

Introduction to Separation Logic

Jean-Marie Madiot

INRIA Paris

Glasgow, 2024 July 30 - August 2 @ SPLV24

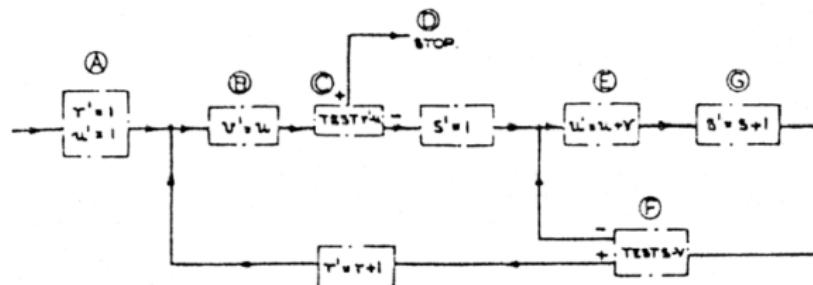
Please interrupt me!

Material from:

- intro talk for math students *in French*
- a separation logic course of *4 * 3 hours*

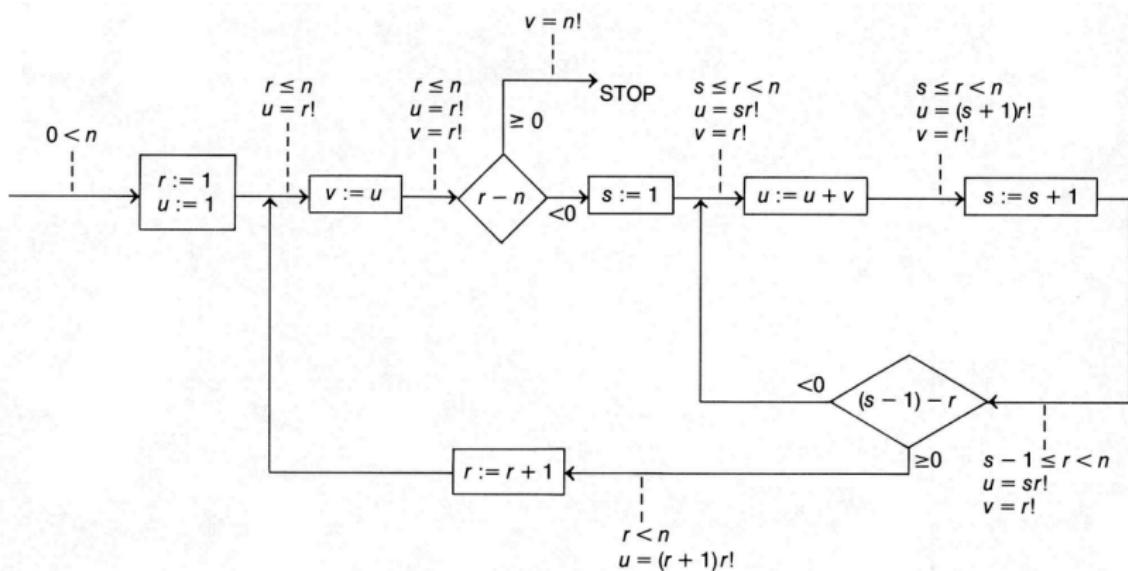
Alan Turing, 1949 : Checking a large routine

"a small routine to obtain $n!$ without the use of a multiplier"



STORAGE LOCATION	(INITIAL) Ⓐ k=6	Ⓑ k=6	Ⓒ k=4	(STOP) Ⓓ k=0	Ⓔ k=3	Ⓕ k=1	Ⓖ k=2
27					S	S+1	S
28		T	T		T	T	T
29	n	n	n	n	T	n	n
30		FF	FF		SET	(S+1) IF (S+1) IF	
31		FF	FF	FF	FF	FF	FF
	TO Ⓐ WITH T'=6 S'=1	TO Ⓑ	TO Ⓒ IF T=n TO Ⓓ IF T<n	TO Ⓗ	TO Ⓕ WITH S'=6 IF S<T	TO Ⓖ	

Rediscovery: Morris, Jones, 1984 : An Early Program Proof by Alan Turing



Turing's argument: no need to have all of the program in mind. It is enough to check, for each box, the consistency between:

- the **precondition** (ingoing annotation)
- the **action** of the instruction
- the **postcondition** (outgoing annotation)

More modern presentation

More structured code (no arrows (no GOTO)), fewer annotations:

- ▶ functions with pre- and postconditions
- ▶ loops with *loop invariants*

```
def fact(n):
    # requires n >= 0, returns n!
    i = 1
    x = 1
    while i < n:
        # invariant: i <= n,  x = i!
        j = 1
        y = x
        while j <= i:
            # invariant: j - 1 <= i < n,  y = j * i!,  x = i!
            y = y + x
            j = j + 1
        i = i + 1
        x = y
    return x
```

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)
- ▶ ignorance/forgetting subtleties depending on the language:
 - ▶ is 2^{63} positive? negative? zero?

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)
- ▶ ignorance/forgetting subtleties depending on the language:
 - ▶ is 2^{63} positive? negative? zero?
 - ▶ is $x + (x = 1)$; forbidden? non-deterministic?

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)
- ▶ ignorance/forgetting subtleties depending on the language:
 - ▶ is 2^{63} positive? negative? zero?
 - ▶ is $x + (x = 1)$; forbidden? non-deterministic?
 - ▶ does the order of evaluation of $f() + g()$; matter?

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)
- ▶ ignorance/forgetting subtleties depending on the language:
 - ▶ is 2^{63} positive? negative? zero?
 - ▶ is $x + (x = 1)$; forbidden? non-deterministic?
 - ▶ does the order of evaluation of $f() + g()$; matter?
 - ▶ worry about cache consistency for parallel programs?

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)
- ▶ ignorance/forgetting subtleties depending on the language:
 - ▶ is 2^{63} positive? negative? zero?
 - ▶ is $x + (x = 1)$; forbidden? non-deterministic?
 - ▶ does the order of evaluation of $f() + g()$; matter?
 - ▶ worry about cache consistency for parallel programs?
- ▶ errors sometimes costly, sometimes dangerous

Problem solved?

Yes, manual proofs are possible, but unsurprisingly:

- ▶ tedious: many many cases, big boring formulas
- ▶ fragile: must be re-checked everytime the code changes
(frequent when coding, and when proving)
- ▶ ignorance/forgetting subtleties depending on the language:
 - ▶ is 2^{63} positive? negative? zero?
 - ▶ is $x + (x = 1)$; forbidden? non-deterministic?
 - ▶ does the order of evaluation of $f() + g()$; matter?
 - ▶ worry about cache consistency for parallel programs?
- ▶ errors sometimes costly, sometimes dangerous

Less boring (and less risky) mathematical problem: *formalize each of those aspects using a computer proof assistant.*

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

`x := 1; x := 2 * x`

does nothing

computes $x = 2$

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

computes $x = 2$

`if x > 0 then r := x else r := 0 - x`

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

computes $x = 2$

`if x > 0 then r := x else r := 0 - x`

computes $r = |x|$

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

computes $x = 2$

`if x > 0 then r := x else r := 0 - x`

computes $r = |x|$

`while n ≠ 0 do n := n - 1`

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

computes $x = 2$

`if x > 0 then r := x else r := 0 - x`

computes $r = |x|$

`while n ≠ 0 do n := n - 1`

computes 0 if \mathbb{N} or $\mathbb{Z}/n\mathbb{Z}$

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$$\begin{aligned} e & ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e \\ s & ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s \end{aligned}$$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

computes $x = 2$

`if x > 0 then r := x else r := 0 - x`

computes $r = |x|$

`while n ≠ 0 do n := n - 1`

computes 0 if \mathbb{N} or $\mathbb{Z}/n\mathbb{Z}$

`r := 1; while n > 0 do (r := r * x; n := n - 1)`

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

does nothing

`x := 1; x := 2 * x`

computes $x = 2$

`if x > 0 then r := x else r := 0 - x`

computes $r = |x|$

`while n ≠ 0 do n := n - 1`

computes 0 if \mathbb{N} or $\mathbb{Z}/n\mathbb{Z}$

`r := 1; while n > 0 do (r := r * x; n := n - 1)`

computes $r = x^n$

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e$

$s ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s$

Examples :

`skip`

`x := 1; x := 2 * x`

`if x > 0 then r := x else r := 0 - x`

`while n ≠ 0 do n := n - 1`

`r := 1; while n > 0 do (r := r * x; n := n - 1)`

`x := 0; s := 1; while s ≤ n do x := x + 1; s := s + 2 * x + 1`

does nothing

computes $x = 2$

computes $r = |x|$

computes 0 if \mathbb{N} or $\mathbb{Z}/n\mathbb{Z}$

computes $r = x^n$

The “While” or “IMP” language

Toy language with variables (x), integers ($n \in \mathbb{Z}$), arithmetical and Boolean expressions, and *while* loops:

$$\begin{aligned} e & ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \leq e_2 \mid e_1 \wedge e_2 \mid \neg e \\ s & ::= \text{skip} \mid x := e \mid s_1; s_2 \mid \text{if } e \text{ then } s_1 \text{ else } s_2 \mid \text{while } e \text{ do } s \end{aligned}$$

Examples :

`skip`

`x := 1; x := 2 * x`

`if x > 0 then r := x else r := 0 - x`

`while n ≠ 0 do n := n - 1`

`r := 1; while n > 0 do (r := r * x; n := n - 1)`

`x := 0; s := 1; while s ≤ n do x := x + 1; s := s + 2 * x + 1`

does nothing

computes $x = 2$

computes $r = |x|$

computes 0 if \mathbb{N} or $\mathbb{Z}/n\mathbb{Z}$

computes $r = x^n$

computes $x = \lfloor \sqrt{n} \rfloor$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s, then holds Q at the end"

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\overline{\{P\} \ s \ \{P\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\overline{\{P\} \ \text{skip} \ \{P\}}$$

$$\overline{\{P[e/x]\} \ x := e \ \{P\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\overline{\{P\} \ \text{skip} \ \{P\}}$$

$$\overline{\{x + 1 = 1\} \ x := x + 1 \ \{x = 1\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\overline{\{P\} \ \text{skip} \ \{P\}}$$

$$\overline{\{P[e/x]\} \ x := e \ \{P\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\overline{\{P\} \ \text{skip} \ \{P\}}$$

$$\overline{\{P[e/x]\} \ x := e \ \{P\}}$$

$$\frac{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}}{\{P\} \ \text{if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\{P\} \ \text{skip} \ \{P\}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \qquad \frac{\{P[e/x]\} \ x := e \ \{P\}}{\{P\} \ \text{if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}{\{P\} \ s_1; s_2 \ \{R\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\{P\} \ \text{skip} \ \{P\}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}}$$
$$\frac{}{\{P\} \ \text{if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}$$
$$\frac{\{I \wedge B\} \ s \ \{I\}}{\{I\} \ \text{while } B \text{ do } s \ \{I \wedge \neg B\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\overline{\{ \quad \} \ x := x + 1 \ \{ \quad \}} \quad \overline{\{ \quad \} \ x := 2 * x \ \{ \quad \}}}{\{ \quad \} \ x := x + 1; x = 2 * x \ \{x > 6\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\overline{\{ \quad \} \ x := x + 1 \ \{ \quad \}} \quad \overline{\{ \quad \} \ x := 2 * x \ \{x > 6\}}}{\{ \quad \} \ x := x + 1; x = 2 * x \ \{x > 6\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\overline{\{ \quad \} \ x := x + 1 \ \{ \quad \}} \quad \overline{\{2x > 6\} \ x := 2 * x \ \{x > 6\}}}{\{ \quad \} \ x := x + 1; x = 2 * x \ \{x > 6\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\overline{\{ \quad \} \ x := x + 1 \ \{2x > 6\}} \quad \overline{\{2x > 6\} \ x := 2 * x \ \{x > 6\}}}{\{ \quad \} \ x := x + 1; x = 2 * x \ \{x > 6\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\overline{\{2(x + 1) > 6\} \ x := x + 1 \ \{2x > 6\}} \quad \overline{\{2x > 6\} \ x := 2 * x \ \{x > 6\}}}{\{ \quad \} \ x := x + 1; x = 2 * x \ \{x > 6\}}$$

Hoare Logic (Floyd 1967, Hoare 1969)

Definition (Hoare triple) we write $\{P\} \ s \ \{Q\}$ for :

"If P holds and we run program s , then holds Q at the end"

Hoare is defined with *inference rules*:

$$\frac{\overline{\{P\} \text{ skip } \{P\}}}{\{P \wedge B\} \ s_1 \ \{Q\} \quad \{P \wedge \neg B\} \ s_2 \ \{Q\}} \quad \frac{\overline{\{P[e/x]\} \ x := e \ \{P\}}}{\{P\} \text{ if } B \text{ then } s_1 \text{ else } s_2 \ \{Q\}}$$
$$\frac{\overline{\{P\} \ s_1 \ \{Q\} \quad \{Q\} \ s_2 \ \{R\}}}{\{P\} \ s_1; s_2 \ \{R\}}$$
$$\frac{\overline{\{I \wedge B\} \ s \ \{I\}}}{\{I\} \text{ while } B \text{ do } s \ \{I \wedge \neg B\}}$$
$$\frac{P \Rightarrow P' \quad \{P'\} \ s \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \ s \ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\overline{\{2(x + 1) > 6\} \ x := x + 1 \ \{2x > 6\}} \quad \overline{\{2x > 6\} \ x := 2 * x \ \{x > 6\}}}{\{2(x + 1) > 6\} \ x := x + 1; x = 2 * x \ \{x > 6\}}$$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(\textcolor{red}{s}, Q)$ such that $\{\text{wp}(\textcolor{red}{s}, Q)\} \textcolor{red}{s} \{Q\}$:

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(\textcolor{red}{s}, Q)$ such that $\{\text{wp}(\textcolor{red}{s}, Q)\} \textcolor{red}{s} \{Q\}$:

$$\text{wp}(\text{skip}, Q) = Q$$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(\textcolor{red}{s}, Q)$ such that $\{\text{wp}(\textcolor{red}{s}, Q)\} \textcolor{red}{s} \{Q\}$:

$$\text{wp}(\text{skip}, Q) = Q$$

$$\text{wp}(\textcolor{red}{x} := e, Q) = Q[e/x]$$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(s, Q)$ such that $\{\text{wp}(s, Q)\} \textcolor{red}{s} \{Q\}$:

$$\begin{aligned}\text{wp}(\text{skip}, Q) &= Q \\ \text{wp}(x := e, Q) &= Q[e/x] \\ \text{wp}(s_1; s_2, Q) &= \text{wp}(s_1, \text{wp}(s_2, Q))\end{aligned}$$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \; s \; \{Q\}$.

Definition of $\text{wp}(s, Q)$ such that $\{\text{wp}(s, Q)\} \; s \; \{Q\}$:

$$\text{wp}(\text{skip}, Q) = Q$$

$$\text{wp}(x := e, Q) = Q[e/x]$$

$$\text{wp}(s_1; s_2, Q) = \text{wp}(s_1, \text{wp}(s_2, Q))$$

$$\text{wp}(\text{if } B \text{ then } s_1 \text{ else } s_2, Q) = (B \Rightarrow \text{wp}(s_1, Q)) \wedge (\neg B \Rightarrow \text{wp}(s_2, Q))$$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(s, Q)$ such that $\{\text{wp}(s, Q)\} \textcolor{red}{s} \{Q\}$:

$$\begin{aligned}\text{wp}(\text{skip}, Q) &= Q \\ \text{wp}(x := e, Q) &= Q[e/x] \\ \text{wp}(s_1; s_2, Q) &= \text{wp}(s_1, \text{wp}(s_2, Q)) \\ \text{wp}(\text{if } B \text{ then } s_1 \text{ else } s_2, Q) &= (B \Rightarrow \text{wp}(s_1, Q)) \wedge (\neg B \Rightarrow \text{wp}(s_2, Q)) \\ \text{wp}(\text{while } B \text{ do } s, Q) &= \exists I \quad I \wedge (B \wedge I \Rightarrow \text{wp}(s, I)) \\ &\quad \wedge (\neg B \wedge I \Rightarrow Q)\end{aligned}$$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(s, Q)$ such that $\{\text{wp}(s, Q)\} \textcolor{red}{s} \{Q\}$:

$$\begin{aligned}\text{wp}(\text{skip}, Q) &= Q \\ \text{wp}(x := e, Q) &= Q[e/x] \\ \text{wp}(s_1; s_2, Q) &= \text{wp}(s_1, \text{wp}(s_2, Q)) \\ \text{wp}(\text{if } B \text{ then } s_1 \text{ else } s_2, Q) &= (B \Rightarrow \text{wp}(s_1, Q)) \wedge (\neg B \Rightarrow \text{wp}(s_2, Q)) \\ \text{wp}(\text{while } B \text{ do } s, Q) &= \exists I \quad I \wedge (B \wedge I \Rightarrow \text{wp}(s, I)) \\ &\quad \wedge (\neg B \wedge I \Rightarrow Q)\end{aligned}$$

Examples :

- ▶ $\text{wp}(x := 4, x \geq 1) = (4 \geq 1)$

Weakest preconditions (Dijkstra, 1975)

Dijkstra decided to really start from the end: there exists a most general precondition P such that $\{P\} \textcolor{red}{s} \{Q\}$.

Definition of $\text{wp}(s, Q)$ such that $\{\text{wp}(s, Q)\} \textcolor{red}{s} \{Q\}$:

$$\begin{aligned}\text{wp}(\text{skip}, Q) &= Q \\ \text{wp}(x := e, Q) &= Q[e/x] \\ \text{wp}(s_1; s_2, Q) &= \text{wp}(s_1, \text{wp}(s_2, Q)) \\ \text{wp}(\text{if } B \text{ then } s_1 \text{ else } s_2, Q) &= (B \Rightarrow \text{wp}(s_1, Q)) \wedge (\neg B \Rightarrow \text{wp}(s_2, Q)) \\ \text{wp}(\text{while } B \text{ do } s, Q) &= \exists I \quad I \wedge (B \wedge I \Rightarrow \text{wp}(s, I)) \\ &\quad \wedge (\neg B \wedge I \Rightarrow Q)\end{aligned}$$

Examples :

- ▶ $\text{wp}(x := 4, x \geq 1) = (4 \geq 1)$
- ▶ $\text{wp}((\text{if } x > 0 \text{ then } r := x \text{ else } r := 0 - x), r = |x|) =$
 $(x > 0 \Rightarrow x = |x|) \wedge (x \leq 0 \Rightarrow (0 - x = |x|))$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$\text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ = \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i - 1)/i][(rx)/x]) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i - 1)/i][(rx)/x]) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= 1x^n = x^n \wedge (i > 0 \wedge rx^i = x^n \Rightarrow rx \cdot x^{i-1} = x^n) \wedge (i \leq 0 \wedge rx^i = x^n \Rightarrow r = x^n) \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i - 1)/i][(rx)/x]) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= 1x^n = x^n \wedge (i > 0 \wedge rx^i = x^n \Rightarrow rx \cdot x^{i-1} = x^n) \wedge (i \leq 0 \wedge rx^i = x^n \Rightarrow r = x^n) \\ &= x^n = x^n \wedge (i > 0 \wedge rx^i = x^n \Rightarrow rx^i = x^n) \wedge (i \leq 0 \wedge rx^i = x^n \Rightarrow r = x^n) = \text{False?} \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n \wedge i \geq 0)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i - 1)/i][(rx)/x]) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= 1x^n = x^n \wedge n \geq 0 \wedge (i > 0 \wedge rx^i = x^n \wedge i \geq 0 \Rightarrow rx \cdot x^{i-1} = x^n \wedge i - 1 \geq 0) \\ &\quad \wedge (i \leq 0 \wedge rx^i = x^n \wedge i \geq 0 \Rightarrow r = x^n) \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n \wedge i \geq 0)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i - 1)/i][(rx)/x]) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= 1x^n = x^n \wedge n \geq 0 \wedge (i > 0 \wedge rx^i = x^n \wedge i \geq 0 \Rightarrow rx \cdot x^{i-1} = x^n \wedge i - 1 \geq 0) \\ &\quad \wedge (i \leq 0 \wedge rx^i = x^n \wedge i \geq 0 \Rightarrow r = x^n) \\ &= n \geq 0 \end{aligned}$$

Example: a program computing x^n

i.e. we want postcondition $r = x^n$ with program:

```
i := n;  
r := 1;  
while i > 0 do  
  (r := r * x;  
   i := i - 1)
```

We come up proudly with invariant $I = (rx^i = x^n \wedge i \geq 0)$

$$\begin{aligned} & \text{wp}(i := n; r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n) \\ &= \text{wp}(r := 1; \text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\ &= \text{wp}(\text{while } i > 0 \text{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x; i := i - 1, I)) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, \text{wp}(i := i - 1, I))) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \text{wp}(r := r * x, I[(i - 1)/i])) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i - 1)/i][(rx)/x]) \wedge (i \leq 0 \wedge I \Rightarrow r = x^n) \\ &= 1x^n = x^n \wedge n \geq 0 \wedge (i > 0 \wedge rx^i = x^n \wedge i \geq 0 \Rightarrow rx \cdot x^{i-1} = x^n \wedge i - 1 \geq 0) \\ &\quad \wedge (i \leq 0 \wedge rx^i = x^n \wedge i \geq 0 \Rightarrow r = x^n) \\ &= n \geq 0 \end{aligned}$$

Frequent: invariant is not general enough + forgot a precondition

The end?

Show that the rules are correct? What does that even mean? One way is formalizing and trusting a small-step **operational semantics** on configurations (m, s) where m is the memory and s the statement/instructions:

$$m, s \rightarrow m', s'$$

The end?

Show that the rules are correct? What does that even mean? One way is formalizing and trusting a small-step **operational semantics** on configurations (m, s) where m is the memory and s the statement/instructions:

$$m, s \rightarrow m', s'$$

Then to define $P(m)$, to mean “ P holds on a memory m ”.

And then to show that if $\{P\} \xrightarrow{s} \{Q\}$ and $P(m)$ then:

- ▶ nothing goes wrong when running (m, s) ,
- ▶ for all m' , if $(m, s) \xrightarrow{*} (m', \text{skip})$, then $Q(m')$.

It was all a lie

The other way around:

Often, *rules* of Hoare logic are in fact a *lemmas*, and $\text{wp}(s, Q)$ is not defined by induction on s but is defined by recursively or inductively

$$\frac{Q(m)}{\text{wp}(\text{skip}, Q)(m)} \quad \frac{\exists m', s' (m, s) \rightarrow (m', s') \quad \forall m', s' (m, s) \rightarrow (m', s') \Rightarrow \text{wp}(s', Q)(m')}{\text{wp}(s, Q)(m)}$$

Many variants exist, inductive, coinductive, predicate on returned values, ghost state, etc

skip operational semantics and jump to slide: 15

Operational semantics

Execution is modelled by a small step **operational semantics**, i.e. a reduction relation $m, s \rightarrow m', s'$

$$\frac{}{m, x := e \rightarrow m(x \mapsto m(e)), \text{skip}}$$

$$\frac{}{m, (\text{skip}; s) \rightarrow m, s}$$

$$\frac{m, s_1 \rightarrow m', s'_1}{m, (s_1; s_2) \rightarrow m', (s'_1; s_2)}$$

$$\frac{m(e) \neq 0}{m, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow m, s_1}$$

$$\frac{m(e) = 0}{m, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow m, s_2}$$

$$\frac{m(e) \neq 0}{m, \text{while } e \text{ do } s \rightarrow m, (s; \text{while } e \text{ do } s)}$$

$$\frac{m(e) = 0}{m, \text{while } e \text{ do } s \rightarrow m, \text{skip}}$$

Operational semantics: example

$(n \mapsto 5), (\textcolor{red}{i := n; r := 1; while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow (n \mapsto 5, i \mapsto 5), (\textcolor{red}{r := 1; while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow (n \mapsto 5, i \mapsto 5, r \mapsto 1), (\textcolor{red}{while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow (n \mapsto 5, i \mapsto 5, r \mapsto 1), (\textcolor{red}{r := r * 2; i := i - 1; while } i > 0 \text{ do } \dots)$
 $\rightarrow (n \mapsto 5, i \mapsto 5, r \mapsto 2), (\textcolor{red}{i := i - 1; while } i > 0 \text{ do } \dots)$
 $\rightarrow (n \mapsto 5, i \mapsto 4, r \mapsto 2), (\textcolor{red}{while } i > 0 \text{ do } \dots)$
 $\rightarrow \rightarrow \rightarrow (n \mapsto 5, i \mapsto 3, r \mapsto 4), (\textcolor{red}{while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow \rightarrow \rightarrow (n \mapsto 5, i \mapsto 2, r \mapsto 8), (\textcolor{red}{while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow \rightarrow \rightarrow (n \mapsto 5, i \mapsto 1, r \mapsto 16), (\textcolor{red}{while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow \rightarrow \rightarrow (n \mapsto 5, i \mapsto 0, r \mapsto 32), (\textcolor{red}{while } i > 0 \text{ do } (r := r * 2; i := i - 1))$
 $\rightarrow (n \mapsto 5, i \mapsto 0, r \mapsto 32), \textcolor{red}{skip}$

Consistent with the earlier example:

$$\{n \geq 0\} \textcolor{red}{i := n; r := 1; while } i > 0 \text{ do } (r := r * 2; i := i - 1) \{r = 2^n\}$$

Separation Logic

Hoare Logic / Floyd-Hoare logic / Program logic / Axiomatic semantics

- ▶ mathematical proofs for imperative programs with variables
- ▶ tedious for pointer aliasing, concurrent programs

Separation Logic: Hoare logic with a more robust notion of memory

- ▶ allocation on the heap
- ▶ operations on pointers
- ▶ many extensions, including concurrent programs

Origins

- ▶ Burstall (1972): reasoning on with no sharing
Distinct Nonrepeating List Systems
- ▶ Reynolds (1999): separating conjunction
Intuitionistic Reasoning about Shared Mutable
- ▶ O'Hearn and Pym (1999): linear resources
The Logic of Bunched Implications
- ▶ O'Hearn, Reynolds, Yang (2001)
Local Reasoning about Programs that Alter Data Structures.

Examples

Micro-controller	Klein et al	ICTA	Isabelle
Assembly language	Chlipala et al	MIT	Coq
Operating system	Shao et al	Yale	Coq
C (drivers)	Yang et al	Oxford	Other
C-light (concurrent)	Appel et al	Princeton	Coq
C11 (concurrent)	Vafeiadis et al	MPI and MSR	Paper
Java	Parkinson et al	MSR and Cambridge	Other
Java	Jacobs et al	Leuven	Verifast
Javascript	Gardner et al	Imperial College	Paper
ML	Morisset et al	Harvard	Coq
OCaml	Charguéraud	Inria	Coq
SML	Myreen et al	U. of Cambridge	HOL
Rust	Jung et al	MPI	Coq-Iris
Time complexity	Guéneau et al	Inria	Coq
Multicore OCaml	Mével et al	Inria	Coq-Iris
Space complexity	Madiot et al	Inria	Coq-Iris
...	Coq-Iris

Interactive vs automated

Automated (Infer, SpaceInvader, Predator, MemCAD, SLAyer)

- ▶ find many bugs, analyse large codebases
- ▶ don't find proofs

Semi-automated (Smallfoot, Heap Hop, VeriFast, Viper)

- ▶ work well on some classes of programs
- ▶ rely on user-provided invariants
- ▶ blackbox problem (hard to debug, extend, prove...)

Interactive (Iris, VST, Ynot, CFML):

- ▶ verified
- ▶ easier to debug, understand, extend
- ▶ expressive
- ▶ often slower

Choice of the logic

Most research projects, including mines, define separation logic inside a logic framework. Here I'll use **Coq** and **Iris**, which is fact a whole proof mode inside Coq:

The screenshot shows the Coq interface with a proof script open. The script is proving a property related to handleability and step functions in Iris. It uses various tactics like `destruct`, `apply`, and `apply invert_pure_wp`. The right pane displays the current state of the proof, showing hypotheses and the goal. The bottom status bar indicates the current file is `pure_wp.v`, the tactic is `Bot`, and the goals count is 17.

```
File Edit Options Buffers Tools Coq Proof-General Outline Holes Hide/Show YASnippet Help
destruct (is_handleable m) as [ h | ] eqn:R.
- (* [m] is handleable *)
  destruct h as [ a | e | eff f ].  

  + (* [ret]'s satisfy [φ] *)
    destruct m as [ [ | [??][ ] | ] ]; discriminate ||  

    by eapply invert_pure_wp_ret in Hm.  

  + (* [throw]'s satisfy [ψ] *)
    destruct m as [ [ | [??][ ] | ] ]; discriminate ||  

    by eapply invert_pure_wp_throw in Hm.  

  + (* [perform]'s are not immediately pure *)
    destruct m as [ [ | [??][ ] | ] ]; discriminate ||  

    by apply invert_pure_wp_stop in Hm.  

- (* [m] is not handleable *)
  intro_state.  

  ewp_mask_intro "Hmod".  

  iSplit.  

  + (* so [m] can step because it is [pure_wp] *)
    destruct (pure_wp_progress m Hm) as [(a, -)]|[e]  

  + (* and no step can change [σ] or escape [pure_wp] *)
    intro_step.  

    ewp_cleanup_mod. ewp_mask_elim.□  

    destruct (pure_wp_preservation Hm Hstep) as (Hm  

    iFrame.  

    by iApply "IH".  

Qed.
```

"IH" : $\forall m : \text{micro A E}$,
 $\lceil \text{pure_wp } m \theta \varphi \psi \rceil \dashv$
 $\text{EWP } m \theta$
 $\lceil E' \lhd \Psi \rceil \triangleright \{ \{ \mid \text{RET } a \Rightarrow \lceil \varphi a \rceil ;$
 $\mid \text{EXN } e \Rightarrow \lceil \psi e \rceil \} \}$

"Hsi" : osiris_state_interp σ

osiris_state_interp σ' *

EWP m'

$\lceil E' \lhd \Psi \rceil \triangleright \{ \{ \mid \text{RET } a \Rightarrow \lceil \varphi a \rceil ;$
 $\mid \text{EXN } e \Rightarrow \lceil \psi e \rceil \} \}$

U:%%- *goals* Bot (17,7) (Coq Goals drag

U:%%- pure_wp.v Bot (1281,37) Git:switching-to- U:%%- *response* All (1,0) (Coq Response dr

Tutorials available at <https://iris-project.org/>

Chapter 1

Separation Logic Operators

The heap in programming

“The heap”

- = the dynamically-allocated memory
- `malloc` in C, `new` in some object-oriented languages,
- sometimes implicit, especially in languages with garbage collection such as Python, Javascript, OCaml
- contains most things (not local variables, which are on the stack)

Mathematical (sub)heaps

Definition

A *map*, or *partial function*, from a set X to a set Y is a subset F of $X \times Y$ such that $(x, y_1) \in F \wedge (x, y_2) \in F \Rightarrow y_1 = y_2$.

Definition

A *subheap*, or more simply *heap*, is a finite map from *locations* (= memory addresses) to *values*.

Examples, with locations = values = \mathbb{N} :

- the empty heap \emptyset
- $\{(1, 2)\}$ and $\{(1, 2), (2, 3)\}$ are heaps,
- $\{(2, 1)\} \cup \{(2, 3)\}$ is not a heap.

Joining

When $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$ we write $h_1 \uplus h_2$ for $h_1 \cup h_2$.

Heap predicates

A *heap predicate* H is a predicate on heaps.
i.e. if h is a heap then $H h$ is a proposition.

In Coq: $H : \text{heap} \rightarrow \text{Prop}$ where Prop is the type of propositions.

Primitive heap predicates:

\top	empty heap
$\lceil P \rceil$	pure fact
$l \mapsto v$	singleton heap
$H * H'$	separating conjunction
$\exists x, H$	existential quantification

Empty heap and pure facts

Definition:

$$\top \equiv \lambda m. m = \emptyset$$

$$\lceil P \rceil \equiv \lambda m. m = \emptyset \wedge P$$

Example: specification of “`let a = 3 and b = a+1`”.

Before: \top

After: $\lceil a = 3 \wedge b = 4 \rceil$

Empty heap and pure facts

Definition:

$$\top \equiv \lambda m. m = \emptyset$$

$$\top P \equiv \lambda m. m = \emptyset \wedge P$$

Example: specification of “`let a = 3 and b = a+1`”.

Before: \top

After: $\top a = 3 \wedge b = 4$

Observe that \top is equivalent to ‘True’.

Singleton heap

Definition:

$$l \mapsto v \quad \equiv \quad \lambda m. \ m = \{(l, v)\} \wedge l \neq \text{null}$$

Example: specification of “`let r = ref 3`”.

Before: `r`

After: `r ↦ 3`

Singleton heap

Definition:

$$l \mapsto v \quad \equiv \quad \lambda m. \ m = \{(l, v)\} \wedge l \neq \text{null}$$

Example: specification of “`let r = ref 3`”.

Before: $\text{ref } \text{null}$

After: $r \mapsto 3$

Example: specification of “`incr s`”.

Before: $s \mapsto n$ for some n

After: $s \mapsto (n + 1)$

Separating conjunction

The heap predicate $H_1 * H_2$ characterizes a heap made of two disjoint parts, one that satisfies H_1 and one that satisfies H_2 .

Example: $(r \mapsto 3) * (s \mapsto 4)$ describes two distinct reference cells.

Separating conjunction

The heap predicate $H_1 * H_2$ characterizes a heap made of two disjoint parts, one that satisfies H_1 and one that satisfies H_2 .

Example: $(r \mapsto 3) * (s \mapsto 4)$ describes two distinct reference cells.

Definition:

$$H_1 * H_2 \quad \equiv \quad \lambda m. \exists m_1 m_2. \begin{cases} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1 m_1 \\ H_2 m_2 \end{cases}$$

where:

$$m_1 \perp m_2 \quad \equiv \quad \text{dom } m_1 \cap \text{dom } m_2 = \emptyset$$

$$m_1 \uplus m_2 \quad \equiv \quad m_1 \cup m_2 \quad \text{when } m_1 \perp m_2$$

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

\top $\lceil 0 = 1 \rceil$ $\lceil 1 = 1 \rceil$ $\lceil 1 = 1 \rceil * \lceil 0 = 1 \rceil$ $1 \mapsto 2$

$(1 \mapsto 2) * \lceil 1 = 1 \rceil$ $(1 \mapsto 2) * (1 \mapsto 3)$ $(1 \mapsto 2) * (2 \mapsto 1)$

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

\top $\lceil 0 = 1 \rceil$ $\lceil 1 = 1 \rceil$ $\lceil 1 = 1 \rceil * \lceil 0 = 1 \rceil$ $1 \mapsto 2$

$(1 \mapsto 2) * \lceil 1 = 1 \rceil$ $(1 \mapsto 2) * (1 \mapsto 3)$ $(1 \mapsto 2) * (2 \mapsto 1)$

Exercise:

- ① specify: `let r = ref 5 and s = ref 3 and t = r.`
- ② specify the state after subsequently executing: `incr r.`
- ③ specify the state after subsequently executing: `incr t.`

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

\top $\lceil 0 = 1 \rceil$ $\lceil 1 = 1 \rceil$ $\lceil 1 = 1 \rceil * \lceil 0 = 1 \rceil$ $1 \mapsto 2$

$(1 \mapsto 2) * \lceil 1 = 1 \rceil$ $(1 \mapsto 2) * (1 \mapsto 3)$ $(1 \mapsto 2) * (2 \mapsto 1)$

Exercise:

- ① specify: `let r = ref 5 and s = ref 3 and t = r.`
- ② specify the state after subsequently executing: `incr r.`
- ③ specify the state after subsequently executing: `incr t.`

Incorrect answer: $(r \mapsto 5) * (s \mapsto 3) * (t \mapsto 5)$.

Heaps and heap predicates

Exercise: give heaps satisfying the following heap predicates

$r \vdash 0 = 1$ $r \vdash 1 = 1$ $r \vdash 1 = 1 * r \vdash 0 = 1$ $1 \mapsto 2$

$(1 \mapsto 2) * r \vdash 1 = 1$ $(1 \mapsto 2) * (1 \mapsto 3)$ $(1 \mapsto 2) * (2 \mapsto 1)$

Exercise:

- ① specify: `let r = ref 5 and s = ref 3 and t = r.`
- ② specify the state after subsequently executing: `incr r.`
- ③ specify the state after subsequently executing: `incr t.`

Incorrect answer: $(r \mapsto 5) * (s \mapsto 3) * (t \mapsto 5)$.

Correct answer:

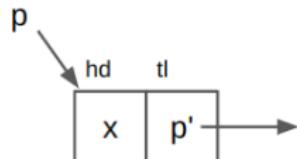
- ① $(r \mapsto 5) * (s \mapsto 3) * r \vdash t = r$
- ② $(r \mapsto 6) * (s \mapsto 3) * r \vdash t = r$
- ③ $(r \mapsto 7) * (s \mapsto 3) * r \vdash t = r$

Record fields

Heap predicate describing the field f of a record at address p :

$$p.f \mapsto v$$

Example:



$$p.hd \mapsto x$$

$$p.tl \mapsto p'$$

Record fields

Heap predicate describing the field f of a record at address p :

$$p.f \mapsto v$$

Example:



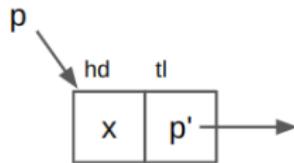
In the C memory model:

$$p.f \mapsto v \equiv (p + f) \mapsto v$$

with

$$\text{hd} \equiv 0 \quad \text{and} \quad \text{tl} \equiv 1$$

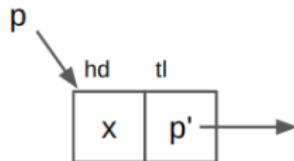
Representation of list cells



$$p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} \quad \equiv \quad p.\text{hd} \mapsto x * p.\text{tl} \mapsto p'$$

Or simply: $p \rightsquigarrow \{x, p'\}$

Representation of list cells



$$p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} \quad \equiv \quad p.\text{hd} \mapsto x * p.\text{tl} \mapsto p'$$

Or simply: $p \rightsquigarrow \{x, p'\}$

Remark: the new arrow symbol will be overloaded later.

Existential quantification

Definition:

$$\exists x. H \equiv \lambda m. \exists x. H m$$

Compare:

$$(\exists x. P) : \text{Prop} \quad \text{when } (P : \text{Prop})$$

$$(\exists x. H) : \text{heap} \rightarrow \text{Prop} \quad \text{when } (H : \text{heap} \rightarrow \text{Prop})$$

Existential quantification

Exercise: give heaps satisfying the following heap predicates

$$\exists x. \lceil (1 \mapsto x) \rceil \quad \exists x. (1 \mapsto x) * (2 \mapsto x) \quad \exists x. \lceil x = x + 1 \rceil$$

$$\exists x. (x \mapsto x + 1) * (x + 1 \mapsto x) \quad \exists x. 1 \mapsto x$$

$$\exists x. (x \mapsto 1) * (x \mapsto 2) \quad \exists P. \lceil P \rceil \quad \exists H. H$$

Summary

$$\text{`True'} \equiv \text{`True'}$$

$$\text{'P'} \equiv \lambda m. m = \emptyset \wedge P$$

$$l \mapsto v \equiv \lambda m. m = \{(l, v)\} \wedge l \neq \text{null}$$

$$H_1 * H_2 \equiv \lambda m. \exists m_1 m_2. \begin{cases} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1 m_1 \\ H_2 m_2 \end{cases}$$

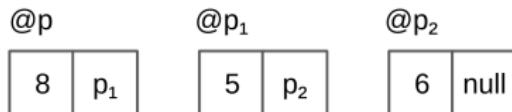
$$\exists x. H \equiv \lambda m. \exists x. H m$$

Chapter 2

Representation Predicate for Lists

Implementation of mutable lists

Mutable lists (C-style), expressed in OCaml extended with null pointers.



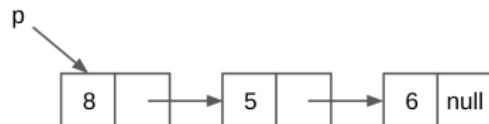
```
type 'a cell = { mutable hd : 'a;  
                 mutable tl : 'a cell }
```

```
{ hd = 8; tl = { hd = 5; tl = { hd = 6; tl = null } } }
```



Representation of mutable lists

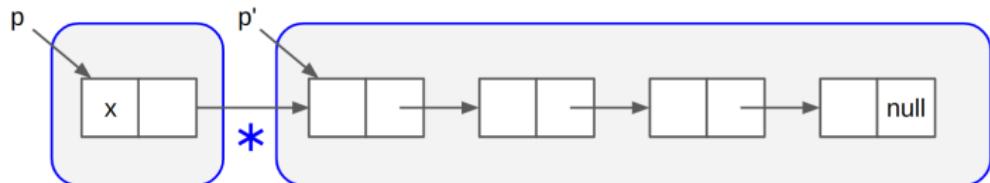
$$L = 8 :: 5 :: 6 :: \text{nil}$$



$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \exists p_1. \quad p \rightsquigarrow \{\text{hd}=8; \text{tl}=p_1\} \\ &\quad * \exists p_2. \quad p_1 \rightsquigarrow \{\text{hd}=5; \text{tl}=p_2\} \\ &\quad * \exists p_3. \quad p_2 \rightsquigarrow \{\text{hd}=6; \text{tl}=p_3\} \\ &\quad * 'p_3 = \text{null}' \end{aligned}$$

Note: $p \rightsquigarrow \text{MList } L$ is notation for $\text{MList } L p$.

Representation predicate



$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

- $| \text{nil} \Rightarrow 'p = \text{null}'$
- $| x :: L' \Rightarrow \exists p'. \ p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$
- $* \quad * \ p' \rightsquigarrow \text{MList } L'$

Separation properties

$p_1 \rightsquigarrow \text{MList } L_1 * p_2 \rightsquigarrow \text{MList } L_2 * p_3 \rightsquigarrow \text{MList } L_3$

Separation enforces: no cycles, and no sharing.

Union heap predicate

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow 'p = \text{null}' \\ &| x :: L' \Rightarrow \exists p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad * \quad p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Equivalent to:

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv 'L = \text{nil} \wedge p = \text{null}' \\ &\vee (\exists x L' p'. \quad 'L = x :: L') \quad) \\ &\quad * \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad * \quad p' \rightsquigarrow \text{MList } L' \end{aligned}$$

where:

$$H_1 \vee H_2 \quad \equiv \quad \lambda m. \quad H_1 m \vee H_2 m$$

List construction

```
let rec build n v =
  if n = 0 then null else
    let p' = build (n-1) v in
    { hd = v; tl = p' }
```

List construction

```
let rec build n v =
  if n = 0 then null else
    let p' = build (n-1) v in
    { hd = v; tl = p' }
```

Pre-condition:

$$n \geq 0$$

Post-condition, where p denotes the result:

$$\exists L. p \rightsquigarrow \text{MList } L * (\text{length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v))$$

List construction: proof (1/2)

$$\exists L. p \rightsquigarrow \text{MList } L * \text{'length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v)$$

Case $n = 0$. We have $p = \text{null}$. We take $L = \text{nil}$.

To produce $p \rightsquigarrow \text{MList } L$, we need to produce $\text{null} \rightsquigarrow \text{MList nil}$.

List construction: proof (1/2)

$$\exists L. p \rightsquigarrow \text{MList } L * \lceil \text{length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v) \rceil$$

Case $n = 0$. We have $p = \text{null}$. We take $L = \text{nil}$.

To produce $p \rightsquigarrow \text{MList } L$, we need to produce $\text{null} \rightsquigarrow \text{MList nil}$. We use:

$$(\text{null} \rightsquigarrow \text{MList nil}) = \lceil \lceil$$

List construction: proof (1/2)

$$\exists L. \ p \rightsquigarrow \text{MList } L * \lceil \text{length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v) \rceil$$

Case $n = 0$. We have $p = \text{null}$. We take $L = \text{nil}$.

To produce $p \rightsquigarrow \text{MList } L$, we need to produce $\text{null} \rightsquigarrow \text{MList nil}$. We use:

$$(\text{null} \rightsquigarrow \text{MList nil}) = \lceil \rceil$$

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &\quad | \text{nil} \Rightarrow \lceil p = \text{null} \rceil \\ &\quad | x :: L' \Rightarrow \exists p'. \ p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad \quad * \ p' \rightsquigarrow \text{MList } L' \end{aligned}$$

List construction: proof (2/2)

$$\exists L. p \rightsquigarrow \text{MList } L * \lceil \text{length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v) \rceil$$

Case $n > 0$. By IH, we have: $p' \rightsquigarrow \text{MList } L'$, with L' of length $n - 1$.

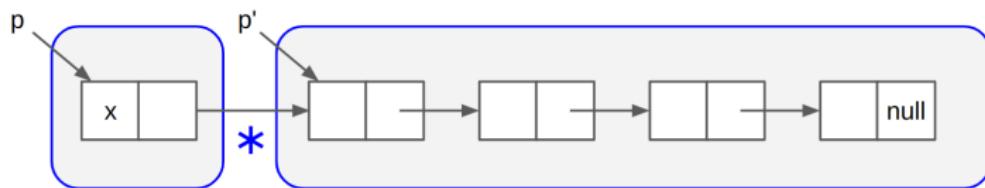
To produce $p \rightsquigarrow \text{MList } L$, we have $p' \rightsquigarrow \text{MList } L'$ and
 $p \rightsquigarrow \{\text{hd} = v; \text{tl} = p'\}$.

List construction: proof (2/2)

$$\exists L. \ p \rightsquigarrow \text{MList } L * \lceil \text{length } L = n \wedge (\forall i. 0 \leq i < n \Rightarrow L[i] = v) \rceil$$

Case $n > 0$. By IH, we have: $p' \rightsquigarrow \text{MList } L'$, with L' of length $n - 1$.

To produce $p \rightsquigarrow \text{MList } L$, we have $p' \rightsquigarrow \text{MList } L'$ and $p \rightsquigarrow \{\text{hd} = v; \text{tl} = p'\}$.



$$(\exists p'. \ p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} * p' \rightsquigarrow \text{MList } L') = p \rightsquigarrow \text{MList } (x :: L')$$

(end of SPLV day 1)

In-place list reversal: code

```
let reverse p0 =
    let r = ref p0 in
    let s = ref null in
    while !r <> null do
        let p = !r in
        r := p.tl;
        p.tl <- !s;
        s := p;
    done;
!s
```

In-place list reversal: code

```
let reverse p0 =
    let r = ref p0 in
    let s = ref null in
    while !r <> null do
        let p = !r in
        r := p.tl;
        p.tl <- !s;
        s := p;
    done;
    !s
```

Exercise:

- ① Specify the state before the loop.
- ② Specify the state after the loop.
- ③ Specify the loop invariant.

In-place list reversal: invariants

Before the loop:

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 * s \mapsto \text{null} * p_0 \rightsquigarrow \text{MList } L$$

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 * s \mapsto \text{null} * p_0 \rightsquigarrow \text{MList } L$$

After the loop:

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 * s \mapsto \text{null} * p_0 \rightsquigarrow \text{MList } L$$

After the loop:

$$\exists q. r \mapsto \text{null} * s \mapsto q * q \rightsquigarrow \text{MList } (\text{rev } L)$$

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 * s \mapsto \text{null} * p_0 \rightsquigarrow \text{MList } L$$

After the loop:

$$\exists q. \ r \mapsto \text{null} * s \mapsto q * q \rightsquigarrow \text{MList } (\text{rev } L)$$

Loop invariant:

In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 * s \mapsto \text{null} * p_0 \rightsquigarrow \text{MList } L$$

After the loop:

$$\exists q. r \mapsto \text{null} * s \mapsto q * q \rightsquigarrow \text{MList } (\text{rev } L)$$

Loop invariant:

$$\begin{aligned} \exists pqL_1L_2. \quad & r \mapsto p * p \rightsquigarrow \text{MList } L_2 \\ & * s \mapsto q * q \rightsquigarrow \text{MList } L_1 \\ & * 'L = \text{rev } L_1 ++ L_2' \end{aligned}$$

In-place list reversal: proof (1/2)

Invariant:

$$\exists pqL_1L_2. \ r \mapsto p * s \mapsto q$$

- * $p \rightsquigarrow \text{MList } L_2$ *
- $q \rightsquigarrow \text{MList } L_1$
- * ' $L = \text{rev } L_1 ++ L_2$ '

Initial state implies the invariant: take $p = p_0$ and $L_1 = \text{nil}$ and $L_2 = L$.

$$r \mapsto p_0 * p_0 \rightsquigarrow \text{MList } L * s \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList nil} * 'L = \text{rev nil} ++ L'$$

In-place list reversal: proof (1/2)

Invariant:

$$\exists pqL_1L_2. \ r \mapsto p * s \mapsto q$$

- * $p \rightsquigarrow \text{MList } L_2$ * $q \rightsquigarrow \text{MList } L_1$
- * $'L = \text{rev } L_1 ++ L_2'$

Initial state implies the invariant: take $p = p_0$ and $L_1 = \text{nil}$ and $L_2 = L$.

$$r \mapsto p_0 * p_0 \rightsquigarrow \text{MList } L * s \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList nil} * 'L = \text{rev nil} ++ L'$$

Invariant implies the final state: exploit $p = \text{null}$.

$$r \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList } L_2 * s \mapsto q * q \rightsquigarrow \text{MList } L_1 * 'L = \text{rev } L_1 ++ L_2'$$

In-place list reversal: proof (1/2)

Invariant:

$$\exists pqL_1L_2. \ r \mapsto p * s \mapsto q$$

- * $p \rightsquigarrow \text{MList } L_2$ * $q \rightsquigarrow \text{MList } L_1$
- * $'L = \text{rev } L_1 ++ L_2'$

Initial state implies the invariant: take $p = p_0$ and $L_1 = \text{nil}$ and $L_2 = L$.

$$r \mapsto p_0 * p_0 \rightsquigarrow \text{MList } L * s \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList nil} * 'L = \text{rev nil} ++ L'$$

Invariant implies the final state: exploit $p = \text{null}$.

$$r \mapsto \text{null} * \text{null} \rightsquigarrow \text{MList } L_2 * s \mapsto q * q \rightsquigarrow \text{MList } L_1 * 'L = \text{rev } L_1 ++ L_2'$$

Derive $L_2 = \text{nil}$ using:

$$(\text{null} \rightsquigarrow \text{MList } L) = 'L = \text{nil}'$$

Conversion rule for empty lists

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

$$\begin{aligned} & | \text{nil} \Rightarrow 'p = \text{null}' \\ & | x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Let us prove: $(\text{null} \rightsquigarrow \text{MList } L) = 'L = \text{nil}'$

– From right to left: we may assume $L = \text{nil}$, thus:

$$'\text{nil} = \text{nil}' = ' = (\text{null} \rightsquigarrow \text{MList nil})$$

Conversion rule for empty lists

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

$$\begin{aligned} & | \text{nil} \Rightarrow 'p = \text{null}' \\ & | x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} * p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Let us prove: $(\text{null} \rightsquigarrow \text{MList } L) = 'L = \text{nil}'$

– From right to left: we may assume $L = \text{nil}$, thus:

$$'\text{nil} = \text{nil}' = ' = (\text{null} \rightsquigarrow \text{MList nil})$$

– From left to right: if $L = \text{nil}$, then easy; otherwise $L = x :: L'$ and:

$$\text{null} \rightsquigarrow \text{MList } (x :: L') = (\exists p'. \text{null} \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} * p' \rightsquigarrow \text{MList } L')$$

contradicts the fact that no data can be allocated at the null address.

In-place list reversal: proof (2/2)

Transition when $p \neq \text{null}$:

$$p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 * 'L = \text{rev } L_1 ++ L_2'$$

to

$$\begin{aligned} & \exists x L'_2 p'. \quad 'L_2 = x :: L'_2 * p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} * p' \rightsquigarrow \text{MList } L'_2 \\ & * q \rightsquigarrow \text{MList } L_1 * 'L = \text{rev } L_1 ++ L_2 \end{aligned}$$

In-place list reversal: proof (2/2)

Transition when $p \neq \text{null}$:

$$p \rightsquigarrow \text{MList } L_2 * q \rightsquigarrow \text{MList } L_1 * 'L = \text{rev } L_1 ++ L_2$$

to

$$\begin{aligned} & \exists x L'_2 p'. \quad 'L_2 = x :: L'_2 * p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} * p' \rightsquigarrow \text{MList } L'_2 \\ & * q \rightsquigarrow \text{MList } L_1 * 'L = \text{rev } L_1 ++ L_2 \end{aligned}$$

After update of $p.\text{tl}$ to the value q :

$$\begin{aligned} & p \rightsquigarrow \{\text{hd}=x; \text{tl}=q\} * q \rightsquigarrow \text{MList } L_1 \\ & * p' \rightsquigarrow \text{MList } L'_2 * 'L = \text{rev } L_1 ++ (x :: L'_2) \end{aligned}$$

to

$$q \rightsquigarrow \text{MList } (x :: L_1) * p' \rightsquigarrow \text{MList } L'_2 * 'L = \text{rev } (x :: L_1) ++ L_2$$

Conversion rules for nonempty lists

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow [p = \text{null}] \\ &| x :: L' \Rightarrow \exists p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad * \quad p' \rightsquigarrow \text{MList } L' \end{aligned}$$
$$\begin{aligned} p \rightsquigarrow \text{MList } L * [p \neq \text{null}] &= \exists x L' p'. \quad [L = x :: L'] \\ &\quad * \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad * \quad p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Summary

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow 'p = \text{null}' \\ &| x :: L' \Rightarrow \exists p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad * \quad p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Chapter 3

Representation Predicate for List Segments

Length of a mutable list using a while loop

```
let rec mlength (p:'a cell) =
  let f = ref p in
  let t = ref 0 in
  while !f != null do
    incr t;
    f := (!f).tl;
  done;
  !t
```

Exercise:

- ① Specify the state before the loop.
- ② Specify the state after the loop.
- ③ Draw a picture describing a state during the loop.
- ④ Try to state a loop invariant. What do you need?

Mlength: initial and final states

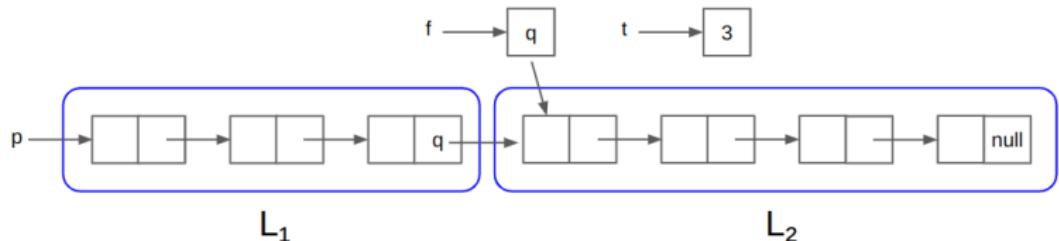
Before the loop:

$$(p \rightsquigarrow \text{MList } L) * (f \mapsto p) * (t \mapsto 0)$$

After the loop:

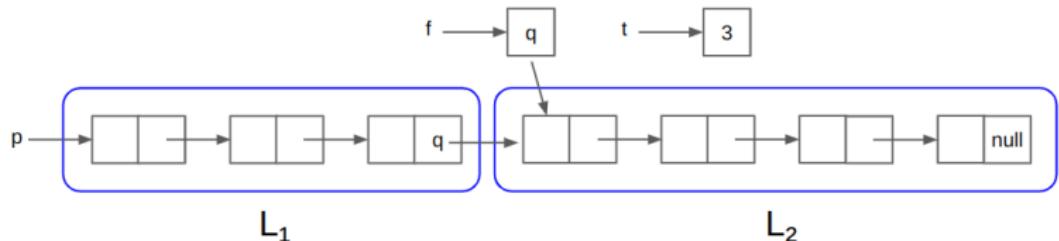
$$(p \rightsquigarrow \text{MList } L) * (f \mapsto \text{null}) * (t \mapsto \text{length } L)$$

Mlength: loop invariant



Loop invariant:

Mlength: loop invariant

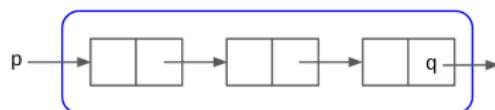


Loop invariant:

$$\exists L_1 L_2 q. \quad 'L = L_1 + L_2' * (t \mapsto \text{length } L_1) * (f \mapsto q) * (p \rightsquigarrow \text{MlistSeg } q L_1) * (q \rightsquigarrow \text{MList } L_2)$$

Representation predicate for list segments

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

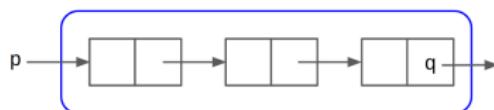
$$\begin{aligned} & | \text{nil} \Rightarrow 'p = \text{null}' \\ & | x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} \\ & \quad * p' \rightsquigarrow \text{MList } L' \end{aligned}$$


Exercise: generalize MList to define $p \rightsquigarrow \text{MlistSeg } q L$, where L denotes the list of items in the list segment from p (inclusive) to q (exclusive).

Representation predicate for list segments

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

- $| \text{nil} \Rightarrow 'p = \text{null}'$
- $| x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$
 - * $p' \rightsquigarrow \text{MList } L'$

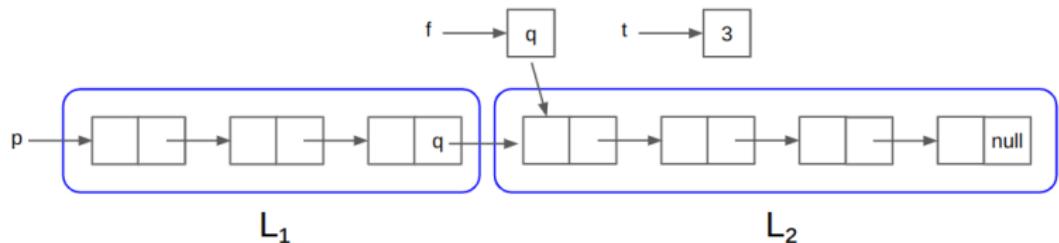


Exercise: generalize MList to define $p \rightsquigarrow \text{MlistSeg } q L$, where L denotes the list of items in the list segment from p (inclusive) to q (exclusive).

$p \rightsquigarrow \text{MlistSeg } q L \equiv \text{match } L \text{ with}$

- $| \text{nil} \Rightarrow 'p = q'$
- $| x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$
 - * $p' \rightsquigarrow \text{MlistSeg } q L'$

Mlength: proof

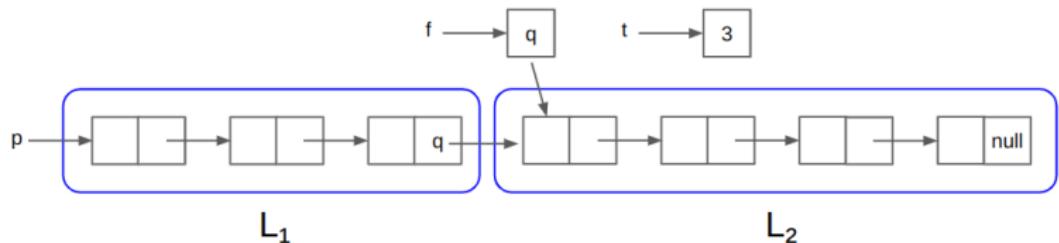


Enter:

$$L_1 = \text{nil} \wedge L_2 = L \wedge q = p$$

$$\textcolor{green}{\vdash} = (p \rightsquigarrow \text{MlistSeg } p \text{ nil})$$

Mlength: proof



Enter:

$$L_1 = \text{nil} \wedge L_2 = L \wedge q = p$$

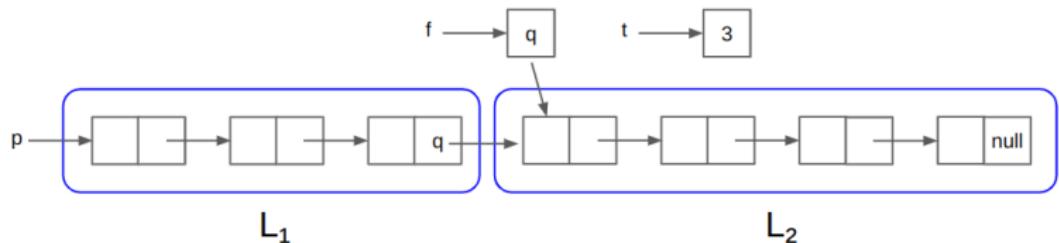
$$\textcolor{green}{\Gamma} = (p \rightsquigarrow \text{MlistSeg } p \text{ nil})$$

Exit:

$$L_1 = L \wedge L_2 = \text{nil} \wedge q = \text{null}$$

$$(p \rightsquigarrow \text{MlistSeg } \text{null } L) = (p \rightsquigarrow \text{MList } L)$$

Mlength: proof



Enter:

$$L_1 = \text{nil} \wedge L_2 = L \wedge q = p$$

$$\textcolor{green}{\Gamma} = (p \rightsquigarrow \text{MlistSeg } p \text{ nil})$$

Exit:

$$L_1 = L \wedge L_2 = \text{nil} \wedge q = \text{null}$$

$$(p \rightsquigarrow \text{MlistSeg } \text{null } L) = (p \rightsquigarrow \text{MList } L)$$

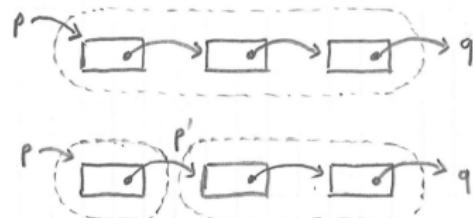
Step:

$$L_2 = x :: L'_2 \wedge q \neq \text{null} \wedge q.\text{tl} = q'$$

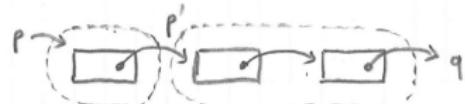
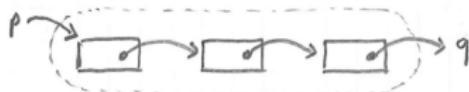
$$\exists q. p \rightsquigarrow \text{MlistSeg } q \ L_1 * q \rightsquigarrow \{\text{hd}=x; \text{tl}=q'\}$$

$$= p \rightsquigarrow \text{MlistSeg } q' (L_1 ++ x :: \text{nil})$$

Splitting rules for list segments

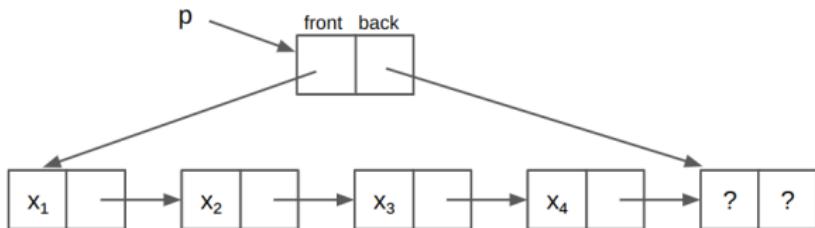

$$p \rightsquigarrow \text{MlistSeg } q (x :: L') = \exists p'. p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} * p' \rightsquigarrow \text{MlistSeg } q L'$$

Splitting rules for list segments


$$p \rightsquigarrow \text{MlistSeg } q (x :: L') = \exists p'. p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} * p' \rightsquigarrow \text{MlistSeg } q L'$$

$$p \rightsquigarrow \text{MlistSeg } q (L_1 + L_2) = \exists p'. p \rightsquigarrow \text{MlistSeg } p' L_1 * p' \rightsquigarrow \text{MlistSeg } q L_2$$

An implementation of mutable queues

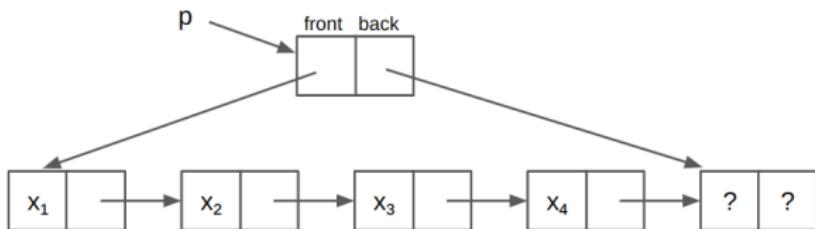


Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked “?” above)

```
type 'a queue = { mutable front : 'a cell;  
                  mutable back : 'a cell; }
```

Exercise: define the representation predicate $p \rightsquigarrow \text{Queue } L$.

An implementation of mutable queues



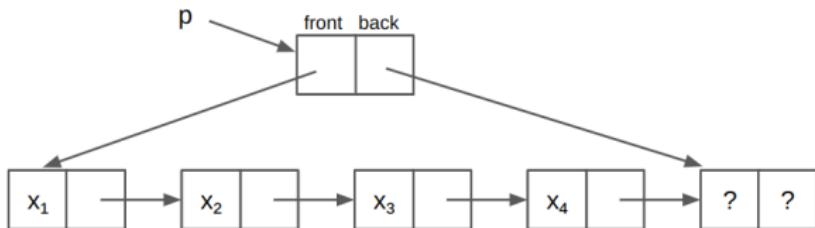
Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked “?” above)

```
type 'a queue = { mutable front : 'a cell;  
                  mutable back : 'a cell; }
```

Exercise: define the representation predicate $p \rightsquigarrow \text{Queue } L$.

$$p \rightsquigarrow \text{Queue } L \equiv \exists fb. \quad p \rightsquigarrow \{\text{front} = f; \text{back} = b\}$$
$$\quad * \quad f \rightsquigarrow \text{MlistSeg } b \ L$$
$$\quad * \quad (b.\text{hd} \mapsto -) \ * \ (b.\text{tl} \mapsto -)$$

An implementation of mutable queues



Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked “?” above)

```
type 'a queue = { mutable front : 'a cell;  
                  mutable back : 'a cell; }
```

Exercise: define the representation predicate $p \rightsquigarrow \text{Queue } L$.

$$\begin{aligned} p \rightsquigarrow \text{Queue } L &\equiv \exists fb. \quad p \rightsquigarrow \{\text{front}=f; \text{back}=b\} \\ &\quad * f \rightsquigarrow \text{MlistSeg } b L \\ &\quad * (b.\text{hd} \mapsto -) * (b.\text{tl} \mapsto -) \end{aligned}$$

Alternative for the last cell: $\exists yq. \quad b \mapsto \{\text{hd}=y; \text{tl}=q\}$.

Summary

$$p \rightsquigarrow \text{MlistSeg } q L \equiv \begin{aligned} &\text{match } L \text{ with} \\ &| \text{nil} \Rightarrow 'p = q' \\ &| x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} \\ &\quad * p' \rightsquigarrow \text{MlistSeg } q L' \end{aligned}$$

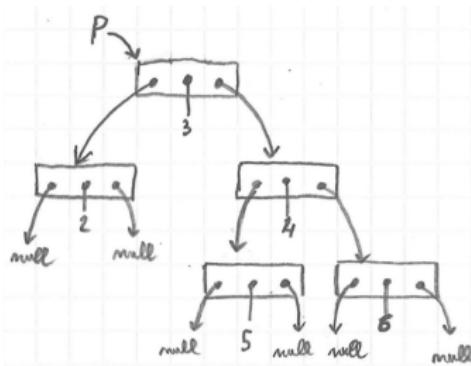
Split and merge of segments:

$$p \rightsquigarrow \text{MlistSeg } q (L_1 + L_2) = \exists p'. \begin{aligned} &p \rightsquigarrow \text{MlistSeg } p' L_1 \\ &\quad * p' \rightsquigarrow \text{MlistSeg } q L_2 \end{aligned}$$

Chapter 4

Representation Predicate for Trees

Implementation of a mutable binary trees



Empty trees represented as null pointers. Nodes represented as records.

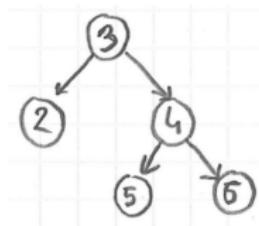
```
type node = {  
    mutable item : int;  
    mutable left : node;  
    mutable right : node; }
```

Logical binary trees

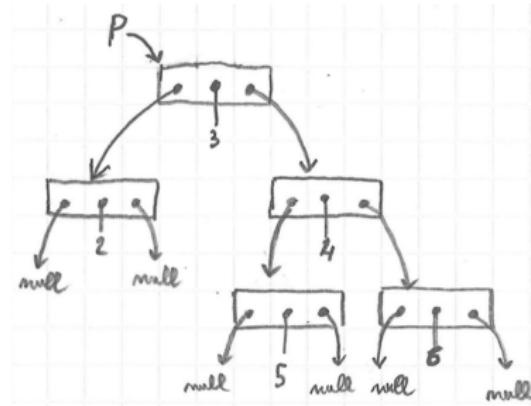
```
Inductive tree : Type :=  
| Leaf : tree  
| Node : int → tree → tree → tree.
```

Example:

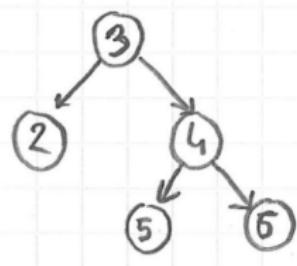
Node 3
(Node 2 Leaf Leaf)
(Node 4 (Node 5 Leaf Leaf)
 (Node 6 Leaf Leaf))



Representation predicate for binary trees



$T =$



Representation predicate:

$$p \rightsquigarrow \text{Mtree } T$$

Representation predicate for binary trees

$$\begin{aligned} p \rightsquigarrow \text{MList } L &\equiv \text{match } L \text{ with} \\ &| \text{nil} \Rightarrow 'p = \text{null}' \\ &| x :: L' \Rightarrow \exists p'. \quad p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \\ &\quad * \quad p' \rightsquigarrow \text{MList } L' \end{aligned}$$

Exercise: define $p \rightsquigarrow \text{Mtree } T$.

Representation predicate for binary trees

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

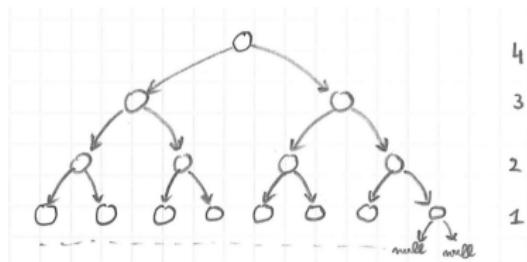
- | nil $\Rightarrow [p = \text{null}]$
- | $x :: L' \Rightarrow \exists p'. \ p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}$
- * $p' \rightsquigarrow \text{MList } L'$

Exercise: define $p \rightsquigarrow \text{Mtree } T$.

$p \rightsquigarrow \text{Mtree } T \equiv \text{match } T \text{ with}$

- | Leaf $\Rightarrow [p = \text{null}]$
- | Node $x T_1 T_2 \Rightarrow \exists p_1 p_2.$
 $p \mapsto \{\text{item} = x; \text{left} = p_1; \text{right} = p_2\}$
- * $p_1 \rightsquigarrow \text{Mtree } T_1$
- * $p_2 \rightsquigarrow \text{Mtree } T_2$

Complete binary tree



$$p \rightsquigarrow \text{MtreeDepth } n T$$

describes a complete binary tree whose leaves are all at depth n .

Complete binary tree (1/2)

```
p ~> Mtree T  ≡  match T with
| Leaf ⇒ 'p = null'
| Node x T1 T2 ⇒ ∃p1p2.
    p ↦ {item=x; left=p1; right=p2}
    * p1 ~> Mtree T1
    * p2 ~> Mtree T2
```

Exercise: define $p \rightsquigarrow \text{MtreeDepth } n T$ by modifying $p \rightsquigarrow \text{Mtree } T$.

Complete binary tree (1/2), solution

```
p ~> MtreeDepth n T  ≡  match T with
| Leaf ⇒ ‘p = null ∧ n = 0’
| Node x T1 T2 ⇒ ∃p1p2. ‘n > 0’ *
  p ↦ {item=x; left=p1; right=p2}
  * p1 ~> MtreeDepth (n - 1) T1
  * p2 ~> MtreeDepth (n - 1) T2
```

Complete binary tree (1/2), solution

$p \rightsquigarrow \text{MtreeDepth } n T \equiv \text{match } T \text{ with}$

- | Leaf \Rightarrow $'p = \text{null} \wedge n = 0'$
- | Node $x T_1 T_2 \Rightarrow \exists p_1 p_2. 'n > 0' *$
 - $p \mapsto \{\text{item} = x; \text{left} = p_1; \text{right} = p_2\}$
 - * $p_1 \rightsquigarrow \text{MtreeDepth } (n - 1) T_1$
 - * $p_2 \rightsquigarrow \text{MtreeDepth } (n - 1) T_2$

Or:

$p \rightsquigarrow \text{MtreeDepth } n T \equiv \text{match } n, T \text{ with}$

- | $O, \text{Leaf} \Rightarrow 'p = \text{null}'$
- | $S m, \text{Node } x T_1 T_2 \Rightarrow \exists p_1 p_2.$
 - $p \mapsto \{\text{item} = x; \text{left} = p_1; \text{right} = p_2\}$
 - * $p_1 \rightsquigarrow \text{MtreeDepth } m T_1$
 - * $p_2 \rightsquigarrow \text{MtreeDepth } m T_2$
- | $-, - \Rightarrow \text{'False'}$

Complete binary tree (2/2)

Exercise: give an alternative definition of " $p \rightsquigarrow \text{MtreeDepth } n T$ " , this time by reusing the definition of $p \rightsquigarrow \text{Mtree } T$ without modification.

Complete binary tree (2/2)

Exercise: give an alternative definition of " $p \rightsquigarrow \text{MtreeDepth } n T$ " , this time by reusing the definition of $p \rightsquigarrow \text{Mtree } T$ without modification.

$$p \rightsquigarrow \text{MtreeDepth } n T \equiv p \rightsquigarrow \text{Mtree } T * \text{'depth } n T'$$

```
Inductive depth : int → tree → Prop :=  
| depth_leaf :  
    depth 0 Leaf  
| depth_node : ∀n x T1 T2,  
    depth n T1 →  
    depth n T2 →  
    depth (n+1) (Node x T1 T2).
```

Complete binary tree of unspecified depth

$$p \rightsquigarrow \text{MtreeDepth } n T \equiv (p \rightsquigarrow \text{Mtree } T) * \lceil \text{depth } n T \rceil$$

Exercise: define a predicate $p \rightsquigarrow \text{MtreeComplete } T$ for describing a mutable complete binary tree, of some unspecified depth.

Complete binary tree of unspecified depth

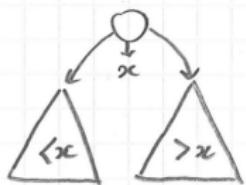
$$p \rightsquigarrow \text{MtreeDepth } n T \equiv (p \rightsquigarrow \text{Mtree } T) * \lceil \text{depth } n T \rceil$$

Exercise: define a predicate $p \rightsquigarrow \text{MtreeComplete } T$ for describing a mutable complete binary tree, of some unspecified depth.

Equivalent definitions for $p \rightsquigarrow \text{MtreeComplete } T$:

- ① $\exists n. p \rightsquigarrow \text{MtreeDepth } n T$
- ② $\exists n. (p \rightsquigarrow \text{Mtree } T) * \lceil \text{depth } n T \rceil$
- ③ $(p \rightsquigarrow \text{Mtree } T) * \lceil \exists n. \text{depth } n T \rceil$

Binary search tree property



The proposition $\text{search } T E$ asserts that the pure tree T describes a valid search tree and that E describes the set integers that it contains.

```
Inductive search : tree → set int → Prop :=  
| search_leaf :  
    search Leaf ∅  
| search_node : ∀x T1 T2,  
    search T1 E1 →  
    search T2 E2 →  
    foreach (is_lt x) E1 →  
    foreach (is_gt x) E2 →  
    search (Node x T1 T2) ({x} ∪ E1 ∪ E2).
```

Binary search tree predicate

Exercise: define a predicate $p \rightsquigarrow \text{MsearchTree } E$ for describing a mutable binary search tree storing the set of elements E .

Binary search tree predicate

Exercise: define a predicate $p \rightsquigarrow \text{MsearchTree } E$ for describing a mutable binary search tree storing the set of elements E .

$$p \rightsquigarrow \text{MsearchTree } E \equiv \exists T. p \rightsquigarrow \text{Mtree } T * [\text{search } T E]$$

Binary search tree predicate

Exercise: define a predicate $p \rightsquigarrow \text{MsearchTree } E$ for describing a mutable binary search tree storing the set of elements E .

$$p \rightsquigarrow \text{MsearchTree } E \equiv \exists T. p \rightsquigarrow \text{Mtree } T * [\text{search } T E]$$

For example, a call “add x p ” can be specified as follows:

- pre-condition: $p \rightsquigarrow \text{MsearchTree } E$
- post-condition: $p \rightsquigarrow \text{MsearchTree } (E \cup \{x\})$

Summary

Common representation predicate for all binary trees:

$$\begin{aligned} p \rightsquigarrow \text{Mtree } T &\equiv \text{match } T \text{ with} \\ &\quad | \text{Leaf} \Rightarrow 'p = \text{null}' \\ &\quad | \text{Node } x \ T_1 \ T_2 \Rightarrow \exists p_1 p_2. \\ &\qquad\qquad p \mapsto \{\text{item} = x; \text{left} = p_1; \text{right} = p_2\} \\ &\qquad\qquad * \ p_1 \rightsquigarrow \text{Mtree } T_1 * \ p_2 \rightsquigarrow \text{Mtree } T_2 \end{aligned}$$

Invariants are expressed on the pure trees:

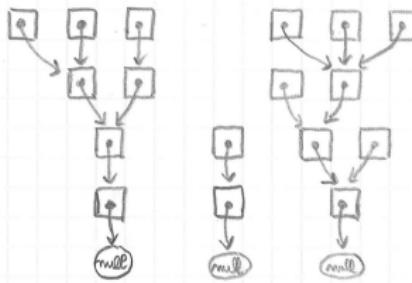
$$p \rightsquigarrow \text{MsearchTree } E \equiv \exists T. \ p \rightsquigarrow \text{Mtree } T * ' \text{search } T \ E'$$

Operations are specified in terms of the model. For example,
add x p changes $p \rightsquigarrow \text{MsearchTree } E$ into
 $p \rightsquigarrow \text{MsearchTree}(E \cup \{x\})$.

Chapter 5

Structures with sharing

The union-find data structure



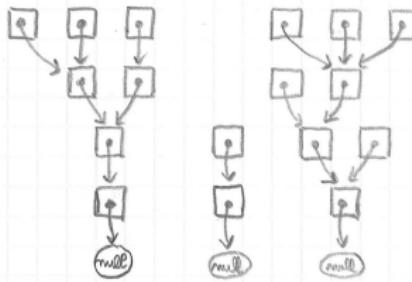
```
type node = node ref
```

Implements an equivalence relation S of type: $\text{loc} \rightarrow \text{loc} \rightarrow \text{Prop}$.

$S a b \Leftrightarrow a$ and b are two valid nodes with the same root

Remark: $S a a$ holds iff a is the location of an existing node.

Representation of union-find cells

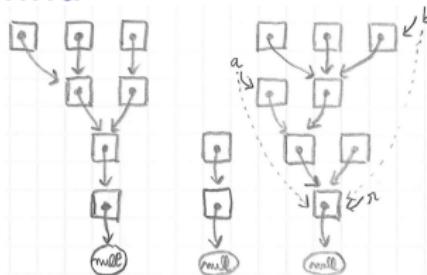


$$(p_1 \mapsto q_1) * (p_2 \mapsto q_2) * \dots * (p_n \mapsto q_n)$$

$$= \circledast_{(p_i, q_i) \in G} (p_i \mapsto q_i)$$

where G is a finite map from locations to locations.

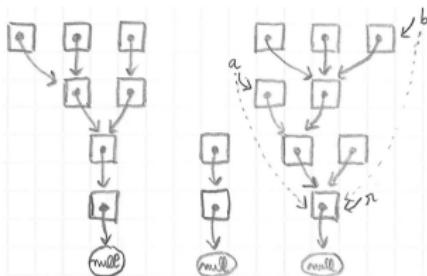
Invariants of union-find



Predicate “ $\text{root } G \ a\ r$ ” asserts that in the graph G , node a has root r .

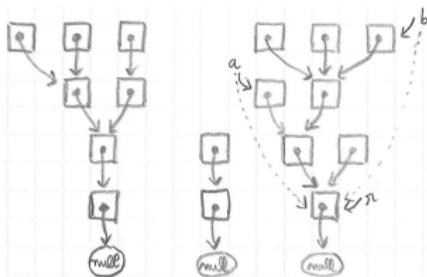
```
Inductive root : fmap loc loc → loc → loc → Prop :=  
| root_init : ∀G x,  
  binds G x null →  
  root G x x  
| root_step : ∀G x y r,  
  binds G x y →  
  y ≠ null →  
  root G y r →  
  root G x r.
```

Specification of the union-find structure



$\text{UnionFind } S \equiv \exists G. (\bigcircledast_{(p,q) \in G} p \mapsto q)$
* $\forall a \in \text{dom } G. \exists r. \text{root } G a r$
* $\forall ab. S a b \Leftrightarrow \exists r. \text{root } G a r \wedge \text{root } G b r$

Specification of the union-find structure



$$\begin{aligned} \text{UnionFind } S &\equiv \exists G. \quad (\bigcircledast_{(p,q) \in G} p \mapsto q) \\ &\quad * \forall a \in \text{dom } G. \exists r. \text{root } G a r \\ &\quad * \forall ab. S a b \Leftrightarrow \exists r. \text{root } G a r \wedge \text{root } G b r \end{aligned}$$

For example, “`let x = is_equiv a b`” is specified as follows:

- pre-condition: ‘ $S a a \wedge S b b$ ’ * UnionFind S
- post-condition: ‘ $x = \text{true} \Leftrightarrow S a b$ ’ * UnionFind S

Summary

Iterated separating conjunction, written \bigcircledast .

For Union-Find:

$$\bigcircledast_{(p,q) \in G} p \mapsto q$$

Chapter 6

Separation Logic Triples

Separation Logic triples

A term t is specified using a Separation Logic triple of the form:

$$\{H\} t \{\lambda x. H'\}$$

- H describes the initial heap
- t is the term being specified
- x is a name for the value produced by t
- H' describes the final heap and the output value x .

Separation Logic triples

A term t is specified using a Separation Logic triple of the form:

$$\{H\} t \{\lambda x. H'\}$$

- H describes the initial heap
- t is the term being specified
- x is a name for the value produced by t
- H' describes the final heap and the output value x .

$$\{H\} t \{Q\}$$

- H (pre-condition) is a predicate of type: $\text{heap} \rightarrow \text{Prop}$
- t has an ML type interpreted in the logic as type A
- Q (post-condition) is a predicate of type: $A \rightarrow \text{heap} \rightarrow \text{Prop}$.

Examples of triples

Example 1:

{ `r` } (`ref` 3) { $\lambda r. r \mapsto 3$ }

Examples of triples

Example 1:

$$\{ \text{'r} \} (\text{ref } 3) \{ \lambda r. r \mapsto 3 \}$$

Example 2:

$$\{ \text{'r} \} (3) \{ \lambda x. \text{'x} = 3 \}$$

Examples of triples

Example 1:

$$\{\text{`r'}\} (\text{ref } 3) \{\lambda r. r \mapsto 3\}$$

Example 2:

$$\{\text{`r'}\} (3) \{\lambda x. \text{'x = 3'}\}$$

Example 3:

$$\{r \mapsto 3\} (!r) \{\lambda x. \text{'x = 3'} * (r \mapsto 3)\}$$

Examples of triples

Example 1:

$$\{\lceil r \rceil\} (\text{ref } 3) \{\lambda r. r \mapsto 3\}$$

Example 2:

$$\{\lceil r \rceil\} (3) \{\lambda x. \lceil x = 3 \rceil\}$$

Example 3:

$$\{r \mapsto 3\} (!r) \{\lambda x. \lceil x = 3 \rceil * (r \mapsto 3)\}$$

Example 4:

$$\{r \mapsto 3\} (\text{incr } r) \{\lambda_. (r \mapsto 4)\}$$

Remark: in “ $\lambda_. (r \mapsto 4)$ ” we do not care about the return value.

Specification of functions

A function f is specified using a triple of the form:

$$\forall a. \quad \{H\} (f a) \{\lambda x. H'\}$$

- H is the pre-condition
- f is the function
- a is the value of the argument
- x is a name for the return value
- H' is the post-condition

Example:

$$\forall rn. \quad \{r \mapsto n\} (\text{incr } r) \{\lambda_. \quad r \mapsto (n + 1)\}$$

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

(ref v)

(!r)

(r := v)

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

(ref v)

(!r)

(r := v)

Solution:

$\forall v. \quad \{^r\} (\text{ref } v) \{\lambda r. (r \mapsto v)\}$

$\forall rv. \quad \{r \mapsto v\} (!r) \{\lambda x. ^r x = v * (r \mapsto v)\}$

Specification of operations on memory cells

Exercise: specify the primitive operations on references.

(ref v)

(!r)

(r := v)

Solution:

$$\forall v. \quad \{^r\} (\text{ref } v) \{\lambda r. (r \mapsto v)\}$$

$$\forall rv. \quad \{r \mapsto v\} (!r) \{\lambda x. {}^r x = v * (r \mapsto v)\}$$

$$\forall rvw. \quad \{r \mapsto w\} (r := v) \{\lambda_. (r \mapsto v)\}$$

$$\forall rv. \quad \{\exists w. r \mapsto w\} (r := v) \{\lambda_. (r \mapsto v)\}$$

$$\forall rv. \quad \{r \mapsto -\} (r := v) \{\lambda_. (r \mapsto v)\}$$

where $(r \mapsto -) \equiv \exists w. r \mapsto w.$

Specification of partial functions

Presentation 1:

$$\forall n. \quad \{^n \geq 0\} (\text{fact } n) \{ \lambda x. ^x = n! \}$$

Presentation 2:

$$\forall n. \, n \geq 0 \Rightarrow \{^n\} (\text{fact } n) \{ \lambda x. ^x = n! \}$$

Interpretation of triples (1/3)

Assume for now that triples describe the entire state.

A triple $\{H\} t \{\lambda x. H'\}$ is interpreted in total correctness as:

$$\forall m. \quad H m \quad \Rightarrow \quad \exists v. \exists m'. \quad \langle t, m \rangle \Downarrow \langle v, m' \rangle \quad \wedge \quad ([x \rightarrow v] H') m'$$

(Assuming a deterministic language)

Interpretation of triples (1/3)

Assume for now that triples describe the entire state.

A triple $\{H\} t \{\lambda x. H'\}$ is interpreted in total correctness as:

$$\forall m. \quad H m \quad \Rightarrow \quad \exists v. \exists m'. \quad \langle t, m \rangle \Downarrow \langle v, m' \rangle \quad \wedge \quad ([x \rightarrow v] H') m'$$

(Assuming a deterministic language)

For $\{H\} t \{Q\}$:

$$\forall m. \quad H m \quad \Rightarrow \quad \exists v. \exists m'. \quad \langle t, m \rangle \Downarrow \langle v, m' \rangle \quad \wedge \quad Q v m'$$

Interpretation of triples (2/3)

In Separation Logic, a triple describes only a part m_1 of the heap. The rest of the heap, call it m_2 , is assumed to remain unchanged.

Recall that:

$$m_1 \perp m_2 \equiv (\text{dom } m_1 \cap \text{dom } m_2 = \emptyset)$$

How is a triple $\{H\} t \{Q\}$ interpreted?

Interpretation of triples (2/3)

In Separation Logic, a triple describes only a part m_1 of the heap. The rest of the heap, call it m_2 , is assumed to remain unchanged.

Recall that:

$$m_1 \perp m_2 \equiv (\text{dom } m_1 \cap \text{dom } m_2 = \emptyset)$$

How is a triple $\{H\} t \{Q\}$ interpreted?

$$\forall m_1 m_2. \left\{ \begin{array}{l} H m_1 \\ m_1 \perp m_2 \end{array} \right. \Rightarrow \exists v. \exists m'_1. \left\{ \begin{array}{l} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m'_1 \uplus m_2 \rangle \\ Q v m'_1 \\ m'_1 \perp m_2 \end{array} \right.$$

Function with garbage collection

What is the *natural* specification of function myref?

```
let myref x =  
    let r = ref x in  
    let s = ref r in  
    r
```

What is missing from our current interpretation of triple?

Function with garbage collection

What is the *natural* specification of function `myref`?

```
let myref x =  
    let r = ref x in  
    let s = ref r in  
    r
```

What is missing from our current interpretation of triple?

From:

$$\{r\} (\text{myref } x) \{\lambda r. r \mapsto x * \exists s. s \mapsto r\}$$

To:

$$\{r\} (\text{myref } x) \{\lambda r. r \mapsto x\}$$

We need the post-condition to describe only a subset of the output heap.

Interpretation of triples (3/3)

Let m_3 describe the *garbage heap*, that is, the part of the final heap that corresponds either to cells from m_1 or to cells allocated during the evaluation of t , and that are not described by the post-condition.

We interpret a triple $\{H\} t \{Q\}$ as:

$$\forall m_1 m_2. \left\{ \begin{array}{l} H m_1 \\ m_1 \perp m_2 \end{array} \right. \Rightarrow \exists v m'_1 m_3. \left\{ \begin{array}{l} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m'_1 \uplus m_2 \uplus m_3 \rangle \\ Q v m'_1 \\ m'_1 \perp m_2 \perp m_3 \end{array} \right.$$

Interpretation of triples (3/3), revisited

We introduce a new heap predicate, written GC , that holds of any heap.

$$\text{GC} \equiv \exists H. H$$

Interpretation of triples (3/3), revisited

We introduce a new heap predicate, written GC , that holds of any heap.

$$\text{GC} \equiv \exists H. H$$

Definition (Separation Logic Triple)

We define $\{H\} t \{Q\}$ as: for all H' and m ,

$$(H * H') m \Rightarrow \exists v m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge (Q v * H' * \text{GC}) m'$$

Summary

Separation Logic triple:

$$\{H\} t \{\lambda x. H'\}$$

Specification of a function:

$$\forall a. \forall \dots \quad \{H\} (f a) \{\lambda x. H'\}$$

Specification of primitive functions:

$$\forall v. \quad \{\text{ref } v\} (\text{ref } v) \{\lambda r. (r \mapsto v)\}$$

$$\forall rv. \quad \{r \mapsto v\} (!r) \{\lambda x. \lceil x = v \rceil * (r \mapsto v)\}$$

$$\forall rv. \quad \{r \mapsto -\} (r := v) \{\lambda_. (r \mapsto v)\}$$

Interpretation of triples: see definition.

Chapter 7

The Frame Rule

Preservation of independent state

We have:

$$\{r \mapsto 2\} (\text{incr } r) \{\lambda_. \ r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2 * s \mapsto 7\} (\text{incr } r) \{\lambda_. \ r \mapsto 3 * s \mapsto 7\}$$

Preservation of independent state

We have:

$$\{r \mapsto 2\} (\text{incr } r) \{\lambda_.\ r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2 * s \mapsto 7\} (\text{incr } r) \{\lambda_.\ r \mapsto 3 * s \mapsto 7\}$$

More generally:

$$\{r \mapsto 2 * H\} (\text{incr } r) \{\lambda_.\ r \mapsto 3 * H\}$$

The frame rule

Principle: a triple remains valid when both the pre-condition and the post-condition are extended with a same heap predicate.

General form:

$$\frac{\{H_1\} \ t \ \{\lambda x. H'_1\}}{\{H_1 * H_2\} \ t \ \{\lambda x. H'_1 * H_2\}}$$

Allows local reasoning / composition of specifications.

The frame rule

Principle: a triple remains valid when both the pre-condition and the post-condition are extended with a same heap predicate.

General form:

$$\frac{\{H_1\} \ t \ \{\lambda x. H'_1\}}{\{H_1 * H_2\} \ t \ \{\lambda x. H'_1 * H_2\}}$$

Allows local reasoning / composition of specifications.

Incompatible with the traditional Hoare Logic rule for assignment

$$\{P[e/x]\} \ x := e \ \{P\}$$

(for this you would need \sim “ t does not modify variables in H_2 ”)

Frame rule and allocation

We have:

$$\{r \mapsto \} (\text{ref } 3) \{\lambda r. (r \mapsto 3)\}$$

By the frame rule, we have:

$$\{s \mapsto 5\} (\text{ref } 3) \{\lambda r. (r \mapsto 3) * (s \mapsto 5)\}$$

Frame rule and allocation

We have:

$$\{r \mapsto \} (\text{ref } 3) \{\lambda r. (r \mapsto 3)\}$$

By the frame rule, we have:

$$\{s \mapsto 5\} (\text{ref } 3) \{\lambda r. (r \mapsto 3) * (s \mapsto 5)\}$$

This post-condition ensures $r \neq s$.

The reference cell r is thus guaranteed to be distinct from any cell that might exist prior to the allocation of r .

(end of SPLV day 2)

Frame rule example: length of a mutable list, recursively

```
let rec mlength (p: 'a cell) =
  if p == null
  then 0
  else let n' = mlength p.tl in 1 + n'
```

Specification:

$$\forall p L. \{p \rightsquigarrow \text{MList } L\} (\text{mlength } p) \{\lambda n. \lceil n = \text{length } L \rceil * p \rightsquigarrow \text{MList } L\}$$

Frame rule example: length of a mutable list, recursively

```
let rec mlength (p: 'a cell) =
  if p == null
  then 0
  else let n' = mlength p.tl in 1 + n'
```

Specification:

$$\forall p L. \{p \rightsquigarrow \text{MList } L\} (\text{mlength } p) \{\lambda n. \lceil n = \text{length } L \rceil * p \rightsquigarrow \text{MList } L\}$$

We prove this specification by induction on L .

Verification of mlength: nil case

Case $L = \text{nil}$. Then $p = \text{null}$. Goal is:

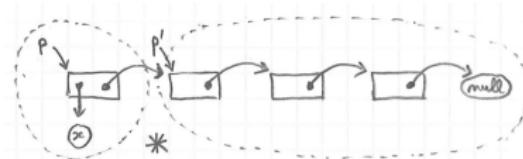
$$\{p \rightsquigarrow \text{MList nil}\} (0) \{\lambda n. \lceil n = \text{length nil} \rceil * p \rightsquigarrow \text{MList nil}\}$$

Same as:

$$\{\lceil p = \text{null} \rceil\} (0) \{\lambda n. \lceil n = 0 \rceil * \lceil p = \text{null} \rceil\}$$

Verification of mlength: using the frame rule

$\forall L p. \quad \{p \rightsquigarrow \text{MList } L\} (\text{mlength } p) \{\lambda n. \lceil n = \text{length } L \rceil * p \rightsquigarrow \text{MList } L\}$



Assume $L = x :: L'$.

$p \rightsquigarrow \text{MList } L$	pre-condition
<u>$p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$</u> * $p' \rightsquigarrow \text{MList } L'$	by unfolding
$p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$ * $p' \rightsquigarrow \text{MList } L' * \lceil n' = L' \rceil$	<u>frame+induction</u>
$p \rightsquigarrow \text{MList } L * \lceil n' + 1 = x :: L' \rceil$	by folding
$p \rightsquigarrow \text{MList } L * \lceil n = L \rceil$	post-condition

Instantiation of the frame rule

Induction hypothesis:

$$\{p' \rightsquigarrow \text{MList } L'\}$$

$$(\text{mlength } p')$$

$$\{\lambda n'. \lceil n = \text{length } L' \rceil * p' \rightsquigarrow \text{MList } L'\}$$

By the frame rule:

$$\{p' \rightsquigarrow \text{MList } L' * p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}\}$$

$$(\text{mlength } p')$$

$$\{\lambda n. \lceil n = \text{length } L' \rceil * p' \rightsquigarrow \text{MList } L' * p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}\}$$

Representation predicate for arrays

Representation predicate for C arrays:

$$p \rightsquigarrow \text{Array } L \quad \equiv \quad \bigcircledast_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

where:

$$p[i] \mapsto v \quad \equiv \quad (p + i) \mapsto v$$

Representation predicate for arrays

Representation predicate for C arrays:

$$p \rightsquigarrow \text{Array } L \equiv \bigcircledast_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

where:

$$p[i] \mapsto v \equiv (p + i) \mapsto v$$

Representation predicate for ML arrays:

$$p \rightsquigarrow \text{Array } L \equiv p.\text{length} \mapsto |L| * \bigcircledast_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

where $p.\text{length} \mapsto n$ and $p[i] \mapsto v$ are abstract definitions for the user.

Dynamic access of ML arrays

Dynamic checks in ocaml:

```
# let v = Array.make 5 0 in Array.get v 7;;
```

Dynamic access of ML arrays

Dynamic checks in ocaml:

```
# let v = Array.make 5 0 in Array.get v 7;;  
Exception: Invalid_argument "index out of bounds".
```

This means preconditions must retain some header information.

Dynamic access of ML arrays

Dynamic checks in ocaml:

```
# let v = Array.make 5 0 in Array.get v 7;;  
Exception: Invalid_argument "index out of bounds".
```

This means preconditions must retain some header information.

When running programs proved in separation logic, one can disable those dynamic checks `ocamlopt -unsafe` and get faster code.

(Same story with `null`.)

Access to a memory cell

In C, record and array accesses are treated uniformly:

$p->hd = v$	'compile' to	$*(p+hd)=v$
$p[i] = v$	'compile' to	$*(p+i)=v$

Access to a memory cell

In C, record and array accesses are treated uniformly:

$p->hd = v$ 'compile' to $*(p+hd)=v$

$p[i] = v$ 'compile' to $*(p+i)=v$

$i[p] = v$ 'compile' to $*(i+p)=v$ (...)

Access to a memory cell

In C, record and array accesses are treated uniformly:

$p->hd = v$	'compile' to	$*(p+hd)=v$
$p[i] = v$	'compile' to	$*(p+i)=v$
$i[p] = v$	'compile' to	$*(i+p)=v \quad (...)$

Common small footprint specification for accessing a memory cell:

$$\begin{aligned} &\{p \mapsto -\} (*p = v) \{\lambda_. \ p \mapsto v\} \\ &\{p \mapsto v\} (*p) \quad \{\lambda x. \ [x = v] * p \mapsto v\} \end{aligned}$$

All other specifications for read and write operations are derivable.

Chapter 9

Heap entailment

Definition of Hprop

Let:

$$\text{Hprop} \equiv \text{heap} \rightarrow \text{Prop}$$

For example:

$$r : \text{Hprop}$$

$$l \mapsto v : \text{Hprop}$$

$$H_1 * H_2 : \text{Hprop}$$

In particular:

$$(*) : \text{Hprop} \rightarrow \text{Hprop} \rightarrow \text{Hprop}$$

Some equations

Associativity: $(H_1 * H_2) * H_3 = H_1 * (H_2 * H_3)$

Commutativity: $H_1 * H_2 = H_2 * H_1$

Neutral element: $H * \top = H$

Extrusion of existentials: $(\exists x. H_1) * H_2 = \exists x. (H_1 * H_2)$ (if $x \notin H_2$)

Extrusion of pure facts: $(\top P * H)m = P \wedge (Hm)$

Assuming functional extensionality ($\forall x. f x = g x \Rightarrow (f = g)$) and propositional extensionality ($P \Leftrightarrow Q \Rightarrow (P = Q)$).

Heap entailment

Definition:

$$H_1 \triangleright H_2 \quad \equiv \quad \forall m. H_1 m \Rightarrow H_2 m$$

For example:

$$(r \mapsto 6) \triangleright \exists n. (r \mapsto n) * \text{'even } n\text{'}$$

Thanks to (\triangleright) , we never need to manipulate heaps explicitly.

\triangleright is a preorder:

REFLEXIVITY

$$\frac{}{H \triangleright H}$$

TRANSITIVITY

$$\frac{H_1 \triangleright H_2 \quad H_2 \triangleright H_3}{H_1 \triangleright H_3}$$

ANTISYMMETRY

$$\frac{H_1 \triangleright H_2 \quad H_2 \triangleright H_1}{H_1 = H_2}$$

Frame property for heap entailment

$$\frac{H_1 \triangleright H'_1}{H_1 * H_2 \triangleright H'_1 * H_2} \text{ENTAIL-FRAME}$$

For example, to prove:

$$(r \mapsto 2) * (s \mapsto 3) \triangleright (r \mapsto 2) * (t \mapsto n)$$

it suffices to prove:

$$(s \mapsto 3) \triangleright (t \mapsto n).$$

Heap implications: true or false?

1. $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$
2. $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$
3. $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$
4. $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$
5. $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$
6. $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$
7. $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$
8. $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$

Heap implications: true or false?

1. $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ true
2. $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$
3. $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$
4. $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$
5. 'False' * $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$
6. $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$
7. $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$
8. $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$

Heap implications: true or false?

- | | | |
|----|---|-------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | |

Heap implications: true or false?

- | | | |
|----|---|-------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | |

Heap implications: true or false?

- | | | |
|----|---|--------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | false* |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | |

Heap implications: true or false?

- | | | |
|----|---|--------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | false* |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | true |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | |

Heap implications: true or false?

- | | | |
|----|---|--------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | false* |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | true |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | false |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | |

Heap implications: true or false?

- | | | |
|----|---|--------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | false* |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | true |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | false |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | true |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | |

Heap implications: true or false?

- | | | |
|----|---|--------|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \triangleright (r \mapsto 3)$ | false* |
| 5. | $\text{'False'} * (r \mapsto 3) \triangleright (s \mapsto 4) * (r \mapsto 4)$ | true |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \triangleright \text{'False'}$ | false |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \triangleright \text{'False'}$ | true |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \triangleright \text{'False'}$ | true |

Instantiation of existentials and propositions

$$(r \mapsto 6) \triangleright (\exists n. (r \mapsto n) * \text{'even } n\text{'})$$

To prove the above, we exhibit an even number n for which $r \mapsto n$.

Rules:

$$\frac{H_1 \triangleright ([x \rightarrow v] H_2)}{H_1 \triangleright (\exists x. H_2)} \text{ EXISTS-R}$$

$$\frac{(H_1 \triangleright H_2) \quad P}{H_1 \triangleright (H_2 * \text{'P'})} \text{ PROP-R}$$

Instantiation of existentials and propositions

$$(r \mapsto 6) \triangleright (\exists n. (r \mapsto n) * \text{'even } n\text{'})$$

To prove the above, we exhibit an even number n for which $r \mapsto n$.

Rules:

$$\frac{H_1 \triangleright ([x \rightarrow v] H_2)}{H_1 \triangleright (\exists x. H_2)} \text{ EXISTS-R}$$

$$\frac{(H_1 \triangleright H_2) \quad P}{H_1 \triangleright (H_2 * \text{'P'})} \text{ PROP-R}$$

Example:

$$\frac{\frac{\frac{\overline{(r \mapsto 6) \triangleright (r \mapsto 6)}}{\text{REFL}} \quad \frac{\overline{\text{even } 6}}{\text{MATH}}}{\overline{(r \mapsto 6) \triangleright (r \mapsto 6) * \text{'even } 6\text'}} \text{ PROP-R}}{\frac{\overline{(r \mapsto 6) \triangleright [n \rightarrow 6] ((r \mapsto n) * \text{'even } n\text')}}{\text{SUBST}}}{(r \mapsto 6) \triangleright \exists n. (r \mapsto n) * \text{'even } n\text'}} \text{ EXISTS-R}$$

Extraction of existentials and propositions

$$(\exists n. \text{'even } n\text{' *} (r \mapsto n)) \triangleright (\exists m. \text{'even } m\text{' *} (r \mapsto m + 2))$$

To prove the above, we show that for any even number n , we have:

$$(r \mapsto n) \triangleright \exists m. \text{'even } m\text{' *} (r \mapsto m + 2)$$

Extraction of existentials and propositions

$$(\exists n. \text{'even } n\text{' * } (r \mapsto n)) \triangleright (\exists m. \text{'even } m\text{' * } (r \mapsto m + 2))$$

To prove the above, we show that for any even number n , we have:

$$(r \mapsto n) \triangleright \exists m. \text{'even } m\text{' * } (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{x \notin H_2 \quad \forall x. (H_1 \triangleright H_2)}{(\exists x. H_1) \triangleright H_2} \text{ EXISTS-L}$$

$$\frac{P \Rightarrow (H_1 \triangleright H_2)}{(\text{'P' * } H_1) \triangleright H_2} \text{ PROP-L}$$

Extraction of existentials and propositions

$$(\exists n. \text{'even } n \text{' * } (r \mapsto n)) \triangleright (\exists m. \text{'even } m \text{' * } (r \mapsto m + 2))$$

To prove the above, we show that for any even number n , we have:

$$(r \mapsto n) \triangleright \exists m. \text{'even } m \text{' * } (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{x \notin H_2 \quad \forall x. (H_1 \triangleright H_2)}{(\exists x. H_1) \triangleright H_2} \text{ EXISTS-L}$$

$$\frac{P \Rightarrow (H_1 \triangleright H_2)}{(\text{'P' * } H_1) \triangleright H_2} \text{ PROP-L}$$

Same with explicit proof contexts:

$$\frac{x \notin H_2 \quad \Gamma, x : A \vdash H_1 \triangleright H_2}{\Gamma \vdash (\exists(x : A). H_1) \triangleright H_2}$$

$$\frac{\Gamma, h : P \vdash H_1 \triangleright H_2}{\Gamma \vdash (\text{'P' * } H_1) \triangleright H_2}$$

Heap implications: true or false?

1. $(r \mapsto 3) \triangleright \exists n. (r \mapsto n)$
2. $\exists n. (r \mapsto n) \triangleright (r \mapsto 3)$
3. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil \triangleright \exists n. \lceil n > 1 \rceil * (r \mapsto (n - 1))$
4. $(r \mapsto 3) * (s \mapsto 3) \triangleright \exists n. (r \mapsto n) * (s \mapsto n)$
5. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil * \lceil n < 0 \rceil \triangleright (r \mapsto m) * (r \mapsto m)$

Heap implications: true or false?

1. $(r \mapsto 3) \triangleright \exists n. (r \mapsto n)$ true
2. $\exists n. (r \mapsto n) \triangleright (r \mapsto 3)$
3. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil \triangleright \exists n. \lceil n > 1 \rceil * (r \mapsto (n - 1))$
4. $(r \mapsto 3) * (s \mapsto 3) \triangleright \exists n. (r \mapsto n) * (s \mapsto n)$
5. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil * \lceil n < 0 \rceil \triangleright (r \mapsto m) * (r \mapsto m)$

Heap implications: true or false?

- | | | |
|----|--|-------|
| 1. | $(r \mapsto 3) \triangleright \exists n. (r \mapsto n)$ | true |
| 2. | $\exists n. (r \mapsto n) \triangleright (r \mapsto 3)$ | false |
| 3. | $\exists n. (r \mapsto n) * \lceil n > 0 \rceil \triangleright \exists n. \lceil n > 1 \rceil * (r \mapsto (n - 1))$ | |
| 4. | $(r \mapsto 3) * (s \mapsto 3) \triangleright \exists n. (r \mapsto n) * (s \mapsto n)$ | |
| 5. | $\exists n. (r \mapsto n) * \lceil n > 0 \rceil * \lceil n < 0 \rceil \triangleright (r \mapsto m) * (r \mapsto m)$ | |

Heap implications: true or false?

1. $(r \mapsto 3) \triangleright \exists n. (r \mapsto n)$ true
2. $\exists n. (r \mapsto n) \triangleright (r \mapsto 3)$ false
3. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil \triangleright \exists n. \lceil n > 1 \rceil * (r \mapsto (n - 1))$ true
4. $(r \mapsto 3) * (s \mapsto 3) \triangleright \exists n. (r \mapsto n) * (s \mapsto n)$
5. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil * \lceil n < 0 \rceil \triangleright (r \mapsto m) * (r \mapsto m)$

Heap implications: true or false?

1. $(r \mapsto 3) \triangleright \exists n. (r \mapsto n)$ true
2. $\exists n. (r \mapsto n) \triangleright (r \mapsto 3)$ false
3. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil \triangleright \exists n. \lceil n > 1 \rceil * (r \mapsto (n - 1))$ true
4. $(r \mapsto 3) * (s \mapsto 3) \triangleright \exists n. (r \mapsto n) * (s \mapsto n)$ true
5. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil * \lceil n < 0 \rceil \triangleright (r \mapsto m) * (r \mapsto m)$

Heap implications: true or false?

1. $(r \mapsto 3) \triangleright \exists n. (r \mapsto n)$ true
2. $\exists n. (r \mapsto n) \triangleright (r \mapsto 3)$ false
3. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil \triangleright \exists n. \lceil n > 1 \rceil * (r \mapsto (n - 1))$ true
4. $(r \mapsto 3) * (s \mapsto 3) \triangleright \exists n. (r \mapsto n) * (s \mapsto n)$ true
5. $\exists n. (r \mapsto n) * \lceil n > 0 \rceil * \lceil n < 0 \rceil \triangleright (r \mapsto m) * (r \mapsto m)$ true

Proving heap entailment relations

Systematic approach to dealing with heap entailment:

- ① extract from left hand side,
- ② instantiate in right hand side,
- ③ cancel equal predicates on both sides.

Example:

$$\frac{a : \text{int}, a > 5 \vdash (r \mapsto 3) * (s \mapsto a) \triangleright (r \mapsto 3) * (s \mapsto a)}{a : \text{int}, a > 5 \vdash (r \mapsto 3) * (s \mapsto a) \triangleright (r \mapsto 3) * (s \mapsto 3 + (a - 3))}$$
$$\frac{a : \text{int}, a > 5 \vdash (r \mapsto 3) * (s \mapsto a) \triangleright \exists m. (r \mapsto 3) * (s \mapsto 3 + m)}{a : \text{int}, a > 5 \vdash (r \mapsto 3) * (s \mapsto a) \triangleright \exists nm. (r \mapsto n) * (s \mapsto n + m)}$$
$$\frac{\emptyset \vdash \exists a. 'a > 5' * (r \mapsto 3) * (s \mapsto a) \triangleright \exists nm. (r \mapsto n) * (s \mapsto n + m)}{\emptyset \vdash (r \mapsto 3) * \exists a. 'a > 5' * (s \mapsto a) \triangleright \exists nm. (s \mapsto n + m) * (r \mapsto n)}$$

Summary

(*) is associative, commutative, and has \top as neutral element.

Existentials and pure facts may be extruded from stars.

(\triangleright) is a partial order, satisfying the frame property.

$\lceil \text{False} \rceil \triangleright H$ is always true.

$\lceil (r \mapsto n) * (r \mapsto m) \rceil$ is equivalent to $\lceil \text{False} \rceil$.

Strategy: extract from the left, instantiate on the right, then cancel out.

Highland Cattle



Chapter 10

Structural rules

Frame rule

$$\frac{\{H_1\} t \{\lambda x. H'_1\}}{\{H_1 * \textcolor{red}{H}_2\} t \{\lambda x. H'_1 * \textcolor{red}{H}_2\}}$$

Reformulation:

$$\frac{\{H_1\} t \{Q_1\}}{\{H_1 * \textcolor{red}{H}_2\} t \{Q_1 \star \textcolor{red}{H}_2\}} \text{ FRAME}$$

with the overloading: $Q \star H \equiv \lambda x. (Q x * H)$

Consequence rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{ CONSEQUENCE}$$

with the overloading:

$$Q' \triangleright Q \equiv \forall x. (Q' x \triangleright Q x)$$

Consequence rule

$$\frac{H \triangleright H' \quad \{H'\} t \{Q'\} \quad Q' \triangleright Q}{\{H\} t \{Q\}} \text{ CONSEQUENCE}$$

with the overloading:

$$Q' \triangleright Q \equiv \forall x. (Q' x \triangleright Q x)$$

Note that H and H' must cover the same set of memory cells, that is, no garbage collection is allowed here. Similarly for Q and Q' .

Recall the need for garbage collection

```
let myref x =  
    let r = ref x in  
    let s = ref r in  
    r
```

From:

$$\{ \top \} (\text{myref } x) \{ \lambda r. r \mapsto x * \exists s. s \mapsto r \}$$

To:

$$\{ \top \} (\text{myref } x) \{ \lambda r. r \mapsto x \}$$

Recall the need for garbage collection

```
let myref x =  
    let r = ref x in  
    let s = ref r in  
    r
```

From:

$$\{ \top \} (\text{myref } x) \{ \lambda r. r \mapsto x * \exists s. s \mapsto r \}$$

To:

$$\{ \top \} (\text{myref } x) \{ \lambda r. r \mapsto x \}$$

Question: can the following rule be used?

$$\frac{\{H\} t \{Q \star H'\}}{\{H\} t \{Q\}} \text{GC-POST},$$

Garbage collection rule

$$\frac{\{H\} t \{Q \star \text{GC}\}}{\{H\} t \{Q\}} \text{ GC-POST} \quad \text{where: } \text{GC} \equiv \exists H'. H'$$

Same as:

$$\frac{\{H\} t \{\lambda x. (Q x * \exists H'. H')\}}{\{H\} t \{Q\}}$$

Observe that H' may depend on the return value x :

For $(\lambda r. r \mapsto x * \exists s. s \mapsto r)$, we may instantiate H' as $(\exists s. s \mapsto r)$.

Garbage collection in the pre-condition

$$\frac{\{H\} t \{Q\}}{\{H * GC\} t \{Q\}} \text{ GC-PRE} \quad \frac{\{H\} t \{Q\}}{\{H * H'\} t \{Q\}} \text{ GC-PRE'}$$

Exercise: show that GC-PRE is derivable from GC-POST and FRAME.

$$\frac{\{H\} t \{Q \star GC\}}{\{H\} t \{Q\}} \text{ GC-POST} \quad \frac{\{H_1\} t \{Q_1\}}{\{H_1 * H_2\} t \{Q_1 \star H_2\}} \text{ FRAME}$$

Garbage collection in the pre-condition

$$\frac{\{H\} t \{Q\}}{\{H * GC\} t \{Q\}} \text{GC-PRE} \quad \frac{\{H\} t \{Q\}}{\{H * H'\} t \{Q\}} \text{GC-PRE'}$$

Exercise: show that GC-PRE is derivable from GC-POST and FRAME.

$$\frac{\{H\} t \{Q * GC\}}{\{H\} t \{Q\}} \text{GC-POST} \quad \frac{\{H_1\} t \{Q_1\}}{\{H_1 * H_2\} t \{Q_1 * H_2\}} \text{FRAME}$$

Proof:

$$\frac{\frac{\frac{\{H\} t \{Q\}}{\{H * GC\} t \{Q * GC\}} \text{FRAME}}{\{H * GC\} t \{Q\}} \text{GC-POST}}{\{H * GC\} t \{Q\}}$$

Garbage collection in the pre-condition

$$\frac{\{H\} t \{Q\}}{\{H * GC\} t \{Q\}} \text{GC-PRE} \quad \frac{\{H\} t \{Q\}}{\{H * H'\} t \{Q\}} \text{GC-PRE'}$$

Exercise: show that GC-PRE is derivable from GC-POST and FRAME.

$$\frac{\{H\} t \{Q \star GC\}}{\{H\} t \{Q\}} \text{GC-POST} \quad \frac{\{H_1\} t \{Q_1\}}{\{H_1 * H_2\} t \{Q_1 \star H_2\}} \text{FRAME}$$

Proof:

$$\frac{\frac{\frac{\{H\} t \{Q\}}{\{H * GC\} t \{Q \star GC\}} \text{FRAME}}{\{H * GC\} t \{Q\}} \text{GC-POST}}{\{H * GC\} t \{Q\}}$$

Remark: no analog in Iris by default, where $P \star Q \vdash P$.

Extraction of existentials and propositions

$$\{\exists n. (r \mapsto n) * \lceil \text{even } n \rceil\} (!r) \{\lambda x. \dots\}$$

To prove the above, we need to show that:

$$\forall n. \text{even } n \Rightarrow \{r \mapsto n\} (!r) \{\lambda x. \dots\}$$

Rules:

$$\frac{x \notin t, Q \quad \forall x. \{H\} t \{Q\}}{\{\exists x. H\} t \{Q\}} \text{ EXISTS}$$

$$\frac{P \Rightarrow \{H\} t \{Q\}}{\{\lceil P \rceil * H\} t \{Q\}} \text{ PROP}$$

Extraction of existentials and propositions

$$\{\exists n. (r \mapsto n) * \lceil \text{even } n \rceil\} (!r) \{\lambda x. \dots\}$$

To prove the above, we need to show that:

$$\forall n. \text{even } n \Rightarrow \{r \mapsto n\} (!r) \{\lambda x. \dots\}$$

Rules:

$$\frac{x \notin t, Q \quad \forall x. \{H\} t \{Q\}}{\{\exists x. H\} t \{Q\}} \text{ EXISTS}$$

$$\frac{P \Rightarrow \{H\} t \{Q\}}{\{\lceil P \rceil * H\} t \{Q\}} \text{ PROP}$$

Chapter 11

Reasoning rules for terms

Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\} (\text{incr } r) \{\lambda _. r \mapsto n + 1\} \\ \{r \mapsto n + 1\} (!r) \{\lambda x. [x = n + 1] * r \mapsto n + 1\}}{\{r \mapsto n\} (\text{incr } r; !r) \{\lambda x. [x = n + 1] * r \mapsto n + 1\}}$$

Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\} (\text{incr } r) \{\lambda _. r \mapsto n + 1\} \\ \{r \mapsto n + 1\} (!r) \{\lambda x. [x = n + 1] * r \mapsto n + 1\}}{\{r \mapsto n\} (\text{incr } r; !r) \{\lambda x. [x = n + 1] * r \mapsto n + 1\}}$$

Exercise: complete the rule for sequences.

$$\frac{\{...\} t_1 \{...\} \quad \{...\} t_2 \{...\}}{\{H\} (t_1 ; t_2) \{Q\}}$$

Reasoning rule for sequences

Solution 1:

$$\frac{\{H\} t_1 \{\lambda_{_} H'\} \quad \{H'\} t_2 \{Q\}}{\{H\} (t_1 ; t_2) \{Q\}}$$

Solution 2:

$$\frac{\{H\} t_1 \{Q'\} \quad \{Q'()\} t_2 \{Q\}}{\{H\} (t_1 ; t_2) \{Q\}} \text{ SEQ}$$

Remark: $Q' = \lambda_{_} H'$ is equivalent to $Q'() = H'$.

Reasoning rule for let-bindings

Exercise: complete the reasoning rule for let-bindings.

$$\frac{\{ \dots \} t_1 \{ \dots \} \quad \forall x. (\{ \dots \} t_2 \{ \dots \})}{\{ H \} (\text{let } x = t_1 \text{ in } t_2) \{ Q \}}$$

Reasoning rule for let-bindings

Exercise: complete the reasoning rule for let-bindings.

$$\frac{\{ \dots \} t_1 \{ \dots \} \quad \forall x. (\{ \dots \} t_2 \{ \dots \})}{\{ H \} (\text{let } x = t_1 \text{ in } t_2) \{ Q \}}$$

Solution:

$$\frac{\{ H \} t_1 \{ Q' \} \quad \forall x. \{ Q' x \} t_2 \{ Q \}}{\{ H \} (\text{let } x = t_1 \text{ in } t_2) \{ Q \}} \text{ LET}$$

Example of let-binding

$$\frac{\{H\} t_1 \{Q'\} \quad \forall x. \{Q' x\} t_2 \{Q\}}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

Exercise: instantiate the rule for let-bindings on the following code.

$$\{r \mapsto 3\} (\text{let } a = !r \text{ in } a+1) \{Q\}$$

Example of let-binding

$$\frac{\{H\} t_1 \{Q'\} \quad \forall x. \{Q' x\} t_2 \{Q\}}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

Exercise: instantiate the rule for let-bindings on the following code.

$$\{r \mapsto 3\} (\text{let } a = !r \text{ in } a+1) \{Q\}$$

Solution:

$$H \equiv (r \mapsto 3)$$

$$Q \equiv \lambda x. \lceil x = 4 \rceil * (r \mapsto 3)$$

$$Q' \equiv \lambda y. \lceil y = 3 \rceil * (r \mapsto 3)$$

Reasoning rule for values

Example:

$$\{ \top \} 3 \{ \lambda x. \top x = 3 \}$$

Rule:

$$\frac{}{\{ \top \} v \{ \lambda x. \top x = v \}} \text{VAL}$$

Exercise: state a reasoning rule for values using a heap implication.

$$\frac{\dots \triangleright \dots}{\{H\} v \{Q\}}$$

Reasoning rule for values

Example:

$$\{ \top \} 3 \{ \lambda x. \top x = 3 \}$$

Rule:

$$\frac{}{\{ \top \} v \{ \lambda x. \top x = v \}} \text{VAL}$$

Exercise: state a reasoning rule for values using a heap implication.

$$\frac{\dots \triangleright \dots}{\{H\} v \{Q\}}$$

Solution:

$$\frac{H \triangleright Q v}{\{H\} v \{Q\}} \text{VAL-FRAME}$$

Reasoning rule for conditionals

Rule:

$$\frac{(v = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (v = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}} \text{ IF}$$

Reasoning rule for conditionals

Rule:

$$\frac{(v = \text{true} \Rightarrow \{H\} t_1 \{Q\}) \quad (v = \text{false} \Rightarrow \{H\} t_2 \{Q\})}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}} \text{ IF}$$

Transformation to A-normal form:

$$(\text{if } t_0 \text{ then } t_1 \text{ else } t_2) = (\text{let } v = t_0 \text{ in } (\text{if } v \text{ then } t_1 \text{ else } t_2))$$

Reasoning rule for top-level functions

Rule:

$$\frac{v_1 = \lambda x. t \quad \{H\} ([x \rightarrow v_2] t) \{Q\}}{\{H\} (v_1 v_2) \{Q\}} \text{ TOP-FUN}$$

Transformation to A-normal form if t_1 or t_2 is not a value:

$$(t_1 t_2) = (\text{let } f = t_1 \text{ in let } v = t_2 \text{ in } (f v))$$

Reasoning rule for top-level functions

Rule:

$$\frac{v_1 = \lambda x. t \quad \{H\} ([x \rightarrow v_2] t) \{Q\}}{\{H\} (v_1 v_2) \{Q\}} \text{ TOP-FUN}$$

Transformation to A-normal form if t_1 or t_2 is not a value:

$$(t_1 t_2) = (\text{let } f = t_1 \text{ in let } v = t_2 \text{ in } (f v))$$

Remark: in general we have $\{H\} t' \{Q\} \Leftrightarrow \{H\} t \{Q\}$ for pure deterministic reductions, i.e. if $\forall m, \langle t, m \rangle \rightarrow_{\det} \langle t', m \rangle$ where

$$\frac{x \rightarrow y \quad \forall z (x \rightarrow z) \Rightarrow y = z}{x \rightarrow_{\det} y}$$

Verification of a simple function

```
let incr r =  
    let a = !r in  
    r := a+1
```

Specification:

$$\forall rn. \quad \{r \mapsto n\} (\text{incr } r) \{\lambda_. \ r \mapsto n + 1\}$$

Verification:

Fix r and n . We need to prove that the body satisfies the specification:

$$\{r \mapsto n\} (\text{let } a = !r \text{ in } r := a+1) \{\lambda_. \ r \mapsto n + 1\}$$

Verification of a simple function

```
let incr r =  
    let a = !r in  
    r := a+1
```

Specification:

$$\forall rn. \quad \{r \mapsto n\} (\text{incr } r) \{\lambda_. \ r \mapsto n + 1\}$$

Verification:

Fix r and n . We need to prove that the body satisfies the specification:

$$\{r \mapsto n\} (\text{let } a = !r \text{ in } r := a+1) \{\lambda_. \ r \mapsto n + 1\}$$

We conclude using the let-binding rule:

$$Q' \equiv \lambda x. [x = n] * (r \mapsto n).$$

Reasoning rule for top-level recursive functions

Rule:

$$\frac{v_1 = \mu f. \lambda x. t \quad \{H\} ([f \rightarrow v_1] [x \rightarrow v_2] t) \{Q\}}{\{H\} (v_1 v_2) \{Q\}}$$
 TOP-FIX

Specification of recursive functions may be established by induction.

Reasoning rule for top-level recursive functions

Rule:

$$\frac{v_1 = \mu f. \lambda x. t \quad \{H\} ([f \rightarrow v_1] [x \rightarrow v_2] t) \{Q\}}{\{H\} (v_1 v_2) \{Q\}}$$
 TOP-FIX

Specification of recursive functions may be established by induction.

Remark: again $v_1 v_2$ makes a pure deterministic reduction to $([f \rightarrow v_1] [x \rightarrow v_2] t)$

Verification of a recursive function

```
let rec mlength (p:'a cell) =
  if p == null
  then 0
  else let p' = p.tl in
        let n' = mlength p' in
        1 + n'
```

Specification:

$$\forall pL. \quad \{p \rightsquigarrow \text{MList } L\} (\text{mlength } p) \{\lambda n. \lceil n = |L| \rceil * p \rightsquigarrow \text{MList } L\}$$

Verification of a recursive function

```
let rec mlength (p:'a cell) =
  if p == null
  then 0
  else let p' = p.tl in
        let n' = mlength p' in
        1 + n'
```

Specification:

$$\forall pL. \quad \{p \rightsquigarrow \text{MList } L\} (\text{mlength } p) \{\lambda n. \lceil n = |L| \rceil * p \rightsquigarrow \text{MList } L\}$$

We prove this specification by induction on L .
Consider p and L . Apply the “if” rule.

Verification of mlength: nil case

Case $p = \text{null}$. Goal is:

$$\{p \rightsquigarrow \text{MList } L\} (0) \{\lambda n. \lceil n = |L| \rceil * p \rightsquigarrow \text{MList } L\}$$

- Replace p with null.
- Rewrite $\text{null} \rightsquigarrow \text{MList } L$ to $\lceil L = \text{nil} \rceil$ in the pre and the post.
- By the PROP rule:

$$L = \text{nil} \Rightarrow \{\lceil \rceil\} (0) \{\lambda n. \lceil n = |L| \rceil * \lceil L = \text{nil} \rceil\}$$

- Replace L with nil.

$$\{\lceil \rceil\} (0) \{\lambda n. \lceil n = |\text{nil}| \rceil * \lceil \text{nil} = \text{nil} \rceil\}$$

- Apply the VAL-FRAME rule.

$$\lceil \rceil \triangleright \lceil 0 = 0 \rceil * \lceil \text{nil} = \text{nil} \rceil$$

Verification of mlength: cons case (1/2)

Case $p \neq \text{null}$. Goal is:

$$\{p \rightsquigarrow \text{MList } L\}$$

$$(\text{let } p' = p.\text{tl} \text{ in let } n' = \text{mlength } p' \text{ in } 1 + n')$$

$$\{\lambda n. \lceil n = |L| \rceil * p \rightsquigarrow \text{MList } L\}$$

– Unfold MList in pre and post, and decompose L as $x :: L'$:

$$p' \rightsquigarrow \text{MList } L' * p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$$

– Apply the let-binding rule, and the read axiom. Remains:

$$\{p' \rightsquigarrow \text{MList } L' * p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}\}$$

$$(\text{let } n' = \text{mlength } p' \text{ in } 1 + n')$$

$$\{\lambda n. \lceil n = |L| \rceil * p' \rightsquigarrow \text{MList } L' * p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}\}$$

– Apply the frame rule to remove: $p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$.

– Apply the let-binding rule with :

$$Q \equiv \lambda n'. \lceil n' = |L'| \rceil * p' \rightsquigarrow \text{MList } L'.$$

Verification of mlength: cons case (2/2)

There remains to prove the two premises of the let-rule.

- First branch, exploit the induction hypothesis:

$$\{p' \rightsquigarrow \text{MList } L'\} (\text{mlength } p') \{\lambda n'. \lceil n' = |L'| \rceil * p' \rightsquigarrow \text{MList } L'\}$$

- Second branch:

$$\{p' \rightsquigarrow \text{MList } L' * \lceil n' = |L'| \rceil\} (1 + n') \{\lambda n. \lceil n = |L| \rceil * p' \rightsquigarrow \text{MList } L'\}$$

- Apply the PROP rule and the VAL-FRAME rule.

$$n' = |L'| \Rightarrow p' \rightsquigarrow \text{MList } L' \triangleright \lceil 1 + n' = |L| \rceil * p' \rightsquigarrow \text{MList } L'$$

- Cancel equal parts, conclude using

$$|L| = |x :: L'| = 1 + |L'| = 1 + n'.$$

Reasoning rule for local functions

Rule template:

$$\frac{\forall f. \ (\dots) \Rightarrow \{H\} t \{Q\}}{\{H\} (\text{let rec } f x = \text{body in } t) \{Q\}}$$

Hypothesis about f :

$$\forall x H' Q'. \ \{H'\} \text{ body } \{Q'\} \Rightarrow \{H'\} (f x) \{Q'\}$$

Rule:

$$\frac{\begin{array}{c} \forall f. \ Pf \Rightarrow \{H\} t \{Q\} \\ Pf = (\forall x H' Q'. \ \{H'\} \text{ body } \{Q'\} \Rightarrow \{H'\} (f x) \{Q'\}) \end{array}}{\{H\} (\text{let rec } f x = \text{body in } t) \{Q\}} \text{ FIX}$$

Summary

$$\overline{\{ \top \} v \{ \lambda x. \top x = v \}}$$

$$\frac{\{H\} t_1 \{Q'\} \quad \forall x. \{Q' x\} t_2 \{Q\}}{\{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

$$\frac{v = \text{true} \Rightarrow \{H\} t_1 \{Q\} \quad v = \text{false} \Rightarrow \{H\} t_2 \{Q\}}{\{H\} (\text{if } v \text{ then } t_1 \text{ else } t_2) \{Q\}}$$

$$\frac{\forall f. \left(\forall x H' Q'. \{H'\} t_1 \{Q'\} \Rightarrow \{H'\} (f x) \{Q'\} \right) \Rightarrow \{H\} t_2 \{Q\}}{\{H\} (\text{let rec } f x = t_1 \text{ in } t_2) \{Q\}}$$

Verification of a for-loop

```
let facto n =
  let r = ref 1 in
  for i = 2 to n do
    let v = !r in
    r := v * i;
  done;
!r
```

Verification of a for-loop

```
let facto n =  
    let r = ref 1 in  
    for i = 2 to n do  
        let v = !r in  
        r := v * i;  
    done;  
    !r
```

Before the loop:

$$r \mapsto 1$$

At each iteration:

$$\text{from } r \mapsto (i-1)! \text{ to } r \mapsto i!$$

After the loop:

$$r \mapsto n!$$

Verification of a for-loop

```
let facto n =  
  let r = ref 1 in  
  for i = 2 to n do  
    let v = !r in  
    r := v * i;  
  done;  
  !r
```

Before the loop:

$$r \mapsto 1$$

At each iteration:

$$\text{from } r \mapsto (i-1)! \text{ to } r \mapsto i!$$

After the loop:

$$r \mapsto n!$$

Loop invariant ($I : \text{int} \rightarrow \text{Hprop}$) that applies for any $i \in [2, n + 1]$:

$$I i \equiv r \mapsto (i-1)!$$

Reasoning rule for for-loops

Reasoning rule for the case $a \leq b$:

$$\frac{\begin{array}{c} H \triangleright I a \\ \forall i \in [a, b]. \quad \{I i\} t \{\lambda _. I(i + 1)\} \\ I(b + 1) \triangleright Q() \end{array}}{\{H\} (\text{for } i = a \text{ to } b \text{ do } t) \{Q\}}$$

Reasoning rule for for-loops

Reasoning rule for the case $a \leq b$:

$$\frac{\begin{array}{c} H \triangleright I a \\ \forall i \in [a, b]. \quad \{I i\} t \{\lambda_. I(i + 1)\} \\ I(b + 1) \triangleright Q() \end{array}}{\{H\} (\text{for } i = a \text{ to } b \text{ do } t) \{Q\}}$$

General rule, also covering the case $a > b$:

$$\frac{\begin{array}{c} H \triangleright I a \\ \forall i \in [a, b]. \quad \{I i\} t \{\lambda_. I(i + 1)\} \\ I(\max a (b + 1)) \triangleright Q() \end{array}}{\{H\} (\text{for } i = a \text{ to } b \text{ do } t) \{Q\}}$$

Reasoning rule for while loops: partial correctness

The loop invariant I describes the state between every iterations.
The post-condition J describes the state after the evaluation of t_1 .

$$\frac{\begin{array}{c} H \triangleright I \\ \{I\} t_1 \{J\} \quad \{J \text{ true}\} t_2 \{\lambda _. I\} \quad J \text{ false} \triangleright Q () \end{array}}{\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}}$$

where ($I : \text{Hprop}$) and ($J : \text{bool} \rightarrow \text{Hprop}$).

Reasoning rule for while loops: partial correctness

The loop invariant I describes the state between every iterations.
The post-condition J describes the state after the evaluation of t_1 .

$$\frac{\begin{array}{c} H \triangleright I \\ \{I\} t_1 \{J\} \quad \{J \text{ true}\} t_2 \{\lambda _. I\} \quad J \text{ false} \triangleright Q () \end{array}}{\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}}$$

where ($I : \text{Hprop}$) and ($J : \text{bool} \rightarrow \text{Hprop}$).

For total correctness: parameterize the invariant with a measure.

Reasoning rule for while loops

We focus on a different approach that:

- inherently supports total correctness;
- allows to apply frame during iterations.

Reasoning rule for while loops

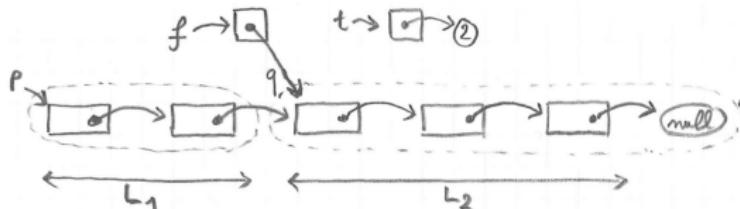
We focus on a different approach that:

- inherently supports total correctness;
- allows to apply frame during iterations.

Prove a triple $\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}$ by induction, using:

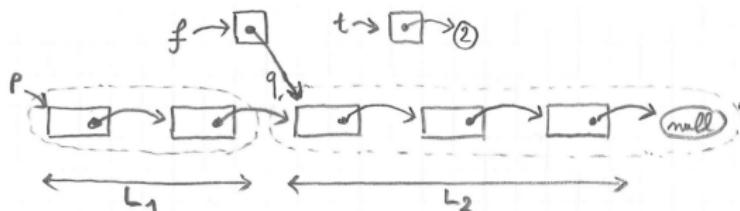
$$\frac{\{H\} (\text{if } t_1 \text{ then } (t_2 ; (\text{while } t_1 \text{ do } t_2)) \text{ else } ()) \{Q\}}{\{H\} (\text{while } t_1 \text{ do } t_2) \{Q\}}$$

Length with a while loop



```
let mlength (p:'a cell) =
  let t = ref 0 in
  let f = ref p in
  while !f != null do
    incr t;
    f := (!f).tl;
  done;
  !t
```

Length with a while loop: induction



We prove by induction on L_2 that for any n and q :

$$\{q \rightsquigarrow \text{MList } L_2 * f \mapsto q * t \mapsto n\}$$

(while !f != null do incr t; f := (!f).tl; done)

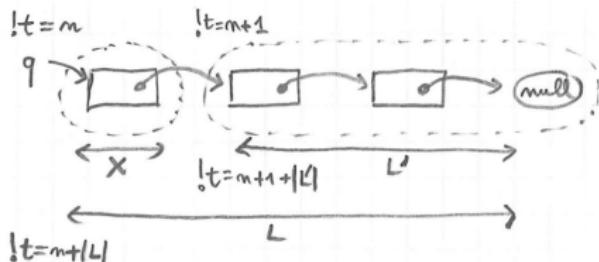
$$\{q \rightsquigarrow \text{MList } L_2 * f \mapsto \text{null} * t \mapsto (n + \text{length } L_2)\}$$

The loop unfolds to:

```
if !f != null
  then (incr t; f := (!f).tl; while .. do .. done)
  else ()
```

Exercise: describe the frame process in the induction for `mlength`.

Length with a while loop: frame process



$q \rightsquigarrow \text{MList } L_2$	$* f \mapsto q$	$* t \mapsto n$	begin
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto q$	$* t \mapsto n$	unfold
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto q$	$* t \mapsto n + 1$	increment
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto q'$	$* t \mapsto n + 1$	shift head
$q \mapsto \{x; q'\} * q' \rightsquigarrow \text{MList } L'_2$	$* f \mapsto \text{null}$	$* t \mapsto n + 1 + L'_2 $	<u>frame+IH</u>
$q \rightsquigarrow \text{MList } L_2$	$* f \mapsto \text{null}$	$* t \mapsto n + L_2 $	fold

Function with local state

Exercise: what is the specification of `f` in the following program?

```
let r = ref 3
let f () =
    incr r
```

Then, show that the code below returns 5.

```
f();
f();
!r
```

Function with local state

Exercise: what is the specification of `f` in the following program?

```
let r = ref 3
let f () =
    incr r
```

Then, show that the code below returns 5.

```
f();
f();
!r
```

Specification:

$$\forall n. \quad \{r \mapsto n\} (f \circ) \{\lambda_. \quad r \mapsto n + 1\}$$

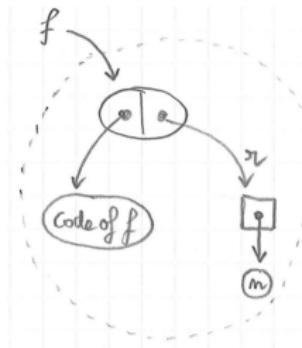
Successive states:

$$r \mapsto 3 \quad r \mapsto 4 \quad r \mapsto 5$$

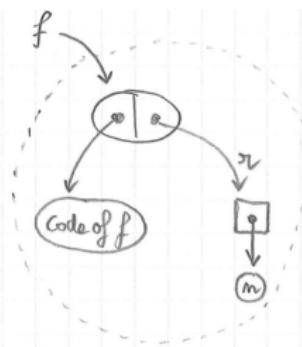
Counter function: code

```
let mkcounter () =  
  let r = ref 0 in  
(fun () -> incr r; !r)
```

```
let c = mkcounter() in  
let x = c() in  
let y = c() in  
assert (x = 1 && y = 2)
```

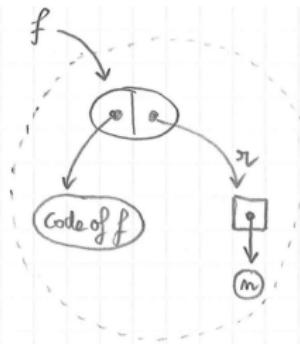


Counter function: specification



$$f \rightsquigarrow \text{Count } n \equiv \exists r. (r \mapsto n) * \forall i. \{r \mapsto i\} (f \circ) \{\lambda x. [x = i + 1] * (r \mapsto i + 1)\}$$

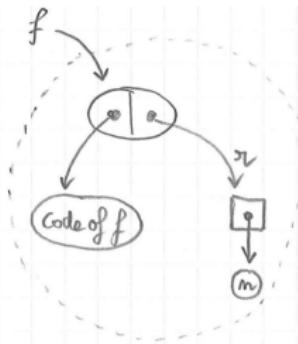
Counter function: specification



$$\begin{aligned} f \rightsquigarrow \text{Count } n &\equiv \exists r. (r \mapsto n) \\ &\quad * \forall i. \{r \mapsto i\} (f \circ) \{\lambda x. [x = i + 1] * (r \mapsto i + 1)\} \end{aligned}$$

Exercise: specify a counter function, only in terms of $f \rightsquigarrow \text{Count } n$.

Counter function: specification



$$f \rightsquigarrow \text{Count } n \equiv \exists r. (r \mapsto n) * \forall i. \{r \mapsto i\} (f ()) \{\lambda x. [x = i + 1] * (r \mapsto i + 1)\}$$

Exercise: specify a counter function, only in terms of $f \rightsquigarrow \text{Count } n$.

$$\{[\cdot]\} (\text{mkcounter}()) \{\lambda f. f \rightsquigarrow \text{Count } 0\}$$

$$\forall fi. \{f \rightsquigarrow \text{Count } i\} (f ()) \{\lambda x. [x = i + 1] * f \rightsquigarrow \text{Count } (i + 1)\}$$

Chapter 14

Basic higher-order functions

Apply

```
let apply f x =  
  f x
```

Specification:

$$\begin{aligned} \forall fxHQ. & \quad \{H\} (f x) \{Q\} \\ \Rightarrow & \quad \{H\} (\text{apply } f x) \{Q\} \end{aligned}$$

Apply

```
let apply f x =  
  f x
```

Specification:

$$\begin{aligned}\forall fxHQ. \quad & \{H\} (f x) \{Q\} \\ \Rightarrow \quad & \{H\} (\text{apply } f x) \{Q\}\end{aligned}$$

This is equivalent to the form below, which involves nested triples:

$$\forall fxHQ. \quad \{H * ^r \{H\} (f x) \{Q\} ^r\} (\text{apply } f x) \{Q\}$$

Function twice

```
let twice f =  
  f(); f()
```

Specification:

$$\begin{aligned}\forall f H' Q. \quad & \{H\} (f()) \{\lambda_. H'\} \\ \wedge \quad & \{H'\} (f()) \{Q\} \\ \Rightarrow \quad & \{H\} (\text{twice } f) \{Q\}\end{aligned}$$

Function repeat

```
let repeat n f =
  for i = 0 to n-1 do
    f()
  done
```

Exercise: specify `repeat`, using an invariant I , of type $\text{int} \rightarrow \text{Hprop}$.

Function repeat

```
let repeat n f =
  for i = 0 to n-1 do
    f()
  done
```

Exercise: specify `repeat`, using an invariant I , of type $\text{int} \rightarrow \text{Hprop}$.

$$\begin{aligned} \forall n f I. \quad & (\forall i \in [0, n]. \quad \{I i\} (f ()) \{\lambda_. \ I (i + 1)\}) \\ \Rightarrow \quad & \{I 0\} (\text{repeat } n f) \{\lambda_. \ I n\} \end{aligned}$$

The premise consists of a family of hypotheses describing the behavior of applications of f to particular arguments.

Chapter 15

Higher order iteration

Iteration over a pure list



For pedagogical purposes, “pure lists” live outside the heap and need no representation predicate. (In practice, most (but not all) lists are allocated on the heap.)



```
let rec iter f l =
  match l with
  | [] -> ()
  | x::t -> f x; iter f t
```

Exercise: specify `iter`, using an invariant I , of type $\text{list } \alpha \rightarrow \text{Hprop}$.

Iteration over a pure list



For pedagogical purposes, “pure lists” live outside the heap and need no representation predicate. (In practice, most (but not all) lists are allocated on the heap.)



```
let rec iter f l =
  match l with
  | [] -> ()
  | x::t -> f x; iter f t
```

Exercise: specify `iter`, using an invariant I , of type $\text{list } \alpha \rightarrow \text{Hprop}$.

$$\begin{aligned} \forall f I. \quad & (\forall x k. \{I k\} (f x) \{\lambda_. I(k\&x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

where $k\&x \equiv k++(x :: \text{nil})$.

Length using iter

$$\begin{aligned} & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k\&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } f l) \{\lambda_. Il\} \end{aligned}$$

```
let length l =
  let r = ref 0 in
  iter (fun x -> incr r) l;
  !r
```

Exercise: give the instantiation of the invariant I for iter ; then, write the specialization of the specification of iter to I and to $(\text{fun } x \rightarrow \text{incr } r)$; finally, check that the premise is provable.

Length using iter

$$\begin{aligned} & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k\&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } fl) \{\lambda_. Il\} \end{aligned}$$

```
let length l =
  let r = ref 0 in
  iter (fun x -> incr r) l;
  !r
```

Exercise: give the instantiation of the invariant I for iter ; then, write the specialization of the specification of iter to I and to $(\text{fun } x \rightarrow \text{incr } r)$; finally, check that the premise is provable.

Invariant: $I \equiv \lambda k. r \mapsto |k|$.

$$\begin{aligned} & (\forall xk. \{r \mapsto |k|\} (\text{incr } r) \{\lambda_. r \mapsto |k| + 1\}) \\ \Rightarrow & \{r \mapsto 0\} (\text{iter } fl) \{\lambda_. r \mapsto |l|\} \end{aligned}$$

Sum using iter

$$\begin{aligned} & (\forall xk. \{I k\} (f x) \{\lambda_. I(k\&x)\}) \\ \Rightarrow & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

```
let sum l =
  let r = ref 0 in
  iter (fun x -> r := !r + x) l;
  !r
```

Exercise: give the invariant I involved in the above call to `iter`.

Sum using iter

$$\begin{aligned} & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } f l) \{\lambda_. Il\} \end{aligned}$$

```
let sum l =
  let r = ref 0 in
  iter (fun x -> r := !r + x) l;
  !r
```

Exercise: give the invariant I involved in the above call to `iter`.

$$I \equiv \lambda k. r \mapsto \text{Sum } k$$

where:

$$\text{Sum } k \equiv \text{Fold } (+) 0 k$$

Constraints over the items

$$\begin{aligned} & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } fl) \{\lambda_. Il\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{x_1} + \frac{10}{x_2} + \dots + \frac{10}{x_n}$$

```
iter (fun x -> r := !r + 10 / x) [2; -3; 4]
```

Constraints over the items

$$\begin{aligned} & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } fl) \{\lambda_. Il\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{x_1} + \frac{10}{x_2} + \dots + \frac{10}{x_n}$$

`iter (fun x -> r := !r + 10 / x) [2; -3; 4]`

The above specification of `iter` is too weak. More general specification:

$$\begin{aligned} \forall fIl. \quad & (\forall xk. \textcolor{red}{x \in l} \Rightarrow \{Ik\} (fx) \{\lambda_. I(k&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } fl) \{\lambda_. Il\} \end{aligned}$$

Constraints over the items, in order

$$\begin{aligned} \forall f l. \quad & (\forall x k. x \in l \Rightarrow \{I k\} (f x) \{\lambda_. I(k \& x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } f l) \{\lambda_. I l\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{\frac{10}{\frac{10}{\frac{10}{\ddots}} + x_1} + x_2} \cdot \cdot \cdot + x_n$$

```
iter (fun x -> r := 10 / (!r + x)) [2; -3; 4]
```

Constraints over the items, in order

$$\begin{aligned} \forall fI l. \quad & (\forall xk. \ x \in l \Rightarrow \{Ik\} (fx) \{\lambda_. \ I(k&x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } fl) \{\lambda_. \ Il\} \end{aligned}$$

Given a list $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$, let us compute:

$$\frac{10}{\begin{array}{c} +x_n \\ \cdot \cdot \cdot \\ \frac{10}{\begin{array}{c} +x_2 \\ \frac{10}{0+x_1} \end{array}} \end{array}}$$

```
iter (fun x -> r := 10 / (!r + x)) [2; -3; 4]
```

The above specification of `iter` is too weak. Most-general specification:

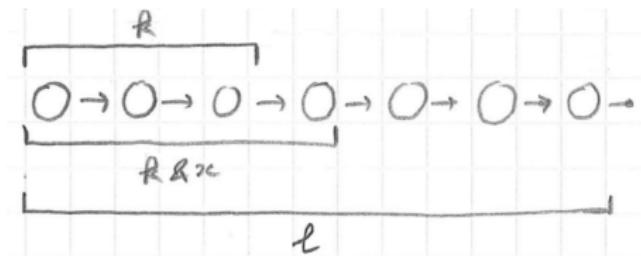
$$\begin{aligned} \forall fI l. \quad & (\forall xks. \ l = k+x :: s \Rightarrow \{Ik\} (fx) \{\lambda_. \ I(k&x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } fl) \{\lambda_. \ Il\} \end{aligned}$$

Verification of iter

$$\begin{aligned} & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}) \\ \Rightarrow & \{Inil\} (\text{iter } f l) \{\lambda_. Il\} \end{aligned}$$

```
let rec iter f l =
  match l with
  | [] -> ()
  | x::t -> f x; iter f t
```

How to prove that the code satisfies its specification?



Verification of iter: generalized principle

Assume:

$$\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}$$

Prove:

$$\{Inil\} (\text{iter } fl) \{\lambda_. Il\}$$

Verification of iter: generalized principle

Assume:

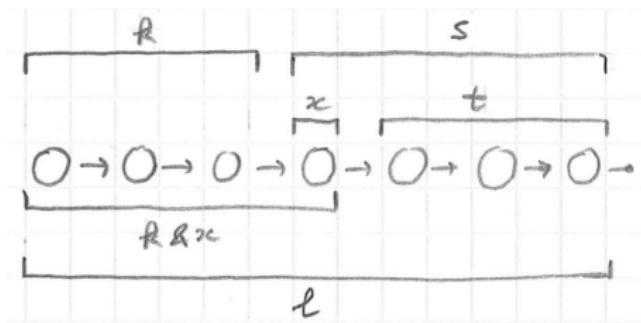
$$\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}$$

Prove:

$$\{Inil\} (\text{iter } fl) \{\lambda_. Il\}$$

Proof by induction over a generalized statement:

$$\forall sk. \{Ik\} (\text{iter } fs) \{\lambda_. I(k+s)\}$$



Verification of iter: induction

```
let rec iter f s =
  match s with
  | [] -> ()
  | x::t -> f x; iter f t
```

Assume: $\forall xk. \{I k\} (f x) \{\lambda_. I(k & x)\}$

Prove: $\forall ks. \{I k\} (\text{iter } f s) \{\lambda_. I(k+s)\}$

By induction on s :

- Case $s = \text{nil}$. Goal is: $\{I k\} (\text{iter } f \text{ nil}) \{\lambda_. I(k + \text{nil})\}$.
This triple simplifies to: $\{I k\} () \{\lambda_. I k\}$, which is correct.

Verification of iter: induction

```
let rec iter f s =
  match s with
  | [] -> ()
  | x::t -> f x; iter f t
```

Assume: $\forall k. \{I k\} (f x) \{\lambda_. I(k&x)\}$

Prove: $\forall ks. \{I k\} (\text{iter } f s) \{\lambda_. I(k+s)\}$

By induction on s :

- Case $s = \text{nil}$. Goal is: $\{I k\} (\text{iter } f \text{ nil}) \{\lambda_. I(k+\text{nil})\}$.
This triple simplifies to: $\{I k\} () \{\lambda_. I k\}$, which is correct.
- Case $s = x :: t$. Goal is:
 $\{I k\} (\text{iter } f (x :: t)) \{\lambda_. I(k+(x :: t))\}$.

HYPOTHESIS-ON-F

INDUCTION-HYPOTHESIS

$\{I k\} (f x) \{\lambda_. I(k&x)\}$	$\{I(k&x)\} (\text{iter } f t) \{\lambda_. I((k&x)+t)\}$	SEQ
	$\{I k\} (f x; \text{iter } f t) \{I((k&x)+t)\}$	

Invariant on remaining items

$$(\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}) \Rightarrow \{Inil\} (\text{iter } f l) \{\lambda_. Il\}$$

$$(\forall \dots \{\dots\} (fx) \{\lambda_. \dots\}) \Rightarrow \{I'l\} (\text{iter } f l) \{\lambda_. I'nil\}$$

Exercise:

- specify `iter` using an invariant that depends on the list of items remaining to process, instead of on the list of items already processed.
- prove the old specification derivable from the new one,
- prove the new specification derivable from the old (most general) one.

Invariant on remaining items

$$(\forall xk. \{Ik\} (fx) \{\lambda_. I(k&x)\}) \Rightarrow \{Inil\} (\text{iter } fl) \{\lambda_. Il\}$$

$$(\forall \dots \{\dots\} (fx) \{\lambda_. \dots\}) \Rightarrow \{I'l\} (\text{iter } fl) \{\lambda_. I'nil\}$$

Exercise:

- specify `iter` using an invariant that depends on the list of items remaining to process, instead of on the list of items already processed.
- prove the old specification derivable from the new one,
- prove the new specification derivable from the old (most general) one.

$$(\forall xs. \{I'(x :: s)\} (fx) \{\lambda_. I's\}) \Rightarrow \{I'l\} (\text{iter } fl) \{\lambda_. I'nil\}$$

$$Ik \equiv \exists s. \lceil l = k ++ s \rceil * I's$$

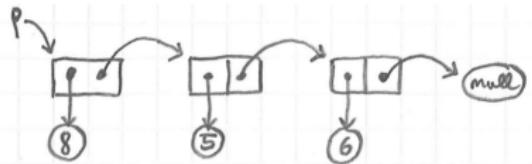
$$I's \equiv \exists k. \lceil l = k ++ s \rceil * Ik$$

More general specification for non-deterministic iterators

- Jean-Christophe Filliâtre and Mário Pereira. 2016.
A Modular Way to Reason About Iteration.
- François Pottier. 2017.
Verifying a hash table and its iterators in higher-order separation logic.

Idea: two predicates *enumerated* and *completed*.

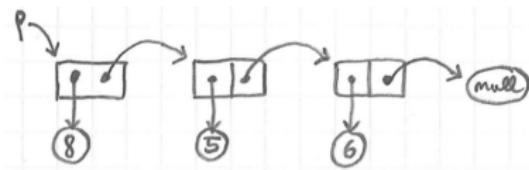
Iterating over a mutable list



```
let rec miter f p =
  if p == null
  then ()
  else (f p.hd; miter f p.tl)
```

Iterating over a mutable list

$$\begin{aligned} \forall f I. \quad & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k\&x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } fl) \{\lambda_. Il\} \end{aligned}$$



Specification:

$$\begin{aligned} \forall fpIl. \quad & (\forall xk. \{Ik\} (fx) \{\lambda_. I(k\&x)\}) \\ \Rightarrow \quad & \{p \rightsquigarrow \text{MList } l * I \text{ nil}\} (\text{miter } fp) \{\lambda_. p \rightsquigarrow \text{MList } l * Il\} \end{aligned}$$

Remark: calls to f will not modify the structure of the list while iterating.

Chapter 17

Higher-order representation predicates

Overview

- ① Higher-order predicate:

$p \rightsquigarrow \text{MList } L$ is generalized into $p \rightsquigarrow \text{Mlistof } R \ L$

- ② Identity representation predicate:

$p \rightsquigarrow \text{Mlistof Id } L$ is the same as $p \rightsquigarrow \text{MList } L$

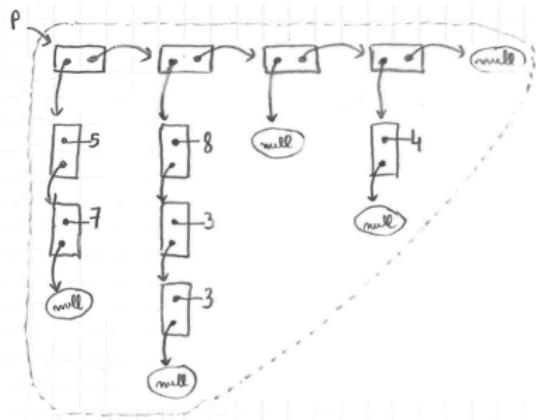
- ③ Control accesses:

$\{p \rightsquigarrow \text{MCellof Id } v_1 \ R_2 \ V_2\} \ (p.\text{hd}) \ \{\lambda x. \lceil x = v_1 \rceil * \dots\}$

- ④ Compose recursively:

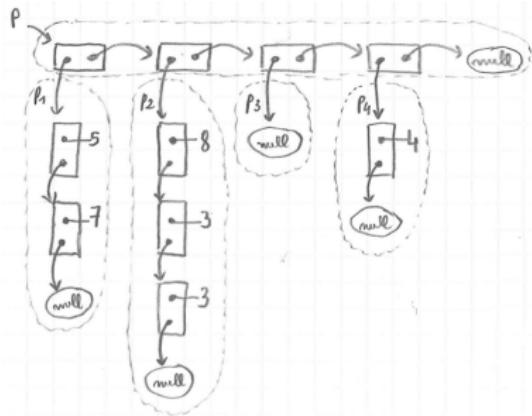
$p \rightsquigarrow \text{Nodeof } R \ X \ (\text{Mlistof} \ (\text{Narytreeof } R)) \ L$

Mutable list of disjoint mutable lists


$$L = (5::7::\text{nil})::(8::3::3::\text{nil})::(\text{nil})::(4::\text{nil}) :: \text{nil}$$
$$p \rightsquigarrow \text{MlistofMlist } L$$

(to be later generalized into: $p \rightsquigarrow \text{Mlistof } RL$)

Representation using iterated star



$$L = (5::7::\text{nil}) :: (8::3::3::\text{nil}) :: (\text{nil}) :: (4::\text{nil}) :: \text{nil}$$

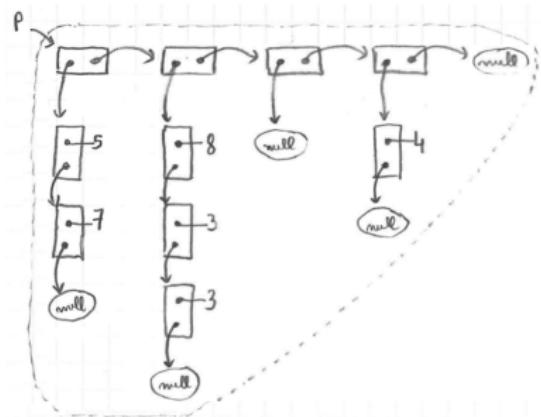
$$K = p_1 :: p_2 :: p_3 :: p_4 :: \text{nil}$$

$$p \rightsquigarrow \text{MlistofMlist } L \equiv \exists K. \quad p \rightsquigarrow \text{MList } K$$

$$* \quad \circledast_{i \in [0, |L|]} (K[i]) \rightsquigarrow \text{MList}(L[i])$$

$$* \quad '|K| = |L|'$$

Representation using a recursive predicate



$$\begin{aligned}L &= (5::7::\text{nil})::(8::3::3::\text{nil}) \\&\quad ::(\text{nil})::(4::\text{nil}) :: \text{nil}\end{aligned}$$

$p \rightsquigarrow \text{MlistofMlist } L \equiv \text{match } L \text{ with}$

- $\mid \text{nil} \Rightarrow 'p = \text{null}'$
- $\mid X :: L' \Rightarrow \exists x p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\}$
 - * $p' \rightsquigarrow \text{MlistofMlist } L'$
 - * $x \rightsquigarrow \text{MList } X$

Generalization to a higher-order predicate

$p \rightsquigarrow \text{MlistofMlist } L \equiv \text{match } L \text{ with}$

- | nil $\Rightarrow [p = \text{null}]$
- | $X :: L' \Rightarrow \exists x p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=\text{p}'\}$
 - * $p' \rightsquigarrow \text{MlistofMlist } L'$
 - * $x \rightsquigarrow \text{MList } X$

Generalization:

$p \rightsquigarrow \text{Mlistof } R L \equiv \text{match } L \text{ with}$

- | nil $\Rightarrow [p = \text{null}]$
- | $X :: L' \Rightarrow \exists x p'. p \rightsquigarrow \{\text{hd}=x; \text{tl}=\text{p}'\}$
 - * $p' \rightsquigarrow \text{Mlistof } R L'$
 - * $x \rightsquigarrow R X$

In particular:

$$p \rightsquigarrow \text{MlistofMlist } L = p \rightsquigarrow \text{Mlistof MList } L$$

The identity representation predicate

$p \rightsquigarrow \text{Mlistof } RL \equiv \text{match } L \text{ with}$

- | nil $\Rightarrow [p = \text{null}]$
- | $X :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}$
 - * $p' \rightsquigarrow \text{Mlistof } RL'$
 - * $x \rightsquigarrow RX$

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

- | nil $\Rightarrow [p = \text{null}]$
- | $x :: L' \Rightarrow \exists p'. p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}$
 - * $p' \rightsquigarrow \text{MList } L'$

Exercise: define the identity representation predicate Id such that

$$p \rightsquigarrow \text{Mlistof Id } L = p \rightsquigarrow \text{MList } L$$

The identity representation predicate

$p \rightsquigarrow \text{Mlistof } RL \equiv \text{match } L \text{ with}$

- | nil \Rightarrow ' $p = \text{null}$ '
- | $X :: L' \Rightarrow \exists p'. \ p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}$
 - * $p' \rightsquigarrow \text{Mlistof } RL'$
 - * $x \rightsquigarrow RX$

$p \rightsquigarrow \text{MList } L \equiv \text{match } L \text{ with}$

- | nil \Rightarrow ' $p = \text{null}$ '
- | $x :: L' \Rightarrow \exists p'. \ p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\}$
 - * $p' \rightsquigarrow \text{MList } L'$

Exercise: define the identity representation predicate Id such that

$$p \rightsquigarrow \text{Mlistof } \text{Id } L = p \rightsquigarrow \text{MList } L$$

Definition:

$$x \rightsquigarrow \text{Id } X \equiv 'x = X'$$

Chapter 18

Separating implication

Separating implication or “magic wand”

Recalling separating conjunction:

$$(P_1 * P_2)h \equiv \exists h_1, h_2. h = h_1 \uplus h_2 \wedge P_1 h_1 \wedge P_2 h_2$$

Introducing *separating implication*:

$$(P \multimap Q)h \equiv \forall h_1. h \perp h_1 \wedge P h_1 \Rightarrow Q(h \uplus h_1)$$

Intuition:

$$(P \multimap Q) * P \Rightarrow Q$$

Rules:

$$\frac{R * P \vdash Q}{R \vdash (P \multimap Q)}$$

$$\frac{R_1 \vdash (P \multimap Q) \quad R_2 \vdash P}{R_1 * R_2 \vdash Q}$$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- ① $\top \rightarrow * (1 \mapsto 2)$
- ② $\text{'False'} \rightarrow * (1 \mapsto 2)$
- ③ $x \geq 1 \rightarrow x \geq 0$
- ④ $(1 \mapsto 4) \rightarrow * (1 \mapsto 4) * (2 \mapsto 3)$
- ⑤ $(1 \mapsto 2) \rightarrow * (1 \mapsto 2)$
- ⑥ $(1 \mapsto 2) \rightarrow \text{'False'}$
- ⑦ $(1 \mapsto 2) \rightarrow \top$
- ⑧ $(1 \mapsto 2) \rightarrow * (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- ① $\top \rightarrow * (1 \mapsto 2)$ $\{(1, 2)\}$
- ② $\text{'False'} \rightarrow * (1 \mapsto 2)$
- ③ $x \geq 1 \rightarrow x \geq 0$
- ④ $(1 \mapsto 4) \rightarrow * (1 \mapsto 4) * (2 \mapsto 3)$
- ⑤ $(1 \mapsto 2) \rightarrow * (1 \mapsto 2)$
- ⑥ $(1 \mapsto 2) \rightarrow \text{'False'}$
- ⑦ $(1 \mapsto 2) \rightarrow \top$
- ⑧ $(1 \mapsto 2) \rightarrow * (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- ① $\top \rightarrow (1 \mapsto 2)$ $\{(1, 2)\}$
- ② $\text{'False'} \rightarrow (1 \mapsto 2)$ all heaps
- ③ $x \geq 1 \rightarrow x \geq 0$
- ④ $(1 \mapsto 4) \rightarrow (1 \mapsto 4) * (2 \mapsto 3)$
- ⑤ $(1 \mapsto 2) \rightarrow (1 \mapsto 2)$
- ⑥ $(1 \mapsto 2) \rightarrow \text{'False'}$
- ⑦ $(1 \mapsto 2) \rightarrow \top$
- ⑧ $(1 \mapsto 2) \rightarrow (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- ① $\top \rightarrow * (1 \mapsto 2)$ $\{(1, 2)\}$
- ② $\text{'False'} \rightarrow * (1 \mapsto 2)$ all heaps
- ③ $x \geq 1 \rightarrow x \geq 0$ only \emptyset
- ④ $(1 \mapsto 4) \rightarrow * (1 \mapsto 4) * (2 \mapsto 3)$
- ⑤ $(1 \mapsto 2) \rightarrow * (1 \mapsto 2)$
- ⑥ $(1 \mapsto 2) \rightarrow \text{'False'}$
- ⑦ $(1 \mapsto 2) \rightarrow \top$
- ⑧ $(1 \mapsto 2) \rightarrow * (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- ① $\top \rightarrow * (1 \mapsto 2)$ $\{(1, 2)\}$
- ② $\text{'False'} \rightarrow * (1 \mapsto 2)$ all heaps
- ③ $x \geq 1 \rightarrow x \geq 0$ only \emptyset
- ④ $(1 \mapsto 4) \rightarrow * (1 \mapsto 4) * (2 \mapsto 3)$ $\{(2, 3)\}$, any h with $1 \in \text{dom}(h)$
- ⑤ $(1 \mapsto 2) \rightarrow * (1 \mapsto 2)$
- ⑥ $(1 \mapsto 2) \rightarrow \text{'False'}$
- ⑦ $(1 \mapsto 2) \rightarrow \top$
- ⑧ $(1 \mapsto 2) \rightarrow * (1 \mapsto 3)$

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|--|
| ① | $\top \rightarrow (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| ② | $\text{'False'} \rightarrow (1 \mapsto 2)$ | all heaps |
| ③ | $x \geq 1 \rightarrow x \geq 0$ | only \emptyset |
| ④ | $(1 \mapsto 4) \rightarrow (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$, any h with $1 \in \text{dom}(h)$ |
| ⑤ | $(1 \mapsto 2) \rightarrow (1 \mapsto 2)$ | \emptyset and any h with $1 \in \text{dom}(h)$ |
| ⑥ | $(1 \mapsto 2) \rightarrow \text{'False'}$ | |
| ⑦ | $(1 \mapsto 2) \rightarrow \top$ | |
| ⑧ | $(1 \mapsto 2) \rightarrow (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|--|
| ① | $\top \rightarrow (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| ② | $\text{'False'} \rightarrow (1 \mapsto 2)$ | all heaps |
| ③ | $x \geq 1 \rightarrow x \geq 0$ | only \emptyset |
| ④ | $(1 \mapsto 4) \rightarrow (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$, any h with $1 \in \text{dom}(h)$ |
| ⑤ | $(1 \mapsto 2) \rightarrow (1 \mapsto 2)$ | \emptyset and any h with $1 \in \text{dom}(h)$ |
| ⑥ | $(1 \mapsto 2) \rightarrow \text{'False'}$ | any h with $1 \in \text{dom}(h)$ |
| ⑦ | $(1 \mapsto 2) \rightarrow \top$ | |
| ⑧ | $(1 \mapsto 2) \rightarrow (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|--|
| ① | $\top \rightarrow (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| ② | $\text{'False'} \rightarrow (1 \mapsto 2)$ | all heaps |
| ③ | $x \geq 1 \rightarrow x \geq 0$ | only \emptyset |
| ④ | $(1 \mapsto 4) \rightarrow (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$, any h with $1 \in \text{dom}(h)$ |
| ⑤ | $(1 \mapsto 2) \rightarrow (1 \mapsto 2)$ | \emptyset and any h with $1 \in \text{dom}(h)$ |
| ⑥ | $(1 \mapsto 2) \rightarrow \text{'False'}$ | any h with $1 \in \text{dom}(h)$ |
| ⑦ | $(1 \mapsto 2) \rightarrow \top$ | any h with $1 \in \text{dom}(h)$ |
| ⑧ | $(1 \mapsto 2) \rightarrow (1 \mapsto 3)$ | |

Separating implication examples

Exercise: Give heaps satisfying the following predicates:

- | | | |
|---|---|--|
| ① | $\top \rightarrow (1 \mapsto 2)$ | $\{(1, 2)\}$ |
| ② | $\text{'False'} \rightarrow (1 \mapsto 2)$ | all heaps |
| ③ | $x \geq 1 \rightarrow x \geq 0$ | only \emptyset |
| ④ | $(1 \mapsto 4) \rightarrow (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2, 3)\}$, any h with $1 \in \text{dom}(h)$ |
| ⑤ | $(1 \mapsto 2) \rightarrow (1 \mapsto 2)$ | \emptyset and any h with $1 \in \text{dom}(h)$ |
| ⑥ | $(1 \mapsto 2) \rightarrow \text{'False'}$ | any h with $1 \in \text{dom}(h)$ |
| ⑦ | $(1 \mapsto 2) \rightarrow \top$ | any h with $1 \in \text{dom}(h)$ |
| ⑧ | $(1 \mapsto 2) \rightarrow (1 \mapsto 3)$ | any h with $1 \in \text{dom}(h)$ |

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$
- ② $(Q \multimap P * Q) \triangleright P$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{''} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{''} \multimap P$
- ⑦ $\text{''} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ yes: unfold and behold the definition of \multimap
- ② $(Q \multimap P * Q) \triangleright P$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{''} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{''} \multimap P$
- ⑦ $\text{''} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q -* P * Q)$ yes: unfold and behold the definition of $*$
- ② $(Q -* P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) -* (1 \mapsto 3) \triangleright \text{'False'}$
- ④ $(1 \mapsto 2) -* (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{''} -* P \triangleright P$
- ⑥ $P \triangleright \text{''} -* P$
- ⑦ $\text{''} \triangleright (P -* Q -* P * Q)$
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P -* Q)$
- ⑨ $(P -* Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q -* P * Q)$ yes: unfold and behold the definition of $*$
- ② $(Q -* P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) -* (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) -* (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$
- ⑤ $\text{''} -* P \triangleright P$
- ⑥ $P \triangleright \text{''} -* P$
- ⑦ $\text{''} \triangleright (P -* Q -* P * Q)$
- ⑧ $'P \triangleright Q' \triangleright (P -* Q)$
- ⑨ $(P -* Q) \triangleright 'P \triangleright Q'$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ yes: unfold and behold the definition of \multimap
- ② $(Q \multimap P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{''} \multimap P \triangleright P$
- ⑥ $P \triangleright \text{''} \multimap P$
- ⑦ $\text{''} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ yes: unfold and behold the definition of \multimap
- ② $(Q \multimap P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{''} \multimap P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{''} \multimap P$
- ⑦ $\text{''} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ yes: unfold and behold the definition of \multimap
- ② $(Q \multimap P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{''} \multimap P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{''} \multimap P$...yes: P and $\text{''} \multimap P$ are equivalent
- ⑦ $\text{''} \triangleright (P \multimap Q \multimap P * Q)$
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P \multimap Q)$
- ⑨ $(P \multimap Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q -* P * Q)$ yes: unfold and behold the definition of $*$
- ② $(Q -* P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) -* (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) -* (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{''} -* P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{''} -* P$...yes: P and $\text{''} -* P$ are equivalent
- ⑦ $\text{''} \triangleright (P -* Q -* P * Q)$ yes: unfold \triangleright and obtain approx. (1)
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P -* Q)$
- ⑨ $(P -* Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q -* P * Q)$ yes: unfold and behold the definition of $*$
- ② $(Q -* P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) -* (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) -* (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{''} -* P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{''} -* P$...yes: P and $\text{''} -* P$ are equivalent
- ⑦ $\text{''} \triangleright (P -* Q -* P * Q)$ yes: unfold \triangleright and obtain approx. (1)
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P -* Q)$ yes
- ⑨ $(P -* Q) \triangleright \lceil P \triangleright Q \rceil$

Separating implication examples

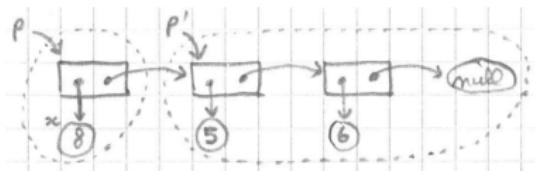
Exercise: Among the following heap entailments, which hold?

- ① $P \triangleright (Q \multimap P * Q)$ yes: unfold and behold the definition of \multimap
- ② $(Q \multimap P * Q) \triangleright P$ no e.g. with $P = Q = \text{'False'}$
- ③ $(1 \mapsto 2) \multimap (1 \mapsto 3) \triangleright \text{'False'}$ no e.g. $\{(1, 4)\}$ satisfies left
- ④ $(1 \mapsto 2) \multimap (1 \mapsto 2 * 2 \mapsto 8) \triangleright 2 \mapsto 8$ no (same)
- ⑤ $\text{''} \multimap P \triangleright P$ yes, and...
- ⑥ $P \triangleright \text{''} \multimap P$...yes: P and $\text{''} \multimap P$ are equivalent
- ⑦ $\text{''} \triangleright (P \multimap Q \multimap P * Q)$ yes: unfold \triangleright and obtain approx. (1)
- ⑧ $\lceil P \triangleright Q \rceil \triangleright (P \multimap Q)$ yes
- ⑨ $(P \multimap Q) \triangleright \lceil P \triangleright Q \rceil$ no, e.g. $P = \text{''}$ and $Q = 1 \mapsto 2$

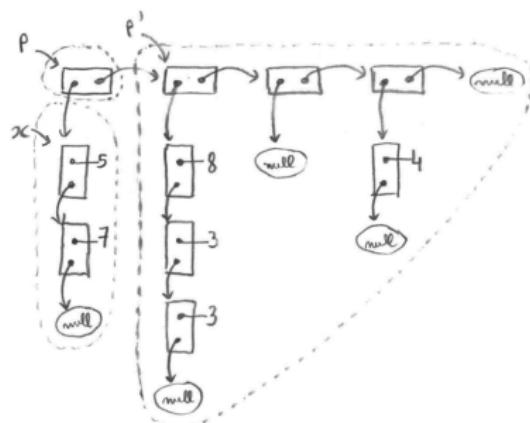
Chapter 19

Higher-order representation predicates and the access problem

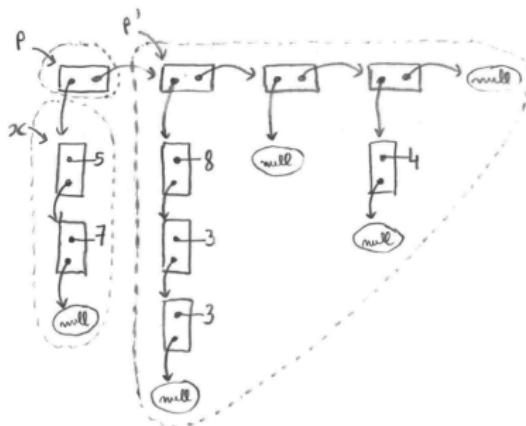
Specification of construction, for basic values


$$\{p' \rightsquigarrow \text{MList } L\} \ (\text{cons } x \ p') \ \{\lambda p. \ p \rightsquigarrow \text{MList } (x :: L)\}$$

Specification of construction


$$\{x \rightsquigarrow RX * p' \rightsquigarrow \text{Mlistof } RL\} (\text{cons } x \ p') \ \{\lambda p. \ p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

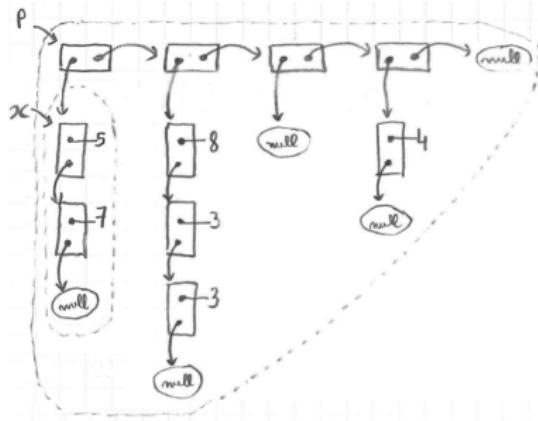
Specification of deconstruction



$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\} \ (\text{uncons } p)$

$\{\lambda(x, p'). x \rightsquigarrow RX * p' \rightsquigarrow \text{Mlistof } RL\}$

Specification of accesses: the problem

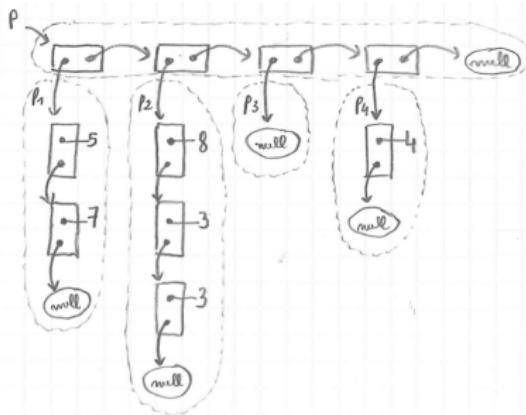


Incorrect specification for head:

$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\} \text{ (head } p\text{)}$$

$$\{\lambda x. x \rightsquigarrow RX * p \rightsquigarrow \text{Mlistof } R(X :: L)\}$$

Specification of accesses: a brute force solution

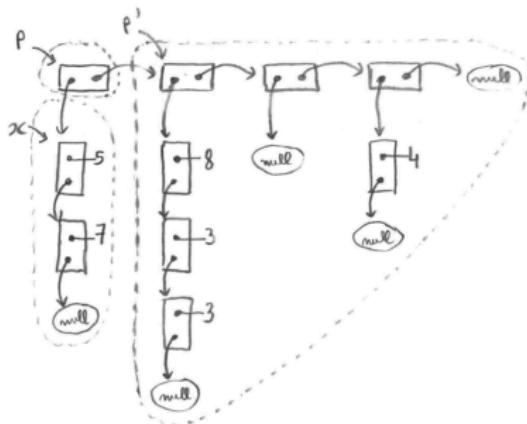


$$p \rightsquigarrow \text{Mlistof } RL = \exists K. \quad p \rightsquigarrow \text{MList } K$$

$$* \quad \circledast_{i \in \{0, \dots, |L| - 1\}} (K[i]) \rightsquigarrow R(L[i])$$

$$* \quad \lceil |K| = |L| \rceil$$

Specification of accesses: focus before read

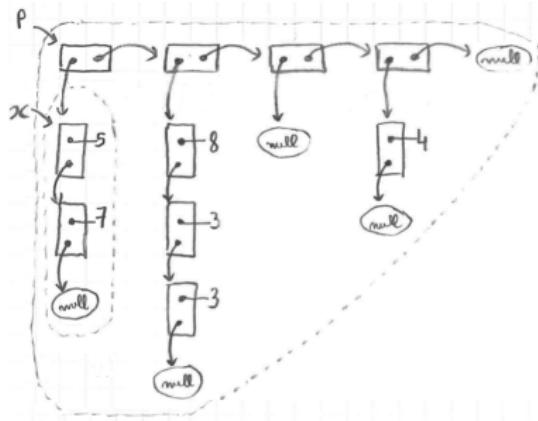


$$\begin{aligned} p \rightsquigarrow \text{Mlistof } R(X :: L) &= \exists x p'. \quad p \rightsquigarrow \{\text{hd} = x; \text{tl} = p'\} \\ * \quad x \rightsquigarrow RX & \\ * \quad p' \rightsquigarrow \text{Mlistof } RL & \end{aligned}$$

Then read using:

$$\{p \mapsto \{\text{hd} = x; \text{tl} = p'\}\} (p.\text{hd}) \{\lambda y. [y = x] * p \mapsto \{\text{hd} = x; \text{tl} = p'\}\}$$

Specification of accesses with separating implication

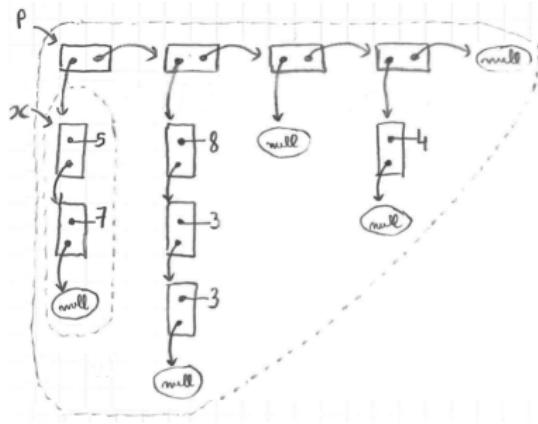


Correct specification with $\rightarrow*$:

$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\} (\text{head } p)$$

$$\{\lambda x. x \rightsquigarrow RX \rightarrow* (x \rightsquigarrow RX \rightarrow* p \rightsquigarrow \text{Mlistof } R(X :: L))\}$$

Specification of accesses with separating implication



Correct specification with $\rightarrow*$:

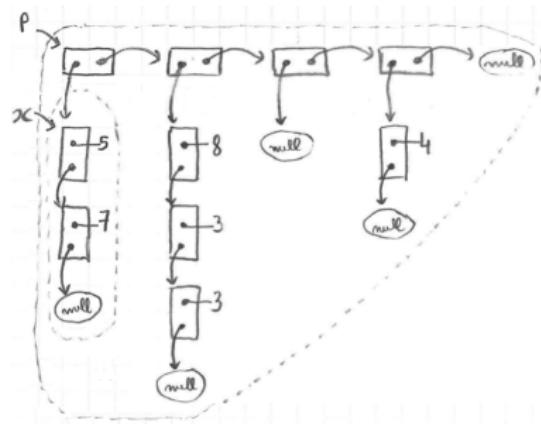
$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\} \ (\text{head } p)$$

$$\{\lambda x. x \rightsquigarrow RX \rightarrow* (x \rightsquigarrow RX \rightarrow p \rightsquigarrow \text{Mlistof } R(X :: L))\}$$

Exercise: What problem is there, e.g. if $x \rightsquigarrow RX$ is $x \mapsto X$ (i.e. $R = \text{Ref}$)?

Exercise: How to generalize the specification to solve this problem?

Specification of accesses with separating implication



Correct specification with $\rightarrow*$, generalized:

$$\{p \rightsquigarrow \text{Mlistof } R(X :: L)\} (\text{head } p)$$

$$\{\lambda x. x \rightsquigarrow RX \rightarrow* (\forall X', x \rightsquigarrow RX' \rightarrow* p \rightsquigarrow \text{Mlistof } R(X' :: L))\}$$

The copy problem

Incorrect specification for copy:

$$\{p \rightsquigarrow \text{Mlistof } R L\}$$

(copy p)

$$\{\lambda p'. p \rightsquigarrow \text{Mlistof } R L * p' \rightsquigarrow \text{Mlistof } R L\}$$

The copy problem

Incorrect specification for copy:

$$\{p \rightsquigarrow \text{Mlistof } R L\}$$

$$(\text{copy } p)$$

$$\{\lambda p'. p \rightsquigarrow \text{Mlistof } R L * p' \rightsquigarrow \text{Mlistof } R L\}$$

Exercise: specify a function $\text{copy } f \, p$ that duplicates a mutable list specified using Mlistof, where f is a function to duplicate items.

The copy problem

Incorrect specification for copy:

$$\{p \rightsquigarrow \text{Mlistof } R L\}$$

$$(\text{copy } p)$$

$$\{\lambda p'. p \rightsquigarrow \text{Mlistof } R L * p' \rightsquigarrow \text{Mlistof } R L\}$$

Exercise: specify a function $\text{copy } f p$ that duplicates a mutable list specified using Mlistof, where f is a function to duplicate items.

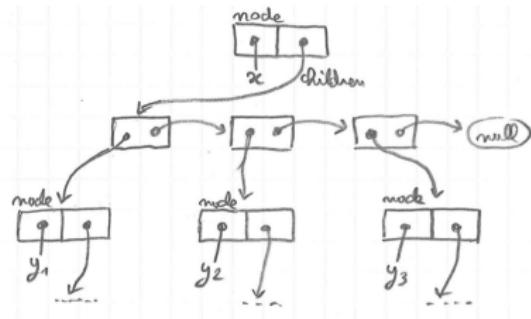
$$(\forall x X. \{x \rightsquigarrow R X\} (f x) \{\lambda x'. x \rightsquigarrow R X * x' \rightsquigarrow R X\})$$

$$\Rightarrow \{p \rightsquigarrow \text{Mlistof } R L\}$$

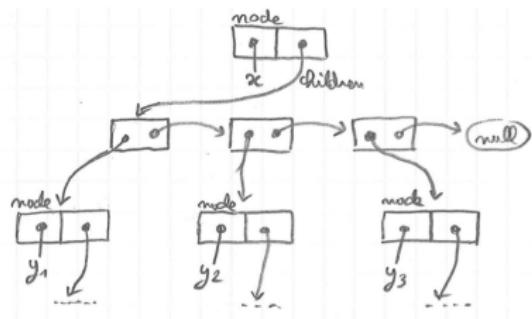
$$(\text{copy } f p)$$

$$\{\lambda p'. p \rightsquigarrow \text{Mlistof } R L * p' \rightsquigarrow \text{Mlistof } R L\}$$

Example of combining higher-order predicates



Example of combining higher-order predicates



$p \rightsquigarrow \text{Narytreeof } RT \equiv$

match T with

| Leaf $\Rightarrow p = \text{null}$

| Node $X L \Rightarrow p \rightsquigarrow \text{Nodeof } RX (\text{Mlistof}(\text{Narytreeof } R)) L$

Chapter 22

Iteration with higher-order representation predicates

Iteration on lists

Recall:

$$\begin{aligned} \forall f l I. \quad & (\forall x k. \{I k\} (f x) \{\lambda _. \ I(k \& x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } f l) \{\lambda _. \ I l\} \end{aligned}$$

$$\begin{aligned} \forall f p l I. \quad & (\forall x k. \{I k\} (f x) \{\lambda _. \ I(k \& x)\}) \\ \Rightarrow \quad & \{p \rightsquigarrow \text{MList } l * I \text{ nil}\} (\text{miter } f p) \{\lambda _. \ p \rightsquigarrow \text{MList } l * I l\} \end{aligned}$$

Iteration on lists

Recall:

$$\begin{aligned} \forall f l I. \quad & (\forall x k. \{I k\} (f x) \{\lambda _. \ I(k \& x)\}) \\ \Rightarrow \quad & \{I \text{ nil}\} (\text{iter } f l) \{\lambda _. \ I l\} \end{aligned}$$

$$\begin{aligned} \forall f p l I. \quad & (\forall x k. \{I k\} (f x) \{\lambda _. \ I(k \& x)\}) \\ \Rightarrow \quad & \{p \rightsquigarrow \text{MList } l * I \text{ nil}\} (\text{miter } f p) \{\lambda _. \ p \rightsquigarrow \text{MList } l * I l\} \end{aligned}$$

Challenge:

$$\begin{aligned} & (\forall x \dots \{\dots\} (f x) \{\lambda _. \ \dots\}) \\ \Rightarrow \quad & \{p \rightsquigarrow \text{Mlistof } R L * \dots\} (\text{miter } f p) \{\lambda _. \ p \rightsquigarrow \dots * \dots\} \end{aligned}$$

Question: Can we use an invariant $I \equiv \lambda K.(\dots)$?

(i.e. with a spec of the form $\{p \rightsquigarrow \dots * I \text{ nil}\}(\dots) \{p \rightsquigarrow \dots * I L\} \ ?$)

Iterating over a mutable list of mutable items

Exercise: specify the function `miter`, using an invariant of the form JKK' , describing the state before and the state after the iteration.

Iterating over a mutable list of mutable items

Exercise: specify the function `miter`, using an invariant of the form JKK' , describing the state before and the state after the iteration.

$$\begin{aligned} & \forall fpRLJ. \quad (\forall xXKK'. \{x \rightsquigarrow RX * JKK'\} \\ & \quad (fx) \\ & \quad \{\lambda_. \exists X'. x \rightsquigarrow RX' * J(K\&X)(K'\&X')\}) \\ \Rightarrow & \quad \{p \rightsquigarrow \text{Mlistof } RL * J \text{ nil nil}\} \\ & \quad (\text{miter } fp) \\ & \quad \{\lambda_. \exists L'. p \rightsquigarrow \text{Mlistof } RL' * JL L'\} \end{aligned}$$

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) p

let example_p =
  { hd = ref 5; tl = { hd = ref 3; tl = null } }
```

$$x \rightsquigarrow \text{Ref } X \equiv x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the function (fun x -> incr x) and incr_all. What is $J \ K \ K'$?

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) p

let example_p =
{ hd = ref 5; tl = { hd = ref 3; tl = null } }
```

$$x \rightsquigarrow \text{Ref } X \equiv x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the function (fun x -> incr x) and incr_all. What is $J \ K \ K'$?

$$\{x \rightsquigarrow \text{Ref } X\} (\text{incr } x) \{\lambda_. \ x \rightsquigarrow \text{Ref } (X + 1)\}$$

Incrementing a mutable list of distinct references (1/2)

```
let incr_all p =
  miter (fun x -> incr x) p
```

```
let example_p =
{ hd = ref 5; tl = { hd = ref 3; tl = null } }
```

$$x \rightsquigarrow \text{Ref } X \equiv x \mapsto X$$

Exercise: using the representation predicates Ref and Mlistof, specify the function (fun x -> incr x) and incr_all. What is J K K' ?

$$\{x \rightsquigarrow \text{Ref } X\} (\text{incr } x) \{\lambda_. x \rightsquigarrow \text{Ref } (X + 1)\}$$

$$\{p \rightsquigarrow \text{Mlistof Ref } L\} (\text{incr_all } p) \{\lambda_. p \rightsquigarrow \text{Mlistof Ref } (\text{map } (+1) L)\}$$

Incrementing a mutable list of distinct references (2/2)

$$\begin{aligned} & \forall fpRLJ. \quad (\forall xXKK'. \quad \{x \rightsquigarrow RX * JKK'\} \\ & \quad (fx) \\ & \quad \{\lambda_. \exists X'. \quad x \rightsquigarrow RX' * J(K\&X)(K'\&X')\}) \\ \Rightarrow & \quad \{p \rightsquigarrow \text{Mlistof } RL * J \text{ nil nil}\} \\ & \quad (\text{miter } fp) \\ & \quad \{\lambda_. \exists L'. \quad p \rightsquigarrow \text{Mlistof } RL' * JL L'\} \end{aligned}$$

Consider:

$$JKK' \equiv 'K' = \text{map } (+1) K$$

Derives:

$$\begin{aligned} & (\forall xX. \quad \{x \rightsquigarrow \text{Ref } X\} (\text{fun } x \rightarrow \text{incr } x)x \{\lambda_. \quad x \rightsquigarrow \text{Ref}(X + 1)\}) \Rightarrow \\ & \{p \rightsquigarrow \text{Mlistof Ref } L\} (\text{incr_all } p) \{\lambda_. \quad p \rightsquigarrow \text{Mlistof Ref } (\text{map } (+1) L)\} \end{aligned}$$

Chapter 23

Resource analysis in Separation Logic

Controlling deallocation

(1) Remove the “GC” part from the definition the triple $\{H\} t \{Q\}$:

$$\forall H'm. (H * H') m \Rightarrow \exists v m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge (Q v * H' * \cancel{GC}) m'$$

(2) Replace the GC rule, add a “free” function for explicit deallocation:

$$\frac{\cancel{\{H\} t \{Q \star \text{GC}\}}}{\cancel{\{H\} t \{Q\}}} \rightsquigarrow \overline{\{r \mapsto \vec{v}\} \text{ free } r \{\lambda _. \ \Gamma\}}$$

Controlling deallocation

- (1) Remove the “GC” part from the definition the triple $\{H\} t \{Q\}$:

$$\forall H' m. (H * H') m \Rightarrow \exists v m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge (Q v * H' * \cancel{GC}) m'$$

- (2) Replace the GC rule, add a “free” function for explicit deallocation:

$$\frac{\cancel{\{H\} t \{Q \star \text{GC}\}}}{\cancel{\{H\} t \{Q\}}} \rightsquigarrow \frac{}{\{r \mapsto \vec{v}\} \text{ free } r \{\lambda _. \text{' } \}'\}}$$

- (3) Theorem: for a full program execution starting in the empty heap, all the data still allocated at the end is described in the post-condition.

- (4) Corollary: terminating on the empty heap ensures no memory leaks.

$$\{\text{' } \}' t \{\lambda n. \text{' } P n \text{' }\}$$

Controlling deallocation

- (1) Remove the “GC” part from the definition the triple $\{H\} t \{Q\}$:

$$\forall H' m. (H * H') m \Rightarrow \exists v m'. \langle t, m \rangle \Downarrow \langle v, m' \rangle \wedge (Q v * H' * \cancel{GC}) m'$$

- (2) Replace the GC rule, add a “free” function for explicit deallocation:

$$\frac{\cancel{\{H\} t \{Q \star \text{GC}\}}}{\cancel{\{H\} t \{Q\}}} \rightsquigarrow \frac{}{\{r \mapsto \vec{v}\} \text{ free } r \{\lambda _. \text{' } \}'\}}$$

- (3) Theorem: for a full program execution starting in the empty heap, all the data still allocated at the end is described in the post-condition.

- (4) Corollary: terminating on the empty heap ensures no memory leaks.

$$\{\text{' } \}' t \{\lambda n. \text{' } P n \text{' }\}$$

(separation logics with GC are sometimes called “affine” or “intuitionistic”)

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \text{File } L$$

where $(f : \text{loc})$ denotes the file handler,
and $(L : \text{list char})$ denotes the remaining bytes to read.

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \text{File } L$$

where $(f : \text{loc})$ denotes the file handler,
and $(L : \text{list char})$ denotes the remaining bytes to read.

$$\{\text{'}'\} (\text{fopen } s) \{\lambda f. \exists L. f \rightsquigarrow \text{File } L\}$$

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \text{File } L$$

where $(f : \text{loc})$ denotes the file handler,
and $(L : \text{list char})$ denotes the remaining bytes to read.

$$\begin{aligned} & \{\text{'}'\} \ (\text{fopen } s) \ \{\lambda f. \exists L. f \rightsquigarrow \text{File } L\} \\ & \{f \rightsquigarrow \text{File } (c :: L)\} \ (\text{fread } f) \ \{\lambda x. 'x = c' * f \rightsquigarrow \text{File } L\} \end{aligned}$$

File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \text{File } L$$

where $(f : \text{loc})$ denotes the file handler,
and $(L : \text{list char})$ denotes the remaining bytes to read.

$$\begin{aligned} & \{\text{'}\}\ (\text{fopen } s) \ \{\lambda f. \exists L. f \rightsquigarrow \text{File } L\} \\ & \{f \rightsquigarrow \text{File } (c :: L)\} \ (\text{fread } f) \ \{\lambda x. 'x = c' * f \rightsquigarrow \text{File } L\} \\ & \{f \rightsquigarrow \text{File } L\} (\text{fclose } f) \ \{\lambda_. \text{'}\}\end{aligned}$$

Complexity analysis

Time credits:

$$\$x : \text{Hprop} \quad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x + y) = \$x * \$y \quad \text{and} \quad \$0 = \top$$

Complexity analysis

Time credits:

$$\$x : \text{Hprop} \quad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x + y) = \$x * \$y \quad \text{and} \quad \$0 = \top$$

Principle:

The execution of every instruction costs \$1.

Simplification:

Entering the body of a function or a loop costs \$1.

Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \text{Array } M * \$c\} (\text{Array.length t}) \{\lambda n. \lceil n = |M| \rceil * t \rightsquigarrow \text{Array } M\}$$

Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \text{Array } M * \$c\} (\text{Array.length } t) \{\lambda n. 'n = |M|' * t \rightsquigarrow \text{Array } M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\} (\text{Array.make } n \text{ v}) \{\lambda t. \exists L. t \rightsquigarrow \text{Array } L * '...' \}$$

Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \text{Array } M * \$c\} (\text{Array.length } t) \{\lambda n. 'n = |M|' * t \rightsquigarrow \text{Array } M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\} (\text{Array.make } n \ v) \{\lambda t. \exists L. t \rightsquigarrow \text{Array } L * '...' \}$$

Quasilinear time:

$$\begin{aligned} & \{t \rightsquigarrow \text{Array } L * \$(c_1 |L| \log |L| + c_2)\} \\ & (\text{Array.sort } t) \\ & \{\lambda t. \exists L'. t \rightsquigarrow \text{Array } L' * '...' \} \end{aligned}$$

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$$\begin{array}{ll} \{\$c\} & (\text{Stack.create}()) \{ \lambda _. s \rightsquigarrow \text{Stack nil} \} \\ \{s \rightsquigarrow \text{Stack } L * \$c\} & (\text{Stack.push } s \ x) \{ \lambda _. s \rightsquigarrow \text{Stack } (x :: L) \} \\ \{s \rightsquigarrow \text{Stack } (x :: L) * \$c\} & (\text{Stack.pop } s) \quad \{ \lambda y. \lceil y = x \rceil * s \rightsquigarrow \text{Stack } L \} \end{array}$$

Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$$\begin{array}{ll} \{\$c\} & (\text{Stack.create}()) \{ \lambda _. s \rightsquigarrow \text{Stack nil} \} \\ \{s \rightsquigarrow \text{Stack } L * \$c\} & (\text{Stack.push } s \ x) \{ \lambda _. s \rightsquigarrow \text{Stack } (x :: L) \} \\ \{s \rightsquigarrow \text{Stack } (x :: L) * \$c\} & (\text{Stack.pop } s) \quad \{ \lambda y. \lceil y = x \rceil * s \rightsquigarrow \text{Stack } L \} \end{array}$$

Representation predicate with a potential function:

$$\begin{aligned} s \rightsquigarrow \text{Stack } L &\equiv \exists ntMk. \quad s \mapsto \{\text{size}=n; \text{data}=t\} \\ &\quad * \quad t \rightsquigarrow \text{Array } M \\ &\quad * \quad \lceil n = |L| \leq |M| = 2^k \rceil \\ &\quad * \quad \lceil \forall i \in [0, n). M[i] = L[i] \rceil \\ &\quad * \quad \$ (c' \cdot \text{abs}(n - |M|/2)) \end{aligned}$$

Space complexity analysis

Suppose $\diamond 1$ represents one *space credit* (Hoffmann, 1999)

Then allocation consumes credits; deallocation produces credits :

$$\{\diamond \text{size}(b)\} \quad \text{alloc}(b) \quad \{\lambda x. x \mapsto b\}$$

$$\{x \mapsto b\} \quad \text{free}(x) \quad \{\lambda _. \diamond \text{size}(b)\}$$

The same mechanisms apply (e.g. $\diamond(a + b) = \diamond a * \diamond b$, amortized analysis, etc.).

Space complexity analysis

Suppose $\diamond 1$ represents one *space credit* (Hoffmann, 1999)

Then allocation consumes credits; deallocation produces credits :

$$\{\diamond \text{size}(b)\} \quad \text{alloc}(b) \quad \{\lambda x. x \mapsto b\}$$

$$\{x \mapsto b\} \quad \text{free}(x) \quad \{\lambda _. \diamond \text{size}(b)\}$$

The same mechanisms apply (e.g. $\diamond(a + b) = \diamond a * \diamond b$, amortized analysis, etc.).

What if we add **garbage collection** ?

- M., Pottier, 2022

A separation logic for heap space under garbage collection.

- Moine, Chaguéraud, Pottier, 2023

A high-level separation logic for heap space under garbage collection.

Fractional permissions

$$(r \xrightarrow{\alpha} v) \quad \text{with } 0 < \alpha \leq 1$$

Splitting and merging:

$$(r \mapsto v) = (r \xrightarrow{1} v) = (r \xrightarrow{1/2} v) * (r \xrightarrow{1/2} v)$$

More generally, if $0 < \alpha, \beta \leq 1$:

$$(r \xrightarrow{\alpha+\beta} v) = (r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} v)$$

Fractional permissions

$$(r \xrightarrow{\alpha} v) \quad \text{with } 0 < \alpha \leq 1$$

Splitting and merging:

$$(r \mapsto v) = (r \xrightarrow{1} v) = (r \xrightarrow{1/2} v) * (r \xrightarrow{1/2} v)$$

More generally, if $0 < \alpha, \beta \leq 1$:

$$(r \xrightarrow{\alpha+\beta} v) = (r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} v)$$

Values must agree: if $0 < \alpha, \beta \leq 1$:

$$\left((r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} w) \right) \triangleright \left((r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} w) * {}^r v = w \right)$$

Fractional permissions

$$(r \xrightarrow{\alpha} v) \quad \text{with } 0 < \alpha \leq 1$$

Splitting and merging:

$$(r \mapsto v) = (r \xrightarrow{1} v) = (r \xrightarrow{1/2} v) * (r \xrightarrow{1/2} v)$$

More generally, if $0 < \alpha, \beta \leq 1$:

$$(r \xrightarrow{\alpha+\beta} v) = (r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} v)$$

Values must agree: if $0 < \alpha, \beta \leq 1$:

$$\left((r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} w) \right) \triangleright \left((r \xrightarrow{\alpha} v) * (r \xrightarrow{\beta} w) * {}^r v = w \right)$$

Operations:

$$\{{}^r\} \ (\mathbf{ref} \ v) \ \{\lambda r. \ r \xrightarrow{1} v\}$$

$$\{r \xrightarrow{1} v'\} \ (r := v) \ \{\lambda_. \ r \xrightarrow{1} v\}$$

$$\forall \alpha. \quad \{r \xrightarrow{\alpha} v\} \quad (!r) \quad \{\lambda x. \ {}^r x = v * (r \xrightarrow{\alpha} v)\}$$

Fractional permissions in practice

$$\begin{aligned} \forall \alpha \beta. \quad & \{a_1 \xrightarrow{\alpha} \text{Array } L_1 * a_2 \xrightarrow{\beta} \text{Array } L_2\} \\ & (\text{concat } a_1 a_2) \\ \{ \lambda a_3. \quad & a_1 \xrightarrow{\alpha} \text{Array } L_1 * a_2 \xrightarrow{\beta} \text{Array } L_2 * a_3 \xrightarrow{1} \text{Array } (L_1 ++ L_2)\} \end{aligned}$$

Fractional permissions in practice

$$\begin{aligned} \forall \alpha \beta. \quad & \{a_1 \xrightarrow{\alpha} \text{Array } L_1 * a_2 \xrightarrow{\beta} \text{Array } L_2\} \\ & (\text{concat } a_1 a_2) \\ & \{\lambda a_3. \quad a_1 \xrightarrow{\alpha} \text{Array } L_1 * a_2 \xrightarrow{\beta} \text{Array } L_2 * a_3 \xrightarrow{1} \text{Array } (L_1 ++ L_2)\} \end{aligned}$$

Limitations:

- need to quantify fractions explicitly,
- need to syntactic sugar to avoid copy-pasting,
- need to re-establish post-conditions,
- a fraction $\frac{1}{2}H$ cannot be defined for arbitrary H .

Those can be alleviated with a duplicable read-only modality $\text{RO}(H)$.

Chapter 24

Parallelism and Concurrency

Parallel pairs

A parallel pair, written $(|t_1, t_2|)$, for evaluating two subterms in parallel.

(Note: one often sees $t_1||t_2$ for $\text{let } (((), () = (|t_1, t_2|) \text{ in } ())$)

Computing: $a[i] + a[i + 1] + \dots + a[j - 1]$.

```
let rec sum a i j =
  if j - i = 1 then a.(i) else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
      s1 + s2
  end
```

Efficient use of parallel pairs with granularity control

```
let rec sum a i j =
  if j - i < sequential_cutoff then begin
    let r = ref 0 in
    for k = i to j-1 do
      r := !r + a.(k)
    done;
    !r
  end else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (|| sum a i m, sum a m j ||) in
    s1 + s2
  end
```

Efficient use of parallel pairs with granularity control

```
let rec sum a i j =
  if j - i < sequential_cutoff then begin
    let r = ref 0 in
    for k = i to j-1 do
      r := !r + a.(k)
    done;
    !r
  end else begin
    let m = (i+j) / 2 in
    let (s1,s2) = (| sum a i m, sum a m j |) in
    s1 + s2
  end
```

Generalizable to map-reduce:

$f(t[0]) \oplus f(a[1]) \oplus \dots \oplus f(a[n-1]).$
(on which condition on \oplus ?)

Reasoning rule for parallel pairs

$$\frac{\{H_1\} t_1 \{Q_1\} \quad \{H_2\} t_2 \{Q_2\}}{\{H_1 * H_2\} (|t_1, t_2|) \{Q_1 \star Q_2\}} \text{ PARALLEL}$$

where $Q_1 \star Q_2 \equiv \lambda(x_1, x_2). Q_1 x_1 * Q_2 x_2$

Reasoning rule for parallel pairs

$$\frac{\{H_1\} t_1 \{Q_1\} \quad \{H_2\} t_2 \{Q_2\}}{\{H_1 * H_2\} (|t_1, t_2|) \{Q_1 \star Q_2\}} \text{ PARALLEL}$$

where $Q_1 \star Q_2 \equiv \lambda(x_1, x_2). Q_1 x_1 * Q_2 x_2$

This rule restricts parallel threads to act on disjoint parts of memory.

(No need for non-interference conditions.)

Concurrent locks: example

```
let r = ref 0
let s = ref n
let p = create_lock()

let concurrent_step () =
    acquire_lock p;
    incr r;
    decr s;
    release_lock p
```

Concurrent locks: example

```
let r = ref 0
let s = ref n
let p = create_lock()

let concurrent_step () =
    acquire_lock p;
    incr r;
    decr s;
    release_lock p
```

Heap predicate $p \rightsquigarrow \text{Lock } H$ asserts that lock p protects an invariant H .

Here:

$$p \rightsquigarrow \text{Lock} (\exists i. (r \mapsto i) * (s \mapsto n - i))$$

Concurrent locks: specification of operations

Duplicable representation predicate:

$$p \rightsquigarrow \text{Lock } H$$

Operations:

$$\forall H. \quad \{H\} (\text{create_lock } ()) \{\lambda p. p \rightsquigarrow \text{Lock } H\}$$

$$\forall pH. \quad \{p \rightsquigarrow \text{Lock } H\} (\text{acquire_lock } p) \{\lambda_. H * p \rightsquigarrow \text{Lock } H\}$$

$$\forall pH. \{H * p \rightsquigarrow \text{Lock } H\} (\text{release_lock } p) \{\lambda_. p \rightsquigarrow \text{Lock } H\}$$

Concurrent locks: example

$$\begin{aligned} \forall H. & \quad \{H\} (\text{create_lock } ()) \{\lambda p. p \rightsquigarrow \text{Lock } H\} \\ \forall pH. & \quad \{p \rightsquigarrow \text{Lock } H\} (\text{acquire_lock } p) \{\lambda_. H * p \rightsquigarrow \text{Lock } H\} \\ \forall pH. & \{H * p \rightsquigarrow \text{Lock } H\} (\text{release_lock } p) \{\lambda_. p \rightsquigarrow \text{Lock } H\} \end{aligned}$$

```
1  let r = ref 0
2  let s = ref n
3  let p = create_lock ()
4
5  let concurrent_step () =
6    acquire_lock p;
7    incr r;
8    decr s;
9    release_lock p
```

Concurrent locks: example

```
1  let r = ref 0
2  let s = ref n
3  let p = create_lock ()
4
5  let concurrent_step () =
6      acquire_lock p;
7      incr r;
8      decr s;
9      release_lock p
```

1: \sqcap . 2: $r \mapsto 0$. 3: $r \mapsto 0 * s \mapsto n$.

4: $p \rightsquigarrow \text{Lock}(\exists i. (r \mapsto i) * (s \mapsto n - i))$.

7: $(r \mapsto i) * (s \mapsto n - i)$. 8: $(r \mapsto i + 1) * (s \mapsto n - i)$.

9: $(r \mapsto i + 1) * (s \mapsto n - i - 1)$. Instantiate the invariant with $i + 1$.

Concurrent locks: non-example

```
let r = ref 0
let p = create_lock()

let f () =
    acquire_lock p;
    incr r;
    release_lock p

let () =
    let _ = (| f(), f() |) in
    acquire_lock p;
    assert (!r == 2)
```

Chapter 25

Ghost state

Same non-example

```
let r = ref 0
let p = create_lock()

acquire_lock p;      ||      acquire_lock p;
r := !r + 1;          ||      r := !r + 1;
release_lock p;      ||      release_lock p;

acquire_lock p;
assert (!r == 2);
```

Same non-example

```
let r = ref 0
let p = create_lock()

acquire_lock p;      ||      acquire_lock p;
r := !r + 1;          ||      r := !r + 1;
release_lock p;      ||      release_lock p;

acquire_lock p;
assert (!r == 2);
```

$$p \rightsquigarrow \text{Lock}(\exists n. r \mapsto n * ^r \dots ? \dots ^r)$$

Same non-example

```
let r = ref 0
let p = create_lock()

acquire_lock p;      ||      acquire_lock p;
r := !r + 1;          ||      r := !r + 1;
release_lock p;      ||      release_lock p;
```

```
acquire_lock p;
assert (!r == 2);
```

$$p \rightsquigarrow \text{Lock}(\exists n. r \mapsto n * ^r \dots ? \dots ^r)$$

Problem: it is impossible to prove, only with invariants, that this program does not crash (i.e. to prove {True} program {True})

More variables! Ghost variables.

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = create_lock()

acquire_lock p;      ||  acquire_lock p;
r := !r + 1;          ||  r := !r + 1;
r1 := !r1 + 1;        ||  r2 := !r2 + 1;
release_lock p;      ||  release_lock p;

acquire_lock p;
assert (!r == 2);
```

More variables! Ghost variables.

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = create_lock()

acquire_lock p;      |||   acquire_lock p;
r := !r + 1;          r := !r + 1;
r1 := !r1 + 1;        r2 := !r2 + 1;
release_lock p;      release_lock p;

acquire_lock p;
assert (!r == 2);
```

Exercise: Give a lock invariant that allows proving {True} program {True} (hint: fractional permissions). Then prove the triple.

More variables! Ghost variables.

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = create_lock()

acquire_lock p;      |||   acquire_lock p;
r := !r + 1;          |||   r := !r + 1;
r1 := !r1 + 1;        |||   r2 := !r2 + 1;
release_lock p;      |||   release_lock p;

acquire_lock p;
assert (!r == 2);
```

Exercise: Give a lock invariant that allows proving {True} program {True} (hint: fractional permissions). Then prove the triple.

$$p \rightsquigarrow \text{Lock}(\exists n, n_1, n_2. r \mapsto n * r_1 \stackrel{1/2}{\mapsto} n_1 * r_2 \stackrel{1/2}{\mapsto} n_2 * \lceil n = n_1 + n_2 \rceil)$$

Proof

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
{r ↦ 0 * r1 ↦ 0 * r2 ↦ 0}
```

Proof

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
{r  $\mapsto$  0 * r1  $\mapsto$  0 * r2  $\mapsto$  0}
{H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
```

Proof

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
{r ↦ 0 * r1 ↦ 0 * r2 ↦ 0}
{H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
let p = create_lock()
```

Proof

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
{r ↦ 0 * r1 ↦ 0 * r2 ↦ 0}
{H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
let p = create_lock()
{p ↠ Lock H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
```

Proof

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
{r ↦ 0 * r1 ↦ 0 * r2 ↦ 0}
{H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
let p = create_lock()
{p ↦ Lock H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
{(p ↦ Lock H * r1  $\xrightarrow{1/2}$  0) * (p ↦ Lock H * r2  $\xrightarrow{1/2}$  0)}
```

Proof

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
{r ↦ 0 * r1 ↦ 0 * r2 ↦ 0}
{H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
let p = create_lock()
{p ↤ Lock H * r1  $\xrightarrow{1/2}$  0 * r2  $\xrightarrow{1/2}$  0}
{(p ↤ Lock H * r1  $\xrightarrow{1/2}$  0) * (p ↤ Lock H * r2  $\xrightarrow{1/2}$  0)}
{p ↤ Lock H * r1  $\xrightarrow{1/2}$  0} || {p ↤ Lock H * r2  $\xrightarrow{1/2}$  0}
    acquire_lock p;           acquire_lock p;
    r := !r + 1;             r := !r + 1;
    ...                      ...
```

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

```
acquire_lock p;
```

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\}$$

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 0 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

r1 := !r1 + 1;

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 0 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

r1 := !r1 + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 1 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 0 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

r1 := !r1 + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 1 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} 1 * r_2 \xrightarrow{1/2} n_2\}$$

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 0 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

r1 := !r1 + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 1 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * H\} \text{ (choosing } n_1 = 1 \text{ and } n_2 = n_2\text{)}$$

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 0 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

r1 := !r1 + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 1 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * H\} \text{ (choosing } n_1 = 1 \text{ and } n_2 = n_2\text{)}$$

release_lock p;

Left thread

$$H \equiv \exists n, n_1, n_2. \ r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2 * \lceil n = n_1 + n_2 \rceil$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0\}$$

acquire_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * H\} \text{ so, for some } n, n_1, n_2 \text{ s. that } n = n_1 + n_2:$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\}$$

r := !r + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 0 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} n_1 * r_2 \xrightarrow{1/2} n_2\} \text{ so } n_1 = 0, \text{ and}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 0 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

r1 := !r1 + 1;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1} 1 * r \mapsto n + 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * r \mapsto n + 1 * r_1 \xrightarrow{1/2} 1 * r_2 \xrightarrow{1/2} n_2\}$$

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1 * H\} \text{ (choosing } n_1 = 1 \text{ and } n_2 = n_2\text{)}$$

release_lock p;

$$\{p \rightsquigarrow \text{Lock } H * r_1 \xrightarrow{1/2} 1\}$$

Right thread

$$H \equiv \exists n, n_1, n_2. r \mapsto n * (r_1 \xrightarrow{1/2} n_1) * (r_2 \xrightarrow{1/2} n_2) * \lceil n = n_1 + n_2 \rceil$$

```
{p ~> Lock H * r2  $\xrightarrow{1/2}$  0}
acquire_lock p;
{p ~> Lock H * r2  $\xrightarrow{1/2}$  0 * H}
r := !r + 1;
r2 := !r2 + 1;
{p ~> Lock H * r2  $\xrightarrow{1/2}$  1 * H}
release_lock p;
{p ~> Lock H * r2  $\xrightarrow{1/2}$  1}
```

Finish up

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = create_lock()

{p ~> Lock H * r1  $\xrightarrow{1/2}$  0} || {p ~> Lock H * r2  $\xrightarrow{1/2}$  0}
acquire_lock p;
r := !r + 1;
r1 := !r1 + 1;
release_lock p;
{p ~> Lock H * r1  $\xrightarrow{1/2}$  1} || {p ~> Lock H * r2  $\xrightarrow{1/2}$  1}
{p ~> Lock H * r1  $\xrightarrow{1/2}$  1 * r2  $\xrightarrow{1/2}$  1}
    acquire_lock p;
{r1  $\xrightarrow{1/2}$  1 * r2  $\xrightarrow{1/2}$  1 * r  $\mapsto$  n * r1  $\xrightarrow{1/2}$  n1 * r2  $\xrightarrow{1/2}$  n2 * `n = n1 + n2`}
{r1  $\mapsto$  1 * r2  $\mapsto$  1 * r  $\mapsto$  n * `n = 1 + 1`}
    assert (!r == 2);
```

Some remarks

Ghost variables are the old way, but:

- what if you have an arbitrary number of threads?
- need some erasure theorem
- reasoning about the program should not be *in* the program

Some remarks

Ghost variables are the old way, but:

- what if you have an arbitrary number of threads?
- need some erasure theorem
- reasoning about the program should not be *in* the program

Ghost state is a more robust approach. How they work in *Iris*:

- allocation of ghost state: for some a any “Resource Algebra”:

$$\text{True} \not\equiv * \exists \gamma. \gamma \rightsquigarrow \text{Ghost}(a)$$

Some remarks

Ghost variables are the old way, but:

- what if you have an arbitrary number of threads?
- need some erasure theorem
- reasoning about the program should not be *in* the program

Ghost state is a more robust approach. How they work in *Iris*:

- allocation of ghost state: for some a any “Resource Algebra”:

$$\text{True} \not\equiv * \exists \gamma. \gamma \rightsquigarrow \text{Ghost}(a)$$

- splitting of ghost state: for any a, b in an RA:

$$\gamma \rightsquigarrow \text{Ghost}(a \cdot b) \Leftrightarrow \gamma \rightsquigarrow \text{Ghost}(a) * \gamma \rightsquigarrow \text{Ghost}(b)$$

Some remarks

Ghost variables are the old way, but:

- what if you have an arbitrary number of threads?
- need some erasure theorem
- reasoning about the program should not be *in* the program

Ghost state is a more robust approach. How they work in *Iris*:

- allocation of ghost state: for some a any “Resource Algebra”:

$$\text{True} \not\equiv * \exists \gamma. \gamma \rightsquigarrow \text{Ghost}(a)$$

- splitting of ghost state: for any a, b in an RA:

$$\gamma \rightsquigarrow \text{Ghost}(a \cdot b) \Leftrightarrow \gamma \rightsquigarrow \text{Ghost}(a) * \gamma \rightsquigarrow \text{Ghost}(b)$$

- validity in RA's e.g. $\text{valid}((r \mapsto n_1) \cdot (r \mapsto n_2)) \Rightarrow n_1 = n_2$
- heaps are a RA (composition of fractional RA + agreement RA)

Chapter 26: weakest preconditions

Presentation with weakest preconditions

WP are generally preferred to triples, in practice:

- more primitive:

$$\{P\}e\{\Phi\} \equiv P \dashv\wp e\Phi$$

- preconditions become hypotheses, more easily managed

Presentation with weakest preconditions

WP are generally preferred to triples, in practice:

- more primitive:

$$\{P\}e\{\Phi\} \equiv P \dashv\wp e\Phi$$

- preconditions become hypotheses, more easily managed

Iris hypotheses go in contexts and can be named, split, rearranged...

$$\begin{array}{ccc} P1 * P2 \dashv\wp e \Phi & \xrightarrow{\hspace{1cm}} & \begin{array}{c} H : P1 * P2 \\ \hline \hline \end{array} \xrightarrow{\hspace{1cm}} \begin{array}{c} H1 : P1 \\ H2 : P2 \\ \hline \hline \end{array} \\ & & \begin{array}{c} wp e \Phi \\ \hline \hline \end{array} \xrightarrow{\hspace{1cm}} \begin{array}{c} wp e \Phi \\ \hline \hline \end{array} \end{array}$$

Builtin consequence rule in postconditions

$\text{wp } e \cdot$ is a monotone predicate transformer, so $\text{wp } e \Phi$ is equivalent to

$$\forall \Psi \ (\forall v \ \Phi v \rightarrow* \Psi v) \rightarrow* \text{wp } e \Psi$$

Builtin consequence rule in postconditions

$\text{wp } e \cdot$ is a monotone predicate transformer, so $\text{wp } e \Phi$ is equivalent to

$$\forall \Psi (\forall v \Phi v \rightarrow* \Psi v) \rightarrow* \text{wp } e \Psi$$

So instead of choosing

$$\{P\}e\{\Phi\} \equiv P \rightarrow* \text{wp } e \Phi$$

the following formulation is preferred:

$$\{P\}e\{\Phi\} \equiv \forall \Psi P \rightarrow* (\forall v \Phi v \rightarrow* \Psi v) \rightarrow* \text{wp } e \Psi$$

Specifications can be applied directly since Ψ is universally quantified.

Note that separating implication $\rightarrow*$ has no easy “heap entailment” \rhd counterpart here.

Simplified rules for load, store, alloc

Exercise:

$$\begin{array}{lll} \forall \ell v \Phi & \dots & \neg* \left(\dots \right) \rightarrow* \text{wp}(\ell) \Phi \\ \forall \ell v v' \Phi & \dots & \neg* \left(\dots \right) \rightarrow* \text{wp}(\ell \leftarrow v) \Phi \\ \forall v \Phi & \dots & \neg* \left(\dots \right) \rightarrow* \text{wp}(\text{ref } v) \Phi \end{array}$$

Simplified rules for load, store, alloc

$\forall \ell v v' \Phi$	$\ell \mapsto v'$	$\rightarrow * (\ell \mapsto v \rightarrow * \Phi())$	$\rightarrow * \text{wp}(\ell \leftarrow v) \Phi$
$\forall v \Phi$	\top	$\rightarrow * (\forall \ell \quad \ell \mapsto v \rightarrow * \Phi \ell)$	$\rightarrow * \text{wp}(\text{ref } v) \Phi$
$\forall \ell v \Phi$	$\ell \mapsto v$	$\rightarrow * (\ell \mapsto v \rightarrow * \Phi v)$	$\rightarrow * \text{wp}(!\ell) \Phi$

Exemple

```
{ℓ ↦ 1}ref 3{ℓ'. ℓ ↦ 1 * ℓ' ↦ 3}
```

Exemple

$$\{\ell \mapsto 1\} \text{ref } 3 \{\ell'. \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi(\ell \mapsto 1) \multimap (\forall \ell'. \ell \mapsto 1 * \ell' \mapsto 3 \multimap \Phi \ell') \multimap \text{wp}(\text{ref } 3) \Phi$$

Exemple

$$\{\ell \mapsto 1\} \text{ref } 3 \{\ell'. \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi(\ell \mapsto 1) \rightarrow* (\forall \ell'. \ell \mapsto 1 * \ell' \mapsto 3 \rightarrow* \Phi \ell') \rightarrow* \text{wp}(\text{ref } 3) \Phi$$

after `iIntros` (Φ) "H1 H Φ " you have:

"H1" : $l \mapsto \#1$

"H Φ " : $\forall l' : \text{loc}, l \mapsto \#1 * l' \mapsto \#3 \rightarrow* \Phi \#l'$

-----*

WP ref #3 {{ v, Φ v }}

Exemple

$$\{\ell \mapsto 1\} \text{ref } 3 \{\ell'. \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi (\ell \mapsto 1) \rightarrow* (\forall \ell'. \ell \mapsto 1 * \ell' \mapsto 3 \rightarrow* \Phi \ell') \rightarrow* \text{wp}(\text{ref } 3) \Phi$$

after iIntros (Φ) "H1 H Φ " you have:

"H1" : $l \mapsto \#1$

"H Φ " : $\forall l' : \text{loc}, l \mapsto \#1 * l' \mapsto \#3 \rightarrow* \Phi \#l'$

-----*

WP ref #3 {{ v, Φ v }}

applying the rule for allocation, we get:

"H1" : $l \mapsto \#1$

"H Φ " : $\forall l' : \text{loc}, l \mapsto \#1 * l' \mapsto \#3 \rightarrow* \Phi \#l'$

-----*

$\forall 10 : \text{loc}, 10 \mapsto \#3 \rightarrow* \Phi \#10$

after `iIntros (l') "Hl'"` we get:

```
"Hl" : l ↪ #1
"HΦ" : ∀ l'0 : loc, l ↪ #1 * l'0 ↪ #3 -* Φ #l'0
"Hl'" : l' ↪ #3
-----*
Φ #l'
```

after iIntros (l') "Hl'" we get:

```
"Hl" : l ↪ #1
"HΦ" : ∀ l'0 : loc, l ↪ #1 * l'0 ↪ #3 -* Φ #l'0
"Hl'" : l' ↪ #3
-----*
Φ #l'
```

after iApply "HΦ" we get:

```
"Hl" : l ↪ #1
"Hl'" : l' ↪ #3
-----*
l ↪ #1 * l' ↪ #3
```

after iIntros (l') "Hl'" we get:

```
"Hl" : l ↪ #1
"HΦ" : ∀ l'0 : loc, l ↪ #1 * l'0 ↪ #3 -* Φ #l'0
"Hl'" : l' ↪ #3
-----*
Φ #l'
```

after iApply "HΦ" we get:

```
"Hl" : l ↪ #1
"Hl'" : l' ↪ #3
-----*
l ↪ #1 * l' ↪ #3
```

iFrame solves the goal.

after iIntros (l') "Hl'" we get:

```
"Hl" : l ↪ #1
"HΦ" : ∀ l'0 : loc, l ↪ #1 * l'0 ↪ #3 -* Φ #l'0
"Hl'" : l' ↪ #3
-----*
Φ #l'
```

after iApply "HΦ" we get:

```
"Hl" : l ↪ #1
"Hl'" : l' ↪ #3
-----*
l ↪ #1 * l' ↪ #3
```

iFrame solves the goal.

⚠ Iris is an affine logic, it can throw away hypotheses. ⚠

Chapter 27: modalities

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * 'P$ is duplicable' (or "persistent")

$\Box P \triangleright P$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * 'P \text{ is duplicable}'$ (or “persistent”)

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * 'P \text{ is duplicable}'$ (or “persistent”)

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * 'P \text{ is duplicable}'$ (or “persistent”)

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

$$\Box P \triangleright \Box P * P * P$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * \text{'P is duplicable'}$ (or “persistent”)

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

$$\Box P \triangleright \Box P * P * P$$

$$\text{'P} \triangleright \Box \text{'P}$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P *$ 'P is duplicable' (or "persistent")

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

$$\Box P \triangleright \Box P * P * P$$

$${}^r P \triangleright \Box {}^r P$$

$$\Box(\ell \mapsto 1) \triangleright \text{'False'}$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * \text{'P is duplicable'}$ (or “persistent”)

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

$$\Box P \triangleright \Box P * P * P$$

$$\text{'P'} \triangleright \Box \text{'P'}$$

$$\Box(\ell \mapsto 1) \triangleright \text{'False'}$$

$$\Box(\ell \stackrel{q}{\mapsto} 1) \triangleright \text{'q=0'} \text{ (if even allowed)}$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P *$ 'P is duplicable' (or "persistent")

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

$$\Box P \triangleright \Box P * P * P$$

$$`P' \triangleright \Box `P'$$

$$\Box(\ell \mapsto 1) \triangleright `False'$$

$$\Box(\ell \stackrel{q}{\mapsto} 1) \triangleright `q=0' \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \equiv \Box(\forall \Psi P \rightarrow* (\forall v \Phi v \rightarrow* \Psi v) \rightarrow* \text{wp } e \Psi)$$

Persistently modality \Box

$\Box P$ is roughly equivalent to $P * \text{'P is duplicable'}$ (or “persistent”)

$$\Box P \triangleright P$$

$$\Box P \triangleright \Box P * P$$

$$\Box P \triangleright \Box P * \Box P$$

$$\Box P \triangleright \Box P * P * P$$

$$\text{'P'} \triangleright \Box \text{'P'}$$

$$\Box(\ell \mapsto 1) \triangleright \text{'False'}$$

$$\Box(\ell \xrightarrow{q} 1) \triangleright \text{'q=0'} \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \equiv \Box(\forall \Psi \ P \multimap (\forall v \ \Phi v \multimap \Psi v) \multimap \text{wp } e \ \Psi)$$

Persistent resources can be shared between threads:

$$\frac{\{P_1 * \Box P\} \ e_1 \ \{Q_1\} \quad \{P_2 * \Box P\} \ e_2 \ \{Q_2\}}{\{P_1 * P_2 * \Box P\} \ (|e_1, e_2|) \ \{Q_1 \star Q_2\}}$$

Persistently modality \square

$\square P$ is roughly equivalent to $P * \lceil P \text{ is duplicable} \rceil$ (or “persistent”)

$$\square P \triangleright P$$

$$\square P \triangleright \square P * P$$

$$\square P \triangleright \square P * \square P$$

$$\square P \triangleright \square P * P * P$$

$$\lceil P \rceil \triangleright \square \lceil P \rceil$$

$$\square(\ell \mapsto 1) \triangleright \lceil \text{False} \rceil$$

$$\square(\ell \xrightarrow{q} 1) \triangleright \lceil q=0 \rceil \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \equiv \square(\forall \Psi \ P \rightarrow* (\forall v \ \Phi v \rightarrow* \Psi v) \rightarrow* \text{wp } e \ \Psi)$$

Persistent resources can be shared between threads:

$$\frac{\{P_1 * \square P\} \ e_1 \ \{Q_1\} \quad \{P_2 * \square P\} \ e_2 \ \{Q_2\}}{\{P_1 * P_2 * \square P\} \ (|e_1, e_2|) \ \{Q_1 \star Q_2\}}$$

Some ghost resources are persistents (e.g. to indicate a task is done), some are not (e.g. to provide a thread with an information it can consume).

Later modality \triangleright

$\triangleright P$ ("later P") can be thought as " P holds after one reduction step".

\triangleright is a modality ($\triangleright P$), \triangleright is a binary predicate ($P \triangleright Q$)

Later modality \triangleright

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

\triangleright is a modality ($\triangleright P$), \triangleright is a binary predicate ($P \triangleright Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

Later modality \triangleright

$\triangleright P$ (“later P”) can be thought as “ P holds after one reduction step”.

\triangleright is a modality ($\triangleright P$), \triangleright is a binary predicate ($P \triangleright Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

\triangleright and \square are modalities in Iris, where “iProp” are not “heap \rightarrow Prop” and have features such as step indexing, and resources are not only heaps.

Later modality \triangleright

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

\triangleright is a modality ($\triangleright P$), \triangleright is a binary predicate ($P \triangleright Q$)

$$P \vdash \triangleright P \quad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \quad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \quad \text{etc}$$

\triangleright and \Box are modalities in Iris, where "iProp" are not "heap \rightarrow Prop" and have features such as step indexing, and resources are not only heaps.

$\triangleright^n \text{False} \vdash P$ iff P holds for n steps

Later modality \triangleright

$\triangleright P$ ("later P") can be thought as "P holds after one reduction step".

\triangleright is a modality ($\triangleright P$), \triangleright is a binary predicate ($P \triangleright Q$)

$$P \vdash \triangleright P \quad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \quad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \quad \text{etc}$$

\triangleright and \Box are modalities in Iris, where "iProp" are not "heap \rightarrow Prop" and have features such as step indexing, and resources are not only heaps.

$\triangleright^n \text{False} \vdash P$ iff P holds for n steps $\vdash \exists k. \triangleright^k \text{False}$

More later

The **Löb rule** is very convenient for partial correctness:

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

More later

The **Löb rule** is very convenient for partial correctness:

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

Step reductions “consume” laters:

$$\frac{\{P\}e'\{Q\} \quad e \rightarrow e'}{\{\triangleright P\}e\{Q\}}$$

More later

The **Löb rule** is very convenient for partial correctness:

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

Step reductions “consume” laters:

$$\frac{\{P\}e'\{Q\} \quad e \rightarrow e'}{\{\triangleright P\}e\{Q\}}$$

Final definition of triples:

$$\{P\}e\{\Phi\} \equiv \square(\forall\Psi P \multimap \triangleright(\forall v \Phi v \multimap \Psi v) \multimap \text{wp } e \Psi)$$

Complete set of rules: [Lecture Notes on Iris](#)

Invariants

New construct \boxed{R}^ℓ : duplicable, but the resource R is lost at allocation:

$$\boxed{R}^\ell \vdash \square \boxed{R}^\ell \quad \frac{S, \boxed{R}^\ell \vdash \{P\}e\{Q\}}{S \vdash \{\triangleright R * P\}e\{Q\}}$$

INV-ALLOC

Invariants

New construct \boxed{R}^ℓ : duplicable, but the resource R is lost at allocation:

$$\boxed{R}^\ell \vdash \square \boxed{R}^\ell \quad \frac{S, \boxed{R}^\ell \vdash \{P\}e\{Q\}}{S \vdash \{\triangleright R * P\}e\{Q\}}$$

INV-ALLOC

Invariant resources can be accessed but must be preserved:

$$e \text{ is atomic} \quad \frac{S, \boxed{R}^\ell \vdash \{\triangleright R * P\}e\{\triangleright R * Q\}}{S, \boxed{R}^\ell \vdash \{P\}e\{Q\}}$$

INV-OPEN

(note: easy to have inconsistent invariants rules, one needs to add stratification not to nest openings)

Locks

Locks can be derived from Compare-And-Swap:

```
let create_lock () = ref false
let acquire p = if CAS p false true then () else
    acquire p
let release p = p := false
```

Locks

Locks can be derived from Compare-And-Swap:

```
let create_lock () = ref false
let acquire p = if CAS p false true then () else
    acquire p
let release p = p := false
```

The logic rules can be derived using invariants:

$$p \rightsquigarrow \text{Lock}(R, \gamma) \equiv \exists \iota. \boxed{p \mapsto \text{true} \And p \mapsto \text{false} * R * \gamma \rightsquigarrow \text{Ghost}(K)}^\iota$$

with resource algebra (ε, K, \perp) s.t. $x \cdot \varepsilon = \varepsilon \cdot x = x$ otherwise
 $x \cdot y = \perp$.

Locks

Locks can be derived from Compare-And-Swap:

```
let create_lock () = ref false
let acquire p = if CAS p false true then () else
    acquire p
let release p = p := false
```

The logic rules can be derived using invariants:

$$p \rightsquigarrow \text{Lock}(R, \gamma) \equiv \exists \iota. \boxed{p \mapsto \text{true} \iff p \mapsto \text{false} * R * \gamma \rightsquigarrow \text{Ghost}(K)}$$

with resource algebra (ε, K, \perp) s.t. $x \cdot \varepsilon = \varepsilon \cdot x = x$ otherwise
 $x \cdot y = \perp$.

$$\forall R. \quad \{R\} (\text{create_lock } ()) \{\lambda p. \exists \gamma. p \rightsquigarrow \text{Lock}(R, \gamma)\}$$

$$\forall pR. \quad \{p \rightsquigarrow \text{Lock}(R, \gamma)\} (\text{acquire_lock } p) \{\lambda_. R * p \rightsquigarrow \text{Lock}(R, \gamma)\}$$

$$\forall pR. \{R * p \rightsquigarrow \text{Lock}(R, \gamma)\} (\text{release_lock } p) \{\lambda_. p \rightsquigarrow \text{Lock}(R, \gamma)\}$$

Prophecy variables

Allows to talk about a future, unknown, value
Invented by Abadi and Lamport [1988, 1991]
implemented Iris by Jung et al. [2019]:

$$\overline{\{ \lceil \rceil \} \text{ newProph()} \{ \lambda p. \exists v. \text{Proph}(p, v) \}}$$

$$\overline{\{ \text{Proph}(p, v) \} \text{ resolve } p \text{ to } w \{ \lambda_. \lceil v = w \rceil \}}$$

Effect handlers

$$\begin{array}{c}
 \text{VALUE} \\
 \frac{\Phi(v)}{ewp_{\mathcal{E}} v \langle \Psi \rangle \{ \Phi \}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{Do} \\
 \frac{\Psi \text{ allows do } v \{ \Phi \}}{ewp_{\mathcal{E}} (\text{do } v) \langle \Psi \rangle \{ \Phi \}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{MONOTONICITY} \\
 \frac{\begin{array}{c} ewp_{\mathcal{E}_1} e \langle \Psi_1 \rangle \{ \Phi_1 \} \\ \mathcal{E}_1 \subseteq \mathcal{E}_2 \\ \Psi_1 \sqsubseteq \Psi_2 \\ \forall v. \Phi_1(v) \multimap \Phi_2(v) \end{array}}{ewp_{\mathcal{E}_2} e \langle \Psi_2 \rangle \{ \Phi_2 \}}
 \end{array}$$

$$\begin{array}{c}
 \text{BIND} \\
 \frac{ewp_{\mathcal{E}} e \langle \Psi \rangle \{ v. ewp_{\mathcal{E}} N[v] \langle \Psi \rangle \{ \Phi \} \}}{ewp_{\mathcal{E}} N[e] \langle \Psi \rangle \{ \Phi \}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{BIND-PURE} \\
 \frac{ewp_{\mathcal{E}} e \langle \perp \rangle \{ v. ewp_{\mathcal{E}} K[v] \langle \Psi \rangle \{ \Phi \} \}}{ewp_{\mathcal{E}} K[e] \langle \Psi \rangle \{ \Phi \}}
 \end{array}$$

$$\frac{\begin{array}{c} ewp_{\mathcal{E}} e \langle \Psi \rangle \{ \Phi \} \\ \forall v. \Phi(v) \multimap \triangleright ewp_{\mathcal{E}} (r v) \langle \Psi' \rangle \{ \Phi' \} \end{array}}{\begin{array}{c} \forall v, k. \left\{ \begin{array}{l} \Psi \text{ allows do } v \{ w. ewp_{\mathcal{E}} (k w) \langle \Psi \rangle \{ \Phi \} \} \multimap \\ \triangleright ewp_{\mathcal{E}} (h v k) \langle \Psi' \rangle \{ \Phi' \} \end{array} \right\} \\ \hline ewp_{\mathcal{E}} (\text{shallow-try } e \text{ with } h \mid r) \langle \Psi' \rangle \{ \Phi' \} \end{array}}$$

$$\frac{\begin{array}{c} ewp_{\mathcal{E}} e \langle \Psi \rangle \{ \Phi \} \\ \forall v. \Phi(v) \multimap \triangleright ewp_{\mathcal{E}} (r v) \langle \Psi' \rangle \{ \Phi' \} \end{array}}{\forall v, k. \left\{ \begin{array}{l} \Psi \text{ allows do } v \{ w. \forall \Psi'', \Phi''. \\ \triangleright \text{deep-handler}_{\mathcal{E}} \langle \Psi \rangle \{ \Phi \} h \mid r \langle \Psi'' \rangle \{ \Phi'' \} \multimap \\ ewp_{\mathcal{E}} (k w) \langle \Psi'' \rangle \{ \Phi'' \} \\ \} \multimap \\ \triangleright ewp_{\mathcal{E}} (h v k) \langle \Psi' \rangle \{ \Phi' \} \end{array} \right\}}$$

Conclusion

Some separation logic features:

- tree-like structures, some sharing,
- abstracting intermediate pointers, internal structure
- higher-order representation predicates
- first-class functions, local state
- ghost state, invariants
- concurrency (rich literature, including weak memory models)
- effects (rich literature here too, e.g. effect handlers)

Bibliography: comprehensive [lecture notes on Iris](#)