② To introduce the operational semantics of IMP < Small-step / Big-step

③ To define a definitional interpreter

④ To relate these notions
correctness of the interpreter for the semantics

Goal of the lecture

① To introduce the syntax of imp, a toy imperative lang.

phrases of imp
- expressions
  - booleans
  - arithmetic
- commands (statements)
  - if - then - else
  - while - do
  - ⋮

⇓

high-level lang

compilation ← { · compound exp
· control structures }

Def. expression = a syntactic entity whose evaluation either produces a value or does not terminate

command = a syntactic entity whose evaluation may not produce a value, but it can hav a side effect

Arithmetic

We first extend the language of arithmetic expressions

$$(\text{aExp} \ni) \quad a ::= \text{CONST}(n) \mid \underbrace{x}_{\text{variable}} \mid \text{PLUS}(a,a) \mid \text{TIMES}(a,a)$$

To extend the interpreter, we need the notion of a store

Def. A store is a function from variables to natural numbers

$$(\text{Store} \ni) \qquad \sigma : \text{Var} \to \mathbb{N}$$

Rem. Better definition would be

$$(\text{aVal} \ni) \quad v ::= \text{CONST}(n)$$

$$(\text{Store} \ni) \quad \sigma : \text{Var} \to \text{aVal}$$

Def (Interpreter)

$$\text{aeval} : \text{aExp} \to \text{Store} \to \underbrace{\mathbb{N}}$$

$$\text{eval}\ c_n\ \sigma = n$$

$$\text{eval}\ x\ \sigma = \sigma(x)$$

$$\text{eval}\ (\text{PLUS}(a_1, a_2))\ \sigma = (\text{eval}\ a_1\ \sigma) + (\text{eval}\ a_2\ \sigma)$$

$$\vdots$$

Q. What if we add

$a ::= \dots \text{DIV}(a, a)$?

$$\text{aeval} : \text{aExp} \to \text{Store} \to \text{option}\ \mathbb{N}$$

Booleans

$(\mathcal{B}exp \ni) \quad b ::= \mathrm{TRUE} \mid \mathrm{FALSE} \mid \mathrm{EQUAL}\,(\underbrace{a\,,\,a}_{aexp}) \cdots \mid \mathrm{AND}\,(b, b) \mid \mathrm{OR}\,(b, b)$

Def. (Interpreter)

$$\mathrm{beval} : \mathcal{B}exp \to \underbrace{\mathrm{Store}} \to \mathrm{bool}$$

$\hookrightarrow$ why ?

$$\mathrm{beval}\ \mathrm{TRUE}\ \sigma = \mathrm{true}$$

$$\vdots$$

$$\mathrm{beval}\ (\mathrm{EQUAL}\,(a_1, a_2))\ \sigma = (\mathrm{aeval}\ a_1\ \sigma) =? (\mathrm{aeval}\ a_2\ \sigma)$$

Commands

(Com) $c ::=$

$\mid$ skip	(do nothing)
$\mid x := a$	(assignment)
$\mid c \,;\, c$	(sequential comp)
$\mid$ if b then c else c	(conditional)
$\mid$ while b do c	(loop)

```
Inductive com : Type :=
  | SKIP
  | ASSIGN (x : ident) (a : aexp)
  | SEQ (c1 : com) (c2 : com)
  | IFTHENELSE (b : bexp) (c1 : com)
  | WHILE (b : bexp) (c : com).
```

Operational Semantics

Goal: describes how programs are executed (in a machine independent fashion)

Many possibilities

① Implementation / Translation to a (lower) language

⇝ give meaning = write a compiler / machine

syntax ⇝ implemented using data structures

(what happens if we change structure?)

semantics ⇝ low level execution mechanism

- Dominant approach before Strachey-Scott (denotational sem)

- Landin ("The next 700 PLs") Encode PLs into λ-calculus (++), give abstract machines for it (ISWIM, CEK, SECD, CAM, ...)

② Structural semantics: specify execution in a machine-independent way $\underbrace{\text{recursively on syntax}}_{\text{syntax is itself a data structure}}$

structural = rules depend on the "outermost" syntactic shape

<u>Formally</u>. A transition system $(S, \to)$

S: states

$\to$: transitions

states = T (syntax)
- configurations (command, store)
- prog + continuations
- ⋮

transitions
- one-step of execution
- many-steps " "
- ⋮

Rem. Modern accounts use coalgebras

$$\delta : S \longrightarrow F(S)$$

$\hookrightarrow$ extra information on execution
(randomness, IO, effects, ...)

(3) Reduction semantics: The subset of structural semantics where

$$S = \text{Syntax}$$

"Semantics is a relation on syntax" (Felleisen)

Ex.

$$(\lambda x. t)\, s \longrightarrow_\beta t[s/x]$$

$$K\, t\, s \longrightarrow_w t$$

$$\text{let } x = (\ell := v ; t) \text{ in } s \longrightarrow \ell := v ; (\text{let } x = t \text{ in } s)$$

We consider a structural operational semantics (SOS) where

$$\left.\begin{array}{l}\text{states} = \underset{(c,\sigma)}{\mathrm{Com}} \times \mathrm{Store}\end{array}\right\}\ \text{configurations}$$

$$\text{transitions} = \begin{cases} \text{small-step} & (c,\sigma) \to (c',\sigma') \\ \text{big-step (natural)} & (c,\sigma) \Downarrow \sigma' \end{cases}$$

<u>Rem</u>. In the notes, small-step is called reduction (wrong!)

Inductive Definition

$$\underset{\text{command}}{(c}, \underset{\text{current store}}{\sigma)} \xrightarrow{\text{one computation step}} \underset{\text{residual command}}{(c'}, \underset{\text{updated store}}{\sigma')}$$

$$\frac{}{(x := a, \sigma) \to (\text{skip}, \sigma\{x \leftarrow \text{aeval}\ a\ \sigma\})} \ (\text{ASSIGN})$$

update operation

$$\frac{}{(\text{skip}; c_2, \sigma) \to (c_2, \sigma)} \ (\text{SEQ-DONE})$$

$$\frac{(c_1, \sigma) \to (c_1', \sigma')}{(c_1; c_2, \sigma) \to (c_1'; c_2, \sigma')} \ (\text{SEQ-STEP})$$

$$\frac{\text{beval } b\ \sigma = \text{true}}{(\text{while } b \text{ do } c, \sigma) \rightarrow (c, \text{while } b \text{ do } c, \sigma)} \quad \text{(WHILE DONE)}$$

$$\frac{\text{beval } b\ \sigma = \text{false}}{(\text{while } b \text{ do } c, \sigma) \rightarrow (\text{skip}, \sigma)} \quad \text{(WHILE LOOP)}$$

How to implement SOS in Coq?

<u>Def</u>. A relation in Coq is a map with codomain Prop

$$R \subseteq A \times B \quad \rightsquigarrow \quad R : A \to B \to \text{Prop}$$

<u>Intuition</u>. $R \subseteq A \times B \;\cong\; R : A \times B \to \{0, 1\}$

Given $a \in A$, $b \in B$, $R(a,b) = 1$ if a is R-related to b,
$R(a,b) = 0$ otherwise

$\Longrightarrow$ boolean point of view

Coq uses <u>constructive logic</u>

Classical logic : a proposition is something that has a truth-value

→ excluded middle : $A \vee \neg A$ (is true)

Constructive logic : a proposition is something that it can be judged true by means of a proof

→ $p : A$ "p is a proof of A"

→ $p : \neg A$ "p is a refutation of A"

Thus $A \vee \neg A$ holds constructively iff we always have either a proof or a refutation of A

Proposition-as-Types

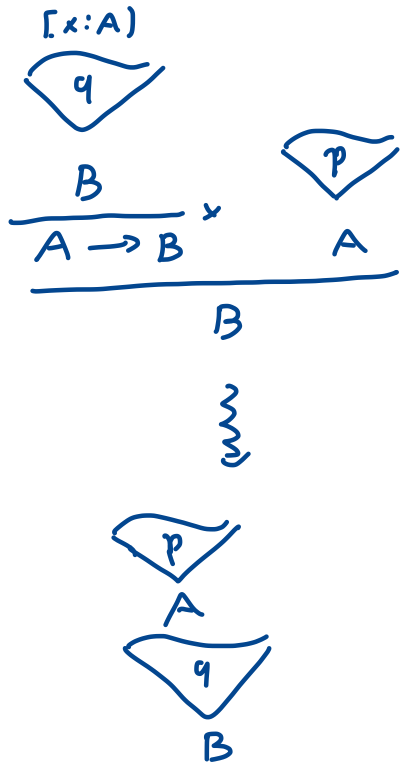
$t : A$ → t is a proof of the proposition A

$t : A$ → t is a program of type A

$$\dfrac{\begin{array}{c}[x:A]\\ \vdots\ p \\ B\end{array}}{A \to B}\ (x) \qquad \qquad \dfrac{x : A \vdash t : B}{\vdash \lambda x.t : A \to B}$$

$$\dfrac{\begin{array}{c}\vdots\ p \\ A \to B\end{array} \qquad \begin{array}{c}\vdots\ q \\ A\end{array}}{B} \qquad \qquad \dfrac{\vdash t : A \to B \qquad \vdash s : A}{\vdash ts : B}$$

Proof normalisation = computation



$$\dfrac{\dfrac{x : A \vdash t : B}{\vdash \lambda x.\, t : A \to B} \qquad \vdash s : A}{\vdash (\lambda x.\, t)\, s : B}$$

$$\rightsquigarrow_{\beta}$$

$$\vdash t[s/x] : B$$

Back to Coq

$R : A \times B \longrightarrow \{0, 1\}$ Classical

$R(a,b)$ is a proposition, hence coincides with its truth-value

$R : A \times B \longrightarrow$ Prop Constructive

$R(a,b)$ is a proposition (type), hence we need a proof (program) that the proposition $R(a,b)$ holds

We define a relation

red : comm * store → comm * store → Prop

What are proof that red (c, σ) (c', σ') holds?

⤳ <u>Our rules</u>!

$$\frac{}{(x := a, \sigma) \to (\text{skip}, \sigma\{x \mapsto \text{aeval } a\ \sigma\})}\ [\text{ASSIGN}]$$

red_assign : ∀ x a σ, red (x := a, σ) (SKIP, σ{x ↦ aeval a σ})

↳ name proof/program

$$\frac{(c_1, \sigma_1) \to (c_2, \sigma_2)}{(c_1; c, \sigma_1) \to (c_2; c, \sigma_2)}\ [\mathrm{SEQ\ STEP}]$$

$$\mathtt{red\_seq\_step} : \forall c_1\ c_2\ \sigma_1\ \sigma_2\ c,\ \underbrace{\mathrm{red}\ (c_1, \sigma_1)\ (c_2, \sigma_2) \to \mathrm{red}\ (\mathrm{SEQ}\ c_1\ c, \sigma_1)\ (\mathrm{SEQ}\ c_2\ c, \sigma_2)}_{\substack{\text{logical statement (proposition)} \\ \text{coq type}}}$$

We also need to say that this is the only way to obtain reductions

```
Inductive red : com * state -> com * state -> Prop :=
  | red_assign : ∀ x a s,
        red (ASSIGN x a, s) (SKIP, update x (aeval s a), s)

  | red_seq_done : ...

  | red_seq_step : ..
     ⋮
  | red_while_loop : ∀ b c s, beval b s = true ->
            red (WHILE b c, s) (c; WHILE b c, s).
```

Comment:

- Each case in the def. of red is a Theorem that allows us to conclude red (c, s) (c', s') for some choices of c, c', s, s'

- The proposition red (c, s) (c', s') holds only if was proved by applying these theorems in a finite number of times

$\Rightarrow$ reasoning on semantics by

- structural induction
- case analysis

Properties of the Semantics

① Determinism

Lemma red-determ : $\forall$ c d_1, red c d_1
$\rightarrow \forall d_2$, red c $d_2 \rightarrow d_1 = d_2$

② Termination

Definition terminates (s : store) (c : com) (s' : store) :=
star red (c, s) (SKIP, s')

Defined in Sequences (library)
star R (usually written as R^*) is the reflexive and transitive closure of R

③ Goes wrong

Given a computation (c, s) we want to say that it does not go wrong: either it reaches a final state (SKIP, s') or diverges

```
Definition goes_wrong (s: state) (c: com) : Prop :=
  ∃ c', ∃ s',
    star red (c, s) (c', s') ∧ ~ ∃ c'', s'', red (c', s') (c'', s'')
      ∧ c' <> SKIP.

Lemma progress : ∀ c s, c = SKIP ∨ ∃ c' s', red (c, s) (c', s').
Theorem not_goes_wrong : ∀ c s, ~ goes_wrong s c.
```

Definitional Interpreter

Goal. Define a function eval that evaluates a command in an initial state, returning the final state

$$\text{eval} : \text{Com} \times \text{Store} \to \text{Store}$$

non-termination

Problem eval (WHILE b c, s)

= if beval b s then eval (c ; WHILE b c, s)
else s

Solution: bounded interpreter

```
Fixpoint eval (fuel : nat) (s : store) (c : com) : option store :=
    match fuel with
      | O => None
      | S fuel' =>
          match c with
            | SKIP => Somme s
            | ASSIGN x a => Some (update x (aeval a s) s)
            :
            | WHILE b c1 =>
                 if beval b s then
                    match eval fuel' s c1 with
                       | None => None
                       | Some s' => eval fuel' s' (WHILE b c1)
                    end
                 else Some s
          end
    end.
```

Correctness

Soundness

Theorem soundness : $\forall$ fuel s c s',
eval fuel s c = Some s' $\rightarrow$ star red (s, c) (SKIP, s')

$(\forall n, s, c, s'.\ \text{eval}\ n\ s\ c = s' \implies (c, s) \rightarrow^{*} (\text{SKIP}, s'))$.

Theorem completeness : $\forall$ s c s', star red (c, s) (SKIP, s')
$\rightarrow \exists$ fuel, eval fuel s c = Some s'.

$(\forall s, c, s'.\ (c, s) \rightarrow^{*} (\text{SKIP}, s') \implies \exists n.\ \text{eval}\ n\ c\ s = s')$