Control structures, eighth lecture

Program logics for control and effects

Xavier Leroy

2024-03-14

Collège de France, chair of software sciences

xavier.leroy@college-de-france.fr

Deductive verification and Hoare logic

Deductive verification of programs

Annotate programs with logical assertions:

- preconditions: expected properties of inputs;
- postconditions: guarantees on outputs;
- invariants: attached to loops, objects, etc.

Example (ACSL specification of a C function)

```
/*@
  requires \valid(a+(0..n-1));
  assigns  a[0..n-1];
  ensures  \forall integer i; 0 <= i < n ==> a[i] == 0;
*/
void set_to_0(int* a, size_t n)
```

Deductive verification of programs

Annotate programs with logical assertions:

Verify the consistency of these annotations:

$$\text{preconditions} \Rightarrow \text{invariants} \Rightarrow \text{postconditions}$$

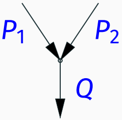
along all the possible execution paths through the program.

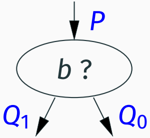
Floyd’s approach

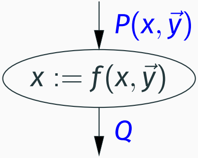
(Alan Turing, *Checking a large routine*, 1949.)
(Robert W. Floyd, *Assigning meanings to programs*, 1967.)

A control-flow graph (flowchart) whose edges are annotated with assertions.

Check the logical consistency of annotations at each node:







$$P_1 \vee P_2 \Rightarrow Q$$

$$P \wedge b \Rightarrow Q_1$$
$$P \wedge \neg b \Rightarrow Q_0$$

$$\left(\exists x_0, x = f(x_0, \vec{y}) \wedge P(x_0, \vec{y})\right) \Rightarrow Q$$

Hoare's approach

(C. A. R. Hoare, *An axiomatic basis for computer programming*, CACM 12, 1969.)

A program logic.

Axioms and deduction rules to prove properties
that hold of all executions of the commands
of an imperative language with structured control.

Strong connections with control structures
and structured programming:

The shape of the verification follows the structure of the program.
Axioms and rule follow the control structures of the language.

Hoare triples

$$\{ P \} \; c \; \{ Q \}$$

c: a command from a structured imperative language (Algol, ...)

P, Q: logical assertions about the program variables.

P: precondition, assumed true “before” the execution of c

Q: postcondition, guaranteed true “after” the execution of c

Hoare triples

"Weak" Hoare logic: (partial correctness)

$\{ P \}\; c\; \{ Q \}$ if P holds "before" and if c terminates,
then Q holds "after"

"Strong" Hoare logic: (full correctness)

$[P]\; c\; [Q]$ if P holds "before",
then c terminates and Q holds "after"

The rules of weak Hoare logic

Structured control:

$$\frac{\{\,P\,\}\; c_1\; \{\,Q\,\} \qquad \{\,Q\,\}\; c_2\; \{\,R\,\}}{\{\,P\,\}\; c_1; c_2\; \{\,R\,\}}$$

$$\frac{\{\,P \land b\,\}\; c_1\; \{\,Q\,\} \qquad \{\,P \land \neg b\,\}\; c_2\; \{\,Q\,\}}{\{\,P\,\}\; \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\; \{\,Q\,\}}$$

$$\frac{\{\,P \land b\,\}\; c\; \{\,P\,\}}{\{\,P\,\}\; \texttt{while}\ b\ \texttt{do}\ c\; \{\,P \land \neg b\,\}}$$

The rules of weak Hoare logic

Empty command:

$$\{\, P \,\}\ \texttt{skip}\ \{\, P \,\}$$

Assignment:

$$\{\, Q[x \leftarrow e] \,\}\ x := e\ \{\, Q \,\}$$

Consequence:

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}$$

Example of verification: Euclidean division

$$\{0 \le a\} \Rightarrow \{a = b \cdot 0 + a \ \wedge \ 0 \le a\}$$

$\mathtt{r} := \mathtt{a};$

$$\{a = b \cdot 0 + r \ \wedge \ 0 \le r\}$$

$\mathtt{q} := 0;$

$$\{a = b \cdot q + r \ \wedge \ 0 \le r\}$$

$\mathtt{while}\ \mathtt{r} \ge \mathtt{b}\ \mathtt{do}$

$$\{a = b \cdot q + r \ \wedge \ 0 \le r \ \wedge \ r \ge b\} \Rightarrow$$
$$\{a = b \cdot (q + 1) + (r - b) \ \wedge \ 0 \le r - b\}$$

$\quad \mathtt{r} := \mathtt{r} - \mathtt{b};$

$$\{a = b \cdot (q + 1) + r \ \wedge \ 0 \le r\}$$

$\quad \mathtt{q} := \mathtt{q} + 1$

$$\{a = b \cdot q + r \ \wedge \ 0 \le r\}$$

$\mathtt{done}$

$$\{a = b \cdot q + r \ \wedge \ 0 \le r \ \wedge \ r < b\} \Rightarrow$$
$$\{q = a/b \ \wedge \ r = a \bmod b\}$$

Extending Hoare logic to various control structures

Other kinds of loops

The `do...while` loop, with exit test at end (C, Java):

$$\frac{\{\,P\,\}\ c\ \{\,Q\,\} \qquad Q \land b \Rightarrow P}{\{\,P\,\}\ \texttt{do}\ c\ \texttt{while}\ b\ \{\,Q \land \neg b\,\}}$$

A loop with exit test in the middle (Ada):

$$\frac{\{\,P\,\}\ c_1\ \{\,Q\,\} \qquad \{\,Q \land \neg b\,\}\ c_2\ \{\,P\,\}}{\{\,P\,\}\ \texttt{loop}\ c_1;\ \texttt{exit when}\ b;\ c_2\ \texttt{end}\ \{\,Q \land b\,\}}$$

A counted `for` loop:

$$\frac{[\,P \land i \leq h\,]\ c\ [\,P[i \leftarrow i+1]\,] \qquad i, h \text{ not assigned in } c}{[\,P[i \leftarrow \ell]\,]\ \texttt{for}\ \ i = \ell\ \ \texttt{to}\ \ h\ \ \texttt{do}\ \ c\ [\,P \land i > h\,]}$$

Non-determinism

Drawing a number between 0 and $N-1$:

$$\{\, \forall i \in [0, N-1],\ Q[\mathrm{x} \leftarrow i\,] \,\}\ x := \mathtt{choose}(N)\ \{\, Q \,\}$$

Dijkstra's "guarded conditional":
executes one of the c_i for which the condition b_i is true.

$$\frac{\{\, P \wedge b_i \,\}\ c_i\ \{\, Q \,\} \quad \text{for } i = 1, \ldots, n}{\{\, P \wedge (b_1 \vee \cdots \vee b_n) \,\}\ \mathtt{if}\ \ b_1 \rightarrow c_1 \mathbin{[\!]} \cdots \mathbin{[\!]} b_n \rightarrow c_n\ \ \mathtt{fi}\ \{\, Q \,\}}$$

A Hoare logic for “goto”?

> Areas which do present real difficulty are labels and jumps, pointers, and name parameters. Proofs of programs which made use of these features are likely to be elaborate, and it is not surprising that this should be reflected in the complexity of the underlying axioms.

(C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969)

A Hoare logic for “goto”?

Consider `goto` in Algol 60: the scope of a label L is the block where it is defined $\Rightarrow$ no jump to the inside of a block.

```
begin
    ...
    goto L
    ...
 L:
    ...
    begin ...     goto L      ... end;
    ...
end
```

Idea: each label L has a precondition R, which is the precondition of the following command. Each `goto` L has precondition R and an arbitrary postcondition.

A Hoare logic for "goto"?

Consider `goto` in Algol 60: the scope of a label L is the block where it is defined $\Rightarrow$ no jump to the inside of a block.

```
begin
    ...
{R} goto L {Q₁}
    ...
 L: {R}
    ...
    begin ... {R} goto L {Q₂} ... end;
    ...
end
```

Idea: each label L has a precondition R, which is the precondition of the following command. Each `goto` L has precondition R and an arbitrary postcondition.

Clint and Hoare's rule for "goto"

(M. Clint, C. A. R. Hoare, *Program proving: jumps and functions*, Acta Informatica 1, 1971.)

$$\frac{\{\,R\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\} \vdash \{\,P\,\}\ c_1\ \{\,R\,\} \qquad \{\,R\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\} \vdash \{\,R\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{begin}\ c_1;\ L : c_2\ \texttt{end}\ \{\,Q\,\}}$$

$X \vdash Y$ reads as a hypothetical deduction in natural deduction: "assuming X we can derive Y".

From the hypothesis $\{\,R\,\}\ \texttt{goto}\ L\ \{\,\texttt{false}\,\}$ we can derive $\{\,R\,\}\ \texttt{goto}\ L\ \{\,Q\,\}$ for any Q, using the consequence rule.

Problems with Clint and Hoare's rule

(M. J. O'Donnell, *A critique of the foundations of Hoare style programming logics*, CACM 25, 1982.)

In case of nested blocks

$$\texttt{begin} \ \ldots \ \texttt{begin} \ \ldots \ L : \ldots \texttt{end} \ldots L : \ldots \texttt{end}$$

"the" precondition associated with L is ambiguous:

$$\{\, R_1 \,\}\ \text{goto}\ L\ \{\, \texttt{false} \,\} \vdash (\{\, R_2 \,\}\ \text{goto}\ L\ \{\, \texttt{false} \,\} \vdash X)$$

Moreover, the logical interpretation of $X \vdash Y$ is delicate. If we read it as "there exists a model where X implies Y", we can take $X = Y =$ false, and deduce

$$\frac{\{\, \texttt{false} \,\}\ \text{goto}\ L\ \{\, \texttt{false} \,\} \implies \{\, \texttt{true} \,\}\ \text{goto}\ L\ \{\, \texttt{false} \,\}}{✘\ \{\, \texttt{true} \,\}\ \text{begin goto}\ L;\ \ L : \text{skip end}\ \{\, \texttt{false} \,\}}$$

The Arbib-Alagic-de Bruin approach

(M. Arbib, S. Alagić, *Proof rules for gotos*, Acta Informatica 11, 1979.
A. de Bruin, *Goto statements: semantics and deduction systems*,
Acta Informatica 15, 1981.)

Idea: `goto` is another way to exit a command c, in addition to normal termination. Let's give `goto` an extra postcondition J.

$$\{ P \}\ c\ \{ Q \}\ \{ J \}$$

J is a function label $\mapsto$ assertion. It can be weakened like the usual postcondition Q.

$$\frac{P' \Rightarrow P \qquad \{ P \}\ c\ \{ Q \}\ \{ J \} \qquad Q \Rightarrow Q' \qquad \forall L,\ J(L) \Rightarrow J'(L)}{\{ P' \}\ c\ \{ Q' \}\ \{ J' \}}$$

The Arbib-Alagic-de Bruin approach

The J postcondition can be false for commands that always terminate normally:

$$\{\, Q[x \leftarrow e] \,\}\; x := e \;\{\, Q \,\}\;\{\, \lambda L.\, \texttt{false} \,\}$$

J is shared between the sub-commands of a sequence and a conditional:

$$\frac{\{\, P \,\}\; c_1 \;\{\, R \,\}\;\{\, J \,\} \qquad \{\, R \,\}\; c_2 \;\{\, Q \,\}\;\{\, J \,\}}{\{\, P \,\}\; c_1; c_2 \;\{\, Q \,\}\;\{\, J \,\}}$$

$$\frac{\{\, P \wedge b \,\}\; c_1 \;\{\, Q \,\}\;\{\, J \,\} \qquad \{\, P \wedge \neg b \,\}\; c_2 \;\{\, Q \,\}\;\{\, J \,\}}{\{\, P \,\}\; \texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2 \;\{\, Q \,\}\;\{\, J \,\}}$$

The rules of Arbib-Alagic-de Bruin

`goto` L can have all its postconditions false except $J(L)$, which is the precondition P of the `goto`:

$$\{\, P \,\}\ \texttt{goto}\ L\ \{\, \texttt{false} \,\}\ \{\, \lambda L'.\ \texttt{if}\ L' = L\ \texttt{then}\ P\ \texttt{else}\ \texttt{false} \,\}$$

In a block that defines L with precondition R, all exits on `goto` L must satisfy R:

$$\frac{\{\, P \,\}\ c_1\ \{\, R \,\}\ \{\, J[L \leftarrow R] \,\} \qquad \{\, R \,\}\ c_2\ \{\, Q \,\}\ \{\, J[L \leftarrow R] \,\}}{\{\, P \,\}\ \texttt{begin}\ c_1;\ L : c_2\ \texttt{end}\ \{\, Q \,\}\ \{\, J \,\}}$$

Early exits from loops

Constructs such as `break` (early loop exit) can also be treated as a special postcondition:

$$\{\,P\,\}\ c\ \{\,Q\,\}\ \{\,B\,\} \qquad (B = \text{precondition for } \texttt{break})$$

Selected rules:

$$\{\,P\,\}\ \texttt{break}\ \{\,\texttt{false}\,\}\ \{\,P\,\}$$

$$\frac{\{\,P \wedge b\,\}\ c\ \{\,P\,\}\ \{\,Q\,\} \qquad P \wedge \neg b \Rightarrow Q}{\{\,P\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\,Q\,\}\ \{\,B\,\}}$$

$$\frac{\{\,P \wedge b\,\}\ c\ \{\,P\,\}\ \{\,Q\,\} \qquad \{\,P \wedge \neg b\,\}\ c'\ \{\,Q\,\}\ \{\,B\,\}}{\{\,P\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{else}\ c'\ \{\,Q\,\}\ \{\,B\,\}}$$

A unified treatment of multiple exits

Instead of having one postcondition for each way of exiting a command, we can have one postcondition that is a function

$$Q : \text{kind of exit} \mapsto \text{assertion}$$

Exit kinds K are, for example,

$K ::= \texttt{norm}$	normal termination
$\mid \texttt{break} \mid \texttt{continue}$	loop exits
$\mid \texttt{break}(n) \mid \texttt{continue}(n)$	multi-level exits
$\mid \texttt{return}(v)$	function return
$\mid \texttt{goto}(L)$	jump
$\mid \texttt{exn}(E)$	exception raising

A unified treatment of multiple exits

The rules for commands that trigger an exit all have the same shape:

$$\{\,P\,\}\ \mathtt{skip}\ \{\,[\mathtt{norm} \mapsto P]\,\}$$
$$\{\,P\,\}\ \mathtt{break}\ \{\,[\mathtt{break}(1) \mapsto P]\,\}$$
$$\{\,P\,\}\ \mathtt{break}\ n\ \{\,[\mathtt{break}(n) \mapsto P]\,\}$$
$$\{\,P\,\}\ \mathtt{goto}\ L\ \{\,[\mathtt{goto}(L) \mapsto P]\,\}$$
$$\{\,P\,\}\ \mathtt{raise}\ E\ \{\,[\mathtt{exn}(E) \mapsto P]\,\}$$

We write $[T \mapsto P] \stackrel{def}{=} \lambda T'.\ \mathtt{if}\ T' = T\ \mathtt{then}\ P\ \mathtt{else}\ \mathtt{false}$.

A unified treatment of multiple exits

The sequence “handles” normal termination:

$$\frac{\{\,P\,\}\; c_1\; \{\,Q[\mathtt{norm} \leftarrow R]\,\} \qquad \{\,R\,\}\; c_2\; \{\,Q\,\}}{\{\,P\,\}\; c_1; c_2\; \{\,Q\,\}}$$

The loop also “handles” the `break` and `continue` exits:

$$Q' = Q\left[\begin{array}{l} \mathtt{norm} \leftarrow P; \\ \mathtt{break}(1) \leftarrow Q(\mathtt{norm}); \\ \mathtt{break}(n+1) \leftarrow Q(\mathtt{break}(n)) \\ \mathtt{continue}(1) \leftarrow P; \\ \mathtt{continue}(n+1) \leftarrow Q(\mathtt{continue}(n)) \end{array}\right]$$

$$\frac{\{\,P \wedge b\,\}\; c\; \{\,Q'\,\} \qquad P \wedge \neg b \Rightarrow Q(\mathtt{norm})}{\{\,P\,\}\; \mathtt{while}\; b\; \mathtt{do}\; c\; \{\,Q\,\}}$$

A unified treatment of multiple exits

The declaration of label L “handles” the $\texttt{goto}(L)$ exit:

$$\frac{\{\,P\,\}\ c_1\ \{\,Q[\texttt{norm} \leftarrow R, \texttt{goto}(L) \leftarrow R]\,\} \qquad \{\,R\,\}\ c_2\ \{\,Q[\texttt{goto}(L) \leftarrow R]\,\}}{\{\,P\,\}\ \texttt{begin}\ c_1;\ L : c_2\ \texttt{end}\ \{\,Q\,\}}$$

Exception handlers “handle” $\texttt{exn}(E)$ exits:

$$\frac{\{\,P\,\}\ c_1\ \{\,Q[\texttt{exn}(E) \leftarrow R]\,\} \qquad \{\,R\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{try}\ c_1\ \texttt{catch}\ E \rightarrow c_2\ \{\,Q\,\}}$$

Coroutines

(M. Clint, *Program proving: coroutines*, Acta Informatica 2, 1973.)

A simple model of asymmetric coroutines:

$$\texttt{coroutine}\ p = c_1\ \texttt{in}\ c_2$$

When the consumer c_2 performs `call` p, the execution of c_1 starts or resumes just after the most recent `yield` p.

When the generator c_1 performs `yield` p, the execution of c_2 resumes just after the most recent `call` p.

The `coroutine` command terminates as soon as c_1 or c_2 terminates.

Exchange of values takes place over shared variables.

An example of a coroutine

```
var obs: int, c: int = 0, h: array [0..N-1] of int = { 0, ... }
coroutine p =
    begin
        while c < N do
            h[obs] := h[obs] + 1; c := c + 1;
            yield p
        done
    end
in
    ... obs = 12; call p; ...
    ... obs = 41; call p; ...
```

The coroutine maintains a histogram `h` of the observed values `obs`, and stops as soon as `N` values have been observed.

The client calls `p` on various values of `obs`.

Clint's rule for coroutines

Two assertions associated with coroutine p:

- A_p: the pre of `call` p, hence also the post of `yield` p;
- B_p: the pre of `yield` p, hence also the post of `call` p.

Clint's rule:

$$\frac{\{\,B_p\,\}\ \texttt{yield}\ p\ \{\,A_p\,\} \vdash \{\,A_p\,\}\ c_1\ \{\,Q\,\} \qquad \{\,A_p\,\}\ \texttt{call}\ p\ \{\,B_p\,\} \vdash \{\,P\,\}\ c_2\ \{\,Q\,\}}{\{\,P\,\}\ \texttt{coroutine}\ p = c_1\ \texttt{in}\ c_2\ \{\,Q\,\}}$$

(Note: same problems with the $X \vdash Y$ notation as for Clint-Hoare rule for `goto`; same solution.)

An example of verification

```
coroutine p =
    begin {Inv ∧ 0 ≤ obs < N}
        while c < N do
            h[obs] := h[obs] + 1;  c := c + 1;
            {Inv} yield p {Inv ∧ 0 ≤ obs < N}
        done
    end
in
    ... obs = 12; {Inv ∧ 0 ≤ obs < N} call p; {Inv} ...
```

The invariant *Inv* is $\mathtt{c} \leq \mathtt{N} \land \mathtt{c} = \Sigma_{i=0}^{\mathtt{N}-1}\, \mathtt{h}[i]$.

The precondition A_p of `call` is $Inv \land 0 \leq \mathtt{obs} < \mathtt{N}$.
It ensures that the access $\mathtt{h}[\mathtt{obs}]$ is within bounds.

The postcondition B_p is *Inv*.

Cooperative threads

A simple model of cooperative threads:

$$\texttt{run}\ c_1 \parallel \cdots \parallel c_n\ \texttt{end}$$

The executions of commands $c_1, \ldots, c_n$ are interleaved.

Each command performs `yield` to offer to suspend itself and give control to another command. Between two `yield`, execution is sequential.

The $\texttt{run} \ldots \texttt{end}$ terminates when all commands c_i have terminated.

Example: a producer-consumer model

```
var full: bool = false; var data: T = null;
run
    while true do                 ||   while true do
        x := produce();           ||       while not full do
        while full do             ||           yield
            yield                 ||       done;
        done;                     ||       y := data;
        data := x;                ||       full := false
        full := true              ||       consume(y)
    done                          ||   done
end
```

A rule for cooperative threads

A symmetrized version of Clint's rule for coroutines:

$$\frac{\{\, P \,\}\ \texttt{yield}\ \{\, P \,\} \vdash \{\, P \,\}\ c_i\ \{\, Q \,\} \quad \text{for } i = 1, \ldots, n}{\{\, P \,\}\ \texttt{run}\ c_1 \parallel \cdots \parallel c_n\ \texttt{end}\ \{\, Q \,\}}$$

The precondition P is the invariant at each "context switch" from a `yield` to the beginning of a c_i, or from `yield` to another `yield`.

Computation can start with any of the c_i and terminate with any of the c_i.

Verifying the producer-consumer schema

```
while true do {P}
    x := produce(); {P ∧ R(x)}
    while full do
       yield {P ∧ R(x)}
    done;
    {full = false ∧ P ∧ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ∧ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ∧ P}
    consume(y) {P}
done
```

Let $R(x)$ be an invariant over values x of type `T`, such that

$\{\, \texttt{true} \,\}\ x := \texttt{produce}()\ \{\, R(x) \,\}$

and $\{\, R(x) \,\}\ \texttt{consume}(x)\ \{\, \texttt{true} \,\}$.

Verifying the producer-consumer schema

```
while true do {P}
    x := produce(); {P ∧ R(x)}
    while full do
        yield {P ∧ R(x)}
    done;
    {full = false ∧ P ∧ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true {P}
done
```

‖

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ∧ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ∧ P}
    consume(y) {P}
done
```

The invariant for the coroutine is

$$P \stackrel{def}{=} \texttt{full} = \texttt{true} \Rightarrow R(\texttt{data})$$

It shows that all the values passed to `consume` satisfy R.

Separation logics for control operators

A small functional and imperative language

In the style of ML languages, using references to represent mutable state.

$$
\begin{array}{rll}
e ::= & cst \mid x \mid \lambda x.\, e \mid e_1\ e_2 & \text{functional constructs} \\
\mid & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 & \text{sequencing} \\
\mid & \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 & \text{conditional} \\
\mid & \ell & \text{location of a reference} \\
\mid & \mathtt{ref}\ e & \text{creating a reference} \\
\mid & !\, e \mid e_1 := e_2 & \text{dereference, assignment} \\
\mid & \mathtt{free}\ e & \text{freeing a reference}
\end{array}
$$

Notation: $e_1; e_2$ est $\mathtt{let}\ z = e_1\ \mathtt{in}\ e_2$ where z is not free in e_2.

Example: $\mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in}\ (\mathtt{if}\ b\ \mathtt{then}\ x := 1\ \mathtt{else}\ ()); !\, x$

Separation logic triples

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

The precondition P is an assertion.

The postcondition Q is a function value of $e \mapsto$ assertion.

$$\frac{v \text{ value} \qquad P \Rightarrow Q\,v}{\{\,P\,\}\; v\; \{\,Q\,\}} \qquad \frac{\{\,P\,\}\; e_1\; \{\,R\,\} \qquad \forall x,\; \{\,R\,x\,\}\; e_2\; \{\,Q\,\}}{\{\,P\,\}\; \mathtt{let}\; x = e_1\; \mathtt{in}\; e_2\; \{\,Q\,\}}$$

Separation logic assertions

Assertions that describe fragments of the memory state (sets of locations and their contents):

emp	the memory is empty
$\langle P \rangle$	the memory is empty and proposition P holds
$\ell \mapsto v$	the memory comprises one location ℓ containing value v
$\ell \mapsto \_$	the memory comprises one location ℓ $\quad (= \exists v, \ell \mapsto v)$
$P \star Q$	separating conjunction: the memory splits in two disjoint fragments, one satisfying P, the other satisfying Q
$P \mathrel{-\!\!\star} Q$	separating implication (*magic wand*): if we add to the memory a fragment satisfying P, we obtain a memory that satisfies Q.

Selected rules for separation logic

The "small" rules for mutable state:

$$
\begin{array}{rcl}
\{\,\mathtt{emp}\,\} & \mathtt{ref}\ v & \{\,\lambda\ell.\ \ell \mapsto v\,\} \\
\{\,\ell \mapsto v\,\} & !\,\ell & \{\,\lambda x.\ \langle x = v\rangle \ast \ell \mapsto v\,\} \\
\{\,\ell \mapsto \_\,\} & \ell := v & \{\,\lambda x.\ \langle x = ()\rangle \ast \ell \mapsto v\,\} \\
\{\,\ell \mapsto \_\,\} & \mathtt{free}\ \ell & \{\,\lambda x.\ \langle x = ()\rangle\,\}
\end{array}
$$

Combine with the frame rule to apply to larger memory states:

$$
\frac{\{\,P\,\}\ e\ \{\,Q\,\}}{\{\,P \ast R\,\}\ e\ \{\,\lambda x.\ Q\,x \ast R\,\}}
$$

Strengths of separation logic

1– We can reason locally on pointer programs without worrying about aliasing:

$$\{\, \ell_1 \mapsto 1 \star \ell_2 \mapsto v \,\}\; \ell_1 := 0 \;\{\, \ell_1 \mapsto 0 \star \ell_2 \mapsto v \,\}$$

No need to handle the case $\ell_1 = \ell_2$: the precondition is false in this case.

2– The logic keeps track of resources (memory, etc) and makes sure that they are used in a linear or affine way:

✔ $\{\, \mathtt{emp} \,\}\; \mathtt{let}\; x = \mathtt{ref}\; v\; \mathtt{in}\; \ldots; \mathtt{free}(x) \;\{\, \lambda\_.\mathtt{emp} \,\}$

✘ $\{\, \mathtt{emp} \,\}\; \mathtt{let}\; x = \mathtt{ref}\; v\; \mathtt{in}\; \ldots; \mathtt{free}(x); !x \;\{\, \lambda\_.\mathtt{emp} \,\}$ (use after free)

✘ $\{\, \mathtt{emp} \,\}\; \mathtt{let}\; x = \mathtt{ref}\; v\; \mathtt{in}\; \ldots; \mathtt{free}(x); \mathtt{free}(x) \;\{\, \lambda\_.\mathtt{emp} \,\}$ (double free)

✘ $\{\, \mathtt{emp} \,\}\; \mathtt{let}\; x = \mathtt{ref}\; v\; \mathtt{in}\; \ldots \;\{\, \lambda\_.\mathtt{emp} \,\}$ (memory leak)

Revisiting the producer-consumer schema

```
while true do {P}
    x := produce(); {P ✶ R(x)}
    while full do
        yield {P ✶ R(x)}
    done;
    {full = false ✶ P ✶ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ✶ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ✶ P}
    consume(y) {P}
done
```

In separation logic, the invariant R also describes the allocation and freeing of resources:

$\{\,\mathtt{emp}\,\}\ x := \mathtt{produce}()\ \{\,R(x)\,\}$ and $\{\,R(x)\,\}\ \mathtt{consume}(x)\ \{\,\mathtt{emp}\,\}$.

Revisiting the producer-consumer schema

$$
\begin{array}{l|l}
\texttt{while true do } \{P\} & \texttt{while true do } \{P\} \\
\quad \texttt{x := produce(); } \{P \star R(x)\} & \quad \texttt{while not full do} \\
\quad \texttt{while full do} & \quad\quad \texttt{yield } \{P\} \\
\quad\quad \texttt{yield } \{P \star R(x)\} & \quad \texttt{done;} \\
\quad \texttt{done;} & \quad \{\texttt{full} = \texttt{true} \star P\} \\
\quad \{\texttt{full} = \texttt{false} \star P \star R(x)\} & \quad \Rightarrow \{R(\texttt{data})\} \\
\quad \Rightarrow \{R(x)\} & \quad \texttt{y := data; } \{R(y)\} \\
\quad \texttt{data := x; } \{R(\texttt{data})\} & \quad \texttt{full := false; } \{R(y) \star P\} \\
\quad \texttt{full := true } \{P\} & \quad \texttt{consume(y) } \{P\} \\
\texttt{done} & \texttt{done}
\end{array}
$$

Take as invariant $P \stackrel{def}{=} \texttt{if full then } R(\texttt{data}) \texttt{ else emp}$
so that $\texttt{full} = \texttt{false} \star P \iff \texttt{emp}$
and $\texttt{full} = \texttt{true} \star P \iff R(\texttt{data})$.

Revisiting the producer-consumer schema

```
while true do {P}
    x := produce(); {P ✶ R(x)}
    while full do
       yield {P ✶ R(x)}
    done;
    {full = false ✶ P ✶ R(x)}
    ⇒ {R(x)}
    data := x; {R(data)}
    full := true {P}
done
```

||

```
while true do {P}
    while not full do
        yield {P}
    done;
    {full = true ✶ P}
    ⇒ {R(data)}
    y := data; {R(y)}
    full := false; {R(y) ✶ P}
    consume(y) {P}
done
```

We see the resources $R($`data`$)$ being transferred from the producer to the consumer. It shows that each resource allocated by `produce` is freed exactly once by `consume`.

An issue with control operators

$$\frac{\{\,P\,\}\;e_1\;\{\,R\,\} \qquad \forall x, \{\,R(x)\,\}\;e_2\;\{\,Q\,\}}{\{\,P\,\}\;\mathtt{let}\;x = e_1\;\mathtt{in}\;e_2\;\{\,Q\,\}}$$

A control operator such as `callcc` can invalidate the rule above: if e_1 captures its continuation, e_2 can be executed multiple times, the first time in a state that satisfies $R(x)$, the second time in a state not satisfying it.

An issue with control operators

(A. Timany, L. Birkedal, *Mechanized relational verification of concurrent programs with continuations*, ICFP 2019.)

$$e \stackrel{def}{=} \begin{array}{l} \mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in} \\ \mathtt{let}\ g = f\ ()\ \mathtt{in} \\ x := !\,x + 1; g\ (); !\,x \end{array}$$

Assuming that f is a pure function that returns a pure function:

$$\{\,\mathtt{emp}\,\}\, f\ ()\ \{\,\lambda g.\,\{\,\mathtt{emp}\,\}\, g\ ()\ \{\,\lambda\_.\mathtt{emp}\,\}\,\}$$

we can prove that e evaluates to 1:

$$\{\,\mathtt{emp}\,\}\, e\,\{\,\lambda v.\ v = 1\,\}$$

An issue with control operators

(A. Timany, L. Birkedal, *Mechanized relational verification of concurrent programs with continuations*, ICFP 2019.)

$$e \overset{def}{=} \begin{array}{l} \texttt{let}\ x = \texttt{ref}\ 0\ \texttt{in} \\ \texttt{let}\ g = f\ ()\ \texttt{in} \\ x := !\,x + 1; g\ (); !\,x \end{array}$$

However, if

$$f = \lambda().\texttt{callcc}\,(\lambda k.\,\lambda().\,\texttt{throw}\ k\,(\lambda().()))$$

the assignment $x := !\,x + 1$ is executed twice, and in the end $!\,x = 2$.

An issue with control operators

(A. Timany, L. Birkedal, *Mechanized relational verification of concurrent programs with continuations*, ICFP 2019.)

$$e \stackrel{def}{=} \begin{array}{l} \mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in} \\ \mathtt{let}\ g = f\ ()\ \mathtt{in} \\ x := !\,x + 1; g\ (); !\,x \end{array}$$

$$f = \lambda().\mathtt{callcc}\,(\lambda k.\,\lambda().\,\mathtt{throw}\,k\,(\lambda().()))$$

f is pure insofar as it does not modify the state. More precisely, when executed in the empty context, it satisfies the contract $\{\,\mathrm{emp}\,\}\,f\ ()\ \{\,\lambda g.\,\{\,\mathrm{emp}\,\}\,g\ ()\ \{\,\lambda\_.\mathrm{emp}\,\}\,\}$.

A logic for whole programs

Timany & Birkedal's approach: define the logic $\{ P \}\ e\ \{ Q \}$ for whole programs e.

The rules apply to decompositions $e = C[e_1]$,
where C is an evaluation context and e_1 an expression that can reduce. They look very much like the reduction rules!

$$\frac{\{ P \}\ C[e[x \leftarrow v]]\ \{ Q \}}{\{ P \}\ C[(\lambda x.\, e)\ v]\ \{ Q \}} \qquad \frac{\{ P \}\ C[e_1]\ \{ Q \}}{\{ P \}\ C[\texttt{if true then}\ e_1\ \texttt{else}\ e_2]\ \{ Q \}}$$

$$\frac{\{ P \}\ C[v\ (\texttt{cont}\ C)]\ \{ Q \}}{\{ P \}\ C[\texttt{callcc}\ v]\ \{ Q \}} \qquad \frac{\{ P \}\ D[v]\ \{ Q \}}{\{ P \}\ C[\texttt{throw}\ (\texttt{cont}\ D)\ v]\ \{ Q \}}$$

A logic for whole programs

Example of a verification:

$$\dfrac{\dfrac{\{\,\mathtt{emp}\,\}\; 5+2\; \{\,\lambda x.\, \langle x = 7\rangle\,\}}{\{\,\mathtt{emp}\,\}\; \mathtt{throw}\; (\mathtt{cont}\; ([\,]+2)\; 5+4)\; \{\,\lambda x.\, \langle x = 7\rangle\,\}}\,(**)}{\{\,\mathtt{emp}\,\}\; \mathtt{callcc}(\lambda k.\, \mathtt{throw}\; k\; 5+4)+2\; \{\,\lambda x.\, \langle x = 7\rangle\,\}}\,(*)$$

(*) uses the `callcc` rule with the context $C = [\,] + 2$.

(**) uses the `throw` rule with the context $C = [\,]$.

(See Timany and Birkedal's paper for more complex examples.)

Triples valid in all contexts

To facilitate verification, we define the Pour faciliter la vérification, on définit les triples valid in all contexts $\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}$ as those triples that validate the context rule contexte

$$\frac{\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, R \,\}\!\} \qquad \forall v,\ \{\, R\ v \,\}\ C[v]\ \{\, Q \,\}}{\{\, P \,\}\ C[e]\ \{\, Q \,\}}$$

We can define rules for $\{\!\{\, P \,\}\!\}\ e\ \{\!\{\, Q \,\}\!\}$ that look very much like the usual separation logic rules, for all kinds of expressions e except `callcc` and `throw`. (See the paper.)

Towards a logic for effect handlers

User-defined effects and effect handlers ought to support reasoning rules that are simpler than those for `callcc`:

- delimited continuations,
- which can be specified in advance by contracts: precondition on the arguments / postcondition on the results (like functions are specified);

provided that

- continuations can only be used once (one-shot continuations).

An issue with multiple-shot continuations

$$\frac{\{\,P\,\}\; e_1\; \{\,R\,\} \qquad \forall x, \{\,R(x)\,\}\; e_2\; \{\,Q\,\}}{\{\,P\,\}\; \texttt{let}\; x = e_1\; \texttt{in}\; e_2\; \{\,Q\,\}}$$

This rule is invalid if e_1 can return several times. Example:

```
handle
    let b = perform Flip in x := !x + 1
with
    val(x) -> x
    Flip(_, k) -> k false; k true
```

`x` is incremented twice, not once as predicted by the `let` rule with $P = \mathtt{x} \mapsto 0$ and $Q = \lambda\_.\ \mathtt{x} \mapsto 1$.

Effect protocols

(P. E. de Vilhena, F. Pottier, *A separation logic for effect handlers*, POPL 2021.)

A specification of the behaviors of effects. Acts as a contract between effect producers and effect handlers.

$$
\begin{array}{rll}
\Psi ::= & \bot & \text{no effect} \\
\mid & !\,\vec{x}\,(F\ v)\ \{\,P\,\}.\ ?\,\vec{y}\,(w)\ \{\,Q\,\} & \text{protocol for } F \\
\mid & \Psi_1 + \Psi_2 & \text{union of two protocols}
\end{array}
$$

The protocol $!\,\vec{x}\,(F\ v)\ \{\,P\,\}.\ ?\,\vec{y}\,(w)\ \{\,Q\,\}$ reads as:
"for all $\vec{x}$, the program can perform effect F with argument value v, provided that the precondition P holds; then, there exists $\vec{y}$ such that the result w of F satisfies the postcondition Q".

Examples of protocols

The `Abort` effect, which never returns:

$$! \; (\texttt{Abort}\ ())\ \{\, \texttt{true} \,\}.\ ?\, y\ (y)\ \{\, \texttt{false} \,\}$$

The `Next` effect, which simulates a counter:

$$!\, n\ (\texttt{Next}\ ())\ \{\, \mathit{Count}\ n \,\}.\ ?\ \ (n)\ \{\, \mathit{Count}\ (n+1) \,\}$$

The abstract predicate *Count n* keeps trace of the current value of the counter.

The `Get` and `Set` effects, which simulate a reference:

$$\begin{aligned} & !\, v\ (\texttt{Get}\ ())\ \{\, \mathit{State}\ v \,\}.\ ?\ \ (v)\ \{\, \mathit{State}\ v \,\} \\ + \quad & !\, v\, v'\ (\texttt{Set}\ v')\ \{\, \mathit{State}\ v \,\}.\ ?\ \ (())\ \{\, \mathit{State}\ v' \,\} \end{aligned}$$

The abstract predicate *State v* keeps trace of the current value of the reference.

A Hoare quadruple

$$\{\, P \,\}\; e \;\langle\, \Psi \,\rangle\; \{\, Q \,\}$$

The protocol Ψ plays the role of an extra postcondition.

In particular, $\{\, P \,\}\; e \;\langle\, \bot \,\rangle\; \{\, Q \,\}$ guarantees that e performs no unhandled effect.

The protocol “distributes over” computations that do not perform effects:

$$\frac{v \text{ value} \qquad P \Rightarrow Q\, v}{\{\, P \,\}\; v \;\langle\, \Psi \,\rangle\; \{\, Q \,\}}$$

$$\frac{\{\, P \,\}\; e_1 \;\langle\, \Psi \,\rangle\; \{\, R \,\} \qquad \forall x,\ \{\, R\, x \,\}\; e_2 \;\langle\, \Psi \,\rangle\; \{\, Q \,\}}{\{\, P \,\}\; \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 \;\langle\, \Psi \,\rangle\; \{\, Q \,\}}$$

Performing an effect

$$\frac{P \Rightarrow \Psi \ \texttt{allows}\ (F\,v')\,\{\,Q\,\}}{\{\,P\,\}\ \texttt{perform}\ F\,v'\ \langle\,\Psi\,\rangle\ \{\,Q\,\}}$$

$\bot\ \texttt{allows}\ (F\,v')\,\{\,Q\,\}$ is always false.

$\Psi_1 + \Psi_2\ \texttt{allows}\ (F\,v')\,\{\,Q\,\}$ is the disjunction
$\Psi_1\ \texttt{allows}\ (F\,v')\,\{Q\} \vee \Psi_2\ \texttt{allows}\ (F\,v')\,\{\,Q\,\}$.

$!\,\vec{x}\,(F\,v)\,\{\,A\,\}.\ ?\,\vec{y}\,(w)\,\{\,B\,\}\ \texttt{allows}\ (F\,v')\,\{\,Q\,\}$ holds if

$$\exists\vec{x},\ \langle v' = v\rangle \ast A \ast (\forall\vec{y},\ B \mathrel{-\!\!\ast} Q(w))$$

Read: if we choose $\vec{x}$ so that the effective argument v' and the formal parameter v are equal, the precondition A of F must hold, and for any choice of $\vec{y}$ that satisfies the postcondition B of F, the postcondition $Q(w)$ holds.

Effect handling

$$\frac{\{\,P\,\}\;e\;\langle\,\Psi\,\rangle\;\{\,Q\,\} \qquad \mathit{isHandler}\;\langle\,\Psi\,\rangle\;\{\,Q\,\}\;(e_{val}, e_{eff})\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}{\{\,P\,\}\;\texttt{handle}\;e\;\texttt{with}\;e_{val}, e_{eff}\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}$$

As always, the purpose of the handler is to transform the results $\langle\Psi\rangle\;\{\,Q\,\}$ of the computation e that is being handled into results $\langle\Psi'\rangle\;\{\,Q'\,\}$.

If e terminates normally, its value v satisfies Q, and $e_{val}\;v$ is executed. Therefore, this computation must satisfy

$$\{\,Q(v)\,\}\;e_{val}\;v\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}$$

Effect handling

$$\frac{\{\,P\,\}\; e\;\langle\,\Psi\,\rangle\;\{\,Q\,\} \qquad \mathit{isHandler}\;\langle\,\Psi\,\rangle\;\{\,Q\,\}\;(e_{val}, e_{eff})\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}{\{\,P\,\}\;\texttt{handle}\;e\;\texttt{with}\;e_{val}, e_{eff}\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}$$

If e terminates by performing an effect with value v and continuation k, $e_{eff}\ v\ k$ is executed, and must satisfy

$$\{\,R\,\}\; e_{eff}\; v\; k\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}$$

The precondition R could say something like: if v is $F\ v'$ and the protocol Ψ associates to F the pre A and the post B, then

- v' satisfies A;
- k is a function with pre B and post $\langle\,\Psi'\,\rangle\;\{\,Q'\,\}$.

Effect handling

$$\frac{\{\, P \,\}\; e \;\langle\, \Psi \,\rangle\; \{\, Q \,\} \qquad \mathit{isHandler}\; \langle\, \Psi \,\rangle\; \{\, Q \,\}\; (e_{val}, e_{eff})\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\}}{\{\, P \,\}\; \texttt{handle}\; e\; \texttt{with}\; e_{val}, e_{eff}\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\}}$$

More simply, R states that v and k are any values and continuations that are permitted by the protocol Ψ:

$$R \stackrel{def}{=} \Psi \;\texttt{allows}\; v\; \{\, \lambda w.\, \{\, \texttt{emp} \,\}\; k\; w\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\} \,\}$$

As a bonus, the Iris formalization of this theory uses "non-persistent triples", hence $\{\, \texttt{emp} \,\}\; k\; w\; \langle\, \Psi' \,\rangle\; \{\, Q' \,\}$ gives the permission to invoke continuation k only once!

Effect handling

$$\frac{\{\,P\,\}\; e \;\langle\,\Psi\,\rangle\;\{\,Q\,\} \qquad \mathit{isHandler}\;\langle\,\Psi\,\rangle\;\{\,Q\,\}\;(e_{val}, e_{eff})\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}{\{\,P\,\}\;\texttt{handle}\;e\;\texttt{with}\;e_{val}, e_{eff}\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}}$$

Putting it all together, we have

$$\begin{aligned}
&\mathit{isHandler}\;\langle\,\Psi\,\rangle\;\{\,Q\,\}\;(e_{val}, e_{eff})\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\} \overset{def}{=} \\
&\qquad (\forall v,\; \{\,Q(v)\,\}\; e_{val}\; v\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}\,) \\
&\quad \wedge\quad (\forall v, k,\; \{\,\Psi\;\;\texttt{allows}\;\; v\;\{\,\lambda w.\,\{\,\texttt{emp}\,\}\;k\;w\;\langle\,\Psi'\,\rangle\;\{\,Q'\,\}\,\}\,\} \\
&\qquad\qquad e_{eff}\; v\; k \\
&\qquad\qquad \langle\,\Psi'\,\rangle\;\{\,Q'\,\}
\end{aligned}$$

This describes a shallow handler. For a deep handler, see the paper.

Summary

Summary

Designed initially for structured control, program logics such as Hoare logic and separation logic extend fairly easily

- to `goto` jumps, `break`/`continue` exits, and exceptions;
- to coroutines and cooperative threads;
- to first-order functions. (not treated in this lecture)

Other language features are more problematic:

- higher-order functions; (not treated in this lecture);
- control operators.

The additional structure provided by effect handlers compared with `callcc` is helpful.

References

References

An introduction to Hoare logic and separation logic:

- My 2020–2021 course on “Program logics”, lectures #1 to #3.

A logic for `callcc`:

- Amin Timany, Lars Birkedal: *Mechanized Relational Verification of Concurrent Programs with Continuations*, PACMPL 3(ICFP), 2019.

A logic for user-defined effects and effect handlers:

- Paulo Emílio de Vilhena, François Pottier: *A Separation Logic for Effect Handlers*, PACMPL 5(POPL), 2021.

THE END