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)**

```c
/*@ 
  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:

preconditions $\Rightarrow$ invariants $\Rightarrow$ 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:

\[ P_1 \lor P_2 \Rightarrow Q \]
\[ P \land b \Rightarrow Q_1 \]
\[ P \land \neg b \Rightarrow Q_0 \]
\[ (\exists x_0, x = f(x_0, \vec{y}) \land P(x_0, \vec{y})) \Rightarrow Q \]
Hoare’s approach


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 \} \quad \text{if } P \text{ holds “before” and if } c \text{ terminates,}

\text{then } Q \text{ holds “after”}

“Strong” Hoare logic: (full correctness)

\left[ P \right] c \left[ Q \right] \quad \text{if } P \text{ holds “before”,}

\text{then } c \text{ terminates and } Q \text{ holds “after”}
Structured control:

\[
\begin{align*}
\{ P \} \ c_1 \ { Q } & \quad \{ Q \} \ c_2 \ { R } \\
\hline
\{ P \} \ c_1 ; \ c_2 \ { R } \\
\{ P ∧ b \} \ c_1 \ { Q } & \quad \{ P ∧ ¬b \} \ c_2 \ { Q } \\
\hline
\{ P \} \ \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \ { Q } \\
\{ P ∧ b \} \ c \ { P } \\
\hline
\{ P \} \ \text{while} \ b \ \text{do} \ c \ { P ∧ ¬b } 
\end{align*}
\]
The rules of weak Hoare logic

Empty command:

\{ P \} \text{skip} \{ P \}

Assignment:

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

Consequence:

\[
\begin{align*}
P \Rightarrow P' & \quad \{ P' \} \ c \ \{ Q' \} \quad Q' \Rightarrow Q \\
\hline
& \quad \{ P \} \ c \ \{ Q \}
\end{align*}
\]

9
Example of verification: Euclidean division

\[ \{0 \leq a\} \implies \{ a = b \cdot 0 + a \land 0 \leq a \} \]

\[ r := a; \]
\[ \{ a = b \cdot 0 + r \land 0 \leq r \} \]

\[ q := 0; \]
\[ \{ a = b \cdot q + r \land 0 \leq r \} \]

while \( r \geq b \) do

\[ \{ a = b \cdot q + r \land 0 \leq r \land r \geq b \} \implies \]
\[ \{ a = b \cdot (q+1) + (r - b) \land 0 \leq r - b \} \]

\[ r := r - b; \]
\[ \{ a = b \cdot (q+1) + r \land 0 \leq r \} \]

\[ q := q + 1 \]
\[ \{ a = b \cdot q + r \land 0 \leq r \} \]

done

\[ \{ a = b \cdot q + r \land 0 \leq r \land r < b \} \implies \]
\[ \{ q = a/b \land r = a \text{ mod } b \} \]
Extending Hoare logic to various control structures
Other kinds of loops

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

\[
\{ P \} \ c \ \{ Q \} \quad Q \land b \Rightarrow P
\]

\[
\{ P \} \ \text{do} \ c \ \text{while} \ b \ \{ Q \land \neg b \}
\]

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

\[
\{ P \} \ c_1 \ \{ Q \} \quad \{ Q \land \neg b \} \ c_2 \ \{ P \}
\]

\[
\{ P \} \ \text{loop} \ c_1; \ \text{exit when} \ b; \ c_2 \ \text{end} \ \{ Q \land b \}
\]

A counted for loop:

\[
[P \land i \leq h] \ c \ [P[i \leftarrow i + 1]] \quad i, h \ \text{not assigned in} \ c
\]

\[
[P[i \leftarrow \ell]] \ \text{for} \ i = \ell \ \text{to} \ h \ \text{do} \ c \ [P \land i > h]
\]
Non-determinism

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

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

Dijkstra’s “guarded conditional”:
executes one of the $c_i$ for which the condition $b_i$ is true.

\[
\{ P \land b_i \} \ c_i \{ Q \} \quad \text{for } i = 1, \ldots, n
\]

\[
\{ P \land (b_1 \lor \cdots \lor b_n) \} \ \text{if} \ b_1 \rightarrow c_1 \quad \cdots \quad b_n \rightarrow c_n \quad \text{fi} \ \{ Q \} 
\]
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.

A Hoare logic for "goto"?

Consider `goto L` 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.

```plaintext
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.
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.

```plaintext
begin
  ...
  \{R\}  goto L \{Q_1\}
  ...
  L: \{R\}
  ...
  begin ... \{R\} goto L \{Q_2\} ... 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”


\[
\begin{align*}
\{ R \} \text{goto} L \{ \text{false} \} & \vdash \{ P \} \ c_1 \{ R \} \\
\{ R \} \text{goto} L \{ \text{false} \} & \vdash \{ R \} \ c_2 \{ Q \} \\
\hline
\{ P \} \begin{array}{l}
\text{begin } c_1; \text{ L: } c_2 \text{ end }
\end{array} \{ Q \}
\end{align*}
\]

\( X \vdash Y \) reads as a hypothetical deduction in natural deduction: “assuming \( X \) we can derive \( Y \).”

From the hypothesis \( \{ R \} \text{goto} L \{ \text{false} \} \) we can derive \( \{ R \} \text{goto} L \{ Q \} \) for any \( Q \), using the consequence rule.
Problems with Clint and Hoare’s rule


In case of nested blocks

```
begin ... begin ... L : ... end ... L : ... end
```

“the” precondition associated with $L$ is ambiguous:

\[
\{ R_1 \} \text{ goto } L \{ \text{false} \} \vdash (\{ R_2 \} \text{ goto } L \{ \text{false} \} \vdash X)
\]

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 $X = Y = \text{false}$, and deduce

\[
\{ \text{false} \} \text{ goto } L \{ \text{false} \} \implies \{ \text{true} \} \text{ goto } L \{ \text{false} \}
\]

\[
\times \{ \text{true} \} \text{ begin goto } L; \ L: \text{skip end } \{ \text{false} \}
\]
The Arbib-Alagic-de Bruin approach

A. de Bruin, *Goto statements: semantics and deduction systems*,
Acta Informatica 15, 1981.)

Idea: \texttt{goto} is another way to exit a command $c$, in addition to normal termination. Let’s give \texttt{goto} an extra postcondition $J$.

\[
\{ P \} \; c \; \{ Q \} \; \{ J \}
\]

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

\[
P' \implies P \quad \{ P \} \; c \; \{ Q \} \; \{ J \} \quad Q \implies Q' \quad \forall L, \; J(L) \implies J'(L)
\]

\[
\{ P' \} \; c \; \{ Q' \} \; \{ J' \}
\]
The Arbib-Alagic-de Bruin approach

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

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

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

$$\{ P \} \ c_1 \ \{ R \} \ \{ J \} \quad \{ R \} \ c_2 \ \{ Q \} \ \{ J \}$$

$$\frac{\{ P \} \ c_1 ; c_2 \ \{ Q \} \ \{ J \}}{\{ P \} \ if \ b \ then \ c_1 \ else \ c_2 \ \{ Q \} \ \{ J \}}$$
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:

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

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

$$\{ P \} c_1 \{ R \} \{ J[L ← R] \} \quad \{ R \} c_2 \{ Q \} \{ J[L ← R] \}$$

$$\{ P \} \text{begin } c_1; L : c_2 \text{ end } \{ Q \} \{ J \}$$
Early exits from loops

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

\[
\{ P \} \ c \ \{ Q \} \ \{ B \} \quad (B = \text{precondition for } \text{break})
\]

Selected rules:

\[
\{ P \} \ \text{break} \ \{ \text{false} \} \ \{ P \}
\]

\[
\{ P \land b \} \ c \ \{ P \} \ \{ Q \} \quad P \land \lnot b \Rightarrow Q
\]

\[
\{ P \} \ \text{while } b \ \text{do } c \ \{ Q \} \ \{ B \}
\]

\[
\{ P \land b \} \ c \ \{ P \} \ \{ Q \} \quad \{ P \land \lnot b \} \ c' \ \{ Q \} \ \{ B \}
\]

\[
\{ P \} \ \text{while } b \ \text{do } c \ \text{else } c' \ \{ Q \} \ \{ B \}
\]
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

\[ Q : \text{kind of exit} \rightarrow \text{assertion} \]

Exit kinds \( K \) are, for example,

\[
K ::= \text{norm} \\
\quad \mid \text{break} \mid \text{continue} \\
\quad \mid \text{break}(n) \mid \text{continue}(n) \\
\quad \mid \text{return}(v) \\
\quad \mid \text{goto}(L) \\
\quad \mid \text{exn}(E)
\]

normal termination

loop exits

multi-level exits

function return

jump

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

\[
\begin{align*}
\{ P \} & \text{ skip } \{ [\text{norm} \mapsto P] \} \\
\{ P \} & \text{ break } \{ [\text{break}(1) \mapsto P] \} \\
\{ P \} & \text{ break } n \{ [\text{break}(n) \mapsto P] \} \\
\{ P \} & \text{ goto } L \{ [\text{goto}(L) \mapsto P] \} \\
\{ P \} & \text{ raise } E \{ [\text{exn}(E) \mapsto P] \}
\end{align*}
\]

We write \( [T \mapsto P] \overset{\text{def}}{=} \lambda T'. \text{ if } T' = T \text{ then } P \text{ else } \text{false} \).
A unified treatment of multiple exits

The sequence “handles” normal termination:

\[
\{ P \} \ c_1 \ \{ Q[\text{norm} \leftarrow R] \} \quad \{ R \} \ c_2 \ \{ Q \}
\]

\[
\{ P \} \ c_1; c_2 \ \{ Q \}
\]

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

\[
Q' = Q \begin{bmatrix}
\text{norm} \leftarrow P; \\
\text{break}(1) \leftarrow Q(\text{norm}); \\
\text{break}(n + 1) \leftarrow Q(\text{break}(n)) \\
\text{continue}(1) \leftarrow P; \\
\text{continue}(n + 1) \leftarrow Q(\text{continue}(n))
\end{bmatrix}
\]

\[
\{ P \land b \} \ c \ \{ Q' \} \quad P \land \lnot b \Rightarrow Q(\text{norm})
\]

\[
\{ P \} \text{ while } b \text{ do } c \ \{ Q \}
\]
The declaration of label $L$ “handles” the $\text{goto}(L)$ exit:

$$
\begin{array}{c}
\{ P \} ~ c_1 \{ Q[\text{norm} \leftarrow R, \text{goto}(L) \leftarrow R] \} \\
\{ R \} ~ c_2 \{ Q[\text{goto}(L) \leftarrow R] \}
\end{array}
$$

$$
\{ P \} \text{begin} ~ c_1; ~ L : c_2 \text{ end} \{ Q \}
$$

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

$$
\begin{array}{c}
\{ P \} ~ c_1 \{ Q[\text{exn}(E) \leftarrow R] \} \quad \{ R \} ~ c_2 \{ Q \}
\end{array}
$$

$$
\{ P \} \text{try} ~ c_1 \text{ catch} ~ E \rightarrow c_2 \{ Q \}
$$
A simple model of asymmetric coroutines:

\[ \text{coroutine } p = c_1 \text{ in } c_2 \]

When the consumer \( c_2 \) performs \text{call } p, the execution of \( c_1 \) starts or resumes just after the most recent \text{yield } p.

When the generator \( c_1 \) performs \text{yield } p, the execution of \( c_2 \) resumes just after the most recent \text{call } p.

The \text{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 \( \text{obs} \), and stops as soon as \( N \) values have been observed.

The client calls \( p \) on various values of \( \text{obs} \).
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:

\[
\begin{align*}
\{ B_p \} & \text{ yield } p \{ A_p \} \vdash \{ A_p \} \ c_1 \{ Q \} \\
\{ A_p \} & \text{ call } p \{ B_p \} \vdash \{ P \} \ c_2 \{ Q \} \\
\hline
\{ P \} & \text{coroutine } p = c_1 \ \text{in} \ c_2 \{ Q \}
\end{align*}
\]

(Note: same problems with the $X \vdash Y$ notation as for Clint-Hoare rule for goto; same solution.)
coroutine p =
  begin \{Inv \land 0 \leq obs < N\}
    while c < N do
      h[obs] := h[obs] + 1; c := c + 1;
      \{Inv\} yield p \{Inv \land 0 \leq obs < N\}
    done
  end
in
  ... obs = 12; \{Inv \land 0 \leq obs < N\} call p; \{Inv\} ...

The invariant Inv is \( c \leq N \land c = \sum_{i=0}^{N-1} h[i] \).

The precondition \( A_p \) of call is \( Inv \land 0 \leq obs < N \).
It ensures that the access \( h[obs] \) is within bounds.

The postcondition \( B_p \) is \( Inv \).
Cooperative threads

A simple model of cooperative threads:

\[
\text{run } c_1 \parallel \cdots \parallel c_n \text{ end}
\]

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

Each command performs \texttt{yield} to offer to suspend itself and give control to another command. Between two \texttt{yield}, execution is sequential.

The \texttt{run} \ldots \texttt{end} terminates when all commands $c_i$ have terminated.
Example: a producer-consumer model

```plaintext
var full: bool = false; var data: T = null;

run

  while true do
    x := produce();
    while full do
      yield
    done;
    data := x;
    full := true
    done
  end

while true do
  while not full do
    yield
  done;
  y := data;
  full := false
  consume(y)
  done
```

end
A rule for cooperative threads

A symmetrized version of Clint’s rule for coroutines:

\[
\begin{align*}
\{ P \} \text{ yield } \{ P \} &\vdash \{ P \} c_i \{ Q \} \quad \text{for } i = 1, \ldots, n \\
\{ P \} \text{ run } c_1 \parallel \cdots \parallel c_n \text{ end } \{ Q \}
\end{align*}
\]

The precondition \( P \) is the invariant at each “context switch” from a \texttt{yield} to the beginning of a \( c_i \), or from \texttt{yield} to another \texttt{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\}
    \(x := \text{produce}(); \{P \wedge R(x)\}\)
    while full do
        yield \{P \wedge R(x)\}
    done;
    \{\text{full} = \text{false} \wedge P \wedge R(x)\} \Rightarrow \{R(x)\}
data := x; \{R(\text{data})\}
full := \text{true} \{P\}
\}
done

while true do \{P\}
    \{\text{while not full do}
        yield \{P\}
    done;
    \{\text{full} = \text{true} \wedge P\} \Rightarrow \{R(\text{data})\\}
y := \text{data}; \{R(y)\}
full := \text{false}; \{R(y) \wedge P\}
\text{consume}(y) \{P\}
done

Let \(R(x)\) be an invariant over values \(x\) of type \(T\), such that
\{true\} \(x := \text{produce}()\) \{R(x)\}
and \{R(x)\} \text{consume}(x) \{\text{true}\}.
Verifying the producer-consumer schema

while true do \{P\}
x := produce(); \{P \land R(x)\}
while full do
    yield \{P \land R(x)\}
done;
\{full = false \land P \land R(x)\}
\Rightarrow \{R(x)\}
data := x; \{R(data)\}
full := true \{P\}
done

while true do \{P\}
    while not full do
        yield \{P\}
done;
\{full = true \land P\}
\Rightarrow \{R(data)\}
y := data; \{R(y)\}
full := false; \{R(y) \land P\}
consume(y) \{P\}
done

The invariant for the coroutine is

\[ P \overset{\text{def}}{=} \text{full} = \text{true} \Rightarrow R(data) \]

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.

\[ e ::= \text{cst} \mid x \mid \lambda x. e \mid e_1 \ e_2 \]

functional constructs

\[ \mid \ \text{let} \ x = e_1 \ \text{in} \ e_2 \]

sequencing

\[ \mid \ \text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3 \]

conditional

\[ \mid \ell \]

location of a reference

\[ \mid \text{ref} \ e \]

creating a reference

\[ \mid !e \mid e_1 := e_2 \]

dereference, assignment

\[ \mid \text{free} \ e \]

freeing a reference

Notation: \( e_1; e_2 \) est let \( z = e_1 \) in \( e_2 \) where \( z \) is not free in \( e_2 \).

Example: let \( x = \text{ref} \ 0 \) in (if \( b \) then \( x := 1 \) 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.

\[\begin{align*}
\text{v value} & \quad P \Rightarrow Q \land v \\
\{P\} & \land \{Q\}
\end{align*}\]

\[\begin{align*}
\{P\} \ e_1 \ \{R\} & \quad \forall x, \ \{R \ x\} \ e_2 \ \{Q\} \\
\{P\} & \text{let } x = e_1 \ \text{in} \ e_2 \ \{Q\}
\end{align*}\]
Separation logic assertions

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

- **emp** the memory is empty
- **⟨P⟩** the memory is empty and proposition P holds
- **ℓ ↷ v** the memory comprises one location ℓ containing value v
- **ℓ ↷ _** the memory comprises one location ℓ (= ∃v, ℓ ↷ v)

- **P ⋆ Q** separating conjunction:
  the memory splits in two disjoint fragments, one satisfying P, the other satisfying Q

- **P →⋆ 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:

\[
\begin{align*}
\{ \text{emp} \} & \quad \text{ref } v & \{ \lambda \ell. \ell \mapsto v \} \\
\{ \ell \mapsto v \} & \quad ! \ell & \{ \lambda x. \langle x = v \rangle \ast \ell \mapsto v \} \\
\{ \ell \mapsto _\bot \} & \quad \ell := v & \{ \lambda x. \langle x = () \rangle \ast \ell \mapsto v \} \\
\{ \ell \mapsto _\bot \} & \quad \text{free } \ell & \{ \lambda x. \langle x = () \rangle \}
\end{align*}
\]

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:

\[
\{ \ell_1 \mapsto 1 \star \ell_2 \mapsto v \} \quad \ell_1 := 0 \quad \{ \ell_1 \mapsto 0 \star \ell_2 \mapsto v \}
\]

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:

\[
\checkmark \quad \{ \text{emp} \} \quad \text{let } x = \text{ref } v \text{ in } \ldots ; \text{free}(x) \quad \{ \lambda_.\text{emp} \}
\]

\[
\xmark \quad \{ \text{emp} \} \quad \text{let } x = \text{ref } v \text{ in } \ldots ; \text{free}(x); !x \quad \{ \lambda_.\text{emp} \} \quad \text{(use after free)}
\]

\[
\xmark \quad \{ \text{emp} \} \quad \text{let } x = \text{ref } v \text{ in } \ldots ; \text{free}(x); \text{free}(x) \quad \{ \lambda_.\text{emp} \} \quad \text{(double free)}
\]

\[
\xmark \quad \{ \text{emp} \} \quad \text{let } x = \text{ref } v \text{ in } \ldots \quad \{ \lambda_.\text{emp} \} \quad \text{(memory leak)}
\]
Revisiting the producer-consumer schema

while true do \{P\}
  \( x := \text{produce}() \); \{P \triangleright R(x)\}
  while full do
    yield \{P \triangleright R(x)\}
  done;
  \{full = \text{false} \triangleright P \triangleright R(x)\}
  \Rightarrow \{R(x)\}
  data := x; \{R(data)\}
  full := \text{true} \{P\}
  done

while true do \{P\}
  while not full do
    yield \{P\}
  done;
  \{full = \text{true} \triangleright P\}
  \Rightarrow \{R(data)\}
  y := data; \{R(y)\}
  full := \text{false}; \{R(y) \triangleright P\}
  consume(y) \{P\}
  done

In separation logic, the invariant \( R \) also describes the allocation and freeing of resources:
\[
\{\text{emp}\}x := \text{produce}() \{ R(x) \} \text{ and } \{ R(x) \} \text{ consume}(x) \{\text{emp}\}.
\]
Revisiting the producer-consumer schema

while true do \{P\}
x := produce(); \{P \star R(x)\}
while full do
  yield \{P \star R(x)\}
done;
{full = false \star P \star R(x)} \Rightarrow \{R(x)\}
data := x; \{R(data)\}
full := true \{P\}
done

while true do \{P\}
while not full do
  yield \{P\}
done;
{full = true \star P} \Rightarrow \{R(data)\}
y := data; \{R(y)\}
full := false; \{R(y) \star P\}
consume(y) \{P\}
done

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

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.)

```
edef = let x = ref 0 in
       let g = f () in
       x := !x + 1; g (); !x
```

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

```
{ emp } f () { λg. { emp } g () { λ_.emp } }
```

we can prove that $e$ evaluates to 1:

```
{ emp } e { λv. v = 1 }
```
An issue with control operators

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

\[ e \overset{def}{=} \text{let } x = \text{ref } 0 \text{ in } \]
\[ \quad \text{let } g = f () \text{ in } \]
\[ \quad x := !x + 1; g (); !x \]

However, if

\[ f = \lambda().\text{callcc}(\lambda k. \lambda().\text{throw } k (\lambda().())) \]

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.)

\[
\begin{align*}
  e & \overset{\text{def}}{=} \text{let } x = \text{ref 0 in} \\
  & \quad \text{let } g = f () \text{ in} \\
  & \quad x := !x + 1; g (); !x
\end{align*}
\]

\[
f = \lambda().\text{callcc} (\lambda k. \lambda().\text{throw } k (\lambda().()))
\]

\(f\) is pure insofar as it does not modify the state. More precisely, when executed in the empty context, it satisfies the contract

\[
\{ \text{emp} \} f () \{ \lambda g. \{ \text{emp} \} g () \{ \lambda_.\text{emp} \} \}.
\]
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!

\[
\begin{align*}
\text{\{P\}} & \ C[e[x \leftarrow v]] \ \{Q\} \\
\text{\{P\}} & \ C[(\lambda x. e) v] \ \{Q\} \\
\text{\{P\}} & \ C[v \ (\text{cont } C)] \ \{Q\} \\
\text{\{P\}} & \ C[\text{callcc } v] \ \{Q\}
\end{align*}
\]

\[
\begin{align*}
\text{\{P\}} & \ C[e_1] \ \{Q\} \\
\text{\{P\}} & \ C[\text{if } \text{true then } e_1 \ \text{else } e_2] \ \{Q\} \\
\text{\{P\}} & \ D[v] \ \{Q\} \\
\text{\{P\}} & \ C[\text{throw } (\text{cont } D) v] \ \{Q\}
\end{align*}
\]
Example of a verification:

\[
\begin{align*}
\{ \text{emp} \} & \ 5 + 2 \ \{ \lambda x. \langle x = 7 \rangle \} \\
\{ \text{emp} \} & \ \text{throw} \ (\text{cont} ([ ] + 2) \ 5 + 4) \ \{ \lambda x. \langle x = 7 \rangle \} \\
\{ \text{emp} \} & \ \text{callcc} (\lambda k. \ \text{throw} \ k \ 5 + 4) + 2 \ \{ \lambda x. \langle x = 7 \rangle \}
\end{align*}
\]

(*) 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.)
Tripples 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

\[
\{\{ P \}\} e \{\{ R \}\} \quad \forall v, \{ R v \} C[v] \{ Q \}
\]

\{ 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.)
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

\[
\begin{align*}
\{ P \} e_1 \{ R \} & \quad \forall x, \{ R(x) \} e_2 \{ Q \} \\
\{ P \} \text{let } x = e_1 \text{ in } e_2 \{ Q \}
\end{align*}
\]

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

```plaintext
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 \texttt{let} rule with \( P = x \mapsto 0 \) and \( Q = \lambda_. \ x \mapsto 1. \)
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.

\[ \Psi ::= \bot \quad \text{no effect} \]
\[ \mid ! \vec{x} (F, v) \{ P \}. ? \vec{y} (w) \{ Q \} \quad \text{protocol for } F \]
\[ \mid \Psi_1 + \Psi_2 \quad \text{union of two protocols} \]

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:

\[ ! (\text{Abort}()) \{ \text{true} \}. \ ? y (y) \{ \text{false} \} \]

The Next effect, which simulates a counter:

\[ ! n (\text{Next}()) \{ \text{Count} n \}. \ ? (n) \{ \text{Count} (n + 1) \} \]

The abstract predicate Count \( n \) keeps trace of the current value of the counter.

The Get and Set effects, which simulate a reference:

\[ ! v (\text{Get}()) \{ \text{State} v \}. \ ? (v) \{ \text{State} v \} \]
\[ + \ ! v v' (\text{Set} v') \{ \text{State} v \}. \ ? (()) \{ \text{State} v' \} \]

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 \( \Psi \) 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:

\[
\begin{align*}
&v \text{ value} \quad P \Rightarrow Q \ v \\
&\{ P \} \ v \langle \Psi \rangle \ { Q } \\
&\{ P \} \ e_1 \langle \Psi \rangle \ { R } \quad \forall x, \ \{ R \ x \} \ e_2 \langle \Psi \rangle \ { Q } \\
&\{ P \} \ \text{let } x = e_1 \ \text{in} \ e_2 \langle \Psi \rangle \ { Q } 
\end{align*}
\]
Performing an effect

\[ P \Rightarrow \Psi \text{ allows } (F v') \{ Q \} \]

\[ \{ P \} \text{ perform } F v' \langle \Psi \rangle \{ Q \} \]

\( \bot \text{ allows } (F v') \{ Q \} \) is always false.

\( \Psi_1 + \Psi_2 \text{ allows } (F v') \{ Q \} \) is the disjunction

\( \Psi_1 \text{ allows } (F v') \{ Q \} \lor \Psi_2 \text{ allows } (F v') \{ Q \} \).

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

\[ \exists \vec{x}, \langle v' = v \rangle \star A \star (\forall \vec{y}, B \star Q(w)) \]

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

\[
\begin{align*}
\{ P \} \ e \langle \Psi \rangle \ \{ Q \} & \quad \text{isHandler} \ \langle \Psi \rangle \ \{ Q \} \ (e_{val}, e_{eff}) \langle \Psi' \rangle \ \{ Q' \} \\
\{ P \} \ \text{handle} \ e \ \text{with} \ e_{val}, e_{eff} \ \langle \Psi' \rangle \ \{ Q' \}
\end{align*}
\]

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

\[
\begin{align*}
\{ P \} \ e \left\langle \psi \right\rangle \ { Q } & \quad \text{isHandler} \left\langle \psi \right\rangle \ { Q } \ (e_{\text{val}}, e_{\text{eff}}) \left\langle \psi' \right\rangle \ { Q' } \\
\{ P \} & \ \text{handle} \ e \ \text{with} \ e_{\text{val}}, e_{\text{eff}} \left\langle \psi' \right\rangle \ { Q' }
\end{align*}
\]

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

\[
\{ R \} \ e_{\text{eff}} v k \left\langle \psi' \right\rangle \ { Q' }
\]

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

- \( v' \) satisfies \( A \);
- \( k \) is a function with pre \( B \) and post \( \left\langle \psi' \right\rangle \ { Q' } \).
Effect handling

\[
\begin{align*}
\{ P \} \ e \ \langle \Psi \rangle \ \{ Q \} &\quad \text{isHandler} \ \langle \Psi \rangle \ \{ Q \} \ (e_{\text{val}}, e_{\text{eff}}) \ \langle \Psi' \rangle \ \{ Q' \} \\
\hline
\{ P \} \ \text{handle} \ e \ \text{with} \ e_{\text{val}}, e_{\text{eff}} \ \langle \Psi' \rangle \ \{ Q' \}
\end{align*}
\]

More simply, \( R \) states that \( v \) and \( k \) are any values and continuations that are permitted by the protocol \( \Psi \):

\[
R \overset{\text{def}}{=} \Psi \ \text{allows} \ v \ \{ \lambda w. \ \{ \text{emp} \} \ k \ w \ \langle \Psi' \rangle \ \{ Q' \} \}
\]

As a bonus, the Iris formalization of this theory uses “non-persistent triples”, hence \( \{ \text{emp} \} \ k \ w \ \langle \Psi' \rangle \ \{ Q' \} \) gives the permission to invoke continuation \( k \) only once!
Effect handling

\[
\{ P \} \, e \, \langle \Psi \rangle \, \{ Q \} \quad \text{isHandler} \, \langle \Psi \rangle \, \{ Q \} \, (e_{val}, e_{eff}) \, \langle \Psi' \rangle \, \{ Q' \}
\]

\[
\{ P \} \, \text{handle} \, e \, \text{with} \, e_{val}, e_{eff} \, \langle \Psi' \rangle \, \{ Q' \}
\]

Putting it all together, we have

\[
isHandler \, \langle \Psi \rangle \, \{ Q \} \, (e_{val}, e_{eff}) \, \langle \Psi' \rangle \, \{ Q' \} \quad \overset{\text{def}}{=} \\
(\forall v, \{ Q(v) \} \, e_{val} \, v \, \langle \Psi' \rangle \, \{ Q' \}) \\
\wedge \quad (\forall v, k, \{ \Psi \, \text{allows} \, v \, \{ \lambda w. \, \{ \text{emp} \} \, k \, w \, \langle \Psi' \rangle \, \{ Q' \} \} \})
\]

\[
e_{eff} \, v \, k \\
\langle \Psi' \rangle \, \{ Q' \}
\]

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
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:

THE END