The Why and How of Deductive Program Verification

Jean-Christophe Filliâtre

Marktoberdorf Summer School 2026

lecture 2

today: from programs to provers

1. weakest preconditions
2. runtime safety
3. functions and contracts
4. termination
5. exceptions (if time permits)

t	$::=$	$\ldots, -1, 0, 1, \ldots, 42, \ldots$	integer constants
	$\mid$	`true` $\mid$ `false`	Boolean constants
	$\mid$	v	immutable variable
	$\mid$	x	mutable variable
	$\mid$	$t \ op \ t$	binary operation
	$\mid$	$op \ t$	unary operation
op	$::=$	$+ \mid - \mid \times$	arithmetic operations
	$\mid$	$= \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant$	arithmetic comparisons
	$\mid$	$\wedge \mid \vee \mid \neg$	Boolean connectives

- two data types: mathematical integers and Booleans
- well-typed terms evaluate without errors (no division)
- evaluation of a term does not change the program memory

e	$::=$	`skip`	do nothing
	$\mid$	t	pure term
	$\mid$	$x \leftarrow t$	assignment
	$\mid$	e `;` e	sequence
	$\mid$	`let` $v = e$ `in` e	binding
	$\mid$	`let ref` $x = e$ `in` e	allocation
	$\mid$	`if` t `then` e `else` e	conditional
	$\mid$	`while` t `do` e `done`	loop

- expressions can allocate and modify memory
- but mutable variables are not first-class values
- well-typed expressions evaluate without errors

three types: $\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

$$
\begin{array}{lcl}
\texttt{skip} & : & \texttt{unit} \\
t_\tau & : & \tau \\
x_\tau \leftarrow t_\tau & : & \texttt{unit} \\
e_{\texttt{unit}} \texttt{ ; } e_\tau & : & \tau \\
\texttt{let } v_\tau = e_\tau \texttt{ in } e'_{\tau'} & : & \tau' \\
\texttt{let ref } x_\tau = e_\tau \texttt{ in } e'_{\tau'} & : & \tau' \\
\texttt{if } t_{\texttt{bool}} \texttt{ then } e_\tau \texttt{ else } e'_\tau & : & \tau \\
\texttt{while } t_{\texttt{bool}} \texttt{ do } e_{\texttt{unit}} \texttt{ done} & : & \texttt{unit}
\end{array}
$$

$$x \leftarrow e \stackrel{\text{def}}{=} \texttt{let}\ v = e\ \texttt{in}\ x \leftarrow v$$

$$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \stackrel{\text{def}}{=} \texttt{let}\ v = e\ \texttt{in}\ \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

$$\texttt{if}\ e_1\ \texttt{then}\ e_2 \stackrel{\text{def}}{=} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else skip}$$

$$e_1\ \texttt{\&\&}\ e_2 \stackrel{\text{def}}{=} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else false}$$

$$e_1\ \texttt{||}\ e_2 \stackrel{\text{def}}{=} \texttt{if}\ e_1\ \texttt{then true else}\ e_2$$

```
let ref p = a in
let ref q = b in
let ref r = 0 in
while q > 0 do
    if odd q then r ← r + p ;
    p ← p + p ;
    q ← half q
done ;
r
```

informal specification:

- at the beginning, we require $b \geqslant 0$
- at the end, we ensure $r = a \times b$

a statement about program correctness:

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

- P precondition property
- e program expression
- Q postcondition property

what is the meaning of a Hoare triple?

$\{P\}\, e\, \{Q\}$ if we execute e in a state that satisfies P, then the computation either diverges or terminates in a state that satisfies Q

this is partial correctness: we say nothing about termination.

examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x \leftarrow x + 2\ \{x = 3\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}\ \texttt{while true do skip done}\ \{\texttt{false}\}$

in our example, we intend to show

$$\{b \geqslant 0\}\ \text{peasant multiplication}\ \{\texttt{result} = a \times b\}$$

examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ \ x \leftarrow x + 2\ \ \{x = 3\}$
- $\{\exists v.\ x = 4v\}\ \ x + 42\ \ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}\ \ \texttt{while true do skip done}\ \ \{\texttt{false}\}$

in our example, we intend to show

$$\{b \geqslant 0\}\ \text{peasant multiplication}\ \{\texttt{result} = a \times b\}$$

how can we establish the validity of Hoare triples?

Hoare's seminal paper introduces deduction rules, such as

$$\frac{\{P \wedge t\}\ e_1\ \{Q\} \qquad \{P \wedge \neg t\}\ e_2\ \{Q\}}{\{P\}\ \texttt{if}\ t\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{Q\}}$$

yet, as such, these rules are not amenable to mechanization

weakest preconditions (Edsger Dijkstra, 1975)

given

- e expression
- Q postcondition

compute the weakest precondition P such that $\{P\}\, e\, \{Q\}$ holds

we write it $\mathrm{WP}(e, Q)$

this is a predicate transformer

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{$ if c then P_1 else P_2 $\}$ $\quad$ `if` c `then` e_1 `else` e_2 $\quad \{\ Q\ \}$

$\{$ if c then P else Q $\}$ $\quad$ `if` c `then` e $\quad \{\ Q\ \}$

?

`while` c `do` e `done` $\quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{$ if c then P_1 else P_2 $\}$ $\quad$ `if` c `then` e_1 `else` e_2 $\quad \{\ Q\ \}$

$\{$ if c then P else Q $\}$ $\quad$ `if` c `then` e $\quad \{\ Q\ \}$

?

`while` c `do` e `done` $\quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{$ if c then P_1 else P_2 $\}$ $\quad$ `if` c `then` e_1 `else` e_2 $\quad \{\ Q\ \}$

$\{$ if c then P else Q $\}$ $\quad$ `if` c `then` e $\quad \{\ Q\ \}$

?

`while` c `do` e `done` $\quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad$ `if` c `then` P_1 e_1 Q `else` P_2 e_2 Q $\quad \{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad$ `if` c `then` P e Q $\quad \{\ Q\ \}$

?

`while` c `do` e `done` $\quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad$ `if` c `then` P_1 e_1 Q `else` P_2 e_2 Q $\quad \{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad$ `if` c `then` P e Q $\quad \{\ Q\ \}$

?

`while` c `do` e `done` $\quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad \texttt{if } c \texttt{ then } P_1\ e_1\ Q \texttt{ else } P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad \texttt{if } c \texttt{ then } P\ e\ Q \quad$ $\{\ Q\ \}$

$?$ $\quad \texttt{while } c \texttt{ do } e \texttt{ done} \quad$ $\{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad \texttt{if } c \texttt{ then } P_1\ e_1\ Q \texttt{ else } P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad \texttt{if } c \texttt{ then } P\ e\ Q \quad$ $\{\ Q\ \}$

$?$ $\quad \texttt{while } c \texttt{ do } e \texttt{ done} \quad$ $\{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \texttt{if}\ c\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \}$ $\quad \texttt{if}\ c\ \texttt{then}\ P_1\ e_1\ Q\ \texttt{else}\ P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q\ \}$ $\quad \texttt{if}\ c\ \texttt{then}\ P\ e\ Q \quad$ $\{\ Q\ \}$

$?$ $\quad \texttt{while}\ c\ \texttt{do}\ e\ \texttt{done} \quad$ $\{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \texttt{if}\ c\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \}$ $\quad \texttt{if}\ c\ \texttt{then}\ P_1\ e_1\ Q\ \texttt{else}\ P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \texttt{if}\ c\ \texttt{then}\ P\ \texttt{else}\ Q\ \}$ $\quad \texttt{if}\ c\ \texttt{then}\ P\ e\ Q \quad$ $\{\ Q\ \}$

$?$ $\quad \texttt{while}\ c\ \texttt{do}\ e\ \texttt{done} \quad$ $\{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad \texttt{if } c \texttt{ then } P_1\ e_1\ Q \texttt{ else } P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad \texttt{if } c \texttt{ then } P\ e\ Q \quad$ $\{\ Q\ \}$

? $\quad \texttt{while } c \texttt{ do } e \texttt{ done} \quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad \texttt{if } c \texttt{ then } P_1\ e_1\ Q \texttt{ else } P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad \texttt{if } c \texttt{ then } P\ e\ Q \quad$ $\{\ Q\ \}$

? $\quad \texttt{while } c \texttt{ do } e \texttt{ done} \quad \{\ Q\ \}$

$\{\ 3 \times x \times y \text{ is even}\ \}$ $\quad x \leftarrow 3 \times x \times y \quad$ $\{\ x \text{ is even}\ \}$

$\{\ Q[t]\ \}$ $\quad x \leftarrow t \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad \texttt{if } c \texttt{ then } P_1\ e_1\ Q \texttt{ else } P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad \texttt{if } c \texttt{ then } P\ e\ Q \quad$ $\{\ Q\ \}$

? $\quad \texttt{while } c \texttt{ do } e \texttt{ done} \quad$ $\{\ Q\ \}$

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q) &\equiv Q \\
\mathrm{WP}(t, Q) &\equiv Q[\texttt{result} \mapsto t] \\
\mathrm{WP}(x \leftarrow t, Q) &\equiv Q[x \mapsto t] \\
\mathrm{WP}(e_1 \texttt{ ; } e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)) \\
\mathrm{WP}(\texttt{let } v = e_1 \texttt{ in } e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[v \mapsto \texttt{result}]) \\
\mathrm{WP}(\texttt{let ref } x = e_1 \texttt{ in } e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \mapsto \texttt{result}]) \\
\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) &\equiv (\ \ t \to \mathrm{WP}(e_1, Q)) \wedge \\
&\quad\ (\neg t \to \mathrm{WP}(e_2, Q))
\end{aligned}
$$

```
if odd q then r ← r + p ;
p ← p + p ;
q ← half q
```

$$(\ \text{odd}\ q \to Q[p + p, \text{half}\ q, r + p]) \wedge$$
$$(\neg\ \text{odd}\ q \to Q[p + p, \text{half}\ q, r\ \ \ ])$$

if odd q **then**

$$Q[p + p, \text{half}\ q, r + p]$$

$$r \leftarrow r + p$$

$$Q[p + p, \text{half}\ q, r]$$

else

$$Q[p + p, \text{half}\ q, r]$$

skip ;

$$Q[p + p, \text{half}\ q, r]$$

$$p \leftarrow p + p\ ;$$

$$Q[p, \text{half}\ q, r]$$

$$q \leftarrow \text{half}\ q$$

$$Q[p, q, r]$$

$(\quad \texttt{odd}\ q \to Q[p + p, \texttt{half}\ q, r + p]) \wedge$
$(\neg\ \texttt{odd}\ q \to Q[p + p, \texttt{half}\ q, r\quad\ ])$

```
if odd q then
```
$Q[p + p, \texttt{half}\ q, r + p]$
$r \leftarrow r + p$
$Q[p + p, \texttt{half}\ q, r]$
```
else
```
$Q[p + p, \texttt{half}\ q, r]$
```
skip ;
```
$Q[p + p, \texttt{half}\ q, r]$
$p \leftarrow p + p$;
$Q[p, \texttt{half}\ q, r]$
$q \leftarrow \texttt{half}\ q$
$Q[p, q, r]$

$$(\ \text{odd}\ q \to Q[p+p, \texttt{half}\ q, r+p]) \wedge$$
$$(\neg\ \text{odd}\ q \to Q[p+p, \texttt{half}\ q, r\ \ \ ])$$

```
  if odd q then
```
$$Q[p+p, \texttt{half}\ q, r+p]$$
```
    r ← r + p
```
$$Q[p+p, \texttt{half}\ q, r]$$
```
  else
```
$$Q[p+p, \texttt{half}\ q, r]$$
```
    skip ;
```
$$Q[p+p, \texttt{half}\ q, r]$$
```
  p ← p + p ;
```
$$Q[p, \texttt{half}\ q, r]$$
```
  q ← half q
```
$$Q[p, q, r]$$

$(\ \text{odd}\ q \to Q[p+p, \texttt{half}\ q, r+p]) \wedge$
$(\neg\ \text{odd}\ q \to Q[p+p, \texttt{half}\ q, r\ \ \ ])$

```
  if odd q then
    Q[p + p, half q, r + p]
      r ← r + p
    Q[p + p, half q, r]
  else
    Q[p + p, half q, r]
      skip ;
Q[p + p, half q, r]
  p ← p + p ;
Q[p, half q, r]
  q ← half q
Q[p, q, r]
```

$$(\ \text{odd}\ q \to Q[p+p, \texttt{half}\ q, r+p]) \wedge$$
$$(\neg\ \text{odd}\ q \to Q[p+p, \texttt{half}\ q, r\ \ \ ])$$

```
  if odd q then
    Q[p + p, half q, r + p]
      r ← r + p
    Q[p + p, half q, r]
  else
    Q[p + p, half q, r]
      skip ;
    Q[p + p, half q, r]
  p ← p + p ;
Q[p, half q, r]
  q ← half q
Q[p, q, r]
```

$$
\begin{array}{l}
(\ \ \mathtt{odd}\ q \to Q[p + p, \mathtt{half}\ q, r + p]) \wedge \\
(\neg\ \mathtt{odd}\ q \to Q[p + p, \mathtt{half}\ q, r\ \ \ \ ])
\end{array}
$$

$$
\begin{array}{l}
\quad \mathtt{if}\ \mathtt{odd}\ q\ \mathtt{then} \\
\quad\quad Q[p + p, \mathtt{half}\ q, r + p] \\
\quad\quad\quad r \leftarrow r + p \\
\quad\quad Q[p + p, \mathtt{half}\ q, r] \\
\quad \mathtt{else} \\
\quad\quad Q[p + p, \mathtt{half}\ q, r] \\
\quad\quad\quad \mathtt{skip}\ ; \\
\quad\quad Q[p + p, \mathtt{half}\ q, r] \\
\quad p \leftarrow p + p\ ; \\
Q[p, \mathtt{half}\ q, r] \\
\quad q \leftarrow \mathtt{half}\ q \\
Q[p, q, r]
\end{array}
$$

$$
\begin{array}{l}
(\quad \texttt{odd}\ q \to Q[p + p, \texttt{half}\ q, r + p]) \wedge \\
(\neg\ \texttt{odd}\ q \to Q[p + p, \texttt{half}\ q, r\quad\ ]) \\
\quad \texttt{if odd}\ q\ \texttt{then} \\
\qquad Q[p + p, \texttt{half}\ q, r + p] \\
\qquad\quad r \leftarrow r + p \\
\qquad Q[p + p, \texttt{half}\ q, r] \\
\quad \texttt{else} \\
\qquad Q[p + p, \texttt{half}\ q, r] \\
\qquad\quad \texttt{skip}\ ; \\
\qquad Q[p + p, \texttt{half}\ q, r] \\
\quad p \leftarrow p + p\ ; \\
Q[p, \texttt{half}\ q, r] \\
\quad q \leftarrow \texttt{half}\ q \\
Q[p, q, r]
\end{array}
$$

$$
\begin{array}{l}
\mathrm{WP}(\texttt{while}\ t\ \texttt{do}\ e\ \texttt{done}, Q) \equiv \\
\quad \exists\, J : \texttt{Prop}. \\
\qquad J \wedge \\
\qquad \forall x_1 \ldots x_k. \\
\qquad\quad (J \wedge\ \ t \to \mathrm{WP}(e, J)) \wedge \\
\qquad\quad (J \wedge \neg t \to Q)
\end{array}
$$

some *invariant property* J
that holds at the loop entry
and is preserved
after a single iteration,
is strong enough to prove Q

where $x_1 \ldots x_k$ are the references modified in e

we cannot know the values of the modified references after n iterations

- therefore, we prove preservation and the post for arbitrary values
- the invariant must provide all the needed information about $x_1 \ldots x_k$

finding an appropriate invariant is difficult in the general case
(this is equivalent to constructing a proof of Q by induction)

we can ease the task of automated tools by providing annotations:

$$
\begin{array}{ll}
\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{do}\ e\ \texttt{done}, Q) \equiv & \text{the given invariant } J \\
\quad J \wedge & \text{holds at the loop entry,} \\
\quad \forall x_1 \ldots x_k. & \text{is preserved after} \\
\qquad (J \wedge \ \ t \to \mathrm{WP}(e, J)) \wedge & \quad \text{a single iteration,} \\
\qquad (J \wedge \neg t \to Q) & \text{and suffices to prove } Q
\end{array}
$$

```
  let ref p = a in
  let ref q = b in
  let ref r = 0 in
  while q > 0 invariant J[p, q, r] do
      if odd q then r ← r + p ;
      p ← p + p ;
      q ← half q
  done ;
  r
result = a × b
```

```
  let ref p = a in
  let ref q = b in
  let ref r = 0 in
  while q > 0 invariant J[p, q, r] do
      if odd q then r ← r + p ;
      p ← p + p ;
      q ← half q
  done ;
r = a × b
  r
```

```
  let ref p = a in
  let ref q = b in
  let ref r = 0 in
  while q > 0 invariant J[p, q, r] do
      if odd q then r ← r + p ;
      p ← p + p ;
      q ← half q
    J[p, q, r]
  done ;
r = a × b
  r
```

$$
\begin{array}{l}
\texttt{let ref } p = a \texttt{ in} \\
\texttt{let ref } q = b \texttt{ in} \\
\texttt{let ref } r = 0 \texttt{ in} \\
\texttt{while } q > 0 \texttt{ invariant } J[p, q, r] \texttt{ do} \\
\quad (\ \ \texttt{odd } q \to J[p + p, \texttt{half } q, r + p]) \wedge \\
\quad (\neg \texttt{ odd } q \to J[p + p, \texttt{half } q, r\ \ \ \ ]) \\
\qquad \texttt{if odd } q \texttt{ then } r \leftarrow r + p\,; \\
\qquad p \leftarrow p + p\,; \\
\qquad q \leftarrow \texttt{half } q \\
\quad J[p, q, r] \\
\texttt{done}\,; \\
r = a \times b \\
\quad r
\end{array}
$$

$$
\begin{array}{l}
\quad \texttt{let ref } p = a \texttt{ in} \\
\quad \texttt{let ref } q = b \texttt{ in} \\
\quad \texttt{let ref } r = 0 \texttt{ in} \\
J[p,q,r] \wedge \\
\forall p\, q\, r.\ J[p,q,r] \rightarrow \\
\quad (q > 0 \rightarrow \\
\qquad (\ \ \texttt{odd } q \rightarrow J[p+p, \texttt{half } q, r+p]) \wedge \\
\qquad (\neg\, \texttt{odd } q \rightarrow J[p+p, \texttt{half } q, r\ \ \ ])) \wedge \\
\quad (q \leqslant 0 \rightarrow \\
\qquad r = a \times b) \\
\quad \texttt{while } q > 0 \texttt{ invariant } J[p,q,r] \texttt{ do} \\
\qquad \texttt{if odd } q \texttt{ then } r \leftarrow r + p\,; \\
\qquad p \leftarrow p + p\,; \\
\qquad q \leftarrow \texttt{half } q \\
\quad \texttt{done}\,; \\
\quad r
\end{array}
$$

$$
\begin{array}{l}
J[a, b, 0] \ \wedge \\
\forall p\, q\, r.\ J[p, q, r] \ \rightarrow \\
\quad (q > 0 \ \rightarrow \\
\qquad (\ \ \mathtt{odd}\ q \rightarrow J[p + p, \mathtt{half}\ q, r + p]) \ \wedge \\
\qquad (\neg\, \mathtt{odd}\ q \rightarrow J[p + p, \mathtt{half}\ q, r\ \ \ \ ])) \ \wedge \\
\quad (q \leqslant 0 \ \rightarrow \\
\qquad r = a \times b) \\
\quad \mathtt{let\ ref}\ p = a\ \mathtt{in} \\
\quad \mathtt{let\ ref}\ q = b\ \mathtt{in} \\
\quad \mathtt{let\ ref}\ r = 0\ \mathtt{in} \\
\quad \mathtt{while}\ q > 0\ \mathtt{invariant}\ J[p, q, r]\ \mathtt{do} \\
\qquad \mathtt{if\ odd}\ q\ \mathtt{then}\ r \leftarrow r + p\,; \\
\qquad p \leftarrow p + p\,; \\
\qquad q \leftarrow \mathtt{half}\ q \\
\quad \mathtt{done}\,; \\
\quad r
\end{array}
$$

find a suitable invariant $J[p, q, r]$ for the peasant multiplication

Theorem

For any e and Q, the triple $\{\mathrm{WP}(e, Q)\}\ e\ \{Q\}$ is valid.

can be proved by induction on the structure of the program e
w.r.t. some reasonable semantics (e.g. operational)

Corollary

To show that $\{P\}\ e\ \{Q\}$ holds, it suffices to prove $P \to \mathrm{WP}(e, Q)$.

this is what Why3 does

we get

$$
\begin{array}{l}
b \geqslant 0 \rightarrow \\
\quad J[a, b, 0] \ \wedge \\
\quad \forall p\, q\, r.\ J[p, q, r] \ \rightarrow \\
\qquad (q > 0 \ \rightarrow \\
\qquad\quad (\quad \texttt{odd}\ \ q \rightarrow J[p + p, \texttt{half}\ \ q, r + p]) \ \wedge \\
\qquad\quad (\neg\ \texttt{odd}\ \ q \rightarrow J[p + p, \texttt{half}\ \ q, r\ \ \ \ ])) \ \wedge \\
\qquad (q \leqslant 0 \ \rightarrow \\
\qquad\quad r = a \times b)
\end{array}
$$

note: no need to include $b \geqslant 0$ in your loop invariant

runtime safety

some operations can fail if their safety preconditions are not met:

- arithmetic operations: division par zero, overflows, etc.
- memory access: `NULL` pointers, buffer overruns, etc.
- assertions

a correct program must not fail:

$\{P\}\, e\, \{Q\}$ if we execute e in a state that satisfies P,
then there will be no runtime errors
and the computation either diverges
or terminates normally in a state that satisfies Q

a new kind of expression:

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \texttt{assert}\ R & \text{fail if } R \text{ does not hold} \end{array}$$

the corresponding weakest precondition rule:

$$\mathrm{WP}(\texttt{assert}\ R, Q) \quad \equiv \quad R \wedge Q \quad \equiv \quad R \wedge (R \to Q)$$

the second version is useful in practical deductive verification

we could add other partially defined operations to the language:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
& | & t \texttt{ div } t & \text{Euclidean division} \\
& | & a\texttt{[}t\texttt{]} & \text{array access} \\
& | & \ldots &
\end{array}
$$

and define the WP rules for them:

$$
\begin{aligned}
\mathrm{WP}(t_1 \texttt{ div } t_2, Q) &\equiv t_2 \neq 0 \ \wedge \ Q[\texttt{result} \mapsto (t_1 \texttt{ div } t_2)] \\
\mathrm{WP}(a\texttt{[}t\texttt{]}, Q) &\equiv 0 \leqslant t < |a| \ \wedge \ Q[\texttt{result} \mapsto a\texttt{[}t\texttt{]}] \\
&\ldots
\end{aligned}
$$

functions and contracts

we may want to delegate some functionality to a function:

$\texttt{let}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n)\ :\ \tau\ \mathcal{C}\ =\ e$	defined function
$\texttt{val}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n)\ :\ \tau\ \mathcal{C}$	abstract function

function behaviour is specified with a contract:

$\mathcal{C} ::=$	$\texttt{requires}\ P$	precondition
	$\texttt{writes}\ x_1 \ldots x_k$	modified global references
	$\texttt{ensures}\ Q$	postcondition

```
val ref r : int  (* declare a global reference *)

let incr_r (v: int) : int
  requires { v > 0 }
  writes   { r }
  ensures  { result = old r /\ r = old r + v }
= let u = r in
  r <- u + v;
  u
```

postcondition may refer to the prestate

verification condition

$$\mathrm{VC}(\texttt{let } f \ \ldots) \quad \equiv \quad \forall \vec{x}\,\vec{v}.\ P \to \mathrm{WP}(e, Q)[\vec{x}^{\circ} \mapsto \vec{x}]$$

where $\vec{x}$ are all global references mentioned in f
$\vec{x}^{\circ}$ are the `old` versions of $\vec{x}$

$$\begin{aligned}\mathrm{VC}(\texttt{incr\_r}) \quad \equiv \quad & \forall rv.\, v > 0 \rightarrow \\ & \mathrm{WP}(\texttt{let } u = r \texttt{ in } r \leftarrow u + v \texttt{ ; } u, \\ & \qquad \texttt{result} = r^\circ \wedge r = r^\circ + v)[r^\circ \mapsto r]\end{aligned}$$

$$\begin{aligned} \mathrm{VC}(\texttt{incr\_r}) \quad &\equiv \quad \forall r v.\, v > 0 \rightarrow \\ &\quad\;\; (r = r^{\circ} \wedge r + v = r^{\circ} + v)[r^{\circ} \mapsto r] \end{aligned}$$

$$\mathrm{VC}(\texttt{incr\_r}) \equiv \forall rv.\, v > 0 \rightarrow r = r \wedge r + v = r + v$$

one more expression,

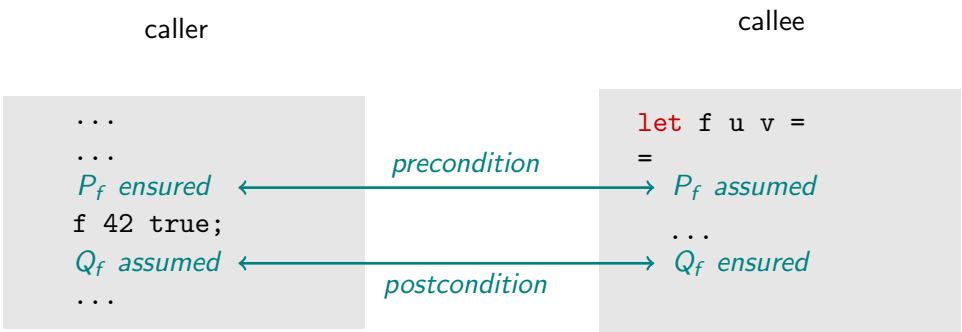
$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & f\ \ t\ \ \ldots\ \ t & \text{function call} \end{array}$$

and its weakest precondition rule:

$$\begin{aligned} \mathrm{WP}(f\ t_1\ \ldots\ t_n, Q) \equiv\ & P_f[\vec{v} \mapsto \vec{t}\,] \wedge \\ & (\forall \vec{\boldsymbol{x}}\, \forall \mathtt{result}. \\ & \quad Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}] \end{aligned}$$

P_f	precondition of f	$\vec{x}$	references used in f
Q_f	postcondition of f	$\vec{\boldsymbol{x}}$	references modified in f ($\vec{\boldsymbol{x}} \subseteq \vec{x}$)
$\vec{v}$	formal parameters of f	$\vec{w}$	fresh variables

the verification proceeds per function, in a modular way



this is similar to what a compiler does (e.g. with static typing)

program function

```
val (/) (x: int) (y: int) : int
  requires { y <> 0 }
  ensures  { result = div x y }
```

logic function

program function

```
val ([]) (a: array 'a) (i: int) : 'a
  requires { 0 <= i < size a }
  ensures  { result = a[i] }
```

logic function

termination

problem: prove that the program terminates for every initial state that satisfies the precondition

it suffices to show that

- every loop makes a finite number of iterations
- recursive function calls cannot go on indefinitely

terminology: partial correctness + termination = total correctness

solution: prove that every loop iteration and every recursive call decreases a certain value, called variant, with respect to some well-founded order

for example, for signed integers, a practical well-founded order is

$$i \prec j \quad \stackrel{\text{def}}{=} \quad i < j \wedge 0 \leqslant j$$

a new annotation:

$$
\begin{array}{rcl}
e & ::= & \ldots \\
 & \mid & \texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{variant}\ t \cdot \prec\ \texttt{do}\ e\ \texttt{done}
\end{array}
$$

the corresponding weakest precondition rule:

$$
\begin{array}{l}
\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{variant}\ s \cdot \prec\ \texttt{do}\ e\ \texttt{done}, Q) \equiv \\
\quad J \wedge \\
\quad \forall x_1 \ldots x_k. \\
\qquad (J \wedge \quad t \to \mathrm{WP}(e, J \wedge s \prec w)[w \mapsto s]) \wedge \\
\qquad (J \wedge \neg t \to Q)
\end{array}
$$

$x_1 \ldots x_k$ references modified in e

w a fresh variable (the variant at the start of the iteration)

find a suitable variant for the peasant multiplication

a new contract clause:

$$
\begin{aligned}
&\texttt{let rec } f\ (v_1 : \tau_1)\ \ldots\ (v_n : \tau_n)\ :\ \tau \\
&\qquad \texttt{requires } P_f \\
&\qquad \texttt{variant } s \cdot \prec \\
&\qquad \texttt{writes } \vec{x} \\
&\qquad \texttt{ensures } Q_f \\
&\quad = e
\end{aligned}
$$

for each recursive call of f in e:

$$
\begin{aligned}
\mathrm{WP}(f\ t_1\ \ldots\ t_n, Q)\ \equiv\ & P_f[\vec{v} \mapsto \vec{t}]\ \wedge\ s[\vec{v} \mapsto \vec{t}] \prec s[\vec{x} \mapsto \vec{x}^{\circ}]\ \wedge \\
& (\forall \vec{x}\ \forall \texttt{result}. \\
& \quad Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}]
\end{aligned}
$$

$s[\vec{v} \mapsto \vec{t}]$	variant at the call site	$\vec{x}$	references used in f
$s[\vec{x} \mapsto \vec{x}^{\circ}]$	variant at the start of f	$\vec{w}$	fresh variables

McCarthy's 91 function

```
let rec f91 (n: int) : int
  ensures { result = if n <= 100 then 91 else n - 10 }
  variant { 101 - n }
= if n <= 100 then
    f91 (f91 (n + 11))
  else
    n - 10
```

exercise #4: find a variant for a non-recursive implementation of `f91` (code is given)

mutually recursive functions must have

- their own variant terms
- a common well-founded order

the WP rule is easily adapted

exceptions

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_E
  - partial correctness: *if* E *is raised, then* Q_E *holds*

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_E
  - partial correctness: *if* E *is raised, then* Q_E *holds*

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_E
  - partial correctness: *if* E *is raised, then* Q_E *holds*

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_{E}
  - partial correctness: *if* E *is raised, then* Q_{E} *holds*

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_{E}
  - partial correctness: *if* E *is raised, then* Q_{E} *holds*

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_E
  - partial correctness: *if* E *is raised, then* Q_E *holds*

execution of a program can lead to

- divergence: the computation never ends
  - total correctness ensures against non-termination
- abnormal termination: the computation fails
  - partial correctness ensures against run-time errors
- normal termination: the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination: produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_E
  - partial correctness: *if* E *is raised, then* Q_E *holds*

```
exception Not_found

val binary_search (a: array int) (v: int) : int
  requires { forall i j. 0 <= i <= j < length a -> a[i] <= a[j] }
  ensures  { 0 <= result < length a  /\  a[result] = v }
  raises   { Not_found -> forall i. 0 <= i < length a -> a[i] <> v }
```

our language keeps growing:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise}\ \mathsf{E} & \text{raise an exception} \\
 & | & \texttt{try}\ e\ \texttt{with}\ \mathsf{E} \to e & \text{catch an exception}
\end{array}
$$

WP handles two postconditions now:

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) &\equiv Q \\
\mathrm{WP}(\texttt{raise}\ \mathsf{E}, Q, Q_{\mathsf{E}}) &\equiv Q_{\mathsf{E}} \\
\mathrm{WP}(e_1\ ;\ e_2, Q, Q_{\mathsf{E}}) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}), Q_{\mathsf{E}}) \\
\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E} \to e_2, Q, Q_{\mathsf{E}}) &\equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}))
\end{aligned}
$$

our language keeps growing:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise}\ \mathsf{E} & \text{raise an exception} \\
 & | & \texttt{try}\ e\ \texttt{with}\ \mathsf{E} \to e & \text{catch an exception}
\end{array}
$$

WP handles two postconditions now:

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_\mathsf{E}) &\equiv Q \\
\mathrm{WP}(\texttt{raise}\ \mathsf{E}, Q, Q_\mathsf{E}) &\equiv Q_\mathsf{E} \\
\mathrm{WP}(e_1\ ;\ e_2, Q, Q_\mathsf{E}) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_\mathsf{E}), Q_\mathsf{E}) \\
\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E} \to e_2, Q, Q_\mathsf{E}) &\equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_\mathsf{E}))
\end{aligned}
$$

our language keeps growing:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise}\ \mathsf{E} & \text{raise an exception} \\
 & | & \texttt{try}\ e\ \texttt{with}\ \mathsf{E} \to e & \text{catch an exception}
\end{array}
$$

WP handles two postconditions now:

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) &\equiv Q \\
\mathrm{WP}(\texttt{raise}\ \mathsf{E}, Q, Q_{\mathsf{E}}) &\equiv Q_{\mathsf{E}} \\
\mathrm{WP}(e_1\ ;\ e_2, Q, Q_{\mathsf{E}}) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}), Q_{\mathsf{E}}) \\
\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E} \to e_2, Q, Q_{\mathsf{E}}) &\equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}))
\end{aligned}
$$

our language keeps growing:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & \mid & \texttt{raise}\ \mathsf{E} & \text{raise an exception} \\
 & \mid & \texttt{try}\ e\ \texttt{with}\ \mathsf{E} \to e & \text{catch an exception}
\end{array}
$$

WP handles two postconditions now:

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_\mathsf{E}) &\equiv Q \\
\mathrm{WP}(\texttt{raise}\ \mathsf{E}, Q, Q_\mathsf{E}) &\equiv Q_\mathsf{E} \\
\mathrm{WP}(e_1\ ;\ e_2, Q, Q_\mathsf{E}) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_\mathsf{E}), Q_\mathsf{E}) \\
\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E} \to e_2, Q, Q_\mathsf{E}) &\equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_\mathsf{E}))
\end{aligned}
$$

exceptions can carry data:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & \mid & \texttt{raise}\ \mathsf{E}\ t & \text{raise an exception} \\
 & \mid & \texttt{try}\ e\ \texttt{with}\ \mathsf{E}\ v\ \to\ e & \text{catch an exception}
\end{array}
$$

still, all needed mechanisms are already in WP:

$$\mathrm{WP}(t, Q, Q_{\mathsf{E}}) \equiv Q[\texttt{result} \mapsto t]$$

$$\mathrm{WP}(\texttt{raise}\ \mathsf{E}\ t, Q, Q_{\mathsf{E}}) \equiv Q_{\mathsf{E}}[\texttt{result} \mapsto t]$$

$$\mathrm{WP}(\texttt{let}\ v = e_1\ \texttt{in}\ e_2, Q, Q_{\mathsf{E}}) \equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}})[v \mapsto \texttt{result}], Q_{\mathsf{E}})$$

$$\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E}\ v\ \to\ e_2, Q, Q_{\mathsf{E}}) \equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}})[v \mapsto \texttt{result}])$$

a new contract clause:

$$\begin{aligned}&\texttt{let } f\,(v_1 : \tau_1)\ \ldots\ (v_n : \tau_n)\ :\ \varsigma\\ &\quad\ \texttt{requires } P_f\\ &\quad\ \texttt{writes } \vec{x}\\ &\quad\ \texttt{ensures } Q_f\\ &\quad\ \texttt{raises } \mathsf{E} \to Q_{\mathsf{E}f}\\ &\ = e\end{aligned}$$

verification condition for the function definition:

$$\mathrm{VC}(\texttt{let } f\ \ldots) \ \equiv\ \forall \vec{x}\,\vec{v}.\ P_f \to \mathrm{WP}(e, Q_f, Q_{\mathsf{E}f})[\vec{x}^{\circ} \mapsto \vec{x}]$$

weakest precondition rule for the function call:

$$\begin{aligned}\mathrm{WP}(f\ \ t_1\ \ldots\ t_n,\ Q,\ Q_{\mathsf{E}}) \ \equiv\ & P_f[\vec{v} \mapsto \vec{t}\,] \ \wedge\\ & (\forall \vec{x}\,\forall \texttt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}] \ \wedge\\ & (\forall \vec{x}\,\forall \texttt{result}.\, Q_{\mathsf{E}f}[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q_{\mathsf{E}})[\vec{w} \mapsto \vec{x}]\end{aligned}$$

takeaways

weakest preconditions turn a program and its specification
into verification conditions, mechanically

the user provides loop invariants and variants

termination is equally as important as partial correctness

thank you