# Introduction to Separation Logic

Jean-Marie Madiot

INRIA Paris

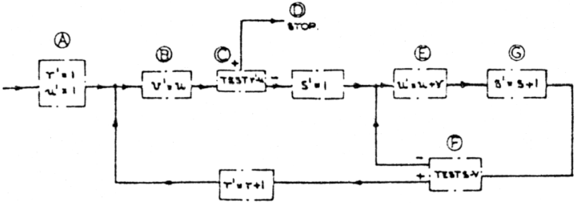
Glasgow, 2024 July 30 - August 2 @ SPLV24

Please interrupt me!

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

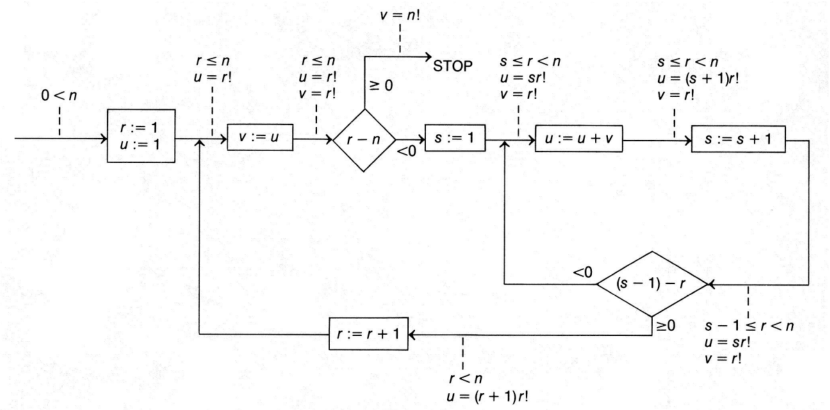
# Alan Turing, 1949 : *Checking a large routine*

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




| STORAGE LOCATION | (INITIAL) Ⓐ k=6 | Ⓑ k=5 | Ⓒ k=4 | (STOP) Ⓓ k=0 | Ⓔ k=3 | Ⓕ k=1 | Ⓖ k=2 |
|---|---|---|---|---|---|---|---|
| 27 | | | | | s | s+1 | s |
| 28 | | r | r | | r | r | r |
| 29 | n | n | n | n | r | n | n |
| 30 | | $\underline{r}$ | $\underline{r}$ | | $s\underline{r}$ | $(s+1)\underline{r}$ | $(s+1)\underline{r}$ |
| 31 | | | $\underline{r}$ | $\underline{n}$ | $\underline{r}$ | $\underline{r}$ | $\underline{r}$ |
| | TO Ⓑ WITH r'=1 u'=1 | TO Ⓒ | TO Ⓓ IF r=n TO Ⓔ IF r<n | | TO Ⓖ | TO Ⓑ WITH r'=r+1 IF s≥r TO Ⓔ WITH s'=s+1 IF s<r | 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:

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

# The "While" or "IMP" language

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

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

Examples :

```
skip
```

# The “While” or “IMP” language

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

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

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

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:

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

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:

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

Examples :

| | |
|---|---|
| `skip` | does nothing |
| $x := 1; x := 2 * x$ | 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:

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

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

Examples :

| | |
|---|---|
| `skip` | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if } x > 0 \texttt{ then } r := x \texttt{ else } r := 0 - x$ | |

# The “While” or “IMP” language

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

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

Examples :

| | |
|---|---|
| `skip` | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if } x > 0 \texttt{ then } r := x \texttt{ else } r := 0 - x$ | computes $r = \lvert x\rvert$ |

# The "While" or "IMP" language

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

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

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

Examples :

| | |
|---|---|
| $\texttt{skip}$ | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if}\ x > 0\ \texttt{then}\ r := x\ \texttt{else}\ r := 0 - x$ | computes $r = \lvert x \rvert$ |
| $\texttt{while}\ n \neq 0\ \texttt{do}\ n := n - 1$ | |

# The "While" or "IMP" language

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

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

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

Examples :

| | |
|---|---|
| `skip` | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if } x > 0 \texttt{ then } r := x \texttt{ else } r := 0 - x$ | computes $r = \|x\|$ |
| $\texttt{while } n \neq 0 \texttt{ do } n := n - 1$ | 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:

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

Examples :

| | |
|---|---|
| $\texttt{skip}$ | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if}\ x > 0\ \texttt{then}\ r := x\ \texttt{else}\ r := 0 - x$ | computes $r = \lvert x \rvert$ |
| $\texttt{while}\ n \neq 0\ \texttt{do}\ n := n - 1$ | computes $0$ if $\mathbb{N}$ or $\mathbb{Z}/n\mathbb{Z}$ |
| $r := 1; \texttt{while}\ n > 0\ \texttt{do}\ (r := r * x; n := n - 1)$ | |

# The “While” or “IMP” language

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

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

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

Examples :

| | |
|---|---|
| $\texttt{skip}$ | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if}\ x > 0\ \texttt{then}\ r := x\ \texttt{else}\ r := 0 - x$ | computes $r = \lvert x \rvert$ |
| $\texttt{while}\ n \neq 0\ \texttt{do}\ n := n - 1$ | computes $0$ if $\mathbb{N}$ or $\mathbb{Z}/n\mathbb{Z}$ |
| $r := 1;\texttt{while}\ n > 0\ \texttt{do}\ (r := r * x; n := n - 1)$ | 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:

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

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

Examples :

| | |
|---|---|
| $\texttt{skip}$ | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if } x > 0 \texttt{ then } r := x \texttt{ else } r := 0 - x$ | computes $r = \lvert x \rvert$ |
| $\texttt{while } n \neq 0 \texttt{ do } n := n - 1$ | computes $0$ if $\mathbb{N}$ or $\mathbb{Z}/n\mathbb{Z}$ |
| $r := 1; \texttt{while } n > 0 \texttt{ do } (r := r * x; n := n - 1)$ | computes $r = x^n$ |
| $x := 0; s := 1; \texttt{while } s \leqslant n \texttt{ do } x := x + 1; s := s + 2 * x + 1$ | |

# The “While” or “IMP” language

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

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

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

Examples :

| | |
|---|---|
| $\texttt{skip}$ | does nothing |
| $x := 1; x := 2 * x$ | computes $x = 2$ |
| $\texttt{if } x > 0 \texttt{ then } r := x \texttt{ else } r := 0 - x$ | computes $r = \lvert x \rvert$ |
| $\texttt{while } n \neq 0 \texttt{ do } n := n - 1$ | computes $0$ if $\mathbb{N}$ or $\mathbb{Z}/n\mathbb{Z}$ |
| $r := 1;\texttt{while } n > 0 \texttt{ do } (r := r * x; n := n - 1)$ | computes $r = x^n$ |
| $x := 0; s := 1;\texttt{while } s \leqslant n \texttt{ do } x := x + 1; s := s + 2 * x + 1$ | 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{x+1=1\}\ x := x+1\ \{x=1\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

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

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \wedge B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \wedge \neg B\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \wedge B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \wedge \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \wedge B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \wedge \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

Example: often easier to start from the end:

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

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \land B\}\ s_1\ \{Q\} \quad \{P \land \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \land B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \land \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

Example: often easier to start from the end:

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

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \quad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \wedge B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \wedge \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

Example: often easier to start from the end:

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

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \wedge B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \wedge \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

Example: often easier to start from the end:

$$\frac{\dfrac{}{\{\qquad\qquad\}\ x := x + 1\ \{2x > 6\}} \qquad \dfrac{}{\{2x > 6\}\ x := 2 * x\ \{x > 6\}}}{\{\qquad\qquad\}\ x := x + 1; x = 2 * x\ \{x > 6\}}$$

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \land B\}\ s_1\ \{Q\} \quad \{P \land \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \land B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \land \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

Example: often easier to start from the end:

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

# 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*:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x := e\ \{P\}}$$

$$\frac{\{P \wedge B\}\ s_1\ \{Q\} \quad \{P \wedge \neg B\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}$$

$$\frac{\{I \wedge B\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ s\ \{I \wedge \neg B\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

Example: often easier to start from the end:

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

# Weakest preconditions (Dijkstra, 1975)

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

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

$$\mathsf{wp}(\texttt{skip}, Q) \quad = \quad Q$$

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

$$
\begin{aligned}
\mathsf{wp}(\mathtt{skip}, Q) &= Q \\
\mathsf{wp}(x := e, Q) &= Q[e/x]
\end{aligned}
$$

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

$$\begin{array}{rcl} \mathsf{wp}(\texttt{skip}, Q) & = & Q \\ \mathsf{wp}(x := e, Q) & = & Q[e/x] \\ \mathsf{wp}(s_1; s_2, Q) & = & \mathsf{wp}(s_1, \mathsf{wp}(s_2, Q)) \end{array}$$

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

$$
\begin{aligned}
\mathsf{wp}(\texttt{skip}, Q) &= Q \\
\mathsf{wp}(x := e, Q) &= Q[e/x] \\
\mathsf{wp}(s_1; s_2, Q) &= \mathsf{wp}(s_1, \mathsf{wp}(s_2, Q)) \\
\mathsf{wp}(\texttt{if}\ B\ \texttt{then}\ s_1\ \texttt{else}\ s_2, Q) &= (B \Rightarrow \mathsf{wp}(s_1, Q)) \wedge (\neg B \Rightarrow \mathsf{wp}(s_2, Q))
\end{aligned}
$$

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

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

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

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

Examples :

- $\mathsf{wp}(x := 4,\ x \geqslant 1) = (4 \geqslant 1)$

# 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 $\mathsf{wp}(s, Q)$ such that $\{\mathsf{wp}(s, Q)\}\ s\ \{Q\}$:

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

Examples :

- $\mathsf{wp}(x := 4,\ x \geqslant 1) = (4 \geqslant 1)$
- $\mathsf{wp}((\texttt{if } x > 0 \texttt{ then } r := x \texttt{ else } r := 0 - x),\ r = |x|) =$
  $(x > 0 \Rightarrow x = |x|) \wedge (x \leqslant 0 \Rightarrow (0 - x = |x|))$

# 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:

$$
\begin{array}{l}
i := n; \\
r := 1; \\
\texttt{while } i > 0 \texttt{ do} \\
\quad (r := r * x; \\
\quad\; i := i - 1)
\end{array}
$$

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

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

# Example: a program computing $x^n$

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

$$\begin{array}{l} i := n; \\ r := 1; \\ \texttt{while } i > 0 \texttt{ do} \\ \quad (r := r * x; \\ \quad\ i := i - 1) \end{array}$$

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

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

# Example: a program computing $x^n$

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

$$
\begin{aligned}
&i := n;\\
&r := 1;\\
&\texttt{while } i > 0 \texttt{ do}\\
&\quad (r := r * x;\\
&\quad\ i := i - 1)
\end{aligned}
$$

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

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

# Example: a program computing $x^n$

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

$$
\begin{array}{l}
i := n; \\
r := 1; \\
\texttt{while } i > 0 \texttt{ do} \\
\quad (r := r * x; \\
\quad\; i := i - 1)
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{wp}(i := n; r := 1; \texttt{while } i > 0 \texttt{ do } (r := r * x; i := i - 1))(r = x^n) \\
= \mathsf{wp}(r := 1; \texttt{while } i > 0 \texttt{ do } (r := r * x; i := i - 1))(r = x^n)[n/i] \\
= \mathsf{wp}(\texttt{while } i > 0 \texttt{ do } (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\
= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x; i := i - 1, I)) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n)
\end{array}
$$

## Example: a program computing $x^n$

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

$$
\begin{aligned}
& i := n; \\
& r := 1; \\
& \texttt{while } i > 0 \texttt{ do} \\
& \quad (r := r * x; \\
& \quad\ i := i - 1)
\end{aligned}
$$

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

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

# Example: a program computing $x^n$

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

$$
\begin{array}{l}
i := n; \\
r := 1; \\
\texttt{while } i > 0 \texttt{ do} \\
\quad (r := r * x; \\
\quad\; i := i - 1)
\end{array}
$$

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

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

# Example: a program computing $x^n$

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

$$\begin{array}{l} i := n; \\ r := 1; \\ \texttt{while}\ i > 0\ \texttt{do} \\ \quad (r := r * x; \\ \quad\ i := i - 1) \end{array}$$

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

$$\begin{array}{l}
\mathsf{wp}(i := n; r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * x; i := i - 1))(r = x^n) \\
\ = \mathsf{wp}(r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * x; i := i - 1))(r = x^n)[n/i] \\
\ = \mathsf{wp}(\texttt{while}\ i > 0\ \texttt{do}\ (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\
\ = I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x; i := i - 1, I)) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
\ = I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x, \mathsf{wp}(i := i - 1, I))) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
\ = I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x, I[(i-1)/i])) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
\ = I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i-1)/i][(rx)/x]) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n)
\end{array}$$

# Example: a program computing $x^n$

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

$$\begin{array}{l} i := n; \\ r := 1; \\ \texttt{while } i > 0 \texttt{ do} \\ \quad (r := r * x; \\ \quad\; i := i - 1) \end{array}$$

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

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

# Example: a program computing $x^n$

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

$$
\begin{array}{l}
i := n; \\
r := 1; \\
\texttt{while}\ i > 0\ \texttt{do} \\
\quad (r := r * x; \\
\quad\ i := i - 1)
\end{array}
$$

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

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

# Example: a program computing $x^n$

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

$$\begin{array}{l} i := n; \\ r := 1; \\ \texttt{while}\ i > 0\ \texttt{do} \\ \quad (r := r * x; \\ \quad\ i := i - 1) \end{array}$$

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

$$\begin{array}{l}
\mathsf{wp}(i := n; r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * x; i := i - 1))(r = x^n) \\
= \mathsf{wp}(r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * x; i := i - 1))(r = x^n)[n/i] \\
= \mathsf{wp}(\texttt{while}\ i > 0\ \texttt{do}\ (r := r * x; i := i - 1))(r = x^n)[n/i][1/r] \\
= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x; i := i - 1, I)) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x, \mathsf{wp}(i := i - 1, I))) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow \mathsf{wp}(r := r * x, I[(i-1)/i])) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
= I[n/i][1/r] \wedge (i > 0 \wedge I \Rightarrow I[(i-1)/i][(rx)/x]) \wedge (i \leqslant 0 \wedge I \Rightarrow r = x^n) \\
= 1x^n = x^n \wedge n \geqslant 0 \wedge (i > 0 \wedge rx^i = x^n \wedge i \geqslant 0 \Rightarrow rx \cdot x^{i-1} = x^n \wedge i - 1 \geqslant 0) \\
\qquad \wedge (i \leqslant 0 \wedge rx^i = x^n \wedge i \geqslant 0 \Rightarrow r = x^n)
\end{array}$$

# Example: a program computing $x^n$

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

$$\begin{aligned}
&i := n;\\
&r := 1;\\
&\texttt{while } i > 0 \texttt{ do}\\
&\quad (r := r * x;\\
&\quad\ i := i - 1)
\end{aligned}$$

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

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

# Example: a program computing $x^n$

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

$$\begin{array}{l} i := n; \\ r := 1; \\ \texttt{while } i > 0 \texttt{ do} \\ \quad (r := r * x; \\ \quad\ i := i - 1) \end{array}$$

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

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

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 $\{P\}\ s\ \{Q\}$ and $P(m)$ then:

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

# It was all a lie

The other way around:

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

$$\frac{Q(m)}{\mathsf{wp}(\texttt{skip}, Q)(m)} \qquad \frac{\begin{array}{c}\exists m', s' \ (m, s) \to (m', s') \\ \forall m', s' \ (m, s) \to (m', s') \Rightarrow \mathsf{wp}(s', Q)(m')\end{array}}{\mathsf{wp}(s, Q)(m)}$$

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

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

$$\frac{}{m, (\texttt{skip}; s) \;\; \rightarrow \;\; m, s} \qquad \frac{m, s_1 \;\; \rightarrow \;\; m', s_1'}{m, (s_1; s_2) \;\; \rightarrow \;\; m', (s_1'; s_2)}$$

$$\frac{m(e) \neq 0}{m, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \;\; \rightarrow \;\; m, s_1} \qquad \frac{m(e) = 0}{m, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \;\; \rightarrow \;\; m, s_2}$$

$$\frac{m(e) \neq 0}{m, \texttt{while } e \texttt{ do } s \;\; \rightarrow \;\; m, (s; \texttt{while } e \texttt{ do } s)} \qquad \frac{m(e) = 0}{m, \texttt{while } e \texttt{ do } s \;\; \rightarrow \;\; m, \texttt{skip}}$$

# Operational semantics: example

$$
\begin{array}{rl}
 & (n \mapsto 5), (i := n; r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to & (n \mapsto 5, i \mapsto 5), (r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to & (n \mapsto 5, i \mapsto 5, r \mapsto 1), (\texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to & (n \mapsto 5, i \mapsto 5, r \mapsto 1), (r := r * 2; i := i - 1; \texttt{while}\ i > 0\ \texttt{do}\ \ldots) \\
\to & (n \mapsto 5, i \mapsto 5, r \mapsto 2), (i := i - 1; \texttt{while}\ i > 0\ \texttt{do}\ \ldots) \\
\to & (n \mapsto 5, i \mapsto 4, r \mapsto 2), (\texttt{while}\ i > 0\ \texttt{do}\ \ldots) \\
\to\ \to\ \to & (n \mapsto 5, i \mapsto 3, r \mapsto 4), (\texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to\ \to\ \to & (n \mapsto 5, i \mapsto 2, r \mapsto 8), (\texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to\ \to\ \to & (n \mapsto 5, i \mapsto 1, r \mapsto 16), (\texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to\ \to\ \to & (n \mapsto 5, i \mapsto 0, r \mapsto 32), (\texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)) \\
\to & (n \mapsto 5, i \mapsto 0, r \mapsto 32), \texttt{skip}
\end{array}
$$

Consistent with the earlier example:

$$\{n \geqslant 0\}\ i := n; r := 1; \texttt{while}\ i > 0\ \texttt{do}\ (r := r * 2; i := i - 1)\ \{r = 2^n\}$$

# 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 | NICTA | 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)

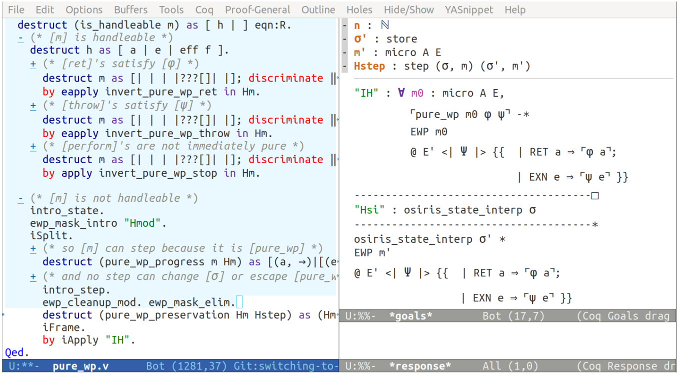
- work well on some classes of programs
- rely on user-provided invariants
- blackbox problem (hard to debug, extend, prove...)

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:




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 langages 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 adresses) to *values*.

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

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

**Joining**

When $\text{dom}(h_1) \cap \text{dom}(h_2) = \varnothing$ 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: $H : \texttt{heap} \to \texttt{Prop}$ where Prop is the type of propositions.

Primitive heap predicates:

| | |
|---|---|
| $\ulcorner\urcorner$ | empty heap |
| $\ulcorner P \urcorner$ | pure fact |
| $l \mapsto v$ | singleton heap |
| $H * H'$ | separating conjunction |
| $\exists x, H$ | existential quantification |

# Empty heap and pure facts

Definition:

$$\ulcorner\,\urcorner \quad \equiv \quad \lambda m.\ m = \varnothing$$
$$\ulcorner P \urcorner \quad \equiv \quad \lambda m.\ m = \varnothing \ \wedge\ P$$

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

Before: $\ulcorner\,\urcorner$
After: $\ulcorner a = 3 \ \wedge\ b = 4 \urcorner$

# Empty heap and pure facts

Definition:

$$\begin{array}{lcl} \ulcorner\,\urcorner & \equiv & \lambda m.\ m = \varnothing \\ \ulcorner P \urcorner & \equiv & \lambda m.\ m = \varnothing \ \wedge \ P \end{array}$$

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

Before: $\ulcorner\,\urcorner$

After: $\ulcorner a = 3 \ \wedge \ b = 4 \urcorner$

Observe that $\ulcorner\,\urcorner$ is equivalent to $\ulcorner$True$\urcorner$.

# Singleton heap

Definition:

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

Example: specification of "`let r = ref 3`".

Before: $\ulcorner\,\urcorner$

After: $r \mapsto 3$

# Singleton heap

Definition:

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

Example: specification of "`let r = ref 3`".

Before: $\ulcorner\,\urcorner$

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:

$$\begin{array}{lcll} m_1 \perp m_2 & \equiv & \mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \varnothing & \\ m_1 \uplus m_2 & \equiv & m_1 \cup m_2 & \text{when } m_1 \perp m_2 \end{array}$$

# Heaps and heap predicates

**Exercise:** give heaps satisfying the following heap predicates

$$\ulcorner\ \urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

# Heaps and heap predicates

**Exercise:** give heaps satisfying the following heap predicates

$$\ulcorner\;\urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

**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

$\ulcorner\ \urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$

$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$

**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

$$\ulcorner\,\urcorner \qquad \ulcorner 0 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner \qquad \ulcorner 1 = 1 \urcorner * \ulcorner 0 = 1 \urcorner \qquad 1 \mapsto 2$$

$$(1 \mapsto 2) * \ulcorner 1 = 1 \urcorner \qquad (1 \mapsto 2) * (1 \mapsto 3) \qquad (1 \mapsto 2) * (2 \mapsto 1)$$

**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:

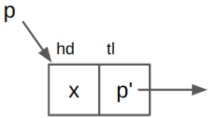
1. $(r \mapsto 5) * (s \mapsto 3) * \ulcorner t = r \urcorner$
2. $(r \mapsto 6) * (s \mapsto 3) * \ulcorner t = r \urcorner$
3. $(r \mapsto 7) * (s \mapsto 3) * \ulcorner t = r \urcorner$

# Record fields

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

$$p.\mathsf{f} \mapsto v$$

Example:




$p.\mathsf{hd} \mapsto x$

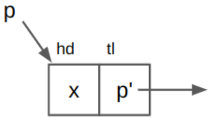
$p.\mathsf{tl} \mapsto p'$

# Record fields

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

$$p.\mathsf{f} \mapsto v$$

Example:




$$p.\mathsf{hd} \mapsto x$$

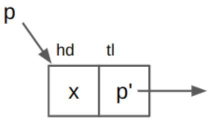
$$p.\mathsf{tl} \mapsto p'$$

In the C memory model:

$$p.\mathsf{f} \mapsto v \quad \equiv \quad (p + \mathsf{f}) \mapsto v$$

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

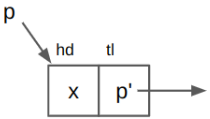
# Representation of list cells




$$p \rightsquigarrow \{\!|\, \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \,|\!\} \quad \equiv \quad p.\mathsf{hd} \mapsto x \ * \ p.\mathsf{tl} \mapsto p'$$

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

# Representation of list cells




$$p \rightsquigarrow \{\!|\, \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \,|\!\} \quad \equiv \quad p.\mathsf{hd} \mapsto x \ * \ p.\mathsf{tl} \mapsto p'$$

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

Remark: the new arrow symbol will be overloaded later.

# Existential quantification

Definition:

$$\exists x.\, H \quad \equiv \quad \lambda m.\; \exists x.\; H\, m$$

Compare:

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

$$(\exists x.\, H) \quad : \quad \texttt{heap} \to \texttt{Prop} \qquad \text{when} \quad (H : \texttt{heap} \to \texttt{Prop})$$

# Existential quantification

**Exercise:** give heaps satisfying the following heap predicates

$$\exists x.\, \ulcorner(1 \mapsto x)\urcorner \qquad \exists x.\, (1 \mapsto x) * (2 \mapsto x) \qquad \exists x.\, \ulcorner x = x + 1\urcorner$$

$$\exists x.\, (x \mapsto x + 1) * (x + 1 \mapsto x) \qquad \exists x.\, 1 \mapsto x$$

$$\exists x.\, (x \mapsto 1) * (x \mapsto 2) \qquad \exists P.\, \ulcorner P\urcorner \qquad \exists H.\, H$$

# Summary

$$\ulcorner\,\urcorner \quad \equiv \quad \ulcorner \mathsf{True} \urcorner$$

$$\ulcorner P \urcorner \quad \equiv \quad \lambda m.\ m = \varnothing \ \wedge\ P$$

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

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

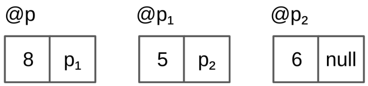
$$\exists x.\, H \quad \equiv \quad \lambda m.\ \exists x.\ H\, m$$

# Chapter 2

## Representation Predicate for Lists

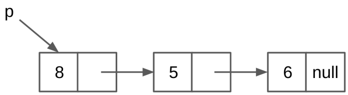
# Implementation of mutable lists

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



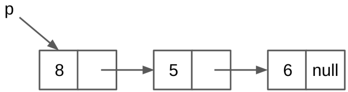

```
type 'a cell = { mutable hd : 'a;
                 mutable tl : 'a cell }

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



# Representation of mutable lists

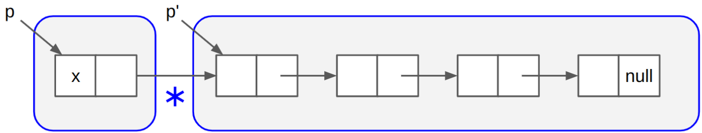
$$L \;=\; 8 :: 5 :: 6 :: \mathsf{nil}$$




$$p \leadsto \mathsf{MList}\, L \quad \equiv \quad \begin{array}{l} \quad \exists p_1.\;\; p \leadsto \{\!|\mathsf{hd}{=}8;\ \mathsf{tl}{=}p_1|\!\} \\ *\;\; \exists p_2.\;\; p_1 \leadsto \{\!|\mathsf{hd}{=}5;\ \mathsf{tl}{=}p_2|\!\} \\ *\;\; \exists p_3.\;\; p_2 \leadsto \{\!|\mathsf{hd}{=}6;\ \mathsf{tl}{=}p_3|\!\} \\ *\;\; \ulcorner p_3 = \mathsf{null} \urcorner \end{array}$$

Note: $p \leadsto \mathsf{MList}\, L$ is notation for $\mathsf{MList}\, L\, p$.

# Representation predicate




$$
\begin{aligned}
p \rightsquigarrow \mathsf{MList}\, L \quad \equiv \quad & \mathsf{match}\, L\, \mathsf{with} \\
& \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\qquad \ast\ \ p' \rightsquigarrow \mathsf{MList}\, L'
\end{aligned}
$$

# Separation properties

$$p_1 \rightsquigarrow \mathsf{MList}\, L_1 \;\;*\;\; p_2 \rightsquigarrow \mathsf{MList}\, L_2 \;\;*\;\; p_3 \rightsquigarrow \mathsf{MList}\, L_3$$

Separation enforces: no cycles, and no sharing.

# Union heap predicate

$$
\begin{array}{rcl}
p \leadsto \mathsf{MList}\, L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\
 & & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
 & & \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad\qquad\quad * \;\; p' \leadsto \mathsf{MList}\, L'
\end{array}
$$

Equivalent to:

$$
\begin{array}{rcl}
p \leadsto \mathsf{MList}\, L & \equiv & \quad \ulcorner L = \mathsf{nil} \;\wedge\; p = \mathsf{null} \urcorner \\
 & & \Cup \;\; \big( \exists x L' p'.\ \ulcorner L = x :: L' \urcorner \\
 & & \qquad\qquad * \;\; p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad * \;\; p' \leadsto \mathsf{MList}\, L' \quad \big)
\end{array}
$$

where:

$$
H_1 \Cup H_2 \quad \equiv \quad \lambda m.\ H_1\, m \;\vee\; H_2\, m
$$

# 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:

$$\ulcorner n \geqslant 0 \urcorner$$

Post-condition, where $p$ denotes the result:

$$\exists L.\ p \leadsto \mathsf{MList}\, L \ * \ \ulcorner \mathsf{length}\, L = n \ \wedge \ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v) \urcorner$$

# List construction: proof (1/2)

$\exists L.\ p \leadsto \mathsf{MList}\, L\ *\ \ulcorner \mathsf{length}\, L = n\ \wedge\ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v) \urcorner$

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

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

# List construction: proof (1/2)

$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \ * \ \ulcorner \mathsf{length}\, L = n \ \wedge \ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v) \urcorner$

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

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

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil}) \ = \ \ulcorner\, \urcorner$$

# List construction: proof (1/2)

$$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \ * \ \ulcorner \mathsf{length}\, L = n \ \wedge \ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v) \urcorner$$

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

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

$$(\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil}) \ = \ \ulcorner\, \urcorner$$

$$\begin{array}{lcl} p \rightsquigarrow \mathsf{MList}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \ \Rightarrow \ \ulcorner p = \mathsf{null} \urcorner \\ & & \mid x :: L' \ \Rightarrow \ \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad\quad * \ p' \rightsquigarrow \mathsf{MList}\, L' \end{array}$$

# List construction: proof (2/2)

$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \ * \ \ulcorner \mathsf{length}\, L = n \ \wedge \ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v) \urcorner$

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

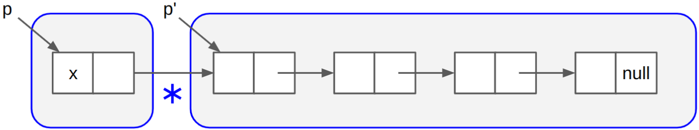
To produce $p \rightsquigarrow \mathsf{MList}\, L$, we have $p' \rightsquigarrow \mathsf{MList}\, L'$ and $p \rightsquigarrow \{\!|\, \mathsf{hd}{=}v;\ \mathsf{tl}{=}p' \,|\!\}$.

# List construction: proof (2/2)

$\exists L.\ p \rightsquigarrow \mathsf{MList}\, L \ * \ \ulcorner \mathsf{length}\, L = n \ \wedge\ (\forall i.\ 0 \leqslant i < n \Rightarrow L[i] = v)\urcorner$

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

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




$$\left(\exists p'.\ \ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \ * \ p' \rightsquigarrow \mathsf{MList}\, L'\right) \ = \ p \rightsquigarrow \mathsf{MList}\,(x :: L')$$

(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:

$$r \mapsto p_0 \;*\; s \mapsto \mathsf{null} \;*\; p_0 \rightsquigarrow \mathsf{MList}\, L$$

# In-place list reversal: invariants

Before the loop:

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

After the loop:

# In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 \ * \ s \mapsto \mathsf{null} \ * \ p_0 \leadsto \mathsf{MList}\, L$$

After the loop:

$$\exists q.\ r \mapsto \mathsf{null} \ * \ s \mapsto q \ * \ q \leadsto \mathsf{MList}\,(\mathsf{rev}\, L)$$

# In-place list reversal: invariants

Before the loop:

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

After the loop:

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

Loop invariant:

# In-place list reversal: invariants

Before the loop:

$$r \mapsto p_0 \ * \ s \mapsto \mathsf{null} \ * \ p_0 \leadsto \mathsf{MList}\, L$$

After the loop:

$$\exists q.\ r \mapsto \mathsf{null} \ * \ s \mapsto q \ * \ q \leadsto \mathsf{MList}\,(\mathsf{rev}\, L)$$

Loop invariant:

$$\begin{aligned} \exists p q L_1 L_2. \quad & r \mapsto p \ * \ p \leadsto \mathsf{MList}\, L_2 \\ * \ & s \mapsto q \ * \ q \leadsto \mathsf{MList}\, L_1 \\ * \ & \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner \end{aligned}$$

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

Invariant:

$$\exists p q L_1 L_2.\ r \mapsto p \ *\ s \mapsto q \ *\ p \rightsquigarrow \mathsf{MList}\, L_2 \ *\ q \rightsquigarrow \mathsf{MList}\, L_1 \ *\ \ulcorner L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} L_2 \urcorner$$

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

$$r \mapsto p_0 \ *\ p_0 \rightsquigarrow \mathsf{MList}\, L \ *\ s \mapsto \mathsf{null} \ *\ \mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil} \ *\ \ulcorner L = \mathsf{rev}\, \mathsf{nil} \mathbin{+\!\!+} L \urcorner$$

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

Invariant:

$$\exists p q L_1 L_2.\ r \mapsto p \ * \ s \mapsto q \ * \ p \leadsto \mathsf{MList}\, L_2 \ * \ q \leadsto \mathsf{MList}\, L_1 \ * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

$$r \mapsto p_0 \ * \ p_0 \leadsto \mathsf{MList}\, L \ * \ s \mapsto \mathsf{null} \ * \ \mathsf{null} \leadsto \mathsf{MList}\, \mathsf{nil} \ * \ \ulcorner L = \mathsf{rev}\, \mathsf{nil} +\!\!+ L \urcorner$$

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

$$r \mapsto \mathsf{null} \ * \ \mathsf{null} \leadsto \mathsf{MList}\, L_2 \ * \ s \mapsto q \ * \ q \leadsto \mathsf{MList}\, L_1 \ * \ \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

Invariant:

$$\exists p q L_1 L_2.\ r \mapsto p \ * \ s \mapsto q \\ * \ p \leadsto \mathsf{MList}\, L_2 \ * \ q \leadsto \mathsf{MList}\, L_1 \\ * \ \ulcorner L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} L_2 \urcorner$$

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

$$r \mapsto p_0 \ * \ p_0 \leadsto \mathsf{MList}\, L \ * \ s \mapsto \mathsf{null} \ * \ \mathsf{null} \leadsto \mathsf{MList}\, \mathsf{nil} \ * \ \ulcorner L = \mathsf{rev}\, \mathsf{nil} \mathbin{+\!\!+} L \urcorner$$

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

$$r \mapsto \mathsf{null} \ * \ \mathsf{null} \leadsto \mathsf{MList}\, L_2 \ * \ s \mapsto q \ * \ q \leadsto \mathsf{MList}\, L_1 \ * \ \ulcorner L = \mathsf{rev}\, L_1 \mathbin{+\!\!+} L_2 \urcorner$$

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

$$(\mathsf{null} \leadsto \mathsf{MList}\, L) \ = \ \ulcorner L = \mathsf{nil} \urcorner$$

# Conversion rule for empty lists

$$\begin{array}{lcl} p \rightsquigarrow \mathsf{MList}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\ & & \mid x :: L' \;\Rightarrow\; \exists p'.\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \;\ast\; p' \rightsquigarrow \mathsf{MList}\, L' \end{array}$$

Let us prove: $(\mathsf{null} \rightsquigarrow \mathsf{MList}\, L) \;=\; \ulcorner L = \mathsf{nil} \urcorner$

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

$$\ulcorner \mathsf{nil} = \mathsf{nil} \urcorner \;=\; \ulcorner\,\urcorner \;=\; (\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil})$$

# Conversion rule for empty lists

$$p \rightsquigarrow \mathsf{MList}\, L \;\equiv\; \begin{array}{l} \mathsf{match}\; L \;\mathsf{with} \\ \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\ \mid x :: L' \;\Rightarrow\; \exists p'.\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \;*\; p' \rightsquigarrow \mathsf{MList}\, L' \end{array}$$

Let us prove: $(\mathsf{null} \rightsquigarrow \mathsf{MList}\, L) \;=\; \ulcorner L = \mathsf{nil} \urcorner$

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

$$\ulcorner \mathsf{nil} = \mathsf{nil} \urcorner \;=\; \ulcorner\,\urcorner \;=\; (\mathsf{null} \rightsquigarrow \mathsf{MList}\, \mathsf{nil})$$

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

$$\mathsf{null} \rightsquigarrow \mathsf{MList}\,(x :: L') \;=\; \big(\exists p'.\; \mathsf{null} \rightsquigarrow \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \;*\; p' \rightsquigarrow \mathsf{MList}\, L'\big)$$

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

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

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

$$p \leadsto \mathsf{MList}\, L_2 \;*\; q \leadsto \mathsf{MList}\, L_1 \;*\; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

to

$$\exists x L_2' p'. \quad \ulcorner L_2 = x :: L_2' \urcorner \;*\; p \leadsto \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \;*\; p' \leadsto \mathsf{MList}\, L_2' \\ \;*\; q \leadsto \mathsf{MList}\, L_1 \;*\; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

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

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

$$p \rightsquigarrow \mathsf{MList}\, L_2 \;*\; q \rightsquigarrow \mathsf{MList}\, L_1 \;*\; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

to

$$\exists x L_2' p'.\quad \ulcorner L_2 = x :: L_2' \urcorner \;*\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}p'|\!\} \;*\; p' \rightsquigarrow \mathsf{MList}\, L_2'$$
$$*\; q \rightsquigarrow \mathsf{MList}\, L_1 \;*\; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ L_2 \urcorner$$

After update of $p$.tl to the value $q$:

$$p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\, \mathsf{tl}{=}q|\!\} \;*\; q \rightsquigarrow \mathsf{MList}\, L_1$$
$$*\; p' \rightsquigarrow \mathsf{MList}\, L_2' \;*\; \ulcorner L = \mathsf{rev}\, L_1 +\!\!+ (x :: L_2') \urcorner$$

to

$$q \rightsquigarrow \mathsf{MList}\,(x :: L_1) \;*\; p' \rightsquigarrow \mathsf{MList}\, L_2' \;*\; \ulcorner L = \mathsf{rev}\,(x :: L_1) +\!\!+ L_2 \urcorner$$

# Conversion rules for nonempty lists

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MList}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& & \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & \qquad\qquad\qquad\quad\; *\;\; p' \rightsquigarrow \mathsf{MList}\, L'
\end{array}
$$

$$
\begin{array}{rcll}
p \rightsquigarrow \mathsf{MList}\, L * \ulcorner p \neq \mathsf{null} \urcorner & = & \exists x L' p'. & \ulcorner L = x :: L' \urcorner \\
& & & *\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & & *\;\; p' \rightsquigarrow \mathsf{MList}\, L'
\end{array}
$$

# Summary

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MList}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& & \mid x :: L' \;\Rightarrow\; \exists p'. \quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& & \qquad\qquad\qquad\quad * \;\; p' \rightsquigarrow \mathsf{MList}\, L'
\end{array}
$$

# 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

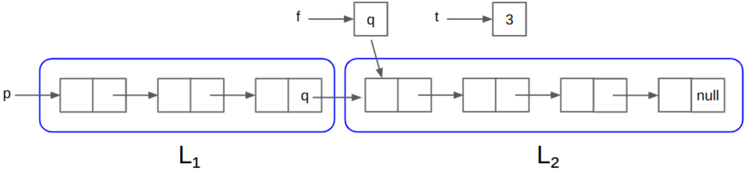
Before the loop:

$$(p \rightsquigarrow \mathsf{MList}\, L) \;*\; (f \mapsto p) \;*\; (t \mapsto 0)$$

After the loop:

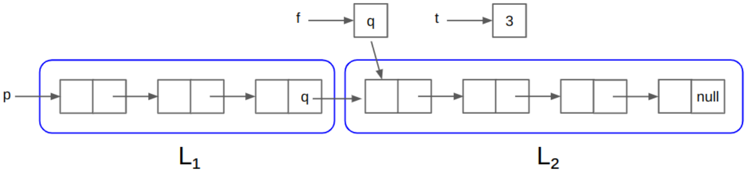
$$(p \rightsquigarrow \mathsf{MList}\, L) \;*\; (f \mapsto \mathsf{null}) \;*\; (t \mapsto \mathsf{length}\, L)$$

# Mlength: loop invariant




Loop invariant:

# Mlength: loop invariant

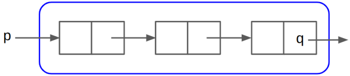



Loop invariant:

$$\exists L_1 L_2 q. \quad \ulcorner L = L_1 +\!\!+ L_2 \urcorner \ast (t \mapsto \mathsf{length}\, L_1) \ast (f \mapsto q)$$
$$\ast \ (p \rightsquigarrow \mathsf{MlistSeg}\, q\, L_1) \ast (q \rightsquigarrow \mathsf{MList}\, L_2)$$

# Representation predicate for list segments

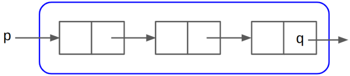
$$\begin{array}{rcl} p \leadsto \mathsf{MList}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\ & & \mid x :: L' \;\Rightarrow\; \exists p'.\; p \leadsto \{\!| \mathsf{hd}{=}x;\, \mathsf{tl}{=}p' |\!\} \\ & & \qquad\qquad\qquad * \; p' \leadsto \mathsf{MList}\, L' \end{array}$$




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

# Representation predicate for list segments

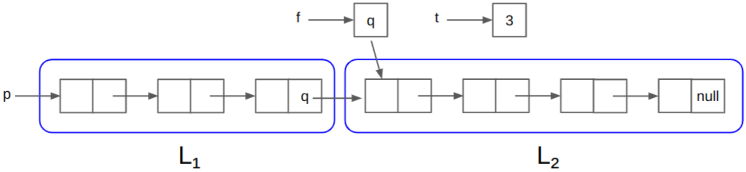
$$p \rightsquigarrow \mathsf{MList}\, L \quad \equiv \quad \mathsf{match}\, L\, \mathsf{with} \;\; | \; \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \;\; | \; x :: L' \Rightarrow \exists p'.\; p \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} \;*\; p' \rightsquigarrow \mathsf{MList}\, L'$$




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

$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, L \quad \equiv \quad \mathsf{match}\, L\, \mathsf{with} \;\; | \; \mathsf{nil} \Rightarrow \ulcorner p = q \urcorner \;\; | \; x :: L' \Rightarrow \exists p'.\; p \rightsquigarrow \{|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\} \;*\; p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L'$$

# Mlength: proof

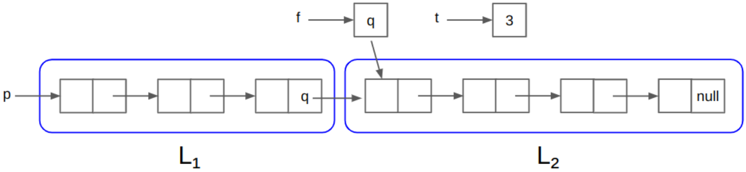



Enter:

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

$$\ulcorner\,\urcorner = (p \rightsquigarrow \mathsf{MlistSeg}\, p\, \mathsf{nil})$$

# Mlength: proof




Enter:

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

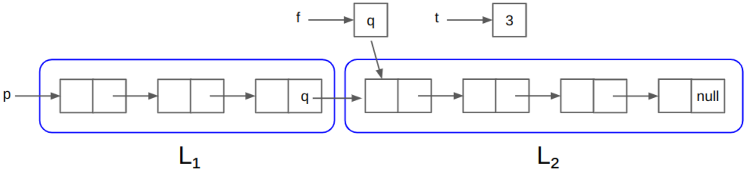
$$\ulcorner\urcorner = (p \rightsquigarrow \mathsf{MlistSeg}\, p\, \mathsf{nil})$$

Exit:

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

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

# Mlength: proof




Enter:

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

$$\ulcorner\,\urcorner \;=\; (p \leadsto \mathsf{MlistSeg}\, p\, \mathsf{nil})$$

Exit:

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

$$(p \leadsto \mathsf{MlistSeg}\, \mathsf{null}\, L) \;=\; (p \leadsto \mathsf{MList}\, L)$$

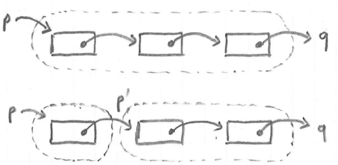
Step:

$$L_2 = x :: L_2' \;\wedge\; q \neq \mathsf{null} \;\wedge\; q.\mathsf{tl} = q'$$

$$\exists q.\; p \leadsto \mathsf{MlistSeg}\, q\, L_1 \;*\; q \leadsto \{\!|\mathsf{hd}{=}x;\; \mathsf{tl}{=}q'|\!\}$$

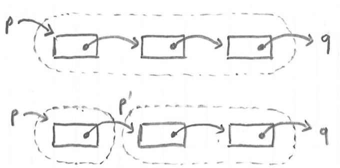
$$=\;\; p \leadsto \mathsf{MlistSeg}\, q'\, (L_1 +\!\!+\, x :: \mathsf{nil})$$

# Splitting rules for list segments

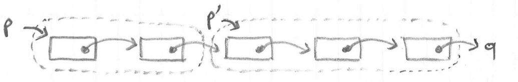



$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, (x :: L') \ = \ \exists p'.\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \ * \ p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L'$$

# Splitting rules for list segments

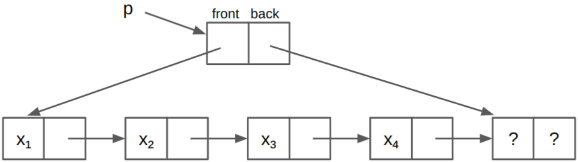



$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, (x :: L') \;=\; \exists p'.\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \;*\; p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L'$$




$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, (L_1 +\!\!+ L_2) \quad = \quad \exists p'. \quad p \rightsquigarrow \mathsf{MlistSeg}\, p'\, L_1 \;*\; p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L_2$$

# An implementation of mutable queues

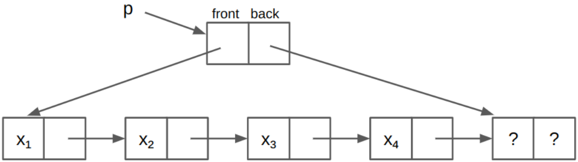



Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked "?" above)

```
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



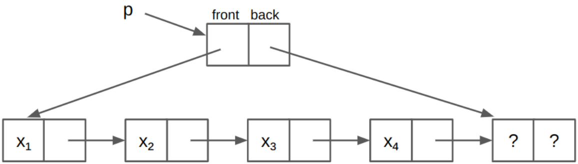

Represent a queue as a list segment, with the last cell storing no item (in fact, storing unknown values, marked "?" above)

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

**Exercise:** define the representation predicate $p \rightsquigarrow \mathsf{Queue}\, L$.

$$p \rightsquigarrow \mathsf{Queue}\, L \;\equiv\; \exists f b.\quad \begin{array}{l} p \rightsquigarrow \{\!|\mathsf{front}{=}f;\ \mathsf{back}{=}b|\!\} \\ \ast\ f \rightsquigarrow \mathsf{MlistSeg}\, b\, L \\ \ast\ (b.\mathsf{hd} \mapsto -)\ \ast\ (b.\mathsf{tl} \mapsto -) \end{array}$$

# 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 $p \rightsquigarrow \text{Queue}\, L$.

$$\begin{array}{rcll} p \rightsquigarrow \text{Queue}\, L & \equiv & \exists f b. & p \rightsquigarrow \{\!| \text{front}{=}f;\ \text{back}{=}b |\!\} \\ & & * & f \rightsquigarrow \text{MlistSeg}\, b\, L \\ & & * & (b.\text{hd} \mapsto -) \ * \ (b.\text{tl} \mapsto -) \end{array}$$

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

# Summary

$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, L \quad \equiv \quad \begin{array}{l} \mathsf{match}\, L\, \mathsf{with} \\ \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = q \urcorner \\ \mid x :: L' \;\Rightarrow\; \exists p'.\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ \qquad\qquad\qquad\qquad\quad \ast\ p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L' \end{array}$$

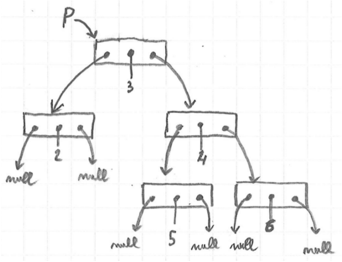
Split and merge of segments:

$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, (L_1 +\!\!+ L_2) \quad = \quad \exists p'. \quad \begin{array}{l} p \rightsquigarrow \mathsf{MlistSeg}\, p'\, L_1 \\ \ast \quad p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L_2 \end{array}$$

# Chapter 4

## Representation Predicate for Trees

# Implementation of a mutable binary trees




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

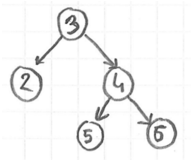
```
type node = {
  mutable item : int;
  mutable left : node;
  mutable right : node; }
```

# Logical binary trees

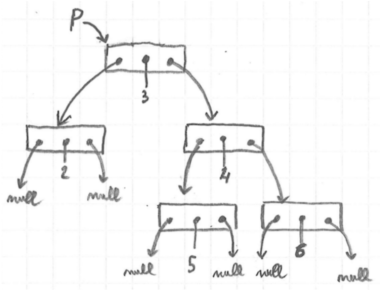
```
Inductive tree : Type :=
  | Leaf : tree
  | Node : int → tree → tree → tree.
```

Example:

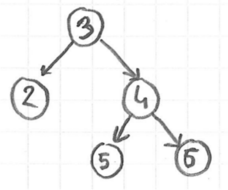
```
Node 3
    (Node 2 Leaf Leaf)
    (Node 4 (Node 5 Leaf Leaf)
            (Node 6 Leaf Leaf))
```



# Representation predicate for binary trees




$T =$




Representation predicate:

$$p \rightsquigarrow \mathsf{Mtree}\, T$$

# Representation predicate for binary trees

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MList}\,L & \equiv & \mathsf{match}\,L\,\mathsf{with} \\
 & & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
 & & \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad\qquad\quad\; \ast\;\; p' \rightsquigarrow \mathsf{MList}\,L'
\end{array}
$$

**Exercise:** define $p \rightsquigarrow \mathsf{Mtree}\,T$.

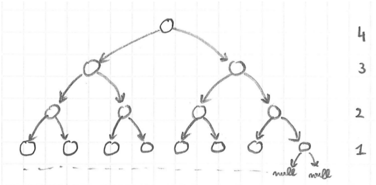
# Representation predicate for binary trees

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{MList}\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
 & & \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
 & & \mid x :: L' \;\Rightarrow\; \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
 & & \qquad\qquad\qquad\quad \ast\;\; p' \rightsquigarrow \mathsf{MList}\, L'
\end{array}
$$

**Exercise:** define $p \rightsquigarrow \mathsf{Mtree}\, T$.

$$
\begin{array}{rcl}
p \rightsquigarrow \mathsf{Mtree}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
 & & \mid \mathsf{Leaf} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
 & & \mid \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; \exists p_1 p_2. \\
 & & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
 & & \qquad\quad \ast\;\; p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \\
 & & \qquad\quad \ast\;\; p_2 \rightsquigarrow \mathsf{Mtree}\, T_2
\end{array}
$$

# Complete binary tree




$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T$$

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

# Complete binary tree (1/2)

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mtree}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \mid \mathsf{Leaf} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& & \mid \mathsf{Node}\, x\, T_1\, T_2 \;\Rightarrow\; \exists p_1 p_2. \\
& & \qquad\qquad p \mapsto \{\!| \mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2 |\!\} \\
& & \qquad\quad\; \ast\;\; p_1 \leadsto \mathsf{Mtree}\, T_1 \\
& & \qquad\quad\; \ast\;\; p_2 \leadsto \mathsf{Mtree}\, T_2
\end{array}
$$

**Exercise:** define $p \leadsto \mathsf{MtreeDepth}\, n\, T$ by modifying $p \leadsto \mathsf{Mtree}\, T$.

# Complete binary tree (1/2), solution

$$
\begin{array}{lcl}
p \leadsto \mathsf{MtreeDepth}\, n\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \mid \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \wedge n = 0 \urcorner \\
& & \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2.\ \ulcorner n > 0 \urcorner \ * \\
& & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad\qquad *\ p_1 \leadsto \mathsf{MtreeDepth}\,(n-1)\, T_1 \\
& & \qquad\qquad *\ p_2 \leadsto \mathsf{MtreeDepth}\,(n-1)\, T_2
\end{array}
$$

# Complete binary tree (1/2), solution

$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T & \equiv & \mathsf{match}\ T\ \mathsf{with} \\
& & \mid \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \ \wedge\ n = 0 \urcorner \\
& & \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2.\ \ulcorner n > 0 \urcorner \, * \\
& & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad\qquad *\ p_1 \rightsquigarrow \mathsf{MtreeDepth}\,(n-1)\,T_1 \\
& & \qquad\qquad *\ p_2 \rightsquigarrow \mathsf{MtreeDepth}\,(n-1)\,T_2
\end{array}
$$

Or:

$$
\begin{array}{lcl}
p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T & \equiv & \mathsf{match}\ n, T\ \mathsf{with} \\
& & \mid O,\ \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid S\, m,\ \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2. \\
& & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad\quad *\ p_1 \rightsquigarrow \mathsf{MtreeDepth}\, m\, T_1 \\
& & \qquad\quad *\ p_2 \rightsquigarrow \mathsf{MtreeDepth}\, m\, T_2 \\
& & \mid \_, \_ \Rightarrow \ulcorner \mathsf{False} \urcorner
\end{array}
$$

# Complete binary tree (2/2)

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

# Complete binary tree (2/2)

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

$$p \leadsto \mathsf{MtreeDepth}\, n\, T \;\equiv\; p \leadsto \mathsf{Mtree}\, T \;*\; \ulcorner \mathsf{depth}\, n\, T \urcorner$$

```
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

$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\; \equiv \;\; (p \rightsquigarrow \mathsf{Mtree}\, T) \; * \; \ulcorner \mathsf{depth}\, n\, T \urcorner$$

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

# Complete binary tree of unspecified depth

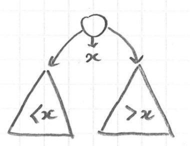
$$p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T \;\;\equiv\;\; (p \rightsquigarrow \mathsf{Mtree}\, T) \;\ast\; \ulcorner \mathsf{depth}\, n\, T \urcorner$$

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

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

1. $\exists n.\; p \rightsquigarrow \mathsf{MtreeDepth}\, n\, T$
2. $\exists n.\; (p \rightsquigarrow \mathsf{Mtree}\, T) \;\ast\; \ulcorner \mathsf{depth}\, n\, T \urcorner$
3. $(p \rightsquigarrow \mathsf{Mtree}\, T) \;\ast\; \ulcorner \exists n.\, \mathsf{depth}\, n\, T \urcorner$

# Binary search tree property



The proposition 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 $p \rightsquigarrow \mathsf{MsearchTree}\, E$ for describing a mutable binary search tree storing the set of elements $E$.

# Binary search tree predicate

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

$$p \leadsto \mathsf{MsearchTree}\, E \;\; \equiv \;\; \exists T.\; p \leadsto \mathsf{Mtree}\, T \; * \; \ulcorner \mathsf{search}\, T\, E \urcorner$$

# Binary search tree predicate

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

$$p \rightsquigarrow \mathsf{MsearchTree}\, E \;\; \equiv \;\; \exists T.\; p \rightsquigarrow \mathsf{Mtree}\, T \;\ast\; \ulcorner \mathsf{search}\, T\, E \urcorner$$

For example, a call "`add x p`" can be specified as follows:

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

# Summary

Common representation predicate for all binary trees:

$$
\begin{array}{rcl}
p \leadsto \mathsf{Mtree}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \\
& & \mid \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& & \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2. \\
& & \qquad\qquad p \mapsto \{\!|\mathsf{item}{=}x;\ \mathsf{left}{=}p_1;\ \mathsf{right}{=}p_2|\!\} \\
& & \qquad\quad * \ p_1 \leadsto \mathsf{Mtree}\, T_1 \ * \ p_2 \leadsto \mathsf{Mtree}\, T_2
\end{array}
$$

Invariants are expressed on the pure trees:

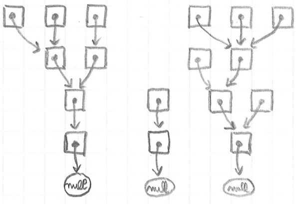
$$
p \leadsto \mathsf{MsearchTree}\, E \ \equiv\ \exists T.\ p \leadsto \mathsf{Mtree}\, T \ * \ \ulcorner \mathsf{search}\, T\, E \urcorner
$$

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

# Chapter 5

## Structures with sharing

# The union-find data structure



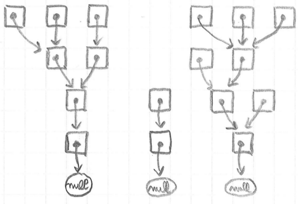

```
type node = node ref
```

Implements an equivalence relation $S$ of type: loc $\rightarrow$ loc $\rightarrow$ `Prop`.

$$S\,a\,b \;\Leftrightarrow\; a \text{ and } b \text{ are two valid nodes with the same root}$$

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

# Representation of union-find cells

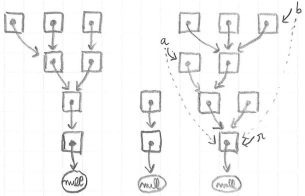



$$(p_1 \mapsto q_1) \;*\; (p_2 \mapsto q_2) \;*\; \ldots \;*\; (p_n \mapsto q_n)$$

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

where $G$ is a finite map from locations to locations.

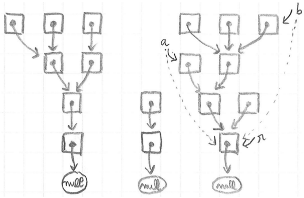
# Invariants of union-find



Predicate "root $G\,a\,r$" asserts that in the graph $G$, node $a$ has root $r$.

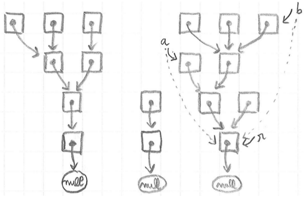
```
Inductive root : fmap loc loc → loc → loc → Prop :=
  | root_init : ∀G x,
      binds G x null →
      root G x x
  | root_step : ∀G x y r,
      binds G x y →
      y ≠null →
      root G y r →
      root G x r.
```

# Specification of the union-find structure




$$\mathsf{UnionFind}\,S \quad \equiv \quad \exists G. \quad \left(\circledast_{(p,q)\in G}\ p \mapsto q\right)$$
$$* \ \ulcorner \forall a \in \mathsf{dom}\,G.\ \exists r.\ \mathsf{root}\,G\,a\,r \urcorner$$
$$* \ \ulcorner \forall ab.\ S\,a\,b \Leftrightarrow \exists r.\ \mathsf{root}\,G\,a\,r \wedge \mathsf{root}\,G\,b\,r \urcorner$$

# Specification of the union-find structure




$$\mathsf{UnionFind}\, S \quad \equiv \quad \exists G. \quad \Big(\circledast_{(p,q)\in G}\; p \mapsto q\Big)$$
$$* \;\ulcorner \forall a \in \mathsf{dom}\, G.\; \exists r.\; \mathsf{root}\, G\, a\, r \urcorner$$
$$* \;\ulcorner \forall ab.\; S\, a\, b \;\Leftrightarrow\; \exists r.\; \mathsf{root}\, G\, a\, r \wedge \mathsf{root}\, G\, b\, r \urcorner$$

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

- pre-condition: $\ulcorner S\, a\, a \;\wedge\; S\, b\, b \urcorner \;*\; \mathsf{UnionFind}\, S$
- post-condition: $\ulcorner x = \mathsf{true} \Leftrightarrow S\, a\, b \urcorner \;*\; \mathsf{UnionFind}\, S$

# Summary

Iterated separating conjunction, written $\circledast$.

For Union-Find:

$$\underset{(p,q)\in G}{\circledast} \; p \mapsto q$$

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

# Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

# Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Example 2:

$$\{\ulcorner\urcorner\}\ (\texttt{3})\ \{\lambda x.\ulcorner x = 3\urcorner\}$$

# Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Example 2:

$$\{\ulcorner\urcorner\}\ (\texttt{3})\ \{\lambda x. \ulcorner x = 3 \urcorner\}$$

Example 3:

$$\{r \mapsto 3\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = 3 \urcorner * (r \mapsto 3)\}$$

# Examples of triples

Example 1:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ r \mapsto 3\}$$

Example 2:

$$\{\ulcorner\urcorner\}\ (\texttt{3})\ \{\lambda x. \ulcorner x = 3\urcorner\}$$

Example 3:

$$\{r \mapsto 3\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = 3\urcorner * (r \mapsto 3)\}$$

Example 4:

$$\{r \mapsto 3\}\ (\texttt{incr r})\ \{\lambda\_. (r \mapsto 4)\}$$

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:

$$\forall rn. \quad \{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto (n+1)\}$$

# Specification of operations on memory cells

**Exercise:** specify the primitive operations on references.

$$\big(\texttt{ref v}\big)$$

$$\big(\texttt{!r}\big)$$

$$\big(\texttt{r := v}\big)$$

# Specification of operations on memory cells

**Exercise:** specify the primitive operations on references.

$$(\texttt{ref v})$$

$$(\texttt{!r})$$

$$(\texttt{r := v})$$

Solution:

$$\forall v. \quad \{\ulcorner\urcorner\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\}$$

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

# Specification of operations on memory cells

**Exercise:** specify the primitive operations on references.

$$(\texttt{ref v})$$
$$(\texttt{!r})$$
$$(\texttt{r := v})$$

Solution:

$$\forall v. \quad \{\ulcorner\urcorner\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\}$$
$$\forall rv. \quad \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = v \urcorner * (r \mapsto v)\}$$

$$\forall rvw. \quad \{r \mapsto w\}\ (\texttt{r := v})\ \{\lambda\_.\,(r \mapsto v)\}$$
$$\forall rv. \quad \{\exists w.\ r \mapsto w\}\ (\texttt{r := v})\ \{\lambda\_.\,(r \mapsto v)\}$$
$$\forall rv. \quad \{r \mapsto -\}\ (\texttt{r := v})\ \{\lambda\_.\,(r \mapsto v)\}$$

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

# Specification of partial functions

Presentation 1:

$$\forall n. \quad \{\ulcorner n \geqslant 0 \urcorner\}\ (\mathsf{fact}\, n)\ \{\lambda x. \ulcorner x = n! \urcorner\}$$

Presentation 2:

$$\forall n.\ n \geqslant 0 \Rightarrow \{\ulcorner\,\urcorner\}\ (\mathsf{fact}\, n)\ \{\lambda x. \ulcorner x = n! \urcorner\}$$

# 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:

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

(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:

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

(Assuming a deterministic language)

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

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

# 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:

$$m_1 \perp m_2 \;\equiv\; (\mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \varnothing)$$

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:

$$m_1 \perp m_2 \;\equiv\; (\mathsf{dom}\, m_1 \cap \mathsf{dom}\, m_2 = \varnothing)$$

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

$$\forall m_1\, m_2.\; \begin{cases} H\, m_1 \\ m_1 \perp m_2 \end{cases} \;\Rightarrow\; \exists v.\, \exists m_1'.\; \begin{cases} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m_1' \uplus m_2 \rangle \\ Q\, v\, m_1' \\ m_1' \perp m_2 \end{cases}$$

# 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:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x \ast \exists s.\, s \mapsto r\}$$

To:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x\}$$

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:

$$\forall m_1\, m_2.\ \begin{cases} H\, m_1 \\ m_1 \perp m_2 \end{cases} \Rightarrow\ \exists v m_1' m_3.\ \begin{cases} \langle t, m_1 \uplus m_2 \rangle \Downarrow \langle v, m_1' \uplus m_2 \uplus m_3 \rangle \\ Q\, v\, m_1' \\ m_1' \perp m_2 \perp m_3 \end{cases}$$

# Interpretation of triples (3/3), revisited

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

$$\mathsf{GC} \;\equiv\; \exists H.\, H$$

# Interpretation of triples (3/3), revisited

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

$$\mathsf{GC} \;\equiv\; \exists H.\, H$$

**Definition (Separation Logic Triple)**

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

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

# Summary

Separation Logic triple:

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

Specification of a function:

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

Specification of primitive functions:

$$\begin{array}{ll} \forall v. & \{\ulcorner\urcorner\}\ (\texttt{ref v})\ \{\lambda r.\ (r \mapsto v)\} \\ \forall r v. & \{r \mapsto v\}\ (\texttt{!r})\ \{\lambda x.\ \ulcorner x = v \urcorner * (r \mapsto v)\} \\ \forall r v. & \{r \mapsto -\}\ (\texttt{r := v})\ \{\lambda \_.\, (r \mapsto v)\} \end{array}$$

Interpretation of triples: see definition.

# Chapter 7

## The Frame Rule

# Preservation of independent state

We have:

$$\{r \mapsto 2\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2 \ast s \mapsto 7\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3 \ast s \mapsto 7\}$$

# Preservation of independent state

We have:

$$\{r \mapsto 2\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3\}$$

We also have:

$$\{r \mapsto 2\ *\ s \mapsto 7\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3\ *\ s \mapsto 7\}$$

More generally:

$$\{r \mapsto 2\ *\ H\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto 3\ *\ H\}$$

# 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:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ (r \mapsto 3)\}$$

By the frame rule, we have:

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

# Frame rule and allocation

We have:

$$\{\ulcorner\urcorner\}\ (\texttt{ref 3})\ \{\lambda r.\ (r \mapsto 3)\}$$

By the frame rule, we have:

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

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:

$\forall pL.\{p \rightsquigarrow \mathsf{MList}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ \ulcorner n = \mathsf{length}\, L \urcorner\ *\ p \rightsquigarrow \mathsf{MList}\, L\}$

# 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:

$$\forall pL.\{p \rightsquigarrow \mathsf{MList}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ \ulcorner n = \mathsf{length}\, L \urcorner\ *\ p \rightsquigarrow \mathsf{MList}\, L\}$$

We prove this specification by induction on $L$.

# Verification of mlength: nil case

**Case** $L = \textbf{nil}$. Then $p = \mathsf{null}$. Goal is:

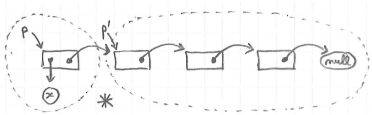
$$\{p \rightsquigarrow \mathsf{MList}\,\mathsf{nil}\}\ (\mathsf{0})\ \{\lambda n.\ \ulcorner n = \mathsf{length}\,\mathsf{nil} \urcorner\ *\ p \rightsquigarrow \mathsf{MList}\,\mathsf{nil}\}$$

Same as:

$$\{\ulcorner p = \mathsf{null} \urcorner\}\ (\mathsf{0})\ \{\lambda n.\ \ulcorner n = 0 \urcorner\ *\ \ulcorner p = \mathsf{null} \urcorner\}$$

# Verification of mlength: using the frame rule

$\forall L p.\quad \{p \leadsto \mathsf{MList}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ \ulcorner n = \mathsf{length}\, L \urcorner \ * \ p \leadsto \mathsf{MList}\, L\}$




Assume $L = x :: L'$.

| | |
|---|---|
| $p \leadsto \mathsf{MList}\, L$ | pre-condition |
| $\underline{p \leadsto \{\!\vert \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \vert\!\}}\ *\ p' \leadsto \mathsf{MList}\, L'$ | by unfolding |
| $p \leadsto \{\!\vert \mathsf{hd}{=}x;\ \mathsf{tl}{=}p' \vert\!\}\ *\ p' \leadsto \mathsf{MList}\, L'\ *\ \ulcorner n' = \vert L' \vert \urcorner$ | frame+induction |
| $p \leadsto \mathsf{MList}\, L\ *\ \ulcorner n' + 1 = \vert x :: L' \vert \urcorner$ | by folding |
| $p \leadsto \mathsf{MList}\, L\ *\ \ulcorner n = \vert L \vert \urcorner$ | post-condition |

# Instantiation of the frame rule

Induction hypothesis:

$$\{p' \rightsquigarrow \mathsf{MList}\, L'\}$$
`(mlength p')`
$$\{\lambda n'.\ \ulcorner n = \mathsf{length}\, L' \urcorner \ * \ p' \rightsquigarrow \mathsf{MList}\, L'\}$$

By the frame rule:

$$\{p' \rightsquigarrow \mathsf{MList}\, L' \ * \ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}$$
`(mlength p')`
$$\{\lambda n.\ \ulcorner n = \mathsf{length}\, L' \urcorner \ * \ p' \rightsquigarrow \mathsf{MList}\, L' \ * \ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}$$

# Representation predicate for arrays

Representation predicate for C arrays:

$$p \rightsquigarrow \mathsf{Array}\, L \quad \equiv \quad \mathop{\circledast}_{v \text{ at index } i \text{ in } L} p[i] \mapsto v$$

where:

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

# Representation predicate for arrays

Representation predicate for C arrays:

$$p \rightsquigarrow \mathsf{Array}\, L \quad \equiv \quad \underset{v \text{ at index } i \text{ in } L}{\circledast} \quad p[i] \mapsto v$$

where:

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

Representation predicate for ML arrays:

$$p \rightsquigarrow \mathsf{Array}\, L \quad \equiv \quad p.\mathsf{length} \mapsto |L| \; * \underset{v \text{ at index } i \text{ in } L}{\circledast} \quad p[i] \mapsto v$$

where $p.\mathsf{length} \mapsto n$ 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` (...) |

Common small footprint specification for accessing a memory cell:

$$\{p \mapsto -\}\ (\texttt{*p = v})\ \{\lambda\_.\ p \mapsto v\}$$
$$\{p \mapsto v\}\ (\texttt{*p})\qquad \{\lambda x.\ \ulcorner x = v \urcorner \ast p \mapsto v\}$$

All other specifications for read and write operations are derivable.

# Chapter 9

## Heap entailment

# Definition of Hprop

Let:

$$\text{Hprop} \equiv \texttt{heap} \to \texttt{Prop}$$

For example:

$$\begin{aligned} \ulcorner\urcorner &: \text{Hprop} \\ l \mapsto v &: \text{Hprop} \\ H_1 * H_2 &: \text{Hprop} \end{aligned}$$

In particular:

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

# 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: | $H * \ulcorner\urcorner$ | $=$ | $H$ | |
| Extrusion of existentials: | $(\exists x.\, H_1) * H_2$ | $=$ | $\exists x.\, (H_1 * H_2)$ | (if $x \notin H_2$) |
| Extrusion of pure facts: | $(\ulcorner P \urcorner * H)\, m$ | $=$ | $P \wedge (H\, m)$ | |

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

# Heap entailment

Definition:

$$H_1 \rhd H_2 \quad \equiv \quad \forall m.\, H_1\, m \;\Rightarrow\; H_2\, m$$

For example:

$$(r \mapsto 6) \;\rhd\; \exists n.\, (r \mapsto n) * \ulcorner \text{even}\, n \urcorner$$

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

$\rhd$ is a preorder:

$$\textsc{Reflexivity}\;\; \frac{}{H \rhd H} \qquad \textsc{Transitivity}\;\; \frac{H_1 \rhd H_2 \qquad H_2 \rhd H_3}{H_1 \rhd H_3} \qquad \textsc{Antisymmetry}\;\; \frac{H_1 \rhd H_2 \qquad H_2 \rhd H_1}{H_1 = H_2}$$

# Frame property for heap entailment

$$\frac{H_1 \quad \rhd H_1'}{H_1 * H_2 \rhd H_1' * H_2} \text{ ENTAIL-FRAME}$$

For example, to prove:

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

it suffices to prove:

$$(s \mapsto 3) \;\; \rhd \;\; (t \mapsto n).$$

# Heap implications: true or false?

1. $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$
2. $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$
3. $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$
4. $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$
5. $\ulcorner \mathsf{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$
6. $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$
7. $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$
8. $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$

# Heap implications: true or false?

| | | |
|---|---|---|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$ | |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$ | |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$ | |
| 5. | $\ulcorner \mathsf{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$ | |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |

# Heap implications: true or false?

1. $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$ — true
2. $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$ — false
3. $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$
4. $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$
5. $\ulcorner \text{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$
6. $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \text{False} \urcorner$
7. $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \text{False} \urcorner$
8. $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \text{False} \urcorner$

# Heap implications: true or false?

| | | |
|---|---|---|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$ | |
| 5. | $\ulcorner \mathsf{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$ | |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |

# Heap implications: true or false?

| | | |
|---|---|---|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$ | false* |
| 5. | $\ulcorner \mathsf{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$ | |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |

# Heap implications: true or false?

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

# Heap implications: true or false?

| | | |
|---|---|---|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$ | false* |
| 5. | $\ulcorner \mathsf{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$ | true |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | false |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | |

# Heap implications: true or false?

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

# Heap implications: true or false?

| | | |
|---|---|---|
| 1. | $(r \mapsto 3) * (s \mapsto 4) \rhd (s \mapsto 4) * (r \mapsto 3)$ | true |
| 2. | $(r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 3)$ | false |
| 3. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 4)$ | false |
| 4. | $(s \mapsto 4) * (r \mapsto 3) \rhd (r \mapsto 3)$ | false* |
| 5. | $\ulcorner \mathsf{False} \urcorner * (r \mapsto 3) \rhd (s \mapsto 4) * (r \mapsto 4)$ | true |
| 6. | $(r \mapsto 4) * (s \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | false |
| 7. | $(r \mapsto 4) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | true |
| 8. | $(r \mapsto 3) * (r \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$ | true |

# Instantiation of existentials and propositions

$$(r \mapsto 6) \rhd (\exists n.\, (r \mapsto n) * \ulcorner \text{even}\, n \urcorner)$$

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

Rules:

$$\frac{H_1 \rhd ([x \to v]\, H_2)}{H_1 \rhd (\exists x.\, H_2)}\ \textsc{exists-r} \qquad\qquad \frac{(H_1 \rhd H_2) \qquad P}{H_1 \rhd (H_2 * \ulcorner P \urcorner)}\ \textsc{prop-r}$$

# Instantiation of existentials and propositions

$$(r \mapsto 6) \;\rhd\; (\exists n.\, (r \mapsto n) * \ulcorner \mathsf{even}\, n \urcorner)$$

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

Rules:

$$\frac{H_1 \rhd ([x \to v]\, H_2)}{H_1 \rhd (\exists x.\, H_2)}\ \textsc{exists-r} \qquad\qquad \frac{(H_1 \rhd H_2) \qquad P}{H_1 \rhd (H_2 * \ulcorner P \urcorner)}\ \textsc{prop-r}$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(r \mapsto 6) \;\rhd\; (r \mapsto 6)}\ \textsc{refl} \qquad \dfrac{}{\mathsf{even}\ 6}\ \textsc{math}}{(r \mapsto 6) \;\rhd\; (r \mapsto 6) * \ulcorner \mathsf{even}\, 6 \urcorner}\ \textsc{prop-r}}{(r \mapsto 6) \;\rhd\; [n \to 6]\, ((r \mapsto n) * \ulcorner \mathsf{even}\, n \urcorner)}\ \textsc{subst}}{(r \mapsto 6) \;\rhd\; \exists n.\, (r \mapsto n) * \ulcorner \mathsf{even}\, n \urcorner}\ \textsc{exists-r}$$

# Extraction of existentials and propositions

$$(\exists n.\, \ulcorner \mathsf{even}\, n \urcorner * (r \mapsto n)) \vartriangleright (\exists m.\, \ulcorner \mathsf{even}\, m \urcorner * (r \mapsto m + 2))$$

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

$$(r \mapsto n) \vartriangleright \exists m.\, \ulcorner \mathsf{even}\, m \urcorner * (r \mapsto m + 2)$$

# Extraction of existentials and propositions

$$(\exists n.\, \ulcorner \textsf{even}\, n \urcorner * (r \mapsto n)) \;\rhd\; (\exists m.\, \ulcorner \textsf{even}\, m \urcorner * (r \mapsto m + 2))$$

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

$$(r \mapsto n) \;\rhd\; \exists m.\, \ulcorner \textsf{even}\, m \urcorner * (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{x \notin H_2 \qquad \forall x.\ (H_1 \rhd H_2)}{(\exists x.\, H_1) \rhd H_2}\ \textsc{exists-l} \qquad\qquad \frac{P \;\Rightarrow\; (H_1 \rhd H_2)}{(\ulcorner P \urcorner * H_1) \rhd H_2}\ \textsc{prop-l}$$

# Extraction of existentials and propositions

$$(\exists n.\, \ulcorner \text{even}\, n \urcorner * (r \mapsto n)) \;\rhd\; (\exists m.\, \ulcorner \text{even}\, m \urcorner * (r \mapsto m + 2))$$

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

$$(r \mapsto n) \;\rhd\; \exists m.\, \ulcorner \text{even}\, m \urcorner * (r \mapsto m + 2)$$

Reasoning rules:

$$\frac{x \notin H_2 \qquad \forall x.\;(H_1 \rhd H_2)}{(\exists x.\, H_1) \rhd H_2}\ \textsc{exists-l} \qquad\qquad \frac{P \;\Rightarrow\; (H_1 \rhd H_2)}{(\ulcorner P \urcorner * H_1) \rhd H_2}\ \textsc{prop-l}$$

Same with explicit proof contexts:

$$\frac{x \notin H_2 \qquad \Gamma,\ x : A \;\vdash\; H_1 \rhd H_2}{\Gamma \;\vdash\; (\exists (x : A).\, H_1) \rhd H_2} \qquad\qquad \frac{\Gamma,\ h : P \;\vdash\; H_1 \rhd H_2}{\Gamma \;\vdash\; (\ulcorner P \urcorner * H_1) \rhd H_2}$$

# Heap implications: true or false?

1. $(r \mapsto 3) \rhd \exists n. (r \mapsto n)$
2. $\exists n. (r \mapsto n) \rhd (r \mapsto 3)$
3. $\exists n. (r \mapsto n) * \ulcorner n > 0 \urcorner \rhd \exists n. \ulcorner n > 1 \urcorner * (r \mapsto (n - 1))$
4. $(r \mapsto 3) * (s \mapsto 3) \rhd \exists n. (r \mapsto n) * (s \mapsto n)$
5. $\exists n. (r \mapsto n) * \ulcorner n > 0 \urcorner * \ulcorner n < 0 \urcorner \rhd (r \mapsto m) * (r \mapsto m)$

# Heap implications: true or false?

1. $(r \mapsto 3) \rhd \exists n.\,(r \mapsto n)$ true
2. $\exists n.\,(r \mapsto n) \rhd (r \mapsto 3)$
3. $\exists n.\,(r \mapsto n) * \ulcorner n > 0 \urcorner \rhd \exists n.\,\ulcorner n > 1 \urcorner * (r \mapsto (n-1))$
4. $(r \mapsto 3) * (s \mapsto 3) \rhd \exists n.\,(r \mapsto n) * (s \mapsto n)$
5. $\exists n.\,(r \mapsto n) * \ulcorner n > 0 \urcorner * \ulcorner n < 0 \urcorner \rhd (r \mapsto m) * (r \mapsto m)$

# Heap implications: true or false?

1. $(r \mapsto 3) \rhd \exists n. (r \mapsto n)$ true
2. $\exists n. (r \mapsto n) \rhd (r \mapsto 3)$ false
3. $\exists n. (r \mapsto n) * \ulcorner n > 0 \urcorner \rhd \exists n. \ulcorner n > 1 \urcorner * (r \mapsto (n-1))$
4. $(r \mapsto 3) * (s \mapsto 3) \rhd \exists n. (r \mapsto n) * (s \mapsto n)$
5. $\exists n. (r \mapsto n) * \ulcorner n > 0 \urcorner * \ulcorner n < 0 \urcorner \rhd (r \mapsto m) * (r \mapsto m)$

# Heap implications: true or false?

1. $(r \mapsto 3) \rhd \exists n.\,(r \mapsto n)$ true
2. $\exists n.\,(r \mapsto n) \rhd (r \mapsto 3)$ false
3. $\exists n.\,(r \mapsto n) * \ulcorner n > 0 \urcorner \rhd \exists n.\,\ulcorner n > 1 \urcorner * (r \mapsto (n-1))$ true
4. $(r \mapsto 3) * (s \mapsto 3) \rhd \exists n.\,(r \mapsto n) * (s \mapsto n)$
5. $\exists n.\,(r \mapsto n) * \ulcorner n > 0 \urcorner * \ulcorner n < 0 \urcorner \rhd (r \mapsto m) * (r \mapsto m)$

# Heap implications: true or false?

1. $(r \mapsto 3) \rhd \exists n.\, (r \mapsto n)$ true
2. $\exists n.\, (r \mapsto n) \rhd (r \mapsto 3)$ false
3. $\exists n.\, (r \mapsto n) * \ulcorner n > 0 \urcorner \rhd \exists n.\, \ulcorner n > 1 \urcorner * (r \mapsto (n-1))$ true
4. $(r \mapsto 3) * (s \mapsto 3) \rhd \exists n.\, (r \mapsto n) * (s \mapsto n)$ true
5. $\exists n.\, (r \mapsto n) * \ulcorner n > 0 \urcorner * \ulcorner n < 0 \urcorner \rhd (r \mapsto m) * (r \mapsto m)$

# Heap implications: true or false?

1. $(r \mapsto 3) \rhd \exists n.\,(r \mapsto n)$ true
2. $\exists n.\,(r \mapsto n) \rhd (r \mapsto 3)$ false
3. $\exists n.\,(r \mapsto n) * \ulcorner n > 0 \urcorner \rhd \exists n.\,\ulcorner n > 1 \urcorner * (r \mapsto (n-1))$ true
4. $(r \mapsto 3) * (s \mapsto 3) \rhd \exists n.\,(r \mapsto n) * (s \mapsto n)$ true
5. $\exists n.\,(r \mapsto n) * \ulcorner n > 0 \urcorner * \ulcorner n < 0 \urcorner \rhd (r \mapsto m) * (r \mapsto m)$ 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:

$$
\begin{array}{c}
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) * (s \mapsto a) \ \rhd\ (r \mapsto 3) * (s \mapsto a) \\
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) * (s \mapsto a) \ \rhd\ (r \mapsto 3) * (s \mapsto 3 + (a - 3)) \\
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) * (s \mapsto a) \ \rhd\ \exists m.\, (r \mapsto 3) * (s \mapsto 3 + m) \\
\hline
a : \mathsf{int},\ a > 5 \ \vdash\ (r \mapsto 3) * (s \mapsto a) \ \rhd\ \exists nm.\, (r \mapsto n) * (s \mapsto n + m) \\
\hline
\emptyset \ \vdash\ \exists a.\, \ulcorner a > 5 \urcorner * (r \mapsto 3) * (s \mapsto a) \ \rhd\ \exists nm.\, (r \mapsto n) * (s \mapsto n + m) \\
\hline
\emptyset \ \vdash\ (r \mapsto 3) * \exists a.\, \ulcorner a > 5 \urcorner * (s \mapsto a) \ \rhd\ \exists nm.\, (s \mapsto n + m) * (r \mapsto n)
\end{array}
$$

# Summary

$(*)$ is associative, commutative, and has $\ulcorner\,\urcorner$ as neutral element.

Existentials and pure facts may be extruded from stars.

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

"$\ulcorner$False$\urcorner \rhd H$" is always true.

"$(r \mapsto n) * (r \mapsto m)$" is equivalent to "$\ulcorner$False$\urcorner$".

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

# Highland Cattle

# Chapter 10

## Structural rules

# Frame rule

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

Reformulation:

$$\frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

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

# Consequence rule

$$\frac{H \rhd H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \rhd Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence}$$

with the overloading:

$$Q' \rhd Q \;\equiv\; \forall x.\, (Q'\,x \rhd Q\,x)$$

# Consequence rule

$$\frac{H \rhd H' \qquad \{H'\}\; t\; \{Q'\} \qquad Q' \rhd Q}{\{H\}\; t\; \{Q\}}\ \textsc{consequence}$$

with the overloading:

$$Q' \rhd Q \ \equiv\ \forall x.\, (Q'\, x \rhd Q\, x)$$

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:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x * \exists s.\, s \mapsto r\}$$

To:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x\}$$

# Recall the need for garbage collection

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

From:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x \ * \ \exists s.\, s \mapsto r\}$$

To:

$$\{\ulcorner\urcorner\}\ (\texttt{myref x})\ \{\lambda r.\ r \mapsto x\}$$

Question: can the following rule be used?

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

# Garbage collection rule

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad \text{where: } \mathsf{GC} \equiv \exists H'.\, H'$$

Same as:

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

Observe that $H'$ may depend on the return value $x$:
For $(\lambda r.\ r \mapsto x\ \ast\ \exists s.\, s \mapsto r)$, we may instantiate $H'$ as $(\exists s.\, s \mapsto r)$.

# Garbage collection in the pre-condition

$$\frac{\{H\}\ t\ \{Q\}}{\{H * \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-pre} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H * H'\}\ t\ \{Q\}}\ \textsc{gc-pre'}$$

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

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad \frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

# Garbage collection in the pre-condition

$$\frac{\{H\}\ t\ \{Q\}}{\{H * \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-pre} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H * H'\}\ t\ \{Q\}}\ \textsc{gc-pre'}$$

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

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad \frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

Proof:

$$\dfrac{\dfrac{\{H\}\ t\ \{Q\}}{\{H * \mathsf{GC}\}\ t\ \{Q \star \mathsf{GC}\}}\ \textsc{frame}}{\{H * \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-post}$$

# Garbage collection in the pre-condition

$$\frac{\{H\}\ t\ \{Q\}}{\{H * \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-pre} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H * H'\}\ t\ \{Q\}}\ \textsc{gc-pre'}$$

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

$$\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}\ \textsc{gc-post} \qquad \frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}\ \textsc{frame}$$

Proof:

$$\frac{\dfrac{\{H\}\ t\ \{Q\}}{\{H * \mathsf{GC}\}\ t\ \{Q \star \mathsf{GC}\}}\ \textsc{frame}}{\{H * \mathsf{GC}\}\ t\ \{Q\}}\ \textsc{gc-post}$$

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

# Extraction of existentials and propositions

$$\{\exists n.\,(r \mapsto n) * \ulcorner \mathsf{even}\, n \urcorner\}\ (\texttt{!r})\ \{\lambda x.\, ...\}$$

To prove the above, we need to show that:

$$\forall n.\ \mathsf{even}\, n \ \ \Rightarrow\ \ \{r \mapsto n\}\ (\texttt{!r})\ \{\lambda x.\, ...\}$$

Rules:

$$\frac{x \notin t, Q \qquad \forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\, H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P\ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{\ulcorner P \urcorner * H\}\ t\ \{Q\}}\ \textsc{prop}$$

# Extraction of existentials and propositions

$$\{\exists n.\,(r \mapsto n) * \ulcorner \mathsf{even}\, n \urcorner\}\ (\texttt{!r})\ \{\lambda x.\, ...\}$$

To prove the above, we need to show that:

$$\forall n.\ \mathsf{even}\, n \ \ \Rightarrow \ \ \{r \mapsto n\}\ (\texttt{!r})\ \{\lambda x.\, ...\}$$

Rules:

$$\frac{x \notin t, Q \qquad \forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\, H\}\ t\ \{Q\}}\ \textsc{exists} \qquad\qquad \frac{P \ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{\ulcorner P \urcorner * H\}\ t\ \{Q\}}\ \textsc{prop}$$

# Chapter 11

## Reasoning rules for terms

# Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\, r \mapsto n+1\} \qquad \{r \mapsto n+1\}\ (\texttt{!r})\ \{\lambda x.\, \ulcorner x = n+1 \urcorner * r \mapsto n+1\}}{\{r \mapsto n\}\ (\texttt{incr r; !r})\ \{\lambda x.\, \ulcorner x = n+1 \urcorner * r \mapsto n+1\}}$$

# Reasoning rule for sequences

Example:

$$\frac{\{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\, r \mapsto n+1\} \qquad \{r \mapsto n+1\}\ (\texttt{!r})\ \{\lambda x.\, \ulcorner x = n+1 \urcorner \ast r \mapsto n+1\}}{\{r \mapsto n\}\ (\texttt{incr r; !r})\ \{\lambda x.\, \ulcorner x = n+1 \urcorner \ast r \mapsto n+1\}}$$

**Exercise:** complete the rule for sequences.

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

# Reasoning rule for sequences

Solution 1:

$$\frac{\{H\}\ t_1\ \{\lambda\_.\ H'\} \qquad \{H'\}\ t_2\ \{Q\}}{\{H\}\ (t_1\ ;\ t_2)\ \{Q\}}$$

Solution 2:

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

Remark: $Q' = \lambda\_.\ H'$ is equivalent to $Q'\,() = H'$.

# Reasoning rule for let-bindings

**Exercise:** complete the reasoning rule for let-bindings.

$$\frac{\{...\}\ t_1\ \{...\} \qquad \forall x.\ \big(\{...\}\ t_2\ \{...\}\big)}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

# Reasoning rule for let-bindings

**Exercise:** complete the reasoning rule for let-bindings.

$$\frac{\{...\}\ t_1\ \{...\} \qquad \forall x.\ \big(\{...\}\ t_2\ \{...\}\big)}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

Solution:

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\, x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}\ \textsc{let}$$

# Example of let-binding

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

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

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

# Example of let-binding

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\ \{Q\}}$$

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

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

Solution:

$$\begin{array}{lcl} H & \equiv & (r \mapsto 3) \\ Q & \equiv & \lambda x.\ \ulcorner x = 4 \urcorner * (r \mapsto 3) \\ Q' & \equiv & \lambda y.\ \ulcorner y = 3 \urcorner * (r \mapsto 3) \end{array}$$

# Reasoning rule for values

Example:

$$\{\ulcorner\urcorner\}\ 3\ \{\lambda x.\ \ulcorner x = 3 \urcorner\}$$

Rule:

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

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

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

# Reasoning rule for values

Example:

$$\{\ulcorner\urcorner\}\ 3\ \{\lambda x.\ \ulcorner x = 3 \urcorner\}$$

Rule:

$$\frac{}{\{\ulcorner\urcorner\}\ v\ \{\lambda x.\ \ulcorner x = v \urcorner\}}\ \textsc{val}$$

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

$$\frac{\ldots \ \rhd\ \ldots}{\{H\}\ v\ \{Q\}}$$

Solution:

$$\frac{H\ \rhd\ Q\, v}{\{H\}\ v\ \{Q\}}\ \textsc{val-frame}$$

# Reasoning rule for conditionals

Rule:

$$\frac{(v = \mathsf{true} \;\Rightarrow\; \{H\}\; t_1\; \{Q\}) \qquad (v = \mathsf{false} \;\Rightarrow\; \{H\}\; t_2\; \{Q\})}{\{H\}\; (\mathsf{if}\, v\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\; \{Q\}}\;\textsc{if}$$

# Reasoning rule for conditionals

Rule:

$$\frac{(v = \mathsf{true} \;\Rightarrow\; \{H\}\; t_1\; \{Q\}) \qquad (v = \mathsf{false} \;\Rightarrow\; \{H\}\; t_2\; \{Q\})}{\{H\}\; (\mathsf{if}\, v\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\; \{Q\}}\ \textsc{if}$$

Transformation to A-normal form:

$$(\mathsf{if}\, t_0\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2) \quad = \quad (\mathsf{let}\, v = t_0\, \mathsf{in}\; (\mathsf{if}\, v\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2))$$

# Reasoning rule for top-level functions

Rule:

$$\frac{v_1 \ = \ \lambda x.\, t \qquad \{H\}\ ([x \to v_2]\, t)\ \{Q\}}{\{H\}\ (v_1\, v_2)\ \{Q\}}\ \textsc{top-fun}$$

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

$$(t_1\, t_2) \quad = \quad (\mathsf{let}\, f = t_1\, \mathsf{in}\, \mathsf{let}\, v = t_2\, \mathsf{in}\, (f\ v))$$

# Reasoning rule for top-level functions

Rule:

$$\frac{v_1 \;=\; \lambda x.\, t \qquad \{H\}\; ([x \to v_2]\, t)\; \{Q\}}{\{H\}\; (v_1\, v_2)\; \{Q\}}\ \textsc{top-fun}$$

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

$$(t_1\, t_2) \quad = \quad (\mathsf{let}\, f \,=\, t_1 \,\mathsf{in}\, \mathsf{let}\, v \,=\, t_2 \,\mathsf{in}\, (f\ v))$$

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

$$\frac{x \to y \qquad \forall z\ (x \to z) \Rightarrow y = z}{x \to_{\mathsf{det}} y}$$

# Verification of a simple function

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

Specification:

$$\forall r n. \quad \{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto n+1\}$$

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

$$\{r \mapsto n\}\ (\texttt{let a = !r in r := a+1})\ \{\lambda\_.\ r \mapsto n+1\}$$

# Verification of a simple function

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

Specification:

$$\forall rn. \quad \{r \mapsto n\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto n+1\}$$

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

$$\{r \mapsto n\}\ (\texttt{let a = !r in r := a+1})\ \{\lambda\_.\ r \mapsto n+1\}$$

We conclude using the let-binding rule:
$Q' \equiv \lambda x. \ulcorner x = n \urcorner * (r \mapsto n)$.

# Reasoning rule for top-level recursive functions

Rule:

$$\frac{v_1 \;=\; \mu f.\lambda x.t \qquad \{H\}\;([f \to v_1]\,[x \to v_2]\,t)\;\{Q\}}{\{H\}\;(v_1\,v_2)\;\{Q\}}\;\textsc{top-fix}$$

Specification of recursive functions may be established by induction.

# Reasoning rule for top-level recursive functions

Rule:

$$\frac{v_1 \; = \; \mu f.\lambda x.t \qquad \{H\} \; ([f \to v_1] \, [x \to v_2] \, t) \; \{Q\}}{\{H\} \; (v_1 \, v_2) \; \{Q\}} \; \textsc{top-fix}$$

Specification of recursive functions may be established by induction.

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

# 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:

$$\forall pL. \quad \{p \rightsquigarrow \mathsf{MList}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ \ulcorner n = |L| \urcorner \ast\ p \rightsquigarrow \mathsf{MList}\, L\}$$

# 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:

$$\forall pL. \quad \{p \leadsto \mathsf{MList}\, L\}\ (\texttt{mlength p})\ \{\lambda n.\ \ulcorner n = |L| \urcorner \ast\ p \leadsto \mathsf{MList}\, L\}$$

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

# Verification of mlength: nil case

**Case** $p =$ **null**. Goal is:

$$\{p \rightsquigarrow \mathsf{MList}\, L\}\ (0)\ \{\lambda n.\ \ulcorner n = |L| \urcorner \ast\ p \rightsquigarrow \mathsf{MList}\, L\}$$

– Replace $p$ with null.
– Rewrite null $\rightsquigarrow$ MList $L$ to $\ulcorner L = \mathsf{nil} \urcorner$ in the pre and the post.
– By the PROP rule:

$$L = \mathsf{nil} \ \Rightarrow\ \{\ulcorner\urcorner\}\ (0)\ \{\lambda n.\ \ulcorner n = |L| \urcorner \ast \ulcorner L = \mathsf{nil} \urcorner\}$$

– Replace $L$ with nil.

$$\{\ulcorner\urcorner\}\ (0)\ \{\lambda n.\ \ulcorner n = |\mathsf{nil}| \urcorner \ast \ulcorner \mathsf{nil} = \mathsf{nil} \urcorner\}$$

– Apply the VAL-FRAME rule.

$$\ulcorner\urcorner \ \rhd\ \ulcorner 0 = 0 \urcorner \ast \ulcorner \mathsf{nil} = \mathsf{nil} \urcorner$$

# Verification of mlength: cons case (1/2)

**Case** $p \neq$ **null**. Goal is:

$$\{p \rightsquigarrow \mathsf{MList}\, L\}$$
$$(\texttt{let p' = p.tl in let n' = mlength p' in 1 + n'})$$
$$\{\lambda n.\ \ulcorner n = |L| \urcorner \ast\ p \rightsquigarrow \mathsf{MList}\, L\}$$

– Unfold MList in pre and post, and decompose $L$ as $x :: L'$:

$$p' \rightsquigarrow \mathsf{MList}\, L' \ \ast\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}$$

– Apply the let-binding rule, and the read axiom. Remains:

$$\{p' \rightsquigarrow \mathsf{MList}\, L' \ \ast\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}$$
$$(\texttt{let n' = mlength p' in 1 + n'})$$
$$\{\lambda n.\ \ulcorner n = |L| \urcorner \ast\ p' \rightsquigarrow \mathsf{MList}\, L' \ \ast\ p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\ \}$$

– Apply the frame rule to remove: $p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}$ .
– Apply the let-binding rule with :
$Q \equiv \lambda n'. \ulcorner n' = |L'| \urcorner \ast p' \rightsquigarrow \mathsf{MList}\, L'$.

# Verification of mlength: cons case (2/2)

There remains to prove the two premises of the let-rule.

– First branch, exploit the induction hypothesis:

$$\{p' \leadsto \mathsf{MList}\, L'\}\ (\texttt{mlength p'})\ \{\lambda n'.\ \ulcorner n' = |L'| \urcorner\ *\ p' \leadsto \mathsf{MList}\, L'\}$$

– Second branch:

$$\{p' \leadsto \mathsf{MList}\, L'\ *\ \ulcorner n' = |L'| \urcorner\}\ (\texttt{1 + n'})\ \{\lambda n.\ \ulcorner n = |L| \urcorner\ *\ p' \leadsto \mathsf{MList}\, L'\}$$

– Apply the PROP rule and the VAL-FRAME rule.

$$n' = |L'| \quad \Rightarrow \quad p' \leadsto \mathsf{MList}\, L'\ \rhd\ \ulcorner 1 + n' = |L| \urcorner\ *\ p' \leadsto \mathsf{MList}\, L'$$

– Cancel equal parts, conclude using
$|L| = |x :: L'| = 1 + |L'| = 1 + n'$.

# Reasoning rule for local functions

Rule template:

$$\frac{\forall f.\ (...)\ \Rightarrow\ \{H\}\ t\ \{Q\}}{\{H\}\ (\mathsf{let\,rec}\ f\ x = \mathit{body}\ \mathsf{in}\ t)\ \{Q\}}$$

Hypothesis about $f$:

$$\forall x H' Q'.\ \ \{H'\}\ \mathit{body}\ \{Q'\}\ \ \Rightarrow\ \ \{H'\}\ (f\ x)\ \{Q'\}$$

Rule:

$$\frac{\begin{array}{c}\forall f.\ \ P f\ \Rightarrow\ \{H\}\ t\ \{Q\} \\ P f\ =\ \big(\forall x H' Q'.\,\{H'\}\ \mathit{body}\ \{Q'\}\ \Rightarrow\ \{H'\}\ (f\ x)\ \{Q'\}\big)\end{array}}{\{H\}\ (\mathsf{let\,rec}\ f\ x = \mathit{body}\ \mathsf{in}\ t)\ \{Q\}}\ \text{FIX}$$

# Summary

$$\frac{}{\{\ulcorner\,\urcorner\}\ v\ \{\lambda x.\ulcorner x = v\urcorner\}}$$

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

$$\frac{v = \mathsf{true}\ \Rightarrow\ \{H\}\ t_1\ \{Q\} \qquad v = \mathsf{false}\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{if}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)\ \{Q\}}$$

$$\frac{\forall f.\ \big(\forall x H' Q'.\ \{H'\}\ t_1\ \{Q'\}\ \Rightarrow\ \{H'\}\ (f\,x)\ \{Q'\}\big)\ \Rightarrow\ \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let\,rec}\,f\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

# 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:

from $\quad r \mapsto (i-1)! \quad$ to $\quad r \mapsto i!$

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:

from $\quad r \mapsto (i-1)! \quad$ to $\quad r \mapsto i!$

After the loop:

$$r \mapsto n!$$

Loop invariant $(I : \text{int} \rightarrow \text{Hprop})$ that applies for any $i \in [2, n+1]$:

$$I\,i \quad \equiv \quad r \mapsto (i-1)!$$

# Reasoning rule for for-loops

Reasoning rule for the case $a \leqslant b$:

$$\frac{\begin{array}{c} H \rhd I\,a \\ \forall i \in [a,b].\quad \{I\,i\}\ t\ \{\lambda\_.\,I\,(i+1)\} \\ I\,(b+1) \rhd Q\,() \end{array}}{\{H\}\ (\mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t)\ \{Q\}}$$

# Reasoning rule for for-loops

Reasoning rule for the case $a \leqslant b$:

$$\frac{\begin{array}{c} H \rhd I\,a \\ \forall i \in [a,b].\quad \{I\,i\}\ t\ \{\lambda\_.\ I\,(i+1)\} \\ I\,(b+1) \rhd Q\,() \end{array}}{\{H\}\ (\mathsf{for}\ i = a\ \mathsf{to}\ b\ \mathsf{do}\ t)\ \{Q\}}$$

General rule, also covering the case $a > b$:

$$\frac{\begin{array}{c} H \rhd I\,a \\ \forall i \in [a,b].\quad \{I\,i\}\ t\ \{\lambda\_.\ I\,(i+1)\} \\ I\,(\mathsf{max}\,a\,(b+1)) \rhd Q\,() \end{array}}{\{H\}\ (\mathsf{for}\ i = a\ \mathsf{to}\ b\ \mathsf{do}\ t)\ \{Q\}}$$

# 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 $t_1$.

$$\frac{H \rhd I \qquad \{I\}\ t_1\ \{J\} \qquad \{J\,\mathsf{true}\}\ t_2\ \{\lambda\_.\, I\} \qquad J\,\mathsf{false} \rhd Q\,()}{\{H\}\ (\mathsf{while}\, t_1\, \mathsf{do}\, t_2)\ \{Q\}}$$

where $(I : \mathsf{Hprop})$ and $(J : \mathsf{bool} \to \mathsf{Hprop})$.

# 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 $t_1$.

$$\frac{H \rhd I \qquad \{I\}\ t_1\ \{J\} \qquad \{J\,\mathsf{true}\}\ t_2\ \{\lambda\_.\, I\} \qquad J\,\mathsf{false} \rhd Q\,()}{\{H\}\ (\mathsf{while}\, t_1\, \mathsf{do}\, t_2)\ \{Q\}}$$

where $(I : \mathsf{Hprop})$ and $(J : \mathsf{bool} \to \mathsf{Hprop})$.

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

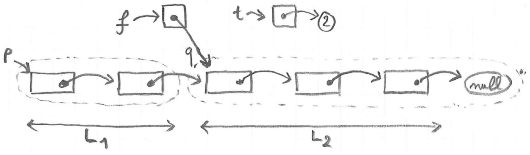
We focus on a different approach that:

- 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:

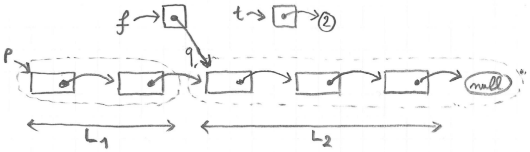
$$\frac{\{H\}\ (\text{if}\, t_1 \,\text{then}\, (t_2\,;\ (\text{while}\, t_1 \,\text{do}\, t_2))\,\text{else}\,())\ \{Q\}}{\{H\}\ (\text{while}\, t_1 \,\text{do}\, t_2)\ \{Q\}}$$

# 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 $q$:

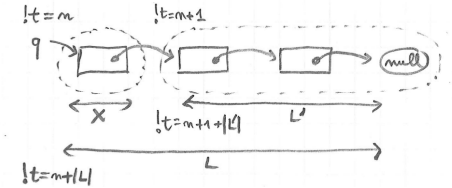
$$\{q \rightsquigarrow \mathsf{MList}\, L_2 \ * \ f \mapsto q \ * \ t \mapsto n\}$$
$$\big(\texttt{while !f != null do incr t; f := (!f).tl; done}\big)$$
$$\{q \rightsquigarrow \mathsf{MList}\, L_2 \ * \ f \mapsto \mathsf{null} \ * \ t \mapsto (n + \mathsf{length}\, L_2)\}$$

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




| | | | |
|---|---|---|---|
| $q \rightsquigarrow \mathsf{MList}\, L_2$ | $* \;\; f \mapsto q$ | $* \;\; t \mapsto n$ | begin |
| $q \mapsto \{\!\vert x; q' \vert\!\} * q' \rightsquigarrow \mathsf{MList}\, L_2'$ | $* \;\; f \mapsto q$ | $* \;\; t \mapsto n$ | unfold |
| $q \mapsto \{\!\vert x; q' \vert\!\} * q' \rightsquigarrow \mathsf{MList}\, L_2'$ | $* \;\; f \mapsto q$ | $* \;\; t \mapsto n + 1$ | increment |
| $\underline{q \mapsto \{\!\vert x; q' \vert\!\}} * q' \rightsquigarrow \mathsf{MList}\, L_2'$ | $* \;\; f \mapsto q'$ | $* \;\; t \mapsto n + 1$ | shift head |
| $q \mapsto \{\!\vert x; q' \vert\!\} * q' \rightsquigarrow \mathsf{MList}\, L_2'$ | $* \;\; f \mapsto \mathsf{null}$ | $* \;\; t \mapsto n + 1 + \vert L_2' \vert$ | frame+IH |
| $q \rightsquigarrow \mathsf{MList}\, L_2$ | $* \;\; f \mapsto \mathsf{null}$ | $* \;\; t \mapsto n + \vert L_2 \vert$ | 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:

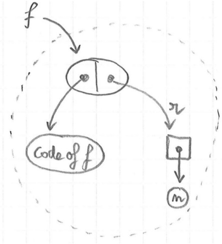
$$\forall n. \quad \{r \mapsto n\}\,(\texttt{f ()})\,\{\lambda\_.\ r \mapsto n+1\}$$

Successive states:

$$r \mapsto 3 \qquad r \mapsto 4 \qquad r \mapsto 5$$

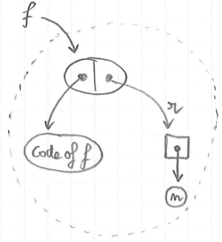
# Counter function: code

```
let mkcounter () =
  let r = ref 0 in
  (fun () -> incr r; !r)
```



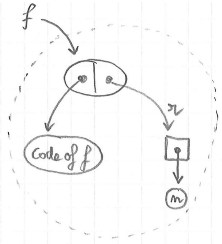

```
let c = mkcounter() in
let x = c() in
let y = c() in
assert (x = 1 && y = 2)
```

# Counter function: specification




$$f \rightsquigarrow \mathsf{Count}\, n \;\equiv\; \exists r.\, (r \mapsto n) \\ \quad * \;\ulcorner \forall i.\ \{r \mapsto i\}\ (f\ ())\ \{\lambda x.\, \ulcorner x = i + 1 \urcorner * (r \mapsto i + 1)\} \urcorner$$

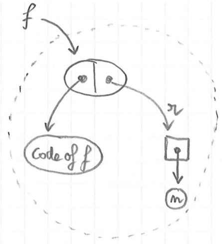
# Counter function: specification




$$
\begin{aligned}
f \rightsquigarrow \mathsf{Count}\, n \;\;\equiv\;\; & \exists r.\, (r \mapsto n) \\
& *\;\; \ulcorner \forall i.\; \{r \mapsto i\}\; (f\;())\; \{\lambda x.\, \ulcorner x = i + 1 \urcorner * (r \mapsto i + 1)\} \urcorner
\end{aligned}
$$

**Exercise:** specify a counter function, only in terms of $f \rightsquigarrow \mathsf{Count}\, n$.

# Counter function: specification




$$f \rightsquigarrow \mathsf{Count}\, n \;\equiv\; \exists r.\, (r \mapsto n) \ast \ulcorner \forall i.\ \{r \mapsto i\}\ (f\ ())\ \{\lambda x.\, \ulcorner x = i+1 \urcorner \ast (r \mapsto i+1)\} \urcorner$$

**Exercise:** specify a counter function, only in terms of $f \rightsquigarrow \mathsf{Count}\, n$.

$$\{\ulcorner\urcorner\}\ (\texttt{mkcounter()})\ \{\lambda f.\ f \rightsquigarrow \mathsf{Count}\, 0\}$$

$$\forall f i.\quad \{f \rightsquigarrow \mathsf{Count}\, i\}\ (f\ ())\ \{\lambda x.\, \ulcorner x = i+1 \urcorner \ast f \rightsquigarrow \mathsf{Count}\,(i+1)\}$$

# Chapter 14

## Basic higher-order functions

# Apply

```
let apply f x =
  f x
```

Specification:

$$\begin{aligned} \forall f x H Q. \quad & \{H\}\ (f\ x)\ \{Q\} \\ \Rightarrow \quad & \{H\}\ (\texttt{apply}\ f\ x)\ \{Q\} \end{aligned}$$

# Apply

```
let apply f x =
  f x
```

Specification:

$$\begin{array}{rl} \forall f x H Q. & \{H\}\ (f\ x)\ \{Q\} \\ \Rightarrow & \{H\}\ (\texttt{apply}\ f\ x)\ \{Q\} \end{array}$$

This is equivalent to the form below, which involves nested triples:

$$\forall f x H Q. \quad \{H \ * \ \ulcorner \{H\}\ (f\ x)\ \{Q\} \urcorner\}\ (\texttt{apply}\ f\ x)\ \{Q\}$$

# Function twice

```
let twice f =
  f(); f()
```

Specification:

$$\begin{array}{rl} \forall f\, H'\, Q. & \{H\}\ (f\ ())\ \{\lambda\_.\ H'\} \\ \wedge & \{H'\}\ (f\ ())\ \{Q\} \\ \Rightarrow & \{H\}\ (\texttt{twice}\, f)\ \{Q\} \end{array}$$

# Function repeat

```
let repeat n f =
  for i = 0 to n-1 do
    f()
  done
```

**Exercise:** specify `repeat`, using an invariant $I$, of type int $\rightarrow$ 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 int → Hprop.

$$\begin{array}{lrl} \forall n f I. & & (\forall i \in [0, n).\quad \{I\, i\}\ (f\ ())\ \{\lambda\_.\ I\,(i+1)\}) \\ & \Rightarrow & \{I\, 0\}\ (\texttt{repeat}\, n\, f)\ \{\lambda\_.\ I\, n\} \end{array}$$

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 list $\alpha \rightarrow$ 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 list $\alpha \to$ Hprop.

$$\begin{array}{ll} \forall f l I. & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big) \\ \Rightarrow & \{I\,\mathsf{nil}\}\ (\mathtt{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

where $k \& x \equiv k +\!\!+ (x :: \mathsf{nil})$.

# Length using iter

$$\begin{array}{rl} & \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

```
let length l =
  let r = ref 0 in
  iter (fun x -> incr r) l;
  !r
```

**Exercise:** give the instantiatiation of the invariant $I$ for `iter`; then, write the specialization of the specification of `iter` to $I$ and to `(fun x -> incr r)`; finally, check that the premise is provable.

# Length using iter

$$\begin{array}{ll} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

```
let length l =
  let r = ref 0 in
  iter (fun x -> incr r) l;
  !r
```

**Exercise:** give the instantiatiation of the invariant $I$ for `iter`; then, write the specialization of the specification of `iter` to $I$ and to `(fun x -> incr r)`; finally, check that the premise is provable.

Invariant: $I \;\equiv\; \lambda k.\ r \mapsto |k|$.

$$\begin{array}{ll} & \big(\forall x k.\ \{r \mapsto |k|\}\ (\texttt{incr r})\ \{\lambda\_.\ r \mapsto |k| + 1\}\big) \\ \Rightarrow & \{r \mapsto 0\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ r \mapsto |l|\} \end{array}$$

# Sum using iter

$$\begin{array}{ll} & \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\} \end{array}$$

```
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

$$\begin{array}{rl} & \big(\forall x k.\ \{I\, k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\ l\} \end{array}$$

```
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`.

$$I \ \equiv\ \lambda k.\ r \mapsto \mathsf{Sum}\, k$$

where:

$$\mathsf{Sum}\, k \ \equiv\ \mathsf{Fold}\,(+)\,0\,k$$

# Constraints over the items

$$\begin{array}{rl} & \big(\forall x k.\ \{I\, k\}\ (f\, x)\ \{\lambda\_.\ I\,(k \& x)\}\big) \\ \Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\, f\, l)\ \{\lambda\_.\ I\, l\} \end{array}$$

Given a list $x_1 :: x_2 :: ... :: x_n ::$ nil, let us compute:

$$\frac{10}{x_1} + \frac{10}{x_2} + ... + \frac{10}{x_n}$$

```
iter (fun x -> r := !r + 10 / x) [2; -3; 4]
```

# Constraints over the items

$$\big(\forall xk.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big)$$
$$\Rightarrow\ \ \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\ l\}$$

Given a list $x_1 :: x_2 :: ... :: x_n ::$ nil, let us compute:

$$\frac{10}{x_1} + \frac{10}{x_2} + ... + \frac{10}{x_n}$$

```
iter (fun x -> r := !r + 10 / x) [2; -3; 4]
```

The above specification of `iter` is too weak. More general specification:

$$\forall f I l.\qquad \big(\forall xk.\ x \in l\ \Rightarrow\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big)$$
$$\Rightarrow\ \ \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\ l\}$$

# Constraints over the items, in order

$$\forall f I l. \quad \big(\forall x k.\ x \in l \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big)$$
$$\Rightarrow \quad \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\,l\}$$

Given a list $x_1 :: x_2 :: \ldots :: x_n ::$ nil, let us compute:

$$\cfrac{10}{\cfrac{10}{\cfrac{10}{0+x_1}+x_2} \cdots +x_n}$$

```
iter (fun x -> r := 10 / (!r + x)) [2; -3; 4]
```

# Constraints over the items, in order

$$\forall f I l. \quad \big(\forall x k.\ x \in l \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \Rightarrow \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\,l\}$$

Given a list $x_1 :: x_2 :: \ldots :: x_n :: \mathsf{nil}$, let us compute:

$$\cfrac{10}{\cfrac{10}{\cfrac{10}{0+x_1}+x_2} \cdot^{\cdot^{\cdot}} +x_n}$$

```
iter (fun x -> r := 10 / (!r + x)) [2; -3; 4]
```

The above specification of `iter` is too weak. Most-general specification:

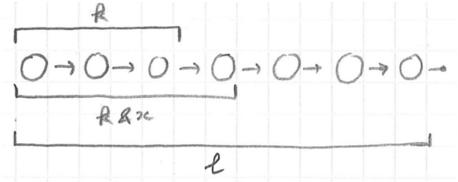
$$\forall f I l. \quad \big(\forall x k s.\ l = k +\!\!+ x :: s \Rightarrow \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \Rightarrow \{I\ \mathsf{nil}\}\ (\texttt{iter}\ f\ l)\ \{\lambda\_.\ I\,l\}$$

# Verification of iter

$$
\begin{array}{ll}
 & \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \\
\Rightarrow & \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\,l\}
\end{array}
$$

```
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:

$$\forall x k.\ \{I\, k\}\ (f\, x)\ \{\lambda\_.\ I\, (k \& x)\}$$

Prove:

$$\{I\, \mathsf{nil}\}\ (\mathsf{iter}\, f\, l)\ \{\lambda\_.\ I\, l\}$$

# Verification of iter: generalized principle

Assume:

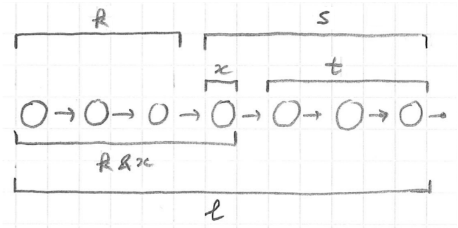
$$\forall x k.\ \{I\, k\}\ (f\ x)\ \{\lambda\_.\ I\, (k \& x)\}$$

Prove:

$$\{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\ l\}$$

Proof by induction over a generalized statement:

$$\forall s k.\quad \{I\ k\}\ (\mathsf{iter}\ f\ s)\ \{\lambda\_.\ I\ (k +\!\!+ s)\}$$



# Verification of iter: induction

```
let rec iter f s =
  match s with
  | [] -> ()
  | x::t -> f x; iter f t
```

Assume: $\forall xk. \quad \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}$

Prove: $\forall ks. \quad \{I\,k\}\ (\mathsf{iter}\,f\,s)\ \{\lambda\_.\ I\,(k \mathbin{+\!\!+} s)\}$

By induction on $s$:

- Case $s = \mathsf{nil}$. Goal is: $\{I\,k\}\ (\mathsf{iter}\,f\,\mathsf{nil})\ \{\lambda\_.\ I\,(k \mathbin{+\!\!+} \mathsf{nil})\}$.
  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: $\forall xk.\quad \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}$

Prove: $\forall ks.\quad \{I\,k\}\ (\text{iter}\,f\,s)\ \{\lambda\_.\ I\,(k +\!\!+ s)\}$

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:
  $\{I\,k\}\ (\text{iter}\,f\,(x :: t))\ \{\lambda\_.\,I\,(k +\!\!+ (x :: t))\}$.

$$\dfrac{\dfrac{\text{HYPOTHESIS-ON-F}}{\{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}} \qquad \dfrac{\text{INDUCTION-HYPOTHESIS}}{\{I\,(k\&x)\}\ (\text{iter}\,f\,t)\ \{\lambda\_.\ I\,((k\&x) +\!\!+ t)\}}}{\{I\,k\}\ (f\,x;\ \text{iter}\,f\,t)\ \{I\,((k\&x) +\!\!+ t)\}}\ \text{SEQ}$$

# Invariant on remaining items

$$\big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \quad\Rightarrow\quad \{I\ \mathsf{nil}\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I\ l\}$$

$$\big(\forall \ldots.\ \{\ldots\}\ (f\,x)\ \{\lambda\_.\ \ldots\}\big) \quad\Rightarrow\quad \{I'\,l\}\ (\mathsf{iter}\ f\ l)\ \{\lambda\_.\ I'\ \mathsf{nil}\}$$

**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

$$\big(\forall xk.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big) \quad\Rightarrow\quad \{I\,\mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$

$$\big(\forall\ldots.\ \{\ldots\}\ (f\,x)\ \{\lambda\_.\ \ldots\}\big) \quad\Rightarrow\quad \{I'\,l\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I'\,\mathsf{nil}\}$$

**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.

$$\big(\forall xs.\ \{I'\,(x :: s)\}\ (f\,x)\ \{\lambda\_.\ I'\,s\}\big) \quad\Rightarrow\quad \{I'\,l\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I'\,\mathsf{nil}\}$$

$$I\,k \;\equiv\; \exists s.\, \ulcorner l = k +\!\!+\, s \urcorner * I'\,s$$

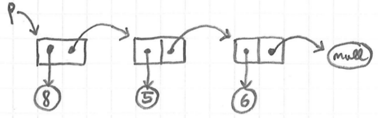
$$I'\,s \;\equiv\; \exists k.\, \ulcorner l = k +\!\!+\, s \urcorner * I\,k$$

# 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*.

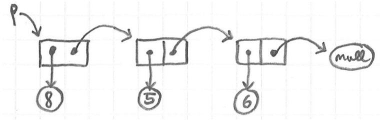
# Iterating over a mutable list




```
let rec miter f p =
  if p == null
    then ()
    else (f p.hd; miter f p.tl)
```

# Iterating over a mutable list

$$\forall f l I. \quad \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big)$$
$$\Rightarrow \quad \{I\ \mathsf{nil}\}\ (\mathsf{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$




Specification:

$$\forall f p I l. \quad \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k\&x)\}\big)$$
$$\Rightarrow \quad \{p \leadsto \mathsf{MList}\,l\ *\ I\ \mathsf{nil}\}\ (\mathtt{miter}\,f\,p)\ \{\lambda\_.\ p \leadsto \mathsf{MList}\,l\ *\ I\,l\}$$

Remark: calls to $f$ will not modify the structure of the list while iterating.

(end of SPLV day 3)

# Chapter 17

## Higher-order representation predicates

# Overview

1. Higher-order predicate:

$$p \rightsquigarrow \mathsf{MList}\, L \qquad \text{is generalized into} \qquad p \rightsquigarrow \mathsf{Mlistof}\, R\, L$$

2. Identity representation predicate:

$$p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Id}\, L \qquad \text{is the same as} \qquad p \rightsquigarrow \mathsf{MList}\, L$$

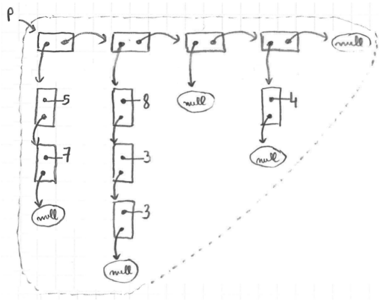
3. Control accesses:

$$\{p \rightsquigarrow \mathsf{MCellof}\,\mathsf{Id}\, v_1\, R_2\, V_2\}\ (p.\mathsf{hd})\ \{\lambda x.\, \ulcorner x = v_1 \urcorner\, *\, \ldots\}$$

4. Compose recursively:

$$p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mlistof}\,(\mathsf{Narytreeof}\, R))\, L$$

# Mutable list of disjoint mutable lists

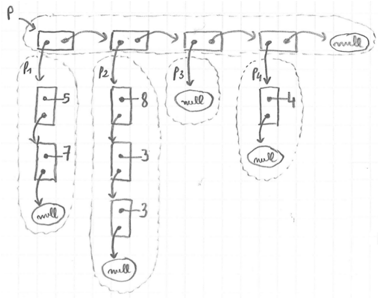



$$L = (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) \\ :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) :: \mathsf{nil}$$

$$p \rightsquigarrow \mathsf{MlistofMlist}\, L$$

(to be later generalized into: $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$)

# Representation using iterated star

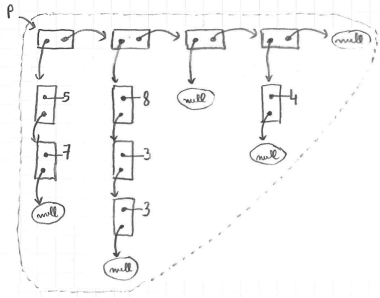



$$L \;=\; (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) \\ :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) :: \mathsf{nil}$$

$$K \;=\; p_1 :: p_2 :: p_3 :: p_4 :: \mathsf{nil}$$

$$p \rightsquigarrow \mathsf{MlistofMlist}\, L \;\equiv\; \exists K.\; p \rightsquigarrow \mathsf{MList}\, K \\ \quad * \circledast_{i \,\in\, [0,|L|)} (K[i]) \rightsquigarrow \mathsf{MList}\,(L[i]) \\ \quad * \ulcorner |K| = |L| \urcorner$$

# Representation using a recursive predicate




$$L = (5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: 3 :: \mathsf{nil}) :: (\mathsf{nil}) :: (4 :: \mathsf{nil}) :: \mathsf{nil}$$

$$\begin{aligned} p \leadsto \mathsf{MlistofMlist}\, L \;\equiv\;& \mathsf{match}\; L \;\mathsf{with} \\ & \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\ & \mid X :: L' \Rightarrow \exists x p'.\;\; p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ & \qquad\qquad\qquad\quad \ast\; p' \leadsto \mathsf{MlistofMlist}\, L' \\ & \qquad\qquad\qquad\quad \ast\; x \leadsto \mathsf{MList}\, X \end{aligned}$$

# Generalization to a higher-order predicate

$$
\begin{aligned}
p \rightsquigarrow \mathsf{MlistofMlist}\, L \;\equiv\; & \mathsf{match}\, L\, \mathsf{with} \\
& \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\qquad\; * \;\; p' \rightsquigarrow \mathsf{MlistofMlist}\, L' \\
& \qquad\qquad\qquad\qquad\; * \;\; x \rightsquigarrow \mathsf{MList}\, X
\end{aligned}
$$

Generalization:

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \;\equiv\; & \mathsf{match}\, L\, \mathsf{with} \\
& \mid \mathsf{nil} \;\Rightarrow\; \ulcorner p = \mathsf{null} \urcorner \\
& \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\qquad\; * \;\; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L' \\
& \qquad\qquad\qquad\qquad\; * \;\; x \rightsquigarrow R\, X
\end{aligned}
$$

In particular:

$$
p \rightsquigarrow \mathsf{MlistofMlist}\, L \;\; = \;\; p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{MList}\, L
$$

# The identity representation predicate

$$
\begin{aligned}
p \leadsto \mathsf{Mlistof}\, R\, L \;\equiv\; & \mathsf{match}\ L\ \mathsf{with} \\
& \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& \mid X :: L' \Rightarrow \exists x p'.\quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\qquad * \; p' \leadsto \mathsf{Mlistof}\, R\, L' \\
& \qquad\qquad\qquad\qquad * \; x \leadsto R\, X
\end{aligned}
$$

$$
\begin{aligned}
p \leadsto \mathsf{MList}\, L \;\equiv\; & \mathsf{match}\ L\ \mathsf{with} \\
& \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& \mid x :: L' \Rightarrow \exists p'.\quad p \leadsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\quad * \; p' \leadsto \mathsf{MList}\, L'
\end{aligned}
$$

**Exercise:** define the identity representation predicate Id such that

$$p \leadsto \mathsf{Mlistof}\, \mathsf{Id}\, L \;=\; p \leadsto \mathsf{MList}\, L$$

# The identity representation predicate

$$
\begin{aligned}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L \;\equiv\; & \mathsf{match}\ L\ \mathsf{with} \\
& \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& \mid X :: L' \Rightarrow \exists x p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad\quad * \; p' \rightsquigarrow \mathsf{Mlistof}\, R\, L' \\
& \qquad\qquad\qquad\quad * \; x \rightsquigarrow R\, X
\end{aligned}
$$

$$
\begin{aligned}
p \rightsquigarrow \mathsf{MList}\, L \;\equiv\; & \mathsf{match}\ L\ \mathsf{with} \\
& \mid \mathsf{nil} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
& \mid x :: L' \Rightarrow \exists p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\
& \qquad\qquad\qquad * \; p' \rightsquigarrow \mathsf{MList}\, L'
\end{aligned}
$$

**Exercise:** define the identity representation predicate Id such that

$$p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Id}\, L \;=\; p \rightsquigarrow \mathsf{MList}\, L$$

Definition:

$$x \rightsquigarrow \mathsf{Id}\, X \;\equiv\; \ulcorner x = X \urcorner$$

# Chapter 18

## Separating implication

# Separating implication or "magic wand"

Recalling separating conjunction:

$$(P_1 * P_2)h \quad \equiv \quad \exists h_1, h_2.\ h = h_1 \uplus h_2 \wedge P_1\, h_1 \wedge P_2\, h_2$$

Introducing *separating implication*:

$$(P \mathrel{-\!\!*} Q)h \quad \equiv \quad \forall h_1.\ h \perp h_1 \wedge P\, h_1 \Rightarrow Q(h \uplus h_1)$$

Intuition:

$$(P \mathrel{-\!\!*} Q) * P \quad \rhd \quad Q$$

Rules:

$$\frac{R * P \vdash Q}{R \vdash (P \mathrel{-\!\!*} Q)} \qquad\qquad \frac{R_1 \vdash (P \mathrel{-\!\!*} Q) \qquad R_2 \vdash P}{R_1 * R_2 \vdash Q}$$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$
2. $\ulcorner \mathsf{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \mathsf{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ $\qquad\qquad \{(1,2)\}$
2. $\ulcorner \text{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \text{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ $\{(1,2)\}$
2. $\ulcorner\text{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ all heaps
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\text{False}\urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner \text{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \text{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$ — $\{(2,3)\}$, any $h$ with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner \text{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$ — $\{(2,3)\}$, any $h$ with $1 \in \text{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$ — $\varnothing$ and any $h$ with $1 \in \text{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \text{False} \urcorner$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

| | |
|---|---|
| 1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ | $\{(1,2)\}$ |
| 2. $\ulcorner \mathsf{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ | all heaps |
| 3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$ | only $\varnothing$ |
| 4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$ | $\{(2,3)\}$, any $h$ with $1 \in \mathrm{dom}(h)$ |
| 5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$ | $\varnothing$ and any $h$ with $1 \in \mathrm{dom}(h)$ |
| 6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \mathsf{False} \urcorner$ | any $h$ with $1 \in \mathrm{dom}(h)$ |
| 7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$ | |
| 8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$ | |

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\,\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner \mathsf{False} \urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1 \urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0 \urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$ — $\{(2,3)\}$, any $h$ with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$ — $\varnothing$ and any $h$ with $1 \in \mathsf{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner \mathsf{False} \urcorner$ — any $h$ with $1 \in \mathsf{dom}(h)$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\,\urcorner$ — any $h$ with $1 \in \mathsf{dom}(h)$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$

# Separating implication examples

**Exercise:** Give heaps satisfying the following predicates:

1. $\ulcorner\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — $\{(1,2)\}$
2. $\ulcorner\mathsf{False}\urcorner \mathrel{-\!\!*} (1 \mapsto 2)$ — all heaps
3. $\ulcorner x \geqslant 1\urcorner \mathrel{-\!\!*} \ulcorner x \geqslant 0\urcorner$ — only $\varnothing$
4. $(1 \mapsto 4) \mathrel{-\!\!*} (1 \mapsto 4) * (2 \mapsto 3)$ — $\{(2,3)\}$, any $h$ with $1 \in \mathsf{dom}(h)$
5. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2)$ — $\varnothing$ and any $h$ with $1 \in \mathsf{dom}(h)$
6. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\mathsf{False}\urcorner$ — any $h$ with $1 \in \mathsf{dom}(h)$
7. $(1 \mapsto 2) \mathrel{-\!\!*} \ulcorner\urcorner$ — any $h$ with $1 \in \mathsf{dom}(h)$
8. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3)$ — any $h$ with $1 \in \mathsf{dom}(h)$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \twoheadrightarrow P * Q)$
2. $(Q \twoheadrightarrow P * Q) \rhd P$
3. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 3) \rhd \ulcorner \mathsf{False} \urcorner$
4. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\,\urcorner \twoheadrightarrow P \rhd P$
6. $P \rhd \ulcorner\,\urcorner \twoheadrightarrow P$
7. $\ulcorner\,\urcorner \rhd (P \twoheadrightarrow Q \twoheadrightarrow P * Q)$
8. $\ulcorner P \rhd Q \urcorner \rhd (P \twoheadrightarrow Q)$
9. $(P \twoheadrightarrow Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q \urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q\urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \twoheadrightarrow P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \twoheadrightarrow P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$
5. $\ulcorner\,\urcorner \twoheadrightarrow P \rhd P$
6. $P \rhd \ulcorner\,\urcorner \twoheadrightarrow P$
7. $\ulcorner\,\urcorner \rhd (P \twoheadrightarrow Q \twoheadrightarrow P * Q)$
8. $\ulcorner P \rhd Q\urcorner \rhd (P \twoheadrightarrow Q)$
9. $(P \twoheadrightarrow Q) \rhd \ulcorner P \rhd Q\urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner \text{False} \urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner \text{False} \urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\,\urcorner \mathrel{-\!\!*} P \rhd P$
6. $P \rhd \ulcorner\,\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\,\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q \urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1,4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\,\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\,\urcorner \mathrel{-\!\!*} P$
7. $\ulcorner\,\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$
8. $\ulcorner P \rhd Q \urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \twoheadrightarrow P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \twoheadrightarrow P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner \text{False} \urcorner$
3. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 3) \rhd \ulcorner \text{False} \urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \twoheadrightarrow P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \twoheadrightarrow P$ ...yes: $P$ and $\ulcorner\urcorner \twoheadrightarrow P$ are equivalent
7. $\ulcorner\urcorner \rhd (P \twoheadrightarrow Q \twoheadrightarrow P * Q)$
8. $\ulcorner P \rhd Q \urcorner \rhd (P \twoheadrightarrow Q)$
9. $(P \twoheadrightarrow Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner \text{False} \urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner \text{False} \urcorner$ no e.g. $\{(1, 4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\urcorner \mathrel{-\!\!*} P$ ...yes: $P$ and $\ulcorner\urcorner \mathrel{-\!\!*} P$ are equivalent
7. $\ulcorner\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$ yes: unfold $\rhd$ and obtain approx. (1)
8. $\ulcorner P \rhd Q \urcorner \rhd (P \mathrel{-\!\!*} Q)$
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \twoheadrightarrow P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \twoheadrightarrow P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner \text{False} \urcorner$
3. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 3) \rhd \ulcorner \text{False} \urcorner$ no e.g. $\{(1,4)\}$ satisfies left
4. $(1 \mapsto 2) \twoheadrightarrow (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\, \urcorner \twoheadrightarrow P \rhd P$ yes, and...
6. $P \rhd \ulcorner\, \urcorner \twoheadrightarrow P$ ...yes: $P$ and $\ulcorner\, \urcorner \twoheadrightarrow P$ are equivalent
7. $\ulcorner\, \urcorner \rhd (P \twoheadrightarrow Q \twoheadrightarrow P * Q)$ yes: unfold $\rhd$ and obtain approx. (1)
8. $\ulcorner P \rhd Q \urcorner \rhd (P \twoheadrightarrow Q)$ yes
9. $(P \twoheadrightarrow Q) \rhd \ulcorner P \rhd Q \urcorner$

# Separating implication examples

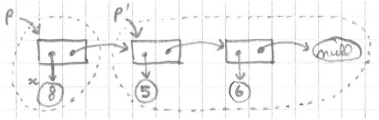
**Exercise:** Among the following heap entailments, which hold?

1. $P \rhd (Q \mathrel{-\!\!*} P * Q)$ yes: unfold and behold the definition of $*$
2. $(Q \mathrel{-\!\!*} P * Q) \rhd P$ no e.g. with $P = Q = \ulcorner\text{False}\urcorner$
3. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 3) \rhd \ulcorner\text{False}\urcorner$ no e.g. $\{(1,4)\}$ satisfies left
4. $(1 \mapsto 2) \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 8) \rhd 2 \mapsto 8$ no (same)
5. $\ulcorner\,\urcorner \mathrel{-\!\!*} P \rhd P$ yes, and...
6. $P \rhd \ulcorner\,\urcorner \mathrel{-\!\!*} P$ ...yes: $P$ and $\ulcorner\,\urcorner \mathrel{-\!\!*} P$ are equivalent
7. $\ulcorner\,\urcorner \rhd (P \mathrel{-\!\!*} Q \mathrel{-\!\!*} P * Q)$ yes: unfold $\rhd$ and obtain approx. (1)
8. $\ulcorner P \rhd Q \urcorner \rhd (P \mathrel{-\!\!*} Q)$ yes
9. $(P \mathrel{-\!\!*} Q) \rhd \ulcorner P \rhd Q \urcorner$ no, e.g. $P = \ulcorner\,\urcorner$ and $Q = 1 \mapsto 2$

# Chapter 19

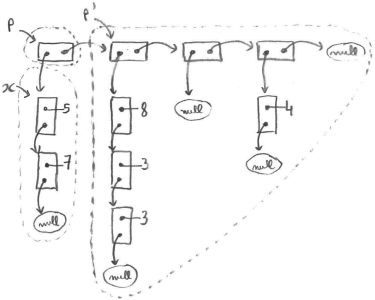
## Higher-order representation predicates and the access problem

# Specification of construction, for basic values



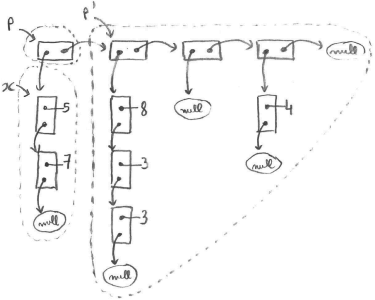

$$\{p' \rightsquigarrow \mathsf{MList}\, L\}\ (\mathsf{cons}\, x\, p')\ \{\lambda p.\ p \rightsquigarrow \mathsf{MList}\, (x :: L)\}$$

# Specification of construction



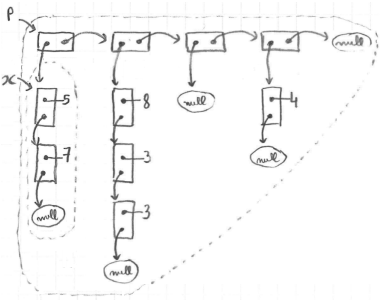

$$\{x \rightsquigarrow R\,X \;*\; p' \rightsquigarrow \mathsf{Mlistof}\,R\,L\}\ (\mathsf{cons}\,x\,p')\ \{\lambda p.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,(X :: L)\}$$

# Specification of deconstruction




$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}\ (\mathsf{uncons}\, p)$$
$$\{\lambda(x, p').\ x \rightsquigarrow R\, X\ *\ p' \rightsquigarrow \mathsf{Mlistof}\, R\, L\}$$

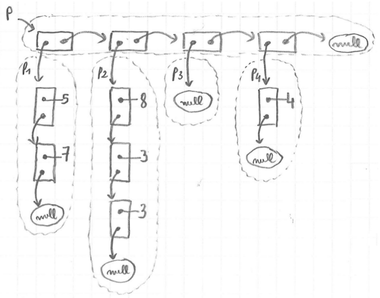
# Specification of accesses: the problem




Incorrect specification for `head`:

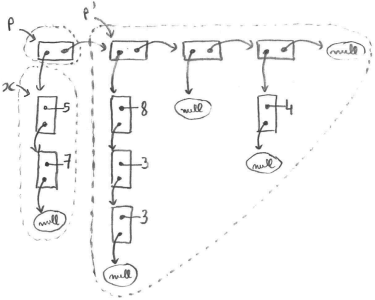
$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}\ (\mathsf{head}\, p)$$
$$\{\lambda x.\ x \rightsquigarrow R\, X\ *\ p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}$$

# Specification of accesses: a brute force solution




$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L \quad = \quad \exists K. \quad p \rightsquigarrow \mathsf{MList}\, K$$
$$* \quad \circledast_{i \,\in\, \{0,\ldots,|L|-1\}} \; (K[i]) \rightsquigarrow R\,(L[i])$$
$$* \quad \ulcorner |K| = |L| \urcorner$$

# Specification of accesses: focus before read

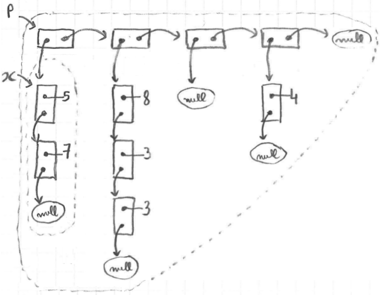



$$p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L) \;=\; \exists x p'.\quad p \rightsquigarrow \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\} \\ *\quad x \rightsquigarrow R\, X \\ *\quad p' \rightsquigarrow \mathsf{Mlistof}\, R\, L$$

Then read using:

$$\{p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}\ (p.\mathsf{hd})\ \{\lambda y.\ \ulcorner y = x \urcorner * p \mapsto \{\!|\mathsf{hd}{=}x;\ \mathsf{tl}{=}p'|\!\}\}$$

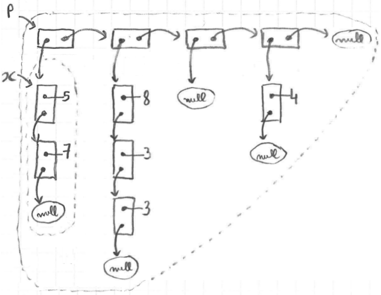
# Specification of accesses with separating implication




Correct specification with $-\!\!*$:

$$\{p \leadsto \mathsf{Mlistof}\, R\, (X :: L)\}\ (\mathtt{head}\, p)$$
$$\{\lambda x.\ x \leadsto R\, X\ *\ (x \leadsto R\, X\ -\!\!*\ \ p \leadsto \mathsf{Mlistof}\, R\, (X :: L))\}$$

# Specification of accesses with separating implication

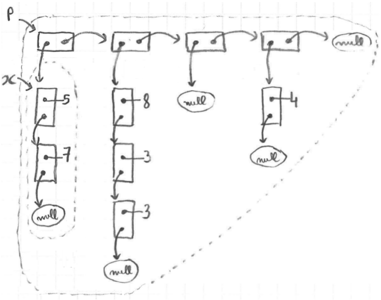



Correct specification with $-\!\!*$:

$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}\ (\texttt{head}\, p)$$
$$\{\lambda x.\ x \rightsquigarrow R\, X\ *\ (x \rightsquigarrow R\, X\ -\!\!*\ \ p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L))\}$$

**Exercise:** What problem is there, e.g. if $x \rightsquigarrow RX$ is $x \mapsto X$ (i.e. $R = \mathsf{Ref}$)?
**Exercise:** How to generalize the specification to solve this problem?

# Specification of accesses with separating implication




Correct specification with $-\!\!*$, generalized:

$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, (X :: L)\}\ (\texttt{head}\, p)$$
$$\{\lambda x.\ x \rightsquigarrow R\, X\ \ast\ (\forall X', x \rightsquigarrow R\, X'\ -\!\!\ast\ \ p \rightsquigarrow \mathsf{Mlistof}\, R\, (X' :: L))\}$$

# The copy problem

Incorrect specification for `copy`:

$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, L\}$$
$$(\mathsf{copy}\, p)$$
$$\{\lambda p'.\ p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ * \ p' \rightsquigarrow \mathsf{Mlistof}\, R\, L\}$$

# The copy problem

Incorrect specification for copy:

$$\{p \rightsquigarrow \mathsf{Mlistof}\, R\, L\}$$
$$(\mathtt{copy}\, p)$$
$$\{\lambda p'.\, p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ * \ p' \rightsquigarrow \mathsf{Mlistof}\, R\, L\}$$

**Exercise:** specify a function $\mathtt{copy}\, f\, p$ that duplicates a mutable list specified using Mlistof, where $f$ is a function to duplicate items.

# The copy problem

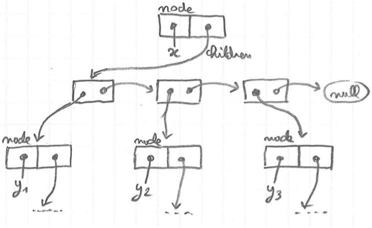
Incorrect specification for `copy`:

$$
\begin{aligned}
&\{p \rightsquigarrow \mathsf{Mlistof}\, R\, L\} \\
&(\mathsf{copy}\, p) \\
&\{\lambda p'.\ p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ *\ p' \rightsquigarrow \mathsf{Mlistof}\, R\, L\}
\end{aligned}
$$

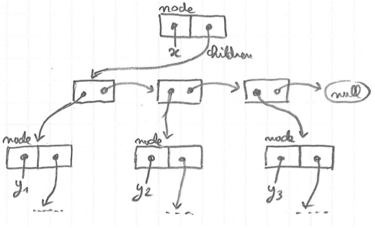
**Exercise:** specify a function `copy` $f\, p$ that duplicates a mutable list specified using Mlistof, where $f$ is a function to duplicate items.

$$
\begin{aligned}
&\big(\forall x X.\ \{x \rightsquigarrow R\, X\}\ (f\, x)\ \{\lambda x'.\ x \rightsquigarrow R\, X \ *\ x' \rightsquigarrow R\, X\}\big) \\
\Rightarrow\ &\{p \rightsquigarrow \mathsf{Mlistof}\, R\, L\} \\
&(\mathsf{copy}\, f\, p) \\
&\{\lambda p'.\ p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ *\ p' \rightsquigarrow \mathsf{Mlistof}\, R\, L\}
\end{aligned}
$$

# Example of combining higher-order predicates



# Example of combining higher-order predicates




$$
\begin{aligned}
&p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv \\
&\quad \mathsf{match}\, T\, \mathsf{with} \\
&\quad \mid \mathsf{Leaf} \Rightarrow \ulcorner p = \mathsf{null} \urcorner \\
&\quad \mid \mathsf{Node}\, X\, L \Rightarrow p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mlistof}\, (\mathsf{Narytreeof}\, R))\, L
\end{aligned}
$$

# Chapter 22

## Iteration with higher-order representation predicates

# Iteration on lists

Recall:

$$\forall f l I.\quad \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big)$$
$$\Rightarrow\quad \{I\,\mathsf{nil}\}\ (\mathtt{iter}\,f\,l)\ \{\lambda\_.\ I\,l\}$$

$$\forall f p l I.\quad \big(\forall x k.\ \{I\,k\}\ (f\,x)\ \{\lambda\_.\ I\,(k \& x)\}\big)$$
$$\Rightarrow\quad \{p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,\mathsf{nil}\}\ (\mathtt{miter}\,f\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{MList}\,l\ *\ I\,l\}$$

# Iteration on lists

Recall:

$$\forall f l I.\quad \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big)$$
$$\Rightarrow\quad \{I\ \mathsf{nil}\}\ (\mathtt{iter}\ f\ l)\ \{\lambda\_.\ I\,l\}$$

$$\forall f p l I.\quad \big(\forall x k.\ \{I\,k\}\ (f\ x)\ \{\lambda\_.\ I\,(k\&x)\}\big)$$
$$\Rightarrow\quad \{p \leadsto \mathsf{MList}\,l\ *\ I\ \mathsf{nil}\}\ (\mathtt{miter}\ f\ p)\ \{\lambda\_.\ p \leadsto \mathsf{MList}\,l\ *\ I\,l\}$$

Challenge:

$$\big(\forall x....\ \{...\}\ (f\ x)\ \{\lambda\_.\ ...\}\big)$$
$$\Rightarrow\quad \{p \leadsto \mathsf{Mlistof}\,R\,L\ *\ ...\}\ (\mathtt{miter}\ f\ p)\ \{\lambda\_.\ p \leadsto ...\ *\ ...\}$$

**Question:** Can we use an invariant $I \equiv \lambda K.(\dots)$?
(i.e. with a spec of the form $\{p \leadsto \dots * I\ \mathsf{nil}\}(\dots)\{p \leadsto \dots * I\ L\}$ ?)

# Iterating over a mutable list of mutable items

**Exercise:** specify the function `miter`, using an invariant of the form $J\,K\,K'$, 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 $J\,K\,K'$, describing the state before and the state after the iteration.

$$
\begin{array}{lll}
\forall f p R L J. & \big(\forall x X K K'. & \{x \rightsquigarrow R\,X \;*\; J\,K\,K'\} \\
 & & (f\,x) \\
 & & \{\lambda\_.\ \exists X'.\ x \rightsquigarrow R\,X' \;*\; J\,(K \& X)\,(K' \& X')\} \big) \\
\Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L \;*\; J\,\mathsf{nil}\,\mathsf{nil}\} & \\
 & (\mathtt{miter}\,f\,p) & \\
 & \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,L' \;*\; J\,L\,L'\} &
\end{array}
$$

# 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 } }
```

$$x \rightsquigarrow \mathsf{Ref}\, X \;\equiv\; x \mapsto X$$

**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 } }
```

$$x \leadsto \mathsf{Ref}\, X \;\; \equiv \;\; x \mapsto X$$

**Exercise:** using the representation predicates Ref and Mlistof, specify the function `(fun x -> incr x)` and `incr_all`. What is $J$ $K$ $K'$?

$$\{x \leadsto \mathsf{Ref}\, X\}\ (\texttt{incr x})\ \{\lambda\_.\ x \leadsto \mathsf{Ref}\,(X+1)\}$$

# 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 } }
```

$$x \rightsquigarrow \mathsf{Ref}\, X \;\equiv\; x \mapsto X$$

**Exercise:** using the representation predicates Ref and Mlistof, specify the function `(fun x -> incr x)` and `incr_all`. What is $J$ $K$ $K'$?

$$\{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{incr x})\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}$$

$$\{p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\,L\}\ (\texttt{incr\_all}\,p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\,\mathsf{Ref}\,(\mathsf{map}\,(+1)\,L)\}$$

# Incrementing a mutable list of distinct references (2/2)

$$
\begin{array}{ll}
\forall f p R L J. & \big(\forall x X K K'.\ \{x \rightsquigarrow R\, X \ * \ J\, K\, K'\} \\
& \qquad\qquad (f\, x) \\
& \qquad\qquad \{\lambda\_.\ \exists X'.\ x \rightsquigarrow R\, X' \ * \ J\, (K \& X)\, (K' \& X')\}\big) \\
\Rightarrow & \{p \rightsquigarrow \mathsf{Mlistof}\, R\, L \ * \ J\, \mathsf{nil}\, \mathsf{nil}\} \\
& (\texttt{miter}\, f\, p) \\
& \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\, R\, L' \ * \ J\, L\, L'\}
\end{array}
$$

Consider:

$$J\, K\, K' \ \equiv\ \ulcorner K' = \mathsf{map}\,(+1)\, K \urcorner$$

Derives:

$$\big(\forall x X.\ \{x \rightsquigarrow \mathsf{Ref}\, X\}\ (\texttt{fun x -> incr x})x\ \{\lambda\_.\ x \rightsquigarrow \mathsf{Ref}\,(X+1)\}\big) \ \Rightarrow$$
$$\{p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Ref}\, L\}\ (\texttt{incr\_all}\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlistof}\, \mathsf{Ref}\, (\mathsf{map}\,(+1)\, L)\}$$

# Chapter 23

## Resource analysis in Separation Logic

# Controlling deallocation

(1) Remove the "GC" part from the definition the triple $\{H\}\ t\ \{Q\}$:

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

(2) Replace the GC rule, add a "free" function for explicit deallocation:

$$\cancel{\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}} \qquad \leadsto \qquad \frac{}{\{r \mapsto \vec{v}\}\ \texttt{free}\ r\ \{\lambda\_.\ \ulcorner\,\urcorner\}}$$

# Controlling deallocation

(1) Remove the "GC" part from the definition the triple $\{H\}\ t\ \{Q\}$:

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

(2) Replace the GC rule, add a "free" function for explicit deallocation:

$$\cancel{\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}} \qquad \rightsquigarrow \qquad \frac{}{\{r \mapsto \vec{v}\}\ \texttt{free}\ r\ \{\lambda\_.\ \ulcorner\urcorner\}}$$

(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.

$$\{\ulcorner\urcorner\}\ t\ \{\lambda n.\ \ulcorner P\, n \urcorner\}$$

# Controlling deallocation

(1) Remove the "GC" part from the definition the triple $\{H\}\ t\ \{Q\}$:

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

(2) Replace the GC rule, add a "free" function for explicit deallocation:

$$\cancel{\frac{\{H\}\ t\ \{Q \star \mathsf{GC}\}}{\{H\}\ t\ \{Q\}}} \qquad \leadsto \qquad \frac{}{\{r \mapsto \vec{v}\}\ \texttt{free}\ r\ \{\lambda\_.\ \ulcorner\urcorner\}}$$

(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.

$$\{\ulcorner\urcorner\}\ t\ \{\lambda n.\ \ulcorner P\, n \urcorner\}$$

(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.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\, char})$ denotes the remaining bytes to read.

# File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \leadsto \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

$$\{\ulcorner\urcorner\}\ \ (\texttt{fopen s})\ \ \{\lambda f.\ \exists L.\ f \leadsto \mathsf{File}\, L\}$$

# File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\,L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\,char})$ denotes the remaining bytes to read.

$$\{\ulcorner\urcorner\}\ \ (\texttt{fopen s})\ \ \{\lambda f.\ \exists L.\ f \rightsquigarrow \mathsf{File}\,L\}$$
$$\{f \rightsquigarrow \mathsf{File}\,(c :: L)\}\ \ (\texttt{fread f})\ \ \{\lambda x.\ \ulcorner x = c\urcorner \ast f \rightsquigarrow \mathsf{File}\,L\}$$

# File handle protocols

Goal: ensure that if a file is open then it is eventually closed.

$$f \rightsquigarrow \mathsf{File}\, L$$

where $(f : \mathsf{loc})$ denotes the file handler,
and $(L : \mathsf{list\, char})$ denotes the remaining bytes to read.

$$\begin{array}{rl} \{\ulcorner\urcorner\} & (\texttt{fopen s}) \ \{\lambda f.\ \exists L.\ f \rightsquigarrow \mathsf{File}\, L\} \\ \{f \rightsquigarrow \mathsf{File}\,(c :: L)\} & (\texttt{fread f}) \ \{\lambda x.\ \ulcorner x = c \urcorner \ * \ f \rightsquigarrow \mathsf{File}\, L\} \\ \{f \rightsquigarrow \mathsf{File}\, L\} & (\texttt{fclose f}) \ \{\lambda \_.\ \ulcorner\urcorner\} \end{array}$$

# Complexity analysis

Time credits:

$$\$\,x \;:\; \mathsf{Hprop} \qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) = \$\,x \;\ast\; \$\,y \quad \text{and} \quad \$\,0 = \ulcorner\urcorner$$

# Complexity analysis

Time credits:

$$\$\,x \;:\; \text{Hprop} \qquad\qquad \text{where } x \in \mathbb{R}^+$$

Properties:

$$\$(x+y) = \$\,x \;*\; \$\,y \qquad \text{and} \qquad \$\,0 = \ulcorner\urcorner$$

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:

$$\{t \rightsquigarrow \mathsf{Array}\, M \ * \ \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\, \ulcorner n = |M| \urcorner * t \rightsquigarrow \mathsf{Array}\, M\}$$

# Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \mathsf{Array}\, M \ * \ \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\ \ulcorner n = |M| \urcorner * t \rightsquigarrow \mathsf{Array}\, M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\}\ (\texttt{Array.make n v})\ \{\lambda t.\ \exists L.\ t \rightsquigarrow \mathsf{Array}\, L \ * \ \ulcorner \ldots \urcorner\}$$

# Time credits in pre-conditions

Constant time:

$$\{t \rightsquigarrow \mathsf{Array}\, M \ * \ \$\, c\}\ (\texttt{Array.length t})\ \{\lambda n.\ \ulcorner n = |M| \urcorner * t \rightsquigarrow \mathsf{Array}\, M\}$$

Linear time:

$$\{\$(c_1 n + c_2)\}\ (\texttt{Array.make n v})\ \{\lambda t.\ \exists L.\ t \rightsquigarrow \mathsf{Array}\, L \ * \ \ulcorner ... \urcorner\}$$

Quasilinear time:

$$\{t \rightsquigarrow \mathsf{Array}\, L \ * \ \$(c_1 |L| \log |L| + c_2)\}$$
$$(\texttt{Array.sort t})$$
$$\{\lambda t.\ \exists L'.\ t \rightsquigarrow \mathsf{Array}\, L' \ * \ \ulcorner ... \urcorner\}$$

# Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$\{\$\, c\}$ (`Stack.create()`) $\{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\, \mathsf{nil}\}$

$\{s \rightsquigarrow \mathsf{Stack}\, L \ast \$\, c\}$ (`Stack.push s x`) $\{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\, (x :: L)\}$

$\{s \rightsquigarrow \mathsf{Stack}\, (x :: L) \ast \$\, c\}$ (`Stack.pop s`) $\{\lambda y.\, \ulcorner y = x \urcorner \ast s \rightsquigarrow \mathsf{Stack}\, L$

# Amortized analysis

Stack of unbounded size with amortized constant-time operations:

$\{\$\, c\}$ $(\texttt{Stack.create()})$ $\{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\, \mathsf{nil}\}$

$\{s \rightsquigarrow \mathsf{Stack}\, L \ast \$\, c\}$ $(\texttt{Stack.push s x})$ $\{\lambda\_.\, s \rightsquigarrow \mathsf{Stack}\, (x :: L)\}$

$\{s \rightsquigarrow \mathsf{Stack}\, (x :: L) \ast \$\, c\}$ $(\texttt{Stack.pop s})$ $\{\lambda y.\, \ulcorner y = x \urcorner \ast s \rightsquigarrow \mathsf{Stack}\, L$

Representation predicate with a potential function:

$$
\begin{aligned}
s \rightsquigarrow \mathsf{Stack}\, L \quad \equiv \quad \exists n t M k. \quad & s \mapsto \{|\mathsf{size}{=}n;\ \mathsf{data}{=}t|\} \\
\ast\ & t \rightsquigarrow \mathsf{Array}\, M \\
\ast\ & \ulcorner n = |L| \leqslant |M| = 2^k \urcorner \\
\ast\ & \ulcorner \forall i \in [0, n).\ M[i] = L[i] \urcorner \\
\ast\ & \$(c' \cdot \mathsf{abs}(n - |M|/2))
\end{aligned}
$$

# Space complexity analysis

Suppose $\Diamond 1$ represents one *space credit* (Hoffmann, 1999)
Then allocation consumes credits; deallocation produces credits :

$$\{\Diamond \mathsf{size(b)}\} \quad \mathsf{alloc}(b) \quad \{\lambda x.\ x \mapsto b\}$$

$$\{x \mapsto b\} \quad \mathsf{free}(x) \quad \{\lambda\_.\ \Diamond \mathsf{size(b)}\}$$

The same mechanisms apply (e.g. $\Diamond(a+b) = \Diamond a * \Diamond b$, amortized analysis, etc.).

# Space complexity analysis

Suppose $\diamondsuit 1$ represents one *space credit* (Hoffmann, 1999)
Then allocation consumes credits; deallocation produces credits :

$$\{\diamondsuit \mathsf{size(b)}\} \quad \mathsf{alloc}(b) \quad \{\lambda x.\ x \mapsto b\}$$

$$\{x \mapsto b\} \quad \mathsf{free}(x) \quad \{\lambda\_.\ \diamondsuit \mathsf{size(b)}\}$$

The same mechanisms apply (e.g. $\diamondsuit(a+b) = \diamondsuit a * \diamondsuit b$, amortized analysis, etc.).

What if we add **garbage collection** ?

- M., Pottier, 2022
  *A separation logic for heap space under garbage collection.*
- Moine, Charguéraud, Pottier, 2023
  *A high-level separation logic for heap space under garbage collection.*

# Fractional permissions

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with} \;\; 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) \;*\; (r \overset{1/2}{\mapsto} v)$$

More generally, if $0 < \alpha, \beta \leqslant 1$:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} v)$$

# Fractional permissions

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with} \;\; 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) \;*\; (r \overset{1/2}{\mapsto} v)$$

More generally, if $0 < \alpha, \beta \leqslant 1$:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} v)$$

Values must agree: if $0 < \alpha, \beta \leqslant 1$:

$$\Big((r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} w)\Big) \;\rhd\; \Big((r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} w) * \ulcorner v = w \urcorner\Big)$$

# Fractional permissions

$$(r \overset{\alpha}{\mapsto} v) \qquad \text{with} \;\; 0 < \alpha \leqslant 1$$

Splitting and merging:

$$(r \mapsto v) \;=\; (r \overset{1}{\mapsto} v) \;=\; (r \overset{1/2}{\mapsto} v) \;*\; (r \overset{1/2}{\mapsto} v)$$

More generally, if $0 < \alpha, \beta \leqslant 1$:

$$(r \overset{\alpha+\beta}{\mapsto} v) \;=\; (r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} v)$$

Values must agree: if $0 < \alpha, \beta \leqslant 1$:

$$\Big((r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} w)\Big) \;\;\rhd\;\; \Big((r \overset{\alpha}{\mapsto} v) \;*\; (r \overset{\beta}{\mapsto} w) * \ulcorner v = w \urcorner\Big)$$

Operations:

$$\{\ulcorner\urcorner\} \;\; (\texttt{ref v}) \;\; \{\lambda r.\; r \overset{1}{\mapsto} v\}$$

$$\{r \overset{1}{\mapsto} v'\} \; (\texttt{r := v}) \; \{\lambda\_.\; r \overset{1}{\mapsto} v\}$$

$$\forall \alpha. \qquad \{r \overset{\alpha}{\mapsto} v\} \qquad (\texttt{!r}) \qquad \{\lambda x.\; \ulcorner x = v \urcorner * (r \overset{\alpha}{\mapsto} v)\}$$

# Fractional permissions in practice

$$
\begin{aligned}
\forall \alpha\beta.\ & \{a_1 \stackrel{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ * \ a_2 \stackrel{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\} \\
& (\texttt{concat}\, a_1\, a_2) \\
& \{\lambda a_3.\ a_1 \stackrel{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \ * \ a_2 \stackrel{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 * a_3 \stackrel{1}{\rightsquigarrow} \mathsf{Array}\, (L_1 +\!\!+ L_2)\}
\end{aligned}
$$

# Fractional permissions in practice

$$\forall \alpha\beta.\ \{a_1 \overset{\alpha}{\leadsto} \mathsf{Array}\, L_1 \ * \ a_2 \overset{\beta}{\leadsto} \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_1 \overset{\alpha}{\leadsto} \mathsf{Array}\, L_1 \ * \ a_2 \overset{\beta}{\leadsto} \mathsf{Array}\, L_2 * a_3 \overset{1}{\leadsto} \mathsf{Array}\, (L_1 +\!\!+ L_2)\}$$

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 $\mathsf{RO}(H)$.

# 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 let $((),()) = (|t_1, t_2|)$ in $()$.)

Computing: $a[i] + a[i+1] + ... + a[j-1]$.

```
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:
$f(t[0]) \oplus f(a[1]) \oplus ... \oplus f(a[n-1])$.
(on which condition on $\oplus$?)

# Reasoning rule for parallel pairs

$$\frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 * H_2\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \textsc{parallel}$$

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

$$\frac{\{H_1\}\ t_1\ \{Q_1\} \qquad \{H_2\}\ t_2\ \{Q_2\}}{\{H_1 * H_2\}\ (|t_1, t_2|)\ \{Q_1 \star Q_2\}}\ \textsc{parallel}$$

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 $p \leadsto \mathsf{Lock}\,H$ asserts that lock $p$ protects an invariant $H$.
Here:

$$p \leadsto \mathsf{Lock}\,(\exists i.\ (r \mapsto i) \ * \ (s \mapsto n - i))$$

# Concurrent locks: specification of operations

Duplicable representation predicate:

$$p \rightsquigarrow \mathsf{Lock}\, H$$

Operations:

$$\forall H. \quad \{H\}\ (\texttt{create\_lock ()})\ \{\lambda p.\ p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$\forall pH. \quad \{p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{acquire\_lock p})\ \{\lambda\_.\ H * p \rightsquigarrow \mathsf{Lock}\, H\}$$

$$\forall pH. \{H\ *\ p \rightsquigarrow \mathsf{Lock}\, H\}\ (\texttt{release\_lock p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Lock}\, H\}$$

# Concurrent locks: example

$$\forall H. \quad \{H\}\ (\texttt{create\_lock ()})\ \{\lambda p.\ p \leadsto \mathsf{Lock}\, H\}$$

$$\forall pH. \quad \{p \leadsto \mathsf{Lock}\, H\}\ (\texttt{acquire\_lock p})\ \{\lambda\_.\ H * p \leadsto \mathsf{Lock}\, H\}$$

$$\forall pH. \{H\ *\ p \leadsto \mathsf{Lock}\, H\}\ (\texttt{release\_lock p})\ \{\lambda\_.\ p \leadsto \mathsf{Lock}\, H\}$$

```
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

```
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: $\ulcorner\urcorner$.  2: $r \mapsto 0$.  3: $r \mapsto 0 * s \mapsto n$.
4: $p \rightsquigarrow \mathsf{Lock}\,(\exists i.\ (r \mapsto i) * (s \mapsto n - i))$.
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);
```

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n.\ r \mapsto n * \ulcorner\ \ldots\ ?\ \ldots\ \urcorner)$$

# 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);
```

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n.\ r \mapsto n * \ulcorner\ \ldots\ ?\ \ldots\ \urcorner)$$

**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;`<br>`r := !r + 1;`<br>`r1 := !r1 + 1;`<br>`release_lock p;` | ‖ | `acquire_lock p;`<br>`r := !r + 1;`<br>`r2 := !r2 + 1;`<br>`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.

$$p \rightsquigarrow \mathsf{Lock}\,(\exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner)$$

# Proof

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

# Proof

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

# Proof

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

```
let p = create_lock()
```

# Proof

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

```
let p = create_lock()
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

# Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

```
let p = create_lock()
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

$\{(p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0) * (p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0)\}$

# Proof

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
```

$\{r \mapsto 0 * r_1 \mapsto 0 * r_2 \mapsto 0\}$

$\{H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

```
let p = create_lock()
```

$\{p \rightsquigarrow \mathsf{Lock}\ H * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0\}$

$\{(p \rightsquigarrow \mathsf{Lock}\ H * r_1 \overset{1/2}{\mapsto} 0) * (p \rightsquigarrow \mathsf{Lock}\ H * r_2 \overset{1/2}{\mapsto} 0)\}$

| | | |
|---|---|---|
| $\{p \rightsquigarrow \mathsf{Lock}\ H * r_1 \overset{1/2}{\mapsto} 0\}$ | $\Vert$ | $\{p \rightsquigarrow \mathsf{Lock}\ H * r_2 \overset{1/2}{\mapsto} 0\}$ |
| `acquire_lock p;` | | `acquire_lock p;` |
| `r := !r + 1;` | | `r := !r + 1;` |
| `...` | | `...` |

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \leadsto \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \leadsto \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \leadsto \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * H\}$ (choosing $n_1 = 1$ and $n_2 = n_2$)

# Left thread

$$H \quad \equiv \quad \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * H\}$ (choosing $n_1 = 1$ and $n_2 = n_2$)

```
release_lock p;
```

# Left thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * H\}$ so, for some $n, n_1, n_2$ s. that $n = n_1 + n_2$:

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r := !r + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2\}$ so $n_1 = 0$, and

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 0 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

```
r1 := !r1 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1}{\mapsto} 1 * r \mapsto n + 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r \mapsto n + 1 * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * H\}$ (choosing $n_1 = 1$ and $n_2 = n_2$)

```
release_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1\}$

# Right thread

$$H \equiv \exists n, n_1, n_2.\ r \mapsto n * (r_1 \overset{1/2}{\mapsto} n_1) * (r_2 \overset{1/2}{\mapsto} n_2) * \ulcorner n = n_1 + n_2 \urcorner$$

$\{p \rightsquigarrow \mathsf{Lock}\ H * r_2 \overset{1/2}{\mapsto} 0\}$

```
acquire_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\ H * r_2 \overset{1/2}{\mapsto} 0 * H\}$

```
r := !r + 1;
r2 := !r2 + 1;
```

$\{p \rightsquigarrow \mathsf{Lock}\ H * r_2 \overset{1/2}{\mapsto} 1 * H\}$

```
release_lock p;
```

$\{p \rightsquigarrow \mathsf{Lock}\ H * r_2 \overset{1/2}{\mapsto} 1\}$

# Finish up

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let p = create_lock()
```

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 0\}$ ‖ $\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 0\}$

| | |
|---|---|
| `acquire_lock p;` | `acquire_lock p;` |
| `r := !r + 1;` | `r := !r + 1;` |
| `r1 := !r1 + 1;` | `r2 := !r2 + 1;` |
| `release_lock p;` | `release_lock p;` |

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1\}$ ‖ $\{p \rightsquigarrow \mathsf{Lock}\, H * r_2 \overset{1/2}{\mapsto} 1\}$

$\{p \rightsquigarrow \mathsf{Lock}\, H * r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1\}$

```
acquire_lock p;
```

$\{r_1 \overset{1/2}{\mapsto} 1 * r_2 \overset{1/2}{\mapsto} 1 * r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner n = n_1 + n_2 \urcorner\}$

$\{r_1 \mapsto 1 * r_2 \mapsto 1 * r \mapsto n * \ulcorner n = 1 + 1 \urcorner\}$

```
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":

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \gamma \rightsquigarrow \mathsf{Ghost}(a)$$

# 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":

$$\mathsf{True} \Rrightarrow \exists \gamma.\, \gamma \rightsquigarrow \mathsf{Ghost}(a)$$

- splitting of ghost state: for any $a, b$ in an RA:

$$\gamma \rightsquigarrow \mathsf{Ghost}(a \cdot b) \Leftrightarrow \gamma \rightsquigarrow \mathsf{Ghost}(a) * \gamma \rightsquigarrow \mathsf{Ghost}(b)$$

# 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":

$$\mathsf{True} \Rrightarrow\!\!\ast\; \exists \gamma.\, \gamma \rightsquigarrow \mathsf{Ghost}(a)$$

- splitting of ghost state: for any $a, b$ in an RA:

$$\gamma \rightsquigarrow \mathsf{Ghost}(a \cdot b) \Leftrightarrow \gamma \rightsquigarrow \mathsf{Ghost}(a) \ast \gamma \rightsquigarrow \mathsf{Ghost}(b)$$

- validity in RA's e.g. $\mathsf{valid}((r \mapsto n_1) \cdot (r \mapsto n_2)) \;\Rightarrow\; n_1 = n_1$
- 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:

$$\{P\}e\{\Phi\} \;\equiv\; P \mathrel{-\!\!*} \mathsf{wp}\, e\, \Phi$$

- preconditions become hypotheses, more easily managed

# Presentation with weakest preconditions

WP are generally preferred to triples, in practice:

- more primitive:

$$\{P\}e\{\Phi\} \;\equiv\; P \twoheadrightarrow* \mathsf{wp}\, e\, \Phi$$

- preconditions become hypotheses, more easily managed

Iris hypotheses go in contexts and can be named, split, rearranged...

```
                                            H1 : P1
                      H : P1 ∗ P2           H2 : P2
P1 ∗ P2 -∗ wp e Φ  →  ----------∗     →     ------∗
                      wp e Φ                wp e Φ
```

# Builtin consequence rule in postconditions

$\mathsf{wp}\, e \cdot$ is a monotone predicate transformer, so $\mathsf{wp}\, e\, \Phi$ is equivalent to

$$\forall \Psi\ (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi$$

# Builtin consequence rule in postconditions

$\mathsf{wp}\, e \cdot$ is a monotone predicate transformer, so $\mathsf{wp}\, e\, \Phi$ is equivalent to

$$\forall \Psi\ (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi$$

So instead of choosing

$$\{P\}e\{\Phi\}\ \equiv\ P \mathrel{-\!\!*} \mathsf{wp}\, e\, \Phi$$

the following formulation is preferred:

$$\{P\}e\{\Phi\}\ \equiv\ \forall \Psi\ P \mathrel{-\!\!*} (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi$$

Specifications can be applied directly since $\Psi$ is universally quantified.

Note that separating implication $\mathrel{-\!\!*}$ has no easy "heap entailment" ▻ counterpart here.

# Simplified rules for load, store, alloc

**Exercise:**

$$
\begin{array}{llll}
\forall \ell v \Phi & \ldots\ldots\ldots\ldots & -\!\!* \; (\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots) & -\!\!* \; \mathsf{wp}\,(!\ell)\,\Phi \\
\forall \ell v v' \Phi & \ldots\ldots\ldots\ldots & -\!\!* \; (\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots) & -\!\!* \; \mathsf{wp}\,(\ell \leftarrow v)\,\Phi \\
\forall v \Phi & \ldots\ldots\ldots\ldots & -\!\!* \; (\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots) & -\!\!* \; \mathsf{wp}\,(\texttt{ref}\; v)\,\Phi
\end{array}
$$

# Simplified rules for load, store, alloc

$$
\begin{array}{llll}
\forall \ell v v' \Phi & \ell \mapsto v' & \mathrel{-\!\!*} (\ell \mapsto v \mathrel{-\!\!*} \Phi()) & \mathrel{-\!\!*} \mathsf{wp}\,(\ell \leftarrow v)\,\Phi \\
\forall v \Phi & \top & \mathrel{-\!\!*} (\forall \ell \;\; \ell \mapsto v \mathrel{-\!\!*} \Phi \ell) & \mathrel{-\!\!*} \mathsf{wp}\,(\mathtt{ref}\,v)\,\Phi \\
\forall \ell v \Phi & \ell \mapsto v & \mathrel{-\!\!*} (\ell \mapsto v \mathrel{-\!\!*} \Phi v) & \mathrel{-\!\!*} \mathsf{wp}\,(!\ell)\,\Phi
\end{array}
$$

# Exemple

$$\{\ell \mapsto 1\}\texttt{ref}\,3\{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

# Exemple

$$\{\ell \mapsto 1\}\texttt{ref}\,3\{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi (\ell \mapsto 1) \mathrel{-\!\!*} (\forall \ell'.\ell \mapsto 1 * \ell' \mapsto 3 \mathrel{-\!\!*} \Phi \ell') \mathrel{-\!\!*} \mathsf{wp}\,(\texttt{ref}\,3)\,\Phi$$

# Exemple

$$\{\ell \mapsto 1\}\texttt{ref}\, 3\{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi (\ell \mapsto 1) \mathrel{-\!\!*} (\forall \ell'.\ell \mapsto 1 * \ell' \mapsto 3 \mathrel{-\!\!*} \Phi \ell') \mathrel{-\!\!*} \mathsf{wp}\, (\texttt{ref}\, 3)\, \Phi$$

after `iIntros (Φ) "Hl HΦ"` you have:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l' : loc, l ↦ #1 ∗ l' ↦ #3 -∗ Φ #l'
--------------------------------------∗
WP ref #3 {{ v, Φ v }}
```

# Exemple

$$\{\ell \mapsto 1\}\texttt{ref}\,3\{\ell'.\ \ell \mapsto 1 * \ell' \mapsto 3\}$$

in other words:

$$\forall \Phi (\ell \mapsto 1) -\!\!* (\forall \ell'.\ell \mapsto 1 * \ell' \mapsto 3 -\!\!* \Phi \ell') -\!\!* \mathsf{wp}\,(\texttt{ref}\,3)\,\Phi$$

after `iIntros (Φ) "Hl HΦ"` you have:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l' : loc, l ↦ #1 ∗ l' ↦ #3 -∗ Φ #l'
--------------------------------------∗
WP ref #3 {{ v, Φ v }}
```

applying the rule for allocation, we get:

```
"Hl" : l ↦ #1
"HΦ" : ∀ l' : loc, l ↦ #1 ∗ l' ↦ #3 -∗ Φ #l'
--------------------------------------∗
∀ l0 : loc, l0 ↦ #3 -∗ Φ #l0
```

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. ⚠

# One-shot, shallow or deep, effect handlers

$$\frac{\Phi(v)}{ewp_{\mathcal{E}}\ v\ \langle\Psi\rangle\{\Phi\}}\ \textsc{Value} \qquad \frac{\Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{\Phi\}}{ewp_{\mathcal{E}}\ (\mathsf{do}\ v)\ \langle\Psi\rangle\{\Phi\}}\ \textsc{Do}$$

$$\frac{ewp_{\mathcal{E}_1}\ e\ \langle\Psi_1\rangle\{\Phi_1\} \qquad \mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad \Psi_1 \sqsubseteq \Psi_2 \qquad \forall v.\, \Phi_1(v) \mathrel{-\!\!*} \Phi_2(v)}{ewp_{\mathcal{E}_2}\ e\ \langle\Psi_2\rangle\{\Phi_2\}}\ \textsc{Monotonicity}$$

$$\frac{ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{v.\, ewp_{\mathcal{E}}\ N[v]\ \langle\Psi\rangle\{\Phi\}\}}{ewp_{\mathcal{E}}\ N[e]\ \langle\Psi\rangle\{\Phi\}}\ \textsc{Bind} \qquad \frac{ewp_{\mathcal{E}}\ e\ \langle\bot\rangle\{v.\, ewp_{\mathcal{E}}\ K[v]\ \langle\Psi\rangle\{\Phi\}\}}{ewp_{\mathcal{E}}\ K[e]\ \langle\Psi\rangle\{\Phi\}}\ \textsc{Bind-Pure}$$

$$\frac{\begin{array}{c} ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\} \\ \forall v.\ \Phi(v) \mathrel{-\!\!*} \triangleright\, ewp_{\mathcal{E}}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\} \end{array} \qquad \forall v, k. \begin{cases} \Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{w.\ ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi\rangle\{\Phi\}\} \mathrel{-\!\!*} \\ \triangleright\, ewp_{\mathcal{E}}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\} \end{cases}}{ewp_{\mathcal{E}}\ (\mathsf{shallow\text{-}try}\ e\ \mathsf{with}\ h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}$$

$$\frac{\begin{array}{c} ewp_{\mathcal{E}}\ e\ \langle\Psi\rangle\{\Phi\} \\ \forall v.\ \Phi(v) \mathrel{-\!\!*} \triangleright\, ewp_{\mathcal{E}}\ (r\ v)\ \langle\Psi'\rangle\{\Phi'\} \end{array} \qquad \forall v, k. \begin{cases} \Psi\ \mathit{allows}\ \mathsf{do}\ v\ \{w.\ \forall\Psi'', \Phi''. \\ \quad \triangleright\, \mathit{deep\text{-}handler}_{\mathcal{E}}\ \langle\Psi\rangle\{\Phi\}\ h \mid r\ \langle\Psi''\rangle\{\Phi''\} \mathrel{-\!\!*} \\ \quad ewp_{\mathcal{E}}\ (k\ w)\ \langle\Psi''\rangle\{\Phi''\} \\ \} \mathrel{-\!\!*} \\ \triangleright\, ewp_{\mathcal{E}}\ (h\ v\ k)\ \langle\Psi'\rangle\{\Phi'\} \end{cases}}{ewp_{\mathcal{E}}\ (\mathsf{deep\text{-}try}\ e\ \mathsf{with}\ h \mid r)\ \langle\Psi'\rangle\{\Phi'\}}$$

# Chapter 27: modalities

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or “persistent”)

$$\square P \vdash P$$

# Persistently modality $\Box$

$\Box P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\Box P \vartriangleright P \qquad \Box P \vartriangleright \Box P * P$$

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

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

# Persistently modality $\Box$

$\Box P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\Box P \vdash P \qquad \Box P \vdash \Box P * P \qquad \Box P \vdash \Box P * \Box P$$

$$\Box P \vdash \Box P * P * P$$

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\square P \vartriangleright P \qquad \square P \vartriangleright \square P * P \qquad \square P \vartriangleright \square P * \square P$$

$$\square P \vartriangleright \square P * P * P \qquad \ulcorner P \urcorner \vartriangleright \square \ulcorner P \urcorner$$

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\square P \vartriangleright P \qquad \square P \vartriangleright \square P * P \qquad \square P \vartriangleright \square P * \square P$$

$$\square P \vartriangleright \square P * P * P \qquad \ulcorner P \urcorner \vartriangleright \square \ulcorner P \urcorner \qquad \square(\ell \mapsto 1) \vartriangleright \ulcorner \mathsf{False} \urcorner$$

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or “persistent”)

$$\square P \vartriangleright P \qquad \square P \vartriangleright \square P * P \qquad \square P \vartriangleright \square P * \square P$$

$$\square P \vartriangleright \square P * P * P \qquad \ulcorner P \urcorner \vartriangleright \square \ulcorner P \urcorner \qquad \square(\ell \mapsto 1) \vartriangleright \ulcorner \text{False} \urcorner$$

$$\square(\ell \overset{q}{\mapsto} 1) \vartriangleright \ulcorner \text{q=0} \urcorner \text{ (if even allowed)}$$

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P \text{ is duplicable} \urcorner$ (or "persistent")

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

$$\square P \vdash \square P * P * P \qquad \ulcorner P \urcorner \vdash \square \ulcorner P \urcorner \qquad \square(\ell \mapsto 1) \vdash \ulcorner \text{False} \urcorner$$

$$\square(\ell \overset{q}{\mapsto} 1) \vdash \ulcorner \text{q=0} \urcorner \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \;\equiv\; \square(\forall \Psi\;\, P \mathrel{-\!\!*} (\forall v\;\, \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

# Persistently modality $\square$

$\square P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\square P \vartriangleright P \qquad \square P \vartriangleright \square P * P \qquad \square P \vartriangleright \square P * \square P$$

$$\square P \vartriangleright \square P * P * P \qquad \ulcorner P \urcorner \vartriangleright \square \ulcorner P \urcorner \qquad \square(\ell \mapsto 1) \vartriangleright \ulcorner \text{False} \urcorner$$

$$\square(\ell \overset{q}{\mapsto} 1) \vartriangleright \ulcorner \text{q=0} \urcorner \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \;\equiv\; \square(\forall \Psi\ P \mathrel{-\!\!*} (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

Persistent resources can be shared between threads:

$$\frac{\{P_1 * \square P\}\ e_1\ \{Q_1\} \qquad \{P_2 * \square P\}\ e_2\ \{Q_2\}}{\{P_1 * P_2 * \square P\}\ (|e_1, e_2|)\ \{Q_1 \star Q_2\}}$$

# Persistently modality $\Box$

$\Box P$ is roughly equivalent to $P * \ulcorner P$ is duplicable$\urcorner$ (or "persistent")

$$\Box P \vdash P \qquad \Box P \vdash \Box P * P \qquad \Box P \vdash \Box P * \Box P$$

$$\Box P \vdash \Box P * P * P \qquad \ulcorner P \urcorner \vdash \Box \ulcorner P \urcorner \qquad \Box(\ell \mapsto 1) \vdash \ulcorner \mathsf{False} \urcorner$$

$$\Box(\ell \overset{q}{\mapsto} 1) \vdash \ulcorner \mathsf{q{=}0} \urcorner \text{ (if even allowed)}$$

$$\{P\}e\{\Phi\} \;\equiv\; \Box(\forall \Psi\ P \mathrel{-\!\!*} (\forall v\ \Phi v \mathrel{-\!\!*} \Psi v) \mathrel{-\!\!*} \mathsf{wp}\, e\, \Psi)$$

Persistent resources can be shared between threads:

$$\frac{\{P_1 * \Box P\}\ e_1\ \{Q_1\} \qquad \{P_2 * \Box P\}\ e_2\ \{Q_2\}}{\{P_1 * P_2 * \Box P\}\ (|e_1, e_2|)\ \{Q_1 \star Q_2\}}$$

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 ($\triangleright P$), $\vartriangleright$ is a binary predicate ($P \vartriangleright Q$)

# Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "$P$ holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

# Later modality $\triangleright$

$\triangleright P$ ("later P") can be thought as "$P$ holds after one reduction step".

$\triangleright$ is a modality ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

$\triangleright$ and $\Box$ are modalities in Iris, where "`iProp`" are not "`heap -> 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 ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

$\triangleright$ and $\square$ are modalities in Iris, where "`iProp`" are not "`heap -> 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 ($\triangleright P$), $\rhd$ is a binary predicate ($P \rhd Q$)

$$P \vdash \triangleright P \qquad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \triangleright (P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \text{etc}$$

$\triangleright$ and $\square$ are modalities in Iris, where "`iProp`" are not "`heap -> Prop`" and have features such as step indexing, and resources are not only heaps.

$$\triangleright^n \text{False} \vdash P \text{ iff } P \text{ holds for } n \text{ steps} \qquad \vdash \exists k.\ \triangleright^k \text{ False}$$

# 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:

$$\frac{\{P\}e'\{Q\} \qquad e \to e'}{\{\triangleright P\}e\{Q\}}$$

# 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:

$$\frac{\{P\}e'\{Q\} \qquad e \rightarrow e'}{\{\triangleright P\}e\{Q\}}$$

Final definition of triples:

$$\{P\}e\{\Phi\} \;\equiv\; \Box(\forall \Psi\; P \twoheadrightarrow \triangleright(\forall v\; \Phi v \twoheadrightarrow \Psi v) \twoheadrightarrow \mathsf{wp}\, e\, \Psi)$$

Complete set of rules: Lecture Notes on Iris

# Invariants

New construct $\boxed{R}^{\iota}$: duplicable, but the resource $R$ is lost at allocation:

$$\boxed{R}^{\iota} \vdash \Box \boxed{R}^{\iota}$$

$$\frac{S, \boxed{R}^{\iota} \vdash \{P\} e \{Q\}}{S \vdash \{\triangleright R * P\} e \{Q\}} \text{ INV-ALLOC}$$

# Invariants

New construct $\boxed{R}^{\iota}$: duplicable, but the resource $R$ is lost at allocation:

$$\boxed{R}^{\iota} \vdash \Box \boxed{R}^{\iota}$$

$$\frac{S, \boxed{R}^{\iota} \vdash \{P\} e \{Q\}}{S \vdash \{\triangleright R * P\} e \{Q\}} \text{ INV-ALLOC}$$

Invariant resources can be accessed but must be preserved:

$$\frac{e \text{ is atomic} \qquad S, \boxed{R}^{\iota} \vdash \{\triangleright R * P\} e \{\triangleright R * Q\}}{S, \boxed{R}^{\iota} \vdash \{P\} e \{Q\}} \text{ INV-OPEN}$$

(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:

$$p \rightsquigarrow \mathsf{Lock}\,(R, \gamma) \equiv \exists \iota. \boxed{p \mapsto \texttt{true} \;\vee\; p \mapsto \texttt{false} * R * \gamma \rightsquigarrow \mathsf{Ghost}(K)}^{\iota}$$

with resource algebra $(\varepsilon, K, \bot)$ s.t. $x \cdot \varepsilon = \varepsilon \cdot x = x$ otherwise $x \cdot y = \bot$.

# 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:

$$p \rightsquigarrow \mathsf{Lock}\,(R,\gamma) \equiv \exists\iota.\, \boxed{p \mapsto \mathsf{true} \;\vee\; p \mapsto \mathtt{false} * R * \gamma \rightsquigarrow \mathsf{Ghost}(K)}^{\iota}$$

with resource algebra $(\varepsilon, K, \bot)$ s.t. $x \cdot \varepsilon = \varepsilon \cdot x = x$ otherwise $x \cdot y = \bot$.

$$\forall R. \quad \{R\}\ (\mathtt{create\_lock\ ()})\ \{\lambda p.\ \exists\gamma.\, p \rightsquigarrow \mathsf{Lock}\,(R,\gamma)\}$$

$$\forall pR. \quad \{p \rightsquigarrow \mathsf{Lock}\,(R,\gamma)\}\ (\mathtt{acquire\_lock\ p})\ \{\lambda\_.\ R * p \rightsquigarrow \mathsf{Lock}\,(R,\gamma)\}$$

$$\forall pR.\ \{R * p \rightsquigarrow \mathsf{Lock}\,(R,\gamma)\}\ (\mathtt{release\_lock\ p})\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Lock}\,(R,\gamma)\}$$

# Prophecy variables

Allows to talk about a future, unknown, value
Invented by Abadi and Lamport [1988, 1991]
implemented Iris by Jung et al. [2019]:

$$\frac{}{\{\ulcorner\urcorner\}\ \texttt{newProph}()\ \{\lambda p. \exists v.\, \mathsf{Proph}(p, v)\}}$$

$$\frac{}{\{\mathsf{Proph}(p, v)\}\ \texttt{resolve}\ p\ \texttt{to}\ w\ \{\lambda \_. \ulcorner v = w \urcorner\}}$$

# 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