IMP$_3$: `Expression`s with Side-Effects

- many imperative languages allow expression evaluation to produce a side-effect on the store
  - if **`func`** calls can have side-effects, this happens by default
  - many languages have `Expression` syntax allowing `Command` execution as part of an `Expression`
    - such constructions are called "Command Expressions" or "Serial Clauses" (Algol 68)
    - Serial clauses analogues of **`func`** bodies: both produce side-effects *and* **`return`** values
    - **`return`** or **`resultis`** syntax establish return values

$\mathbf{IMP_3}$ = $\mathbf{IMP_{2c}}$ + Serial Clauses

- Features
  - `Expression` evaluation can produce a changed store
  - `Function` execution can alter the store

Syntax of $\mathbf{IMP_3}$

- change only the syntax of `Expression`s

```
Expression::=...
  | begin Command ; return Expression end
```

$\mathbf{IMP_3}$ Semantics

- Specify semantic domains
  - most exactly as before in $\mathbf{IMP_{2c}}$
    Value = *truth-value* Boolean + *integer* Integer
    Argument = *value* Value + *variable* Location
    Procedure = Argument → Store → Store
    Bindable = *value* Value + *variable* Location
    + *function* Function + *procedure* Procedure
  - *new* domain reflects return value and side-effect
    Function = Argument → Store → Value × Store
- Specify Contextual Constraints
  - type of a serial clause is the type of its **`return`** expression
- Specify semantic functions
  - now an `Expression` produces a value and a new store
    evaluate : `Expression` →
    *(*Environ → Store → Value × Store *)*
  - the signatures of *execute* and *elaborate* are unchanged
    execute : `Command` → *(*Environ → Store → Store *)*
    elaborate : `Declaration` →
    *(*Environ → Store → Environ × Store *)*

Expression Semantics

- Serial Clauses

  evaluate⟦**begin** *C*; **return** *E* **end**⟧ *env sto* =
  let *sto′* = *execute*⟦*C*⟧ *env sto* **in**
  evaluate⟦*E*⟧ *env sto′* — a pair *(v, s)*

- Now *all* Expression rules must change to yield Value × Store pairs even when there is no store effect

  evaluate⟦*N*⟧ *env sto* =
  (integer valuation N, sto)

  evaluate⟦**true**⟧ *env sto* =
  (truth−value **true**, *sto)* — similarly for **false**

  evaluate⟦*I*⟧ *env sto* =
  (coerce(sto, find(env, I)), sto)

  evaluate⟦E_1 + E_2⟧ *env sto* =
  let *(integer* v_1, s_1*)* = *evaluate*⟦E_1⟧ *env sto* **in**
  let *(integer* v_2, s_2*)* = *evaluate*⟦E_2⟧ *env* s_1 **in**
  (integer(v_1+v_2*)*, s_2 *)* — order is left to right

  . . .

  evaluate⟦**not** *E*⟧ *env sto* =
  let *(truth−value b, s)* = *evaluate*⟦*E*⟧ *env sto* **in**
  (truth−value(¬ *b)*, *s)*

`Expression` Semantics (cont'd)

- In function calls, store can change during argument evaluation:

$give\text{-}argument$: `Actual-Parameter` $\rightarrow$
($\mathsf{Environ} \rightarrow \mathsf{Store} \rightarrow \mathsf{Argument} \times \mathsf{Store}$)

$give\text{-}argument[\![E]\!]\ env\ sto\ =$
let $(\ val,\ sto') = evaluate[\![E]\!]\ env\ sto$ **in**
$(value\ val,\ sto')$

$give\text{-}argument[\![$ **`var`** $I]\!]\ env\ sto\ =$
let $variable\ loc = find(env,\ I)$ **in**
$(variable\ loc,\ sto)$

$evaluate[\![I\ (\ AP\ )\ ]\!]\ env\ sto\ =$
let $function\ func = find(env,\ I)$ **in**
let $(arg,\ sto') = give\text{-}argument[\![AP]\!]\ env\ sto$ **in**
$func\ arg\ sto'$

Command Semantics

- Effects of *evaluate* change—it returns a pair in Value × Store —propagate to everywhere that *evaluate* is used in other semantic functions
- Note the sto', σ' arguments in the following

execute⟦*I* **:=** *E*⟧ *env sto* =
  let *(val, sto′) = evaluate*⟦*E*⟧ *env sto* **in**
  let *variable loc = find(env, I)* **in**
  update(sto′, loc, val)

execute⟦**if** *E* **then** C_1 **else** C_2⟧ *env sto* =
  let *(val, sto′) = evaluate*⟦*E*⟧ *env sto* **in**
    if *val = truth−value* **true**
     then *execute*⟦C_1⟧ *env sto′*
     else *execute*⟦C_2⟧ *env sto′*

execute⟦**while** *E* **do** *C*⟧ =
  let $f\ \rho\ \sigma$ =
    let $(v, \sigma') = evaluate⟦E⟧\ \rho\ \sigma$ **in**
      if *v = truth−value* **false**
       then σ'
       else $f\ \rho$ *(execute*⟦*C*⟧ $\rho\ \sigma'$*)*
   in
  f

Command Semantics (cont'd)

$$execute[\![I\ (\ AP\ )\ ]\!]\ env\ sto\ =$$

let $procedure\ proc = find(env, I)$ **in**
let $(arg, sto') = give-argument[\![AP]\!]\ env\ sto$ **in**
$proc\ arg\ sto'$

`Declaration` Semantics

- Only **`const`** declarations involve *evaluate*; **`var`** declaration does not initialize
- Both kinds of declarations are now "store-altering", but for distinct reasons

elaborate⟦**`const`** *I* ~ *E*⟧ *env sto* =
    let *(val, sto′) = evaluate*⟦*E*⟧ *env sto* **in**
    (bind(I, value val), sto′)) — new store

elaborate⟦**`var`** *I* **:** *T*⟧ *env sto* =
    let *(sto′, l) = allocate sto* **in**
    (bind(I, variable l), sto′)) — as before

Example: A Program with a Serial Clause

- Find the result of executing the program `C` starting with environment *u* and store *s*: Suppose that `y` has been allocated location *u*(`y`) and `x` has been allocated location *u*(`x`)

  `C = y := 7 ; x := begin y := y + 1 ; return y end`

  $execute[\![\texttt{C}]\!]\ u\ s$
  $= execute[\![\texttt{x := begin y := y + 1 ; return y end}]\!]\ u\ (s[u(\texttt{y}) \mapsto 7])$

  Denote the right side of this assignment by *E*. Then

  $evaluate[\![E]\!]\ u\ (s[u(\texttt{y}) \mapsto 7])$
  $= evaluate[\![\texttt{y}]\!]\ u\ (s[u(\texttt{y}) \mapsto 7, u(\texttt{y}) \mapsto evaluate[\![\texttt{y + 1}]\!]\ u\ s[u(\texttt{y}) \mapsto 7]])$
  $= evaluate[\![\texttt{y}]\!]\ u\ (s[u(\texttt{y}) \mapsto 7, u(\texttt{y}) \mapsto 8])$
  $= (8, s[u(\texttt{y}) \mapsto 8])$

  So:
  $= execute[\![\texttt{x := } E]\!]\ u\ (s[u(\texttt{y}) \mapsto 7])$
  $= update(s[u(\texttt{y}) \mapsto 8], u(\texttt{x}), 8)$

  Therefore:
  $execute[\![\texttt{C}]\!]\ u\ s$
  $= s[u(\texttt{y}) \mapsto 8, u(\texttt{x}) \mapsto 8]$

- very different from the following seemingly analogous program:
  `C′ = y := 7 ; let const y ~ y + 1 in x := y`

  $execute[\![\texttt{C}']\!]\ u\ s$
  $= s[u(\texttt{y}) \mapsto 7, u(\texttt{x}) \mapsto 8]$