Introduction to Separation Logic

Jean-Marie Madiot

INRIA Paris

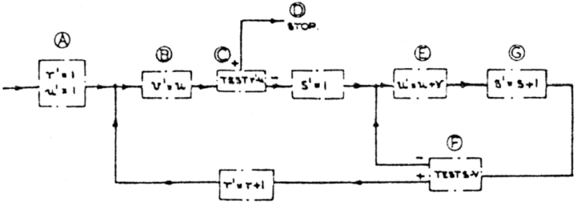
Glasgow, 2024 July 30 - August 2 @ SPLV24

Please interrupt me!

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

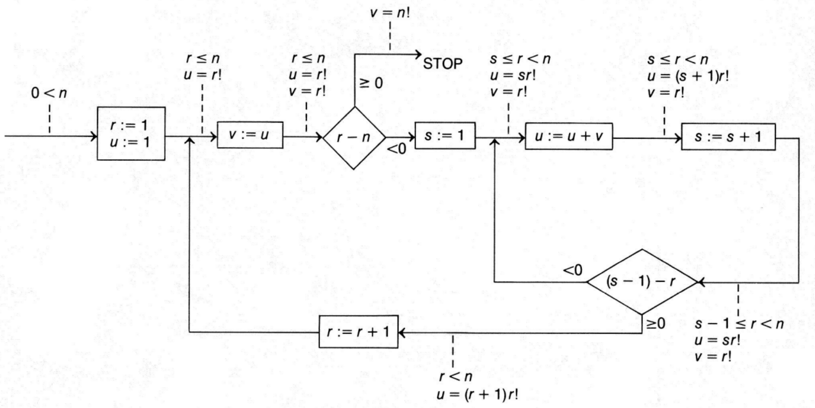
Alan Turing, 1949 : *Checking a large routine*

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



STORAGE LOCATION	(INITIAL) A k=6	B k=5	C k=4	(STOP) D k=0	E k=3	F k=1	G 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 B WITH r'=1 u'=1	TO C	TO D IF r=n TO E IF r<n		TO G	TO B WITH r'=r+1 IF s≥r TO E WITH s'=s+1 IF s<r	TO F

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:

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

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:

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

Examples :

`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:

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

Examples :

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

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:

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

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

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 $r = \lvert x \rvert$
$\texttt{while } n \neq 0 \texttt{ do } n := n - 1$	

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:

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

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

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 $r = \lvert x\rvert$
$\texttt{while}\ n \neq 0\ \texttt{do}\ n := n - 1$	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:

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

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

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 $r = x^n$

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 $\mathbb{Z}/n\mathbb{Z}$
$r := 1;\texttt{while } n > 0 \texttt{ do } (r := r * x; n := n - 1)$	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:

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

Examples :

`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*:

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

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*:

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

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*:

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

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

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

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:

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

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:

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

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:

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

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*:

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

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

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

Example: often easier to start from the end:

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

Examples :

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

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 $I \ = \ (rx^i = x^n)$

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 $I \;=\; (rx^i = x^n)$

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

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 $I \ = \ (rx^i = x^n)$

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

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 $I \;=\; (rx^i = x^n)$

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

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:

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

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

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

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 $I = (rx^i = x^n)$

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

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 $I \;=\; (rx^i = x^n)$

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

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:

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

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

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

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 $I \;=\; (rx^i = x^n \wedge i \geqslant 0)$

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

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 $I = (rx^i = x^n \wedge i \geqslant 0)$

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

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

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

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)

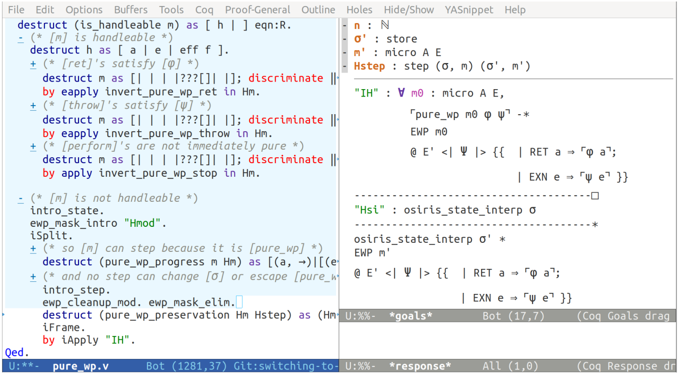
- work well on some classes of programs
- rely on user-provided invariants
- blackbox problem (hard to debug, extend, prove...)

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

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

Heap predicates

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

In Coq: H : `heap` $\rightarrow$ `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:

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

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

Before: $\ulcorner\,\urcorner$

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

Empty heap and pure facts

Definition:

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

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 $\ulcorner \text{True} \urcorner$.

Singleton heap

Definition:

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

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

Before: $\ulcorner\,\urcorner$

After: $r \mapsto 3$

Singleton heap

Definition:

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

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

$$\begin{array}{ll} \text{Before:} & \ulcorner\urcorner \\ \text{After:} & r \mapsto 3 \end{array}$$

Example: specification of "`incr s`".

$$\begin{array}{lll} \text{Before:} & s \mapsto n & \text{for some } n \\ \text{After:} & s \mapsto (n+1) & \end{array}$$

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:

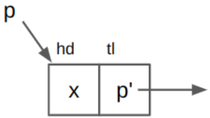
1. $(r \mapsto 5) * (s \mapsto 3) * \ulcorner t = r \urcorner$
2. $(r \mapsto 6) * (s \mapsto 3) * \ulcorner t = r \urcorner$
3. $(r \mapsto 7) * (s \mapsto 3) * \ulcorner t = r \urcorner$

Record fields

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

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

Example:



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

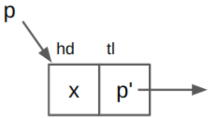
$$p.\mathsf{tl} \mapsto p'$$

Record fields

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

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

Example:



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

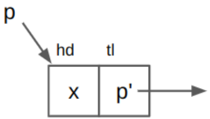
$$p.\mathsf{tl} \mapsto p'$$

In the C memory model:

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

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

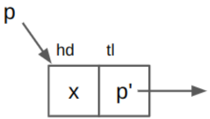
Representation of list cells



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

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:

$$\begin{array}{rcl@{\qquad}l}
(\exists x.\, P) & : & \texttt{Prop} & \text{when} \quad (P : \texttt{Prop}) \\
(\exists x.\, H) & : & \texttt{heap} \to \texttt{Prop} & \text{when} \quad (H : \texttt{heap} \to \texttt{Prop})
\end{array}$$

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

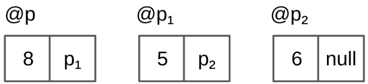
$$
\begin{array}{lcl}
\ulcorner\,\urcorner & \equiv & \ulcorner \mathsf{True} \urcorner \\[2ex]
\ulcorner P \urcorner & \equiv & \lambda m.\ m = \varnothing\ \wedge\ P \\[2ex]
l \mapsto v & \equiv & \lambda m.\ m = \{(l, v)\} \wedge l \neq \mathsf{null} \\[2ex]
H_1 * H_2 & \equiv & \lambda m.\ \exists m_1 m_2. \left\{ \begin{array}{l} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1\, m_1 \\ H_2\, m_2 \end{array} \right. \\[2ex]
\exists x.\, H & \equiv & \lambda m.\ \exists x.\ H\, m
\end{array}
$$

Chapter 2

Representation Predicate for Lists

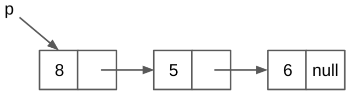
Implementation of mutable lists

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



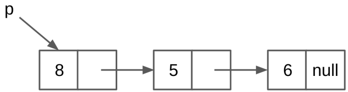

```
type 'a cell = { mutable hd : 'a;
                 mutable tl : 'a cell }

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



Representation of mutable lists

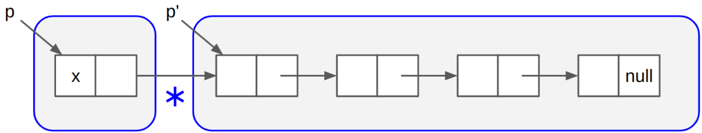
$$L \;=\; 8 :: 5 :: 6 :: \mathsf{nil}$$



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

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

Representation predicate



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

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

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

Equivalent to:

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

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:

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

List construction: proof (1/2)

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

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

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

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 $\mathsf{null} \leadsto \mathsf{MList}\, \mathsf{nil}$. We use:

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

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:

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

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

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 $n - 1$.

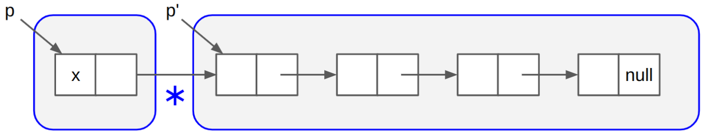
To produce $p \rightsquigarrow \mathsf{MList}\, L$, we have $p' \rightsquigarrow \mathsf{MList}\, L'$ and $p \rightsquigarrow \{\!|\mathsf{hd}{=}v;\ \mathsf{tl}{=}p'|\!\}$.

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 $p \rightsquigarrow \{|\mathsf{hd}{=}v;\ \mathsf{tl}{=}p'|\}$.



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

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.