An Imperative Language

Abstract Grammar - IMP

```
      Program::=Command
      Command::=skip
               | Identifier := Expression
               | let Declaration in Command
               | Command ; Command
               | if Expression then Command
                        else Command
               | while Expression do Command

   Expression::=Numeral
               | false | true
               | Identifier
               | Expression + Expression
               | Expression < Expression
               | not Expression
               | ...

  Declaration::=const Identifier ~ Expression
               | var Identifier : Type-denoter

 Type-denoter::=bool | int
```

- a simple imperative language IMP (Watt)

IMP Semantics

- Specify semantic domains
  - first-class values that are stored, passed, etc.
    Value = *truth-value* Boolean + *integer* Integer
  - storables are all first-class objects
    Storable = Value
  - **`var`** names are bound to refs; **`const`** to values
    Bindable = *value* Value + *variable* Location
  - Bindable also called Denotable
  - Location also called Reference or Ref
- Specify semantic functions
  - an expression, given an env and sto, yields a value
    evaluate : `Expression` → (Environ → Store → Value)
  - a command, given an env and sto, yields a new sto
    execute : `Command` → (Environ → Store → Store)
  - a declaration, given an env and sto, yields a new env, and may alter the store by allocating more locations
    elaborate : `Declaration` →
    (Environ → Store → Environ × Store)

IMP `Expression` Semantics

- *evaluate :*
  `Expression` $\rightarrow$ *(*Environ $\rightarrow$ Store $\rightarrow$ Value *)*
  - — expressions are *evaluated* in an environment *and* store
  - — produce an *expressible value*
  - — no expression side-effects in IMP$_0$
  - — syntactic metavariables are N *:* `Numeral`, E *:* `Expression`, I *:* `Identifier`

Constants

$$evaluate[\![N]\!]\ env\ sto\ =\ integer(valuation\ N)$$

$$evaluate[\![\texttt{true}\ ]\!]\ env\ sto\ =\ truth\text{-}value(\mathbf{true})$$

$$evaluate[\![\texttt{false}\ ]\!]\ env\ sto\ =\ truth\text{-}value(\mathbf{false})$$

IMP `Expression` Semantics (cont'd)

Names

$evaluate[\![I]\!]\ env\ sto\ =$
$\quad coerce(sto,\ find(env, I)\ )$

— auxiliary function:
$coerce : \text{Store} \times \text{Bindable} \rightarrow \text{Value}$

$coerce(sto, value\ val) = val$
$coerce(sto, variable\ loc) = fetch(sto, loc)$

— *coerce* does "implied dereferencing" on assignment RHSs

Operators

$evaluate[\![E_1 + E_2]\!]\ env\ sto\ =$
$\quad \textbf{let} integer\ i_1 = evaluate\ [\![E_1]\!]\ env\ sto \textbf{in}$
$\quad \textbf{let} integer\ i_2 = evaluate\ [\![E_2]\!]\ env\ sto \textbf{in}$
$\quad integer\ (i_1 + i_2)$

$evaluate[\![E_1 < E_2]\!]\ env\ sto\ =$
$\quad \textbf{let} integer\ i_1 = evaluate\ [\![E_1]\!]\ env\ sto \textbf{in}$
$\quad \textbf{let} integer\ i_2 = evaluate\ [\![E_2]\!]\ env\ sto \textbf{in}$
$\quad truth\text{-}value\ (i_1 < i_2)$

$evaluate[\![\texttt{not}\ E]\!]\ env\ sto\ =$
$\quad \textbf{let} truth\text{-}value\ b = evaluate\ [\![E]\!]\ env\ sto \textbf{in}$
$\quad truth\text{-}value\ (\neg\ b)$

IMP `Command` Semantics

- *execute :*
  `Command` $\rightarrow$ *(*Environ $\rightarrow$ Store $\rightarrow$ Store *)*
  - — commands are *executed* in an environment *and* store
  - — produce a new *store*
  - — commands, given an env, are store transformers, and produce pure side-effect
  - — syntactic metavariables are *C :* `Command` , *D :* `Declaration` , *I* and *E*.

Non-iterative

execute⟦**`skip`** ⟧ *env sto* =
sto

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

execute⟦**`let`** *D* **`in`** *C*⟧ *env sto* =
let *(env′, sto′)* = *elaborate*⟦*D*⟧ *env sto* **in**
execute⟦*C*⟧ *(overlay(env′, env))* *sto′*

IMP `Command` Semantics (cont'd)

$$execute[\![C_1 \texttt{;}\ C_2]\!]\ env\ sto\ =$$
$$\textbf{let}\ sto' = execute[\![C_1]\!]\ env\ sto\ \textbf{in}$$
$$execute[\![C_2]\!]\ env\ sto'$$

$$execute[\![\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\ env\ sto\ =$$
$$\textbf{if}\ evaluate[\![E]\!]\ env\ sto = truth\text{-}value\ \textbf{true}$$
$$\textbf{then}\ execute[\![C_1]\!]\ env\ sto$$
$$\textbf{else}\ execute[\![C_2]\!]\ env\ sto$$

Iterative: meaning of a **`while`** in an env is a recursively defined mapping from sto to sto

$$execute[\![\texttt{while}\ E\ \texttt{do}\ C]\!]\ env\ sto\ =$$
$$\textbf{if}\ evaluate[\![E]\!]\ env\ sto = truth\text{-}value\ \textbf{false}$$
$$\textbf{then}\ sto$$
$$\textbf{else}\ execute[\![\texttt{while}\ E\ \texttt{do}\ C]\!]\ env\ (execute[\![C]\!]\ env\ sto)$$

alternatively:

$$execute[\![\texttt{while}\ E\ \texttt{do}\ C]\!] =$$
$$\textbf{let}\ f\ u\ s\ =$$
$$\textbf{if}\ evaluate[\![E]\!]\ u\ s = truth\text{-}value\ \textbf{false}$$
$$\textbf{then}\ s$$
$$\textbf{else}\ f\ u\ (execute[\![C]\!]\ u\ s)$$
$$\textbf{in}$$
$$f$$

Note f : Environ $\rightarrow$ Store $\rightarrow$ Store

IMP `Declaration` Semantics

elaborate : `Declaration` $\rightarrow$
*(*Environ $\rightarrow$ Store $\rightarrow$ Environ $\times$ Store *)*

- declarations are *elaborated* in an environment *and* store
- declarations produce a new elementary *environment (name, denotation)*
- *could* alter store if location *allocated*
- syntactic metavariables are *T :* `Type-denoter`, *I* and *E*.

Store-invariant elaboration

elaborate⟦**const** *I* ~ *E*⟧ *env sto* =
let *v* = *evaluate*⟦*E*⟧ *env sto* **in**
(bind(I, value v), sto)
— result is a pair

Store-altering elaboration

elaborate⟦**var** *I* : *T*⟧ *env sto* =
let *(sto′, l)* = *allocate sto* **in**
(bind(I, variable l), sto′) — result is a pair

Example

- Assume *any-unused-location* returns next integer and increments it.

code	*u* : Environ	*s* : Store
	$u = \lambda I.unbound = \Omega$	$s = \lambda l.unused$
let var *x* **: int**		
in (	$\Omega[\mathbf{x} \mapsto 0]$	$s[0 \mapsto undefined]$
x := 12		
	$\Omega[\mathbf{x} \mapsto 0]$	$s[0 \mapsto 12]$
) ;		
	Ω	$s[0 \mapsto 12]$
let var y : int		
in (	$\Omega[\mathbf{y} \mapsto 1]$	$s[0 \mapsto 12, 1 \mapsto undefined]$
y := 21**;**		
	$\Omega[\mathbf{y} \mapsto 1]$	$s[0 \mapsto 12, 1 \mapsto 21]$
y := y + 1		
	$\Omega[\mathbf{y} \mapsto 1]$	$s[0 \mapsto 12, 1 \mapsto 22]$
)		
	Ω	$s[0 \mapsto 12, 1 \mapsto 22]$

- locations never re-used in this interpretation of ‘‘*any-unused-location*"
- environment ‘‘pops back" after block exit
- $execute[\![C_1 \ ; \ C_2]\!] \ e \ s$
  $= execute[\![C_2]\!] \ e \ (execute[\![C_1]\!] \ e \ s)$
  — Same *e* to both commands, but altered *s* to second command

`:=` versus **`let`**

s : Store *e :* Environ

execute⟦*I* `:=` *E* `;` *C*⟧ *e s* =
execute⟦*C*⟧ *e* (*s*[*find*(*e*, *I*) ↦ *evaluate*⟦*E*⟧ *e s*])

execute⟦**`let const`** *I* `~` *E* **`in`** *C*⟧ *e s* =
execute⟦*C*⟧ (*e*[*I* ↦ *evaluate*⟦*E*⟧ *e s*]) *s*

- assignment (`:=`) alters the *store*, but not the environment
- **`let`** alters the *environment*, but not the store
  - — (unless allocation occurs because of a **`var`** declaration)

Persistence of Store

- altered store state after assignment persists to next command C_2

$$
\begin{aligned}
&\mathit{execute}[\![\,\texttt{(}\ I\ \texttt{:=}\ E\ \texttt{;}\ C_1\ \texttt{)}\ \texttt{;}\ C_2\,]\!]\ e\ s\ = \\
&\qquad \mathit{execute}[\![C_2]\!]\ e\ (\mathit{execute}[\![C_1]\!] \\
&\qquad\qquad e \\
&\qquad\qquad (s[\mathit{find}(e, I) \mapsto \mathit{evaluate}[\![E]\!]\ e\ s]) \\
&\qquad)
\end{aligned}
$$

- altered environment in **let** affects only the block body C_1 and not C_2

$$
\begin{aligned}
&\mathit{execute}[\![\,\texttt{let const}\ I\ \texttt{~}\ E\ \texttt{in}\ C_1\ \texttt{;}\ C_2\,]\!]\ e\ s\ = \\
&\qquad \mathit{execute}[\![C_2]\!]\ e\ (\mathit{execute}[\![C_1]\!] \\
&\qquad\qquad (e[I \mapsto \mathit{evaluate}[\![E]\!]\ e\ s]) \\
&\qquad\qquad s \\
&\qquad)
\end{aligned}
$$