Copy Parameters

- all require *local* allocation of refs
- copy-in: copy from the *r*-value of caller's variable/const
- copy-out: copy to the *l*-value of caller's variable
- copy-in/copy-out: same variable used for both

$$\mathbf{IMP_{2c}} = \mathbf{IMP_1} + \text{Copy Parameters}$$

- Features
  - argument can be first-class value or reference (Location , *l*-value).
  - latter are often called **`var`** parameters
  - formals are declared **`value`** (ADA **`in`**) or **`result`** (ADA **`out`**)
  - *elaborate* allocates locations for parms and so local *sto* is altered at call time
  - still no *l*-valued expressions, but *actual* parm names can be interpreted as *l*-values by *give – argument*

Syntax of IMP$_{2c}$

```
Formal-Parameter::=
          value Identifier : Type-denoter
        | result Identifier : Type-denoter

Actual-Parameter::=Expression
        | var Identifier
```

IMP$_{2c}$ Semantics

- Specify semantic domains
  - exactly as before in **IMP$_{2d}$**
    Value = $\cdots$
    Argument = *value* Value + *variable* Location
    Function = $\cdots$
    Procedure = $\cdots$
    Bindable = *value* Value + *variable* Location
    + *function* Function + *procedure* Procedure
- Specify Contextual Constraints
  - **proc** *p* **(result** *x* **:** $\cdots$ **)** declaration $\Rightarrow$ *p* **(var** *I* **)** call (*I* a simple variable name only)
  - **proc** *p* **(value** *v* **:** $\cdots$ **)** declaration $\Rightarrow$ *p* **(** *E* **)** call
- Specify semantic functions — as before with edits:

Semantic Function for `Actual-Parameter`

- semantics for **`func`**/**`proc`** calls exactly as before. Reprise:

give – argument : `Actual-Parameter` $\rightarrow$
(Environ $\rightarrow$ Store $\rightarrow$ Argument)

give – argument ⟦**`var`** *I*⟧ *env sto* =
let *variable loc = find(env, I)* **in**
variable loc

give – argument ⟦*E*⟧ *env sto* =
value(evaluate ⟦*E*⟧ *env sto)*

`Expression` Semantics

evaluate ⟦*I* **(** *AP* **)** ⟧ *env sto* =
let *function func = find(env, I)* **in**
let *arg = give – argument* ⟦*AP*⟧ *env sto* **in**
func arg sto

`Command` Semantics

execute ⟦*I* **(** *AP* **)** ⟧ *env sto* =
let *procedure proc = find(env, I)* **in**
let *arg = give – argument* ⟦*AP*⟧ *env sto* **in**
proc arg sto

Semantic Function for `Formal-Parameter`

- What does a copy *FP* mean?
- Semantics of *FP* depends on whether it is **`value`** or **`result`**

$$copy\text{-}in : \texttt{Formal-Parameter} \rightarrow (\text{Argument} \rightarrow \text{Store} \rightarrow \text{Environ} \times \text{Store})$$

$$copy\text{-}in[\![\textbf{value } I : T]\!]\ (value\ v)\ sto = \textbf{let } (sto', local) = allocate\ sto \textbf{ in } (\ bind(I, variable\ local),\ update(sto', local, v)\ )$$

$$copy\text{-}in[\![\textbf{result } I : T]\!]\ (variable\ l)\ sto = \textbf{let } (sto', local) = allocate\ sto \textbf{ in } (\ bind(I, variable\ local),\ sto'\ )$$

— note that they are identical except that **`value`** parms cause newly allocated local to be initialized

— thus it is possible to declare parms both **`value`** and **`result`** (use $copy\text{-}in$ for **`value`**)

`Declaration` Semantics

- Declaration semantics must handle binding actions on call and return
- **`value`** parameters have no affect on store at return
- **`result`** parameters cause argument *l*-value to be updated by formal parameter's *r*-value at return
- Auxiliary function for return semantics

  $copy-out$: `Formal-Parameter` $\rightarrow$
  (Environ $\rightarrow$ Argument $\rightarrow$ Store $\rightarrow$ Store)

  $copy-out[\![$ **`value`** I : $T]\!]$ env $(value\ v)$ sto =
  sto

  $copy-out[\![$ **`result`** I : $T]\!]$ env $(variable\ l)$ sto =
  let $variable\ local$ = $find(env, I)$ **in**
  $update(sto, l, fetch(sto, local))$

  — *env* used is the parameter binding at time of declaration (see *env* use in *elaborate* below)

  — *sto* used is that at time of return (see σ'' in *elaborate* below)

`Declaration` Semantics (cont'd)

- *procedure* declaration changed to perform copy-in and copy-out at call and return time

$$
\begin{array}{l}
\mathit{elaborate}[\![\texttt{proc}\ I(FP) \sim C]\!]\ \mathit{env}\ \mathit{sto} = \\
\quad \textbf{let}\ \mathit{proc} = \lambda x\,.\,\lambda\sigma\,. \\
\qquad \textbf{let}\ (\mathit{parmbinding}, \sigma') = \mathit{copy\text{-}in}[\![FP]\!]\ x\ \sigma\ \textbf{in} \\
\qquad \textbf{let}\ \sigma'' = \mathit{execute}[\![C]\!]\ \mathit{overlay}(\mathit{parmbinding}, \mathit{env})\ \sigma'\ \textbf{in} \\
\qquad \mathit{copy\text{-}out}[\![FP]\!]\ \mathit{parmbinding}\ x\ \sigma'' \\
\quad \textbf{in} \\
\quad (\ \mathit{bind}(I, \mathit{procedure\ proc}),\ \mathit{sto}\ )
\end{array}
$$

  - σ is the call-time store
  - at time of entry, *parmbinding* is produced and σ' is produced by local allocation
  - execution of body is in declaration *env* overlain by *parmbinding*
  - execution of body alters store to σ''
  - at time of return, $\mathit{copy\text{-}out}$ uses σ'' and makes further storage changes

- **Exercise:** Supply the semantics for $\mathit{elaborate}[\![\texttt{func}\ I(FP) \sim E]\!]\ \mathit{env}\ \mathit{sto} = \cdots$