Principles of Programming Languages

Lecture 08

Control Semantics & Continuations

Semantics of Control Flow

- *Sequencers:* commands that cause control transfers:
  - **`goto`**
  - **`return`**
  - **`exit`**
  - **`break`**
  - **`continue`**
  - **`resultis`** (break from serial clause with a value)
  - "jumps"
- We shall see that
  - Prior semantic ideas inadquate to explain denotation of jumps & labels
  - Repaired by introduction of *continuations* [Strachey, C. and C.P. Wadsworth, *PRG-11,* Oxford Programming Research Group, 1974]

Continuations

- Provide semantic description for jumps and very complex features, e.g., co-routines, exception handling, co-expressions in Icon, etc.)
- Command semantics—*execute* ⟦ ⟧ *env sto*—takes a new *continuation* argument: *execute* ⟦ ⟧ *env cont sto* and yields a *sto*
- New signature of *execute:*
  - `Command` → Environ → Cont → Store → Store

  will propagate changes to other semantic functions
- When there are expression side-effects, even *evaluate* will be considerably changed
  - More on *expression continuations* later

Two Kinds of Semantics

1) *Direct Semantics.* This is what we have been doing: a command, given an environment, directly denotes a Store → Store transformation. Example:
$execute\ [\![\ \mathbf{skip}\ ]\!]\ env = \lambda s.s \in$ [Store → Store]
Drawback: no way a construct can avoid passing its result to the part of the program following it—no way to skip part of the execution. Example:
$execute\ [\![\ C_1\ \mathbf{;}\ C_2\ ]\!]\ env\ sto =$
$execute\ [\![\ C_2\ ]\!]\ env\ (\ execute\ [\![C_1\ ]\!]\ env\ sto\ )$
If C_1 produces error or jump, then the following part of the program must cope (tests for abnormal values, transfers around normal code, etc.); this leads to a very unnatural semantics
2) *Continuation Semantics.* Denotations depend upon a new parameter *cont* : Cont

Two Kinds of Semantics (cont.)

- *cont* represents the ``rest of the program assuming normal flow of control"
- Each construct "decides" where to deliver its resulting altered store
  - Deliver it to the code following the construct—"normal continuation" *cont*
  - Ignore the normal continuation and deliver the store to some other continuation, e.g., one following a label.

Need for Continuations

- Recall the (direct) semantics for a compound command
- *execute* : `Command` $\rightarrow$ Environ $\rightarrow$ (Store $\rightarrow$ Store)
- $execute \llbracket C_1 \texttt{;} C_2 \rrbracket\ env\ sto$
  $= execute \llbracket C_2 \rrbracket\ env\ (execute \llbracket C_1 \rrbracket\ env\ sto)$
- Suppose add new syntax
  - `Command ::= goto Label ... | Label: Command`
- ✖ this won't work

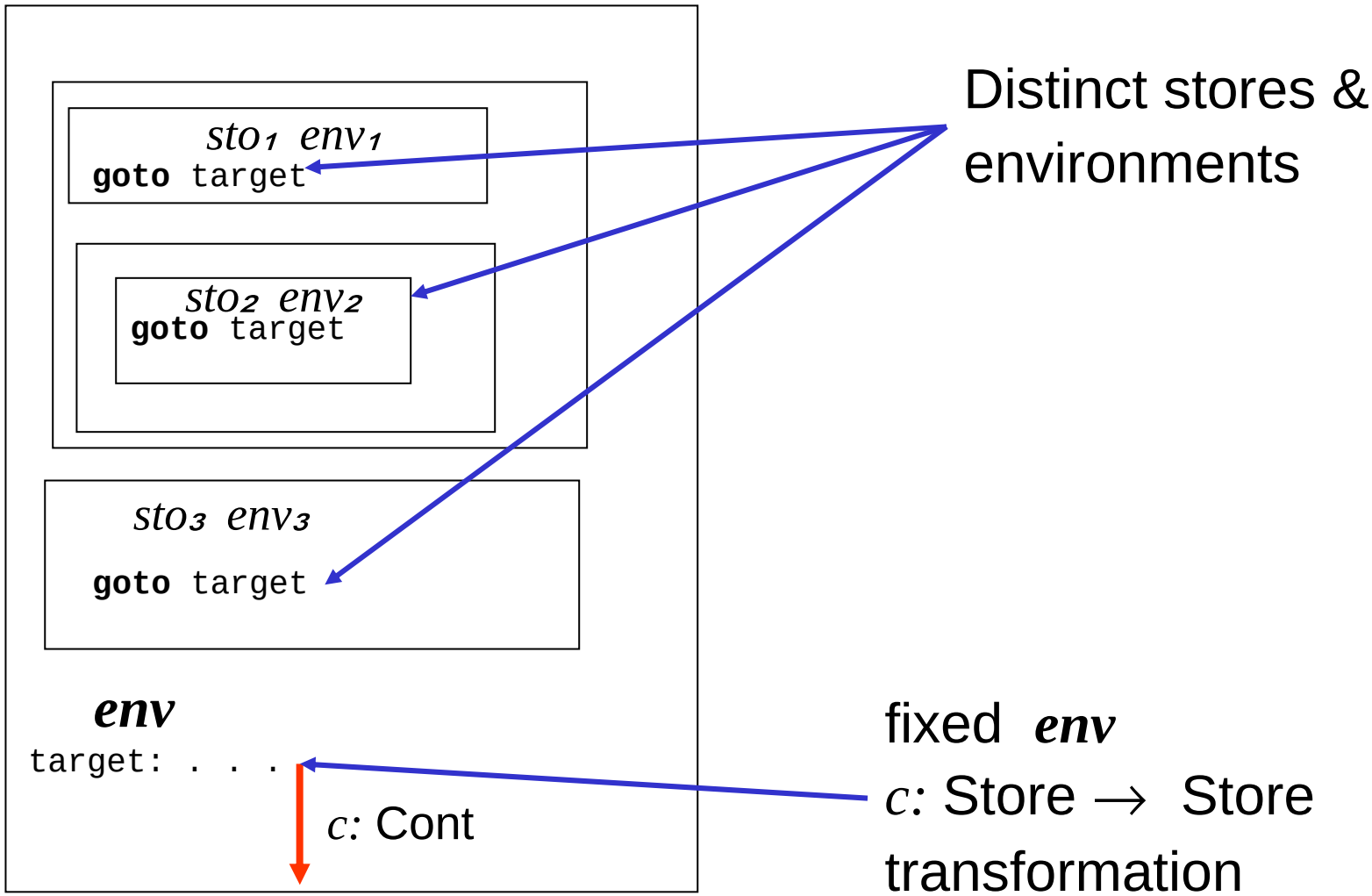
  $execute \llbracket \texttt{goto}\ L\ \texttt{;}\ C_2 \rrbracket\ env\ sto =$
  $execute \llbracket C_2 \rrbracket\ env\ (execute \llbracket \texttt{goto}\ L \rrbracket\ env\ sto)$
  - Result depends on C_2 !
  - Jumps don't do this--C_2 is avoided
  - ∴ $execute \llbracket C_2 \rrbracket$ must be discarded in equation ✖ above

What are Continuations? Labels?

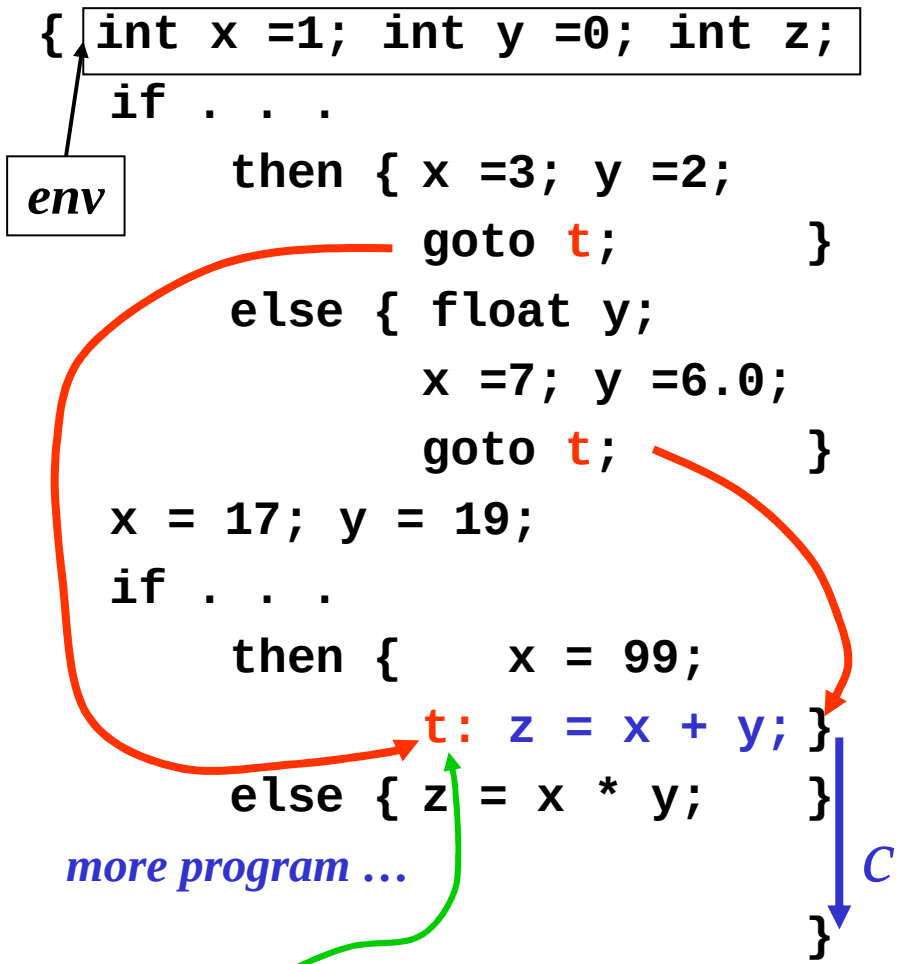
- A *continuation* is an element of domain [Store → Store] representing a transformation of the memory
  - Just like the result of a `Command` given an *env*
- Problems with semantics of *execute* ⟦ C_1 ; C_2 ⟧
  1) How can *execute* ⟦ C_2 ⟧ —the future—be thrown out in case C_1 contains a jump? Somehow C_1 must "decide"
  2) What exactly does a label L : `Label` denote? (It must be the name of *something*!)
- Solutions
  1) Make *execute* ⟦ C_2 ⟧ a parameter to *execute* ⟦C_1 ⟧ so that *execute* ⟦C_1 ⟧ can keep it (if the "normal continuation" is taken) or ignore it (if C_1 contains a jump). (Note that "direct semantics" does just the opposite!)
  2) A label (in implementation) is a point in the program where computation will continue if that label is jumped to—a "continuation computation". A computation is an element of [Store → Store]. So a *label* denotes a *continuation*

Continuations & Labels (cont.)

- Example:



Continuations & Labels (cont.)



- "come from" many places, each with distinct *sto* : **Store**

- **goto t** like a "call" to a procedure with 0 parms:
  Proc⁰ = [Store → Store]
  (that will never return …)
- ∴ **t** denotes a continuation
  c: Store → Store
- **t** is bound to its denotation in the environment: $env' = env\,[t \mapsto c\,]$
- $env'(t) = c$

Label Use (**`goto`**)

- Bindable = *value* Value + *variable* Location
  + *contin* Cont
- Cont = [Store → Store]
- *execute* : `Command`→Environ →Cont →Store → Store
- I.e., *execute* : `Command` → Environ → Cont → Cont
- `Command ::= . . . | Command ; Command`
- *execute* ⟦ C_1 ; C_2 ⟧ *env cont* =

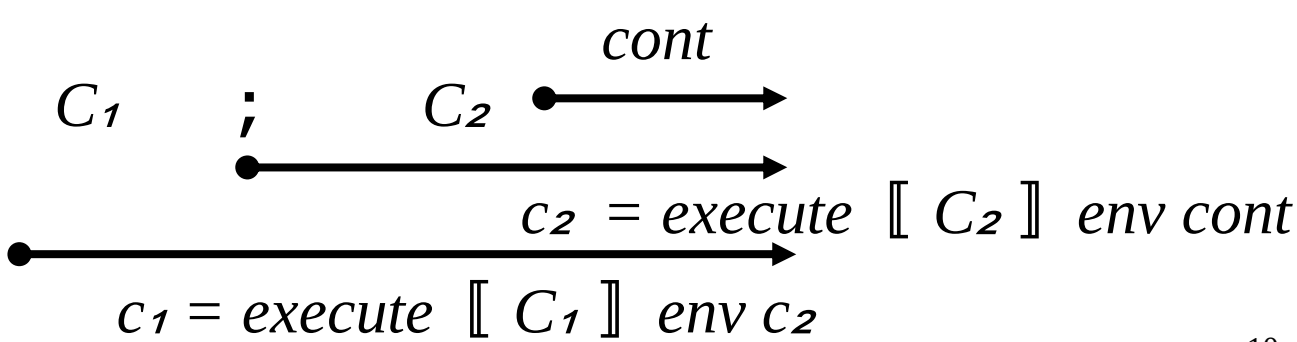
  let c_2 = *execute* ⟦ C_2 ⟧ *env cont* **in**

  let c_1 = *execute* ⟦ C_1 ⟧ *env* c_2 **in**

  c_1

execute is a continuation transformer

- Diagram



Label Use (cont.)

- **`goto`** —a broken promise
- *execute* ⟦ **`goto`** *L* ⟧ *env cont sto* =

  let *contin c* = *find*(*env*, *L*) **in**

  c sto

normal continuation discarded

look up new continuation bound to *L* & apply to current store

- Example:

execute ⟦ **`goto`** `t; y = 2; t: x=3` ⟧ *env cont sto* =
execute ⟦ **`goto`** `t` ⟧ *env*
{ *execute* ⟦ `y=2` ⟧ *env*
{ *execute* ⟦ `t: x=3` ⟧ *env cont* }
} *sto* =
find(*env*, `t`) *sto* = *execute* ⟦ `x=3` ⟧ *env cont sto* =
update(*sto*, *env* `x`, 3)

  - Above assumes the binding of label `t` to the continuation
    c = *execute* ⟦ `x=3` ⟧ *env cont*
  - How does this binding [`t` ↦ *c*] get declared?

IMPg=IMP$_o$+**goto**+labels

- Command ::= **skip**
  | Identifier **:=** Expression
  | **let** Declaration **in** Command
  | Command ; Command
  | **if** Expression **then** Command
  else Command
  | **while** Expression **do** Command
  | Label : Command
  | **goto** Label
- Expression ::= . . .
- Declaration ::= **const** Identifier **~** Expression
  | **var** Identifier : Type-denoter
- Type-denoter ::= **bool** | **int**

`Goto+`Label Declarations: IMPg

- Bindable = *value* Value + *variable* Location + *contin* Cont
- Cont = [Store → Store]
- Changes to `Command` semantics to handle arguments *env cont sto*—add continuations throughout
- No `Expression` side-effects ⇒ no expression continuations ⇒ no changes to `Expression` semantics

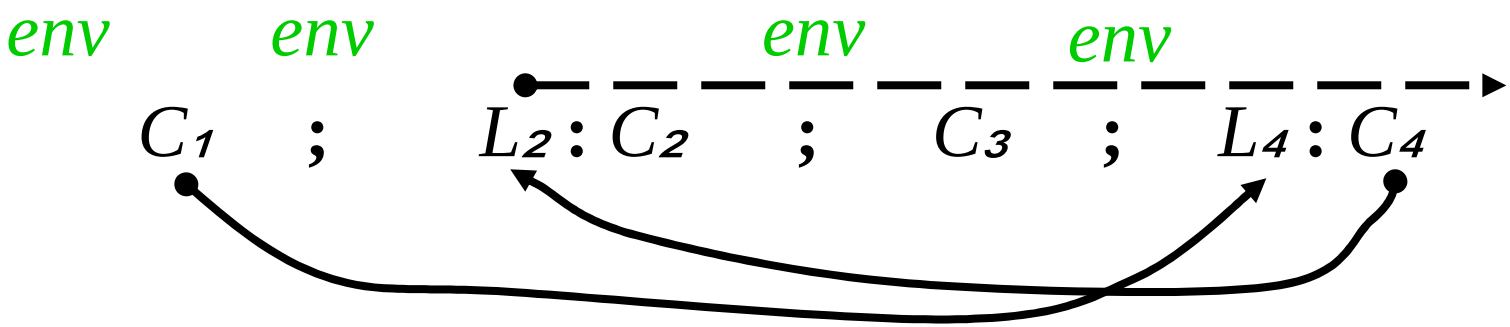
  evaluate : `Expression`→ Environ → Store → Value

- No new `Declaration;` labels declared by attachment to `Commands` ⇒ handled in *execute*
  - With help from auxiliary function *bind-labels*

  elaborate : `Declaration`→Environ →Store →
  Environ × Store

Goto+Label Declarations: IMPg

- Features of the jump semantics
  - No jumps into **`let`** blocks from outside (labels in **`let`** blocks are local; not exported)
  - Jumps to surrounding **`let`** blocks allowed (visibility outward)
  - No jumps into **`while`** loops (labels in **`while`** bodies are local; not exported)
  - Jumps into and across **`then/else`** arms allowed
  - Jumps forward and backward along sequential chains allowed:



Common *env* contains all label bindings at same "level"

Label Declarations (cont.)

bind-labels: `Command` → Env → Cont → Env

bind-labels ⟦ **goto** *L* ⟧ *env cont* = *env*

bind-labels ⟦ **skip** ⟧ *env cont* = *env*

bind-labels ⟦ *I* **:=** *E* ⟧ *env cont* = *env*

bind-labels ⟦ **while** *E* **do** *C* ⟧ *env cont* = *env*

bind-labels ⟦ **let** *D* **in** *C* ⟧ *env cont* = *env*

No label bindings inside *C* are visible from outside

bind-labels ⟦ **if** *E* **then** C_1 **else** C_2 ⟧ *env cont* =

let e_1 = *bind-labels* ⟦ C_1 ⟧ *env′cont* **in**

let e_2 = *bind-labels* ⟦ C_2 ⟧ *env′cont* **in**

let *env′* = *overlay*(e_2, *overlay*(e_1, *env*)) **in**

env′

New *env′* makes label bindings available to <u>both</u> Commands—for both fwd and cross jumps—and visible outside

Label Declarations (cont.)

- Note definition of *env′* is recursive:

  e_1 refers to *env′* and e_2 refers to *env′*

  env′ refers to e_1 , e_2
- Easier to see it making substitutions:

  let *env′* = *overlay*(*bind-labels* ⟦ C_2 ⟧ *env′cont* ,
  overlay(*bind-labels* ⟦C_1 ⟧ *env′cont* , *env*))
- Next two rules also have recursive definition of environment

Label Declarations (cont.)

bind-labels ⟦ C_1 ; C_2 ⟧ *env cont* =

let e_2 = *bind-labels* ⟦ C_2 ⟧ *env′ cont* **in**

let c_2 = *execute* ⟦ C_2 ⟧ *env′ cont* **in**

let e_1 = *bind-labels* ⟦ C_1 ⟧ *env′* c_2 **in**

let *env′* = *overlay*(e_1 , *overlay*(e_2 , *env*)) **in**

env′

bind-labels ⟦ *L* : *C* ⟧ *env cont* =

let *c* = *execute* ⟦ *C* ⟧ *env′ cont* **in**

let *e* = *bind*(*L*, *contin c*) **in**

let *env′* = *overlay*(*e* , *env*) **in**

env′

New *env′* makes label bindings available to both Commands—for both fwd and back jumps—and visible outside

Commands with Continuations

execute: Command → Env → Cont → Sto → Sto

execute ⟦ **goto** *L* ⟧ *env cont* =

let *contin c* = *find*(*env*, *L*) **in**

c

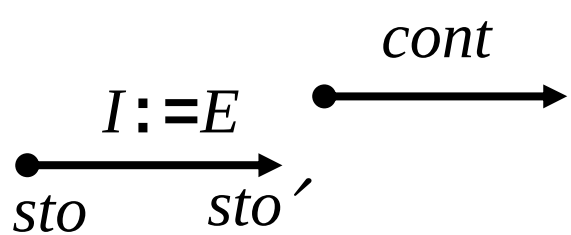
execute ⟦ **skip** ⟧ *env cont* = *cont*

execute ⟦*I* **:=** *E*⟧ *env cont sto* =

let *val* = *evaluate* ⟦*E*⟧ *env sto* **in**

let *variable loc* = find(*env*, *I*) **in**

cont update(sto, loc, val)



Commands with Continuations

execute ⟦ **while** *E* **do** *C* ⟧ *env cont sto* =

if *evaluate* ⟦*E*⟧ *env sto* = *truth-value false*

then *cont sto*

else *execute* ⟦*C*⟧ *env*

{*execute* ⟦**while** *E* **do** *C*⟧ *env cont* }

sto

execute: Command → Env → Cont → Cont

execute ⟦ **while** *E* **do** *C* ⟧ *env cont* =

let *f s* = **if** *evaluate* ⟦*E*⟧ *env s* = *truth-value false*

then *cont s*

else *execute* ⟦*C*⟧ *env* { *f* } *s*

in

f — *f*: Sto → Sto

Commands with Continuations

execute: `Command` → Env → Cont → Sto → Sto

execute ⟦ **`let`** *D* **`in`** *C* ⟧ *env cont sto* =

let (*e*, *s*) = *elaborate* ⟦*D* ⟧ *env sto* **in**

execute ⟦*C* ⟧ *overlay*(*e*, *env*) *cont s*

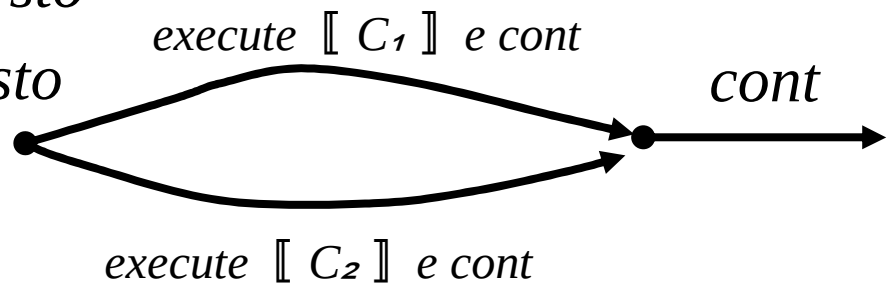
execute ⟦ **`if`** *E* **`then`** C_1 **`else`** C_2⟧ *env cont sto* =

let *e* = *bind-labels* ⟦ **`if`** *E* **`then`** C_1 **`else`** C_2⟧ *env cont* **in**

if *evaluate* ⟦ *E* ⟧ *env sto* = *truth-value true*

then *execute* ⟦C_1 ⟧ *e cont sto*

else *execute* ⟦C_2 ⟧ *e cont sto*



Label Declarations (cont.)

execute ⟦ C_1 ; C_2 ⟧ *env cont* =

let *e* = *bind-labels* ⟦ C_1 ; C_2 ⟧ *env cont* **in**

let c_2 = *execute* ⟦ C_2 ⟧ *e cont* **in**

let c_1 = *execute* ⟦ C_1 ⟧ *e* c_2 **in**

c_1

execute ⟦ *L* : *C* ⟧ *env cont* =

let *e* = *bind-labels* ⟦ *L* : *C* ⟧ *env cont* **in**

execute ⟦ *C* ⟧ *e cont*

Label Declarations (cont.)

- Net effect of these rules

$execute\ [\![\ \mathbf{let}\ D\ \mathbf{in}\ (L_1 : C_1\ ;\ L_2 : C_2\ ;\ \ldots\ ;\ L_n : C_n)\ ]\!]\ env\ cont\ sto$

$=$ **let** $(env', sto') = elaborate\ [\![D]\!]\ env\ sto$ **in**

let $c_n = execute\ [\![C_n]\!]\ e\ cont$ **in**

let $c_{n-1} = execute\ [\![C_{n-1}]\!]\ e\ c_n$ **in**

. . .

let $c_2 = execute\ [\![C_2]\!]\ e\ c_3$ **in**

let $c_1 = execute\ [\![C_1]\!]\ e\ c_2$ **in**

let $e = overlay(env', env)\ [L_1 \mapsto c_1, L_2 \mapsto c_2, \ldots, L_n \mapsto c_n]$ **in**

$c_1\ sto'$

Labels: Example

- C = `l: x:=2; while true do goto t; x:=7; t: x:=x+1;`
- $e_0 = [x \mapsto 1]$ $\quad c_0 = \lambda s.s$ $\quad s_0 = \lambda l.\perp$
- *execute* $[\![C]\!]\ e_0\ c_0\ s_0 = c_1\ s_0$ where

  c_1 = *execute* $[\![$`x:=2; while true do goto t; x:=7`$]\!]\ e_0'\ c_2$

  c_2 = *execute* $[\![$ `x:=x+1` $]\!]\ e_0'\ c_0$

  $e_0' = [x \mapsto 1, l \mapsto c_1, t \mapsto c_2]$
- $c_1\ s_0$ = *execute* $[\![$ `x:=2` $]\!]\ e_0'$ {*execute* $[\![$ **`while …`** $]\!]\ e_0'c_2$} s_0

  = *execute* $[\![$ **`while …`** $]\!]\ e_0'c_2\ s_0[1 \mapsto 2]$

  = *execute* $[\![$ **`while true do goto t`** $]\!]\ e_0'$ {*execute* $[\![$ `x:=7` $]\!]\ e_0'c_2$} $s_0[1 \mapsto 2]$

Example (cont.)

- Now

  $execute\ [\![\ \texttt{while true do goto t}\ ]\!]\ e\ c\ s = f\ s$

  where $f\ s =$ **if** $evaluate\ [\![\texttt{true}]\!]\ e\ s = truth\text{-}value\ false$

  then $c\ s$

  else $execute\ [\![\texttt{goto t}]\!]\ e\ \{f\}\ s$

  $= execute\ [\![\texttt{goto t}]\!]\ e\ \{f\}\ s = e\ \texttt{t}\ s$

- $\therefore\ c_1\ s_0 = execute\ [\![\ \texttt{while true do goto t}\ ]\!]\ e_0'$

  $\{execute\ [\![\ \texttt{x:=7}\ ]\!]\ e_0'\ c_2\}\ s_0[1\mapsto 2] = e_0'\ \texttt{t}\ s_0[1\mapsto 2]$

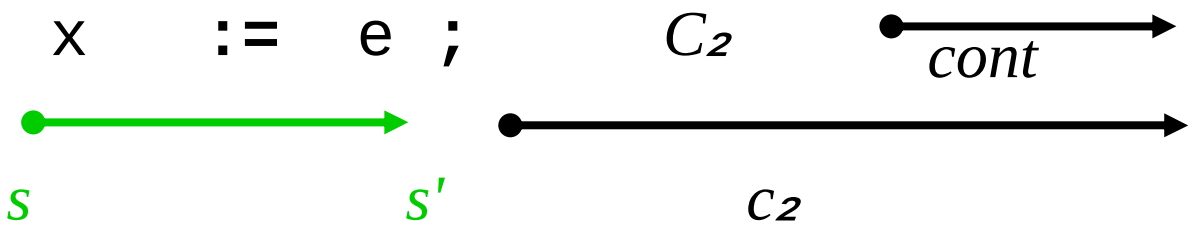
  $= c_2\ s_0[1\mapsto 2] = execute\ [\![\ \texttt{x:=x+1}\ ]\!]\ e_0'\ c_0\ s_0[1\mapsto 2]$

  $= s_0[1\mapsto 3]$

- $\therefore\ execute\ [\![C]\!]\ e_0\ c_0\ s_0 = s_0[1\mapsto 3] \qquad e_0 = [\texttt{x}\mapsto 1]$

Expression Continuations

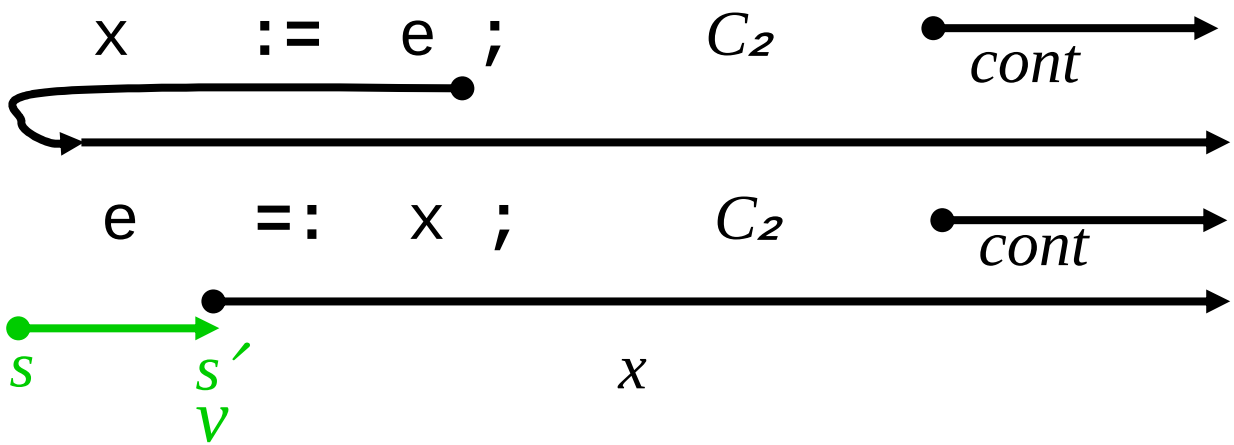
- Expression side-effects + sequencers ⇒ "expresssion continuations"
- Review "rest of the program" after assignment `Command`
  - (Command) continuations —Cont



  - Q: After `x:=e :Command` what is passed to "rest of the program" c_2 ?
  - A: s' : Store. ∴ "rest of the program" after C_1 is c_2 : Cont

Expression Continuations (cont.)

- "rest of the program" after `Expression`: Expression continuations—Econt
  - (Command) continuations —Cont



  - Q: After `e : Expression` what is passed to "rest of the program" x ? x is result of commands `x:=` v `;` C_2
  - A: v : Value s' : Store ∴ "rest of the program" after C_1 is
    x: Value → Store → Store
    x: Value → Cont

∴ Econt = Value → Cont

Expression Continuations (cont.)

Necessary revisions to “assignment” rule:

- Old: only command continuations

$$execute\ [\![\texttt{x := e}]\!]\ env\ cont\ sto$$
$$= cont\ (sto\ [env\ \texttt{x} \mapsto evaluate\ [\![\texttt{e}]\!]\ env\ sto])$$

  - No expression side-effects
  - Does not allow e to fail to fully evaluate
  - No way to “abandon” the assignment to the LHS x := …
  - No way to discard *cont* (normal command continuation)

Expression Continuations (cont.)

- New: with expression continuations

  x: Econt = Value → Store → Store = Value → Cont

  evaluate: `Expression` → Environ → Econt →

  Store → Store

  execute ⟦`x:= e`⟧ *env cont sto*

  = *evaluate* ⟦`e`⟧ *env* {λ*v*. λσ. *cont* (σ[*env* x ↦ *v*])} *sto*

x: Econt

Promise to store the value *v*: Value received from the expression e into location *env* x (thus altering the store)

if the normal expression continuation is followed

Expression Continuations (cont.)

- Let us work on a typical `Expression` to show how expession continuations are propagate

$$
\begin{aligned}
&\mathit{evaluate}\ [\![E_1 + E_2]\!]\ e\ x\ = \\
&\qquad \textbf{let}\ x_2 = \lambda v_2\ .\ x(v_1 + v_2)\ \textbf{in} \\
&\qquad \textbf{let}\ x_1 = \lambda v_1\ .\mathit{evaluate}\ [\![E_2]\!]\ e\ x_2\ \textbf{in} \\
&\qquad \mathit{evaluate}\ [\![E_1]\!]\ e\ x_1
\end{aligned}
$$

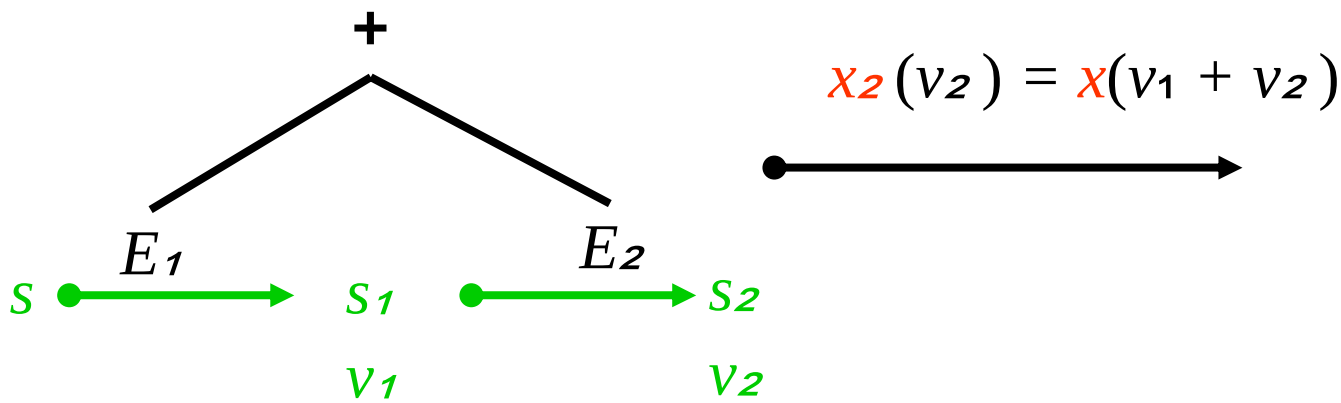
Promise when v_2 is known to use value $v_1 + v_2$ to select command continuation $x(v_1 + v_2)$

Promise when v_1 is known to use value v_1 to select command continuation

$\mathit{evaluate}\ [\![E_2]\!]\ \mathit{env}\ \{\lambda v_2\ .\ x(v_1 + v_2)\}$

Expression Continuations (cont.)

$$evaluate\ [\![E_1 + E_2]\!]\ e\ x\ =$$
$$evaluate\ [\![E_1]\!]\ e\ \{\lambda v_1\ .evaluate\ [\![\ E_2\ ]\!]\ e\ \{\lambda v_2\ .\ x(v_1 + v_2)\ \}\}$$



IMP$_{1g}$=IMP$_g$+serial clauses

- `Expression ::= . . .`
  `| begin Command ; return Expression end`
  `Command ::= . . .`
  `| Label : Command`
  `| goto Label`
- `POP2` had syntax `do Command resultis Expression`
- Cont = [Store → Store] —command continuations
- Econt = [Value → Cont] —expression continuations
- *execute:* `Command` → Environ → Cont → Cont
- *evaluate:* `Expression` → Environ → Econt → Cont

IMP1g=IMPg+serial clauses (cont.)

- Expression ::=

```
        |    Numeral
        |    false  |  true
        |    Identifier
        |    Expression + Expression
        |    Expression < Expression
        |    not Expression
        |    . . .
        | begin Command ; return Expression end
```

Semantics of IMP1g

- Expression Semantics

evaluate: `Expression` $\rightarrow$ Environ $\rightarrow$ Econt $\rightarrow$ Cont

evaluate $[\![N]\!]$ *e x* = **let** *n* = *valuation* $[\![N]\!]$ **in** *x* (*integer n*)

evaluate $[\![$**`false`**$]\!]$ *e x* = *x* (*truth-value false*)

evaluate $[\![$**`true`**$]\!]$ *e x* = *x* (*truth-value true*)

evaluate $[\![I]\!]$ *e x* = $\lambda\sigma$. *x* (*coerce*(σ, *find*(*e*, *I*))) σ

evaluate $[\![E_1$ **+** $E_2]\!]$ *e x* =

let x_2 = λv_2 . $x(v_1 + v_2)$ **in**

let x_1 = λv_1 .*evaluate* $[\![E_2]\!]$ *e* x_2 **in**

evaluate $[\![E_1]\!]$ *e* x_1

evaluate $[\![$**`not`** $E]\!]$ *e x* =

let x_1 = λv_1 . *x* ($\neg v_1$) **in**

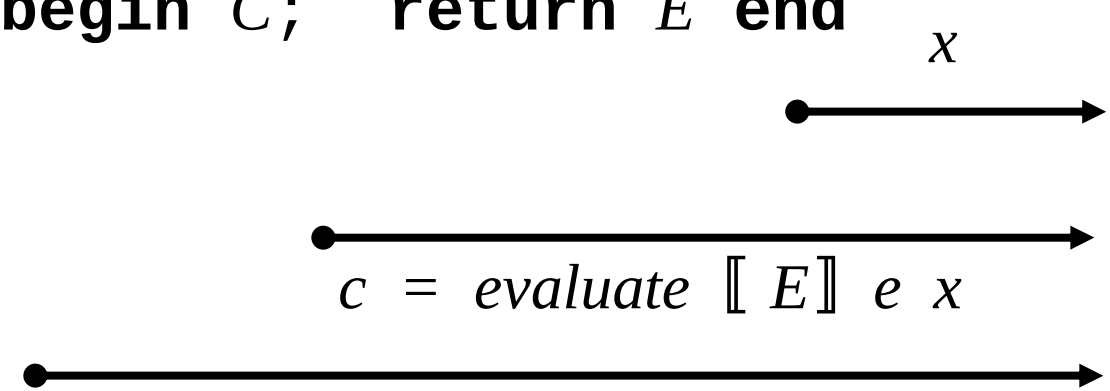
evaluate $[\![E]\!]$ *e* x_1

Semantics of IMP1g (cont.)

- Expression Semantics (cont.)

evaluate ⟦**begin** *C*; **return** *E* **end**⟧ *e x* =
let *c* = *evaluate* ⟦ *E*⟧ *e x* **in**
execute ⟦ *C* ⟧ *e c*

E = **begin** *C*; **return** *E* **end**



evaluate ⟦*E*⟧ *e x* = *execute* ⟦*C*⟧ *e c*

Semantics of IMP$_{1g}$ (cont.)

- Semantics of Commands: changed wherever `Expressions` are present

  execute: `Command` $\rightarrow$ Env $\rightarrow$ Cont $\rightarrow$ Cont

- No changes in continuation semantics of:

  $execute\ [\![\ \mathbf{skip}\ ]\!]\ env\ cont\ =$

  $execute\ [\![C_1\ \mathbf{;}\ C_2]\!]\ env\ cont\ =$

  $execute\ [\![\ L\ \mathbf{:} C\ ]\!]\ env\ cont\ =$

  $execute\ [\![\ \mathbf{goto}\ L\ ]\!]\ env\ cont\ =$

- We will handle declarations and **`let`** commands later

Semantics of IMP$_{1g}$ (cont.)

- Assignment: as in our earlier motivating example:

$execute\ [\![I := E]\!]\ env\ cont$

$= evaluate\ [\![E]\!]\ env\ \{\lambda v.\ \lambda\sigma.\ cont\ update\ (\sigma,\ env\ I,\ v)\}$

- Example: suppose $e_0 = e_0[\mathrm{x} \mapsto 1]$ i.e., $e_0\,\mathrm{x} = 1$

$execute\ [\![\mathtt{x:=x+1}]\!]\ e_0\ \{\ \lambda\sigma.\ c_0\ update\ (\sigma, 2, \sigma 1)\ \}\ s_0$

$= evaluate\ [\![\mathtt{x+1}]\!]\ e_0\ \{\lambda v.\ \lambda\sigma.$

$(\ \lambda s.\ c_0\ update\ (s, 2, s\ 1\ ))$

$update\ (\sigma, e_0\,\mathrm{x}, v)\ \}\ s_0$

$= evaluate\ [\![\mathtt{x+1}]\!]\ e_0\ \{\lambda v.\ \lambda\sigma.$

$(\ \lambda s.\ c_0\ update\ (s, 2, s\ 1\ ))$

$\sigma[1 \mapsto v\ ]\ \}\ s_0$

$= evaluate\ [\![\mathtt{x+1}]\!]\ e_0\ \{\lambda v.\ \lambda\sigma.$

$c_0\ update\ (\sigma[1 \mapsto v\ ],\ 2,\ v\ )\}\ s_0$

— note $\sigma[1 \mapsto\ v\ ]\ 1 = v$

Semantics of IMP$_{1g}$ (cont.)

- Next apply the evaluate rule:

  $evaluate\ [\![\ \texttt{x+1}\ ]\!]\ e\ \{x\} = evaluate\ [\![\ \texttt{x}\ ]\!]\ e\ \{\lambda v.\ x(v+1)\}$

- Thus

$evaluate\ [\![\ \texttt{x+1}\ ]\!]\ e_o\ \{\lambda v.\ \lambda\sigma.$
$c_o\ update\ (\sigma[1 \mapsto v\ ], 2, v\ )\}\ \ s_o$

$= evaluate\ [\![\ \texttt{x}\ ]\!]\ e_o\ \{\lambda v.\ \lambda\sigma.$
$c_o\ update\ (\sigma[1 \mapsto v+1\ ], 2, v+1\ )\}\ \ s_o$

- Next apply $evaluate\ [\![\ \texttt{x}\ ]\!]\ e\ \{x\}\ s = x\ fetch(s, e\ \texttt{x})\ s$

$= (\lambda v.\ \lambda\sigma.\ c_o\ update\ (\sigma[1 \mapsto v+1\ ], 2, v+1\ ))\ (s_o\ (e_o\ \texttt{x}\ ))\ s_o$

$= (\lambda v.\ \lambda\ \sigma.\ c_o\ update\ (\sigma[1 \mapsto v+1\ ], 2, v+1\ ))\ (s_o\ 1)\ \ s_o$

$= \lambda\ \sigma.\ c_o\ update\ (\sigma[1 \mapsto s_o 1 +1\ ], 2, s_o\ 1 +1\ )\ s_o$

$= c_o\ update\ (s_o[1 \mapsto s_o 1 +1\ ], 2, s_o\ 1 +1\ )$

$= c_o\ s_o[1 \mapsto s_o 1 +1, 2 \mapsto s_o 1 +1]$

Semantics of IMP1g (cont.)

- Command semantics (cont.): changes owing to presence of `Expressions`

execute ⟦ **`if`** *E* **`then`** *C₁* **`else`** *C₂* ⟧ *env cont* =

let *e* = *bind-labels* ⟦ **`if`** *E* **`then`** *C₁* **`else`** *C₂* ⟧ *env cont* **in**

let *x* = (λ*b*. **if** *b* **then** *execute* ⟦ *C₁* ⟧ *e cont*

else *execute* ⟦ *C₂* ⟧ *e cont*) **in**

evaluate ⟦ *E* ⟧ *env x*

Semantics of IMP$_{1g}$ (cont.)

- **`While`** is like **`if`** but with identity continuation on the fale branch and continuation recursion on the true branch

$$execute\ [\![\ \texttt{while}\ E\ \texttt{do}\ C\ ]\!]\ env\ cont\ =$$

$$\textbf{let}\quad x = \{\lambda b.\ \textbf{if}\ b\ \textbf{then}\ execute\ [\![\ C\ ]\!]\ e$$

$$\{execute\ [\![\ \texttt{while}\ E\ \texttt{do}\ C\ ]\!]\ env\ cont\ \}$$

$$\textbf{else}\ \{\ cont\ \}$$

$$\}$$

—recursively define *continuation*

$$\textbf{in}$$

$$evaluate\ [\![\ E\ ]\!]\ env\ x$$

Semantics of IMP1g (cont.)

- Or using the fixed point operator Y

$$
\begin{aligned}
&\mathit{execute}\ [\![\ \texttt{while}\ E\ \texttt{do}\ C\ ]\!]\ \mathit{env}\ \mathit{cont}\ = \\
&\quad \textbf{let}\ \tau = \lambda c.\ \mathit{evaluate}\ [\![E]\!] \\
&\qquad\qquad\qquad\qquad \mathit{env} \\
&\qquad\qquad\qquad\qquad \{\lambda b.\ \textbf{if}\ b\ \ \textbf{then}\ \ \mathit{execute}\ [\![C]\!]\ e\ \{\ c\ \} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \{\ c\ \}\ \} \\
&\quad \textbf{in}\ \ \mathsf{Y}\ \tau
\end{aligned}
$$

Declaration in IMP_{1g}

- One more problem area in declaration. Recall that constant declarations can contain expression evaluations:
- `Declaration ::= | . . .`

  `const` `Identifier ~ Expression`
- Old *elaborate*: `Declaration` $\rightarrow$ Environ $\rightarrow$ Store
  $\rightarrow$ Environ $\times$ Store
- But jumps out of `Expression` are possible!

Example:

L_1 **:** C_1 **; let const** c **~ begin** C_2 **; return** E
in L_3 **:** C_3 **;** L_4 **:** C_4

↑ end of **`let`** block

- Jumps from C_2 to C_1 or C_4 are OK, but jumps to C_3 would abandon a needed declaration

Semantics of IMP$_{1g}$ (cont.)

- Solution?
- Correctamundo!: add *declaration continuations*

- Dcont = [Environ $\rightarrow$ Store $\rightarrow$ Store] = [Environ $\rightarrow$ Cont]
  - Promise to follow the normal command continuation if make it through the declaration with a particular Environ value

elaborate: `Declaration` $\rightarrow$ Environ $\rightarrow$ Dcont $\rightarrow$ Cont

- Compare:

  evaluate: `Expression` $\rightarrow$ Environ $\rightarrow$ Econt $\rightarrow$ Cont

Semantics of IMP$_{1g}$ (cont.)

elaborate: `Declaration` $\rightarrow$ Environ $\rightarrow$ Dcont $\rightarrow$ Cont

Dcont = [Environ $\rightarrow$ Cont]

elaborate $[\![$ **`const`** I **`~`** $E]\!]$ *env d* =

let $x = \{ \lambda v.\ d\ overlay\ (bind(I, v), env)) \}$

—an Econt with a promise to add new binding

in

evaluate $[\![E]\!]$ *env x* —a Cont

Semantics of IMP1g (cont.)

elaborate ⟦ **var** *I* **:** *T* ⟧ *env d sto* =

let (*sto′, loc*) = *allocate sto* **in**

d overlay(*bind*(*I*, *variable loc*), *env*) *sto′*

—a new store

- Or to see result type of Cont explicitly

elaborate ⟦ **var** *I* **:** *T* ⟧ *env d* =

$\lambda\sigma.$

let (σ', *loc*) = *allocate* σ **in**

d overlay(*bind*(*I*, *variable loc*), *env*) σ'

Semantics of IMP$_{1g}$ (cont.)

- Reminder of how declarations formerly interfaced with Commands:

execute ⟦ **let** *D* **in** *C* ⟧ *env cont sto* =

let (*e*, *s*) = *elaborate* ⟦*D* ⟧ *env sto* **in**

execute ⟦*C* ⟧ *overlay*(*e*, *env*) *cont* *s*

- In the new semantics , the normal continuation of a declaration is the command body followed by the normal continuation of the **let**

elaborate: `Declaration` → Environ → Dcont → Cont

Dcont = [Environ → Cont]

execute ⟦ **let** *D* **in** *C* ⟧ *env cont* =

elaborate ⟦*D* ⟧ *env* (λ*e*. *execute* ⟦*C* ⟧ *e cont*)

—a Cont

Example

C = **let** c ~ (**begin if** x >0 **then** x := x+1 **else goto** t; **return** x) **in** y := c ;

- e_0 = [x ↦ 1 , y ↦ 2] $c_0 = \lambda s.s$ s_0 = [1 ↦5 , 2 ↦3]
- *execute* ⟦C⟧ e_0 c_0 = *elaborate* ⟦ c ~ (**begin if** … ⟧ e_0 ($\lambda e.$ *execute* ⟦ y := c ⟧ e c_0) =

  evaluate ⟦ (**begin if** … ⟧ e_0 { $\lambda v.$ (($\lambda e.$ *execute* ⟦ y := c ⟧ e c_0) *overlay* (*bind*(c, v), e_0) } =

  evaluate ⟦ (**begin if** … ⟧ e_0

  { $\lambda v.$ ($\lambda \sigma.$ c_0 *update* (σ, e_0 y, v)) }

= *execute* ⟦**if** …⟧ e_0 { *evaluate* ⟦ x⟧ e_0

{ $\lambda v.$ ($\lambda \sigma.$ c_0 *update* (σ, 2, v)) } }

Example (cont.)

$= execute\ [\![\texttt{if}\ \ldots]\!]\ e_o\ \{\ evaluate\ [\![\texttt{x}]\!]\ e_o$

$\{\ \lambda v.\ (\lambda\sigma.\ c_o\ update\ (\sigma, 2, v)\ )\ \}\ \}$

$= execute\ [\![\texttt{if}\ \ldots]\!]\ e_o\ \{$

$\lambda\sigma.\ c_o\ update\ (\sigma, 2, \sigma\ (e_o\ \texttt{x})\ )\ \}$

$= execute\ [\![\texttt{if}\ \ldots]\!]\ e_o\ \{$

$\lambda\sigma.\ c_o\ update\ (\sigma, 2, \sigma\ 1)\ \}$

Now in s_o we have that $[\![\texttt{x > 0}]\!]$ is true, with no side effect and so for all $e\ c\ s$

$execute\ [\![\texttt{if}\ \ldots]\!]\ e\ c\ s = execute\ [\![\texttt{x:=x+1}]\!]\ e\ c\ s$

Applying to our case, then $execute\ [\![C]\!]\ e_o\ c_o\ s_o =$

$execute\ [\![\texttt{x:=x+1}]\!]\ e_o\ \{\ \lambda\sigma.\ c_o\ update\ (\sigma, 2, \sigma\ 1)\ \}\ s_o$.

Example (cont.)

- The example on pp. 36-37 showed that

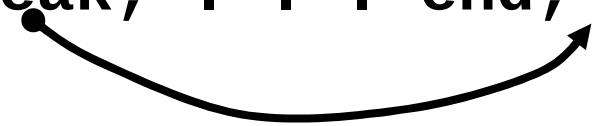
$$execute\ [\![\ \texttt{x:=x+1}\ ]\!]\ e_o\ \{\ \lambda\sigma.\ c_o\ update\ (\sigma, 2, \sigma\, 1\ )\ \}\ \ s_o$$
$$= \ \ c_o\ s_o[1\ \mapsto\ s_o 1\ +1, 2 \mapsto\ s_o 1\ +1]$$

- So we get

$$execute\ [\![C]\!]\ e_o\ c_o s_o\ = c_o\ s_o[1\ \mapsto\ 5\ +1, 2 \mapsto\ 5\ +1]$$
$$= \ \ c_o\ s_o[1 \mapsto\ 6, 2 \mapsto\ 6]$$

Other Commands

- **while** E **do . . . break; . . . end;** C



- *execute* ⟦ **break** ⟧ *env cont* = ?
  - Want the command continuation for the surrounding **while** to be available in *env*
  - So every *execute* ⟦ **while** E **do** C ⟧ *e c* must add a binding to the environment of execution of C
  - Do details

Other Commands (cont.)

- **while** *E* **do . . . continue; . . . end;**

- *execute* ⟦ **continue** ⟧ *env cont* = ?
  - Want the command continuation starting with eval of *E*

execute ⟦ **while** *E* **do** *C* ⟧ *env cont*

let $x = \{\lambda b.$ **if** *b* **then** *execute* ⟦ *C* ⟧ *e*

{*execute* ⟦ **while** *E* **do** *C* ⟧ *env cont* }

else { *cont* } } **in**

evaluate ⟦ *E* ⟧ *env x*

- So every *execute* ⟦ **while** *E* **do** *C* ⟧ *e c* must add a binding to the command continuation *evaluate* ⟦ *E* ⟧ *env x*

Exception Handling

- *Exception:* an infrequent event
  - An event has a name
- *Raise, Signal* (an exception): occurrence of the event
  - By definition this is what it means for an event to “occur”
  - Can be explicit (``**`raise OVERFLOW''`**) or implicit (storage error)
- *Handler:* subroutine/code sequence to be executed when the corresponding named exception is raised
- Why exceptions?
  - Makes code more readable and modular by removing tedious, repetitive explict tests for exceptional conditions
  - Eliminates user coding of explicit subroutine call or unconditional transfer

Exception Handling (cont.)

- *Exception continuation:* the combination of control point and access environment that begins execution after the handler exits.
  - *Resumption model* of continuations: take up execution at point just after where exception occurred
    - Example: **BASIC**

```
 . . .
ON ERROR GOTO 100
X = Y +2                    //  resumption point
 . . .
100 . . .
    . . .
    RESUME NEXT
```

Exception Handling (cont.)

- *Termination model* of continuations: take up execution at point just after the return or exit from the block where exception occurred
- Example: **Ada**

```
SINGULAR: exception;
begin
 . . .
exception
 when SINGULAR | NUMERIC_ERROR  =>
      PUT("matrix is singular");
 when others =>
      PUT("fatal error");
      raise ERROR;
end;
```

block

Exception Handling (cont.)

- When an exception is raised, the handler in the current block is executed and then the block is exited
- If no handler exists, the block is exited, the *dynamic link* is followed, and the exception is re-raised in the new block (*dynamic propagation*)
- Handlers use *dynamic binding*: free names have their latest meanings at time that exception was raised
- Some exceptions are pre-defined and raised implicitly:
  - STORAGE_ERROR – heap storage exhausted
  - CONSTRAINT_ERROR - attempt to index out of bounds, to access through a null pointer, to divide by zero, &c

Exception Handling: Semantics

- A simplified model of termination semantics
- Abstract syntax

```
Declaration ::= Declaration ; Declaration
              | on Identifier do Command | . . .
Command ::= begin Declaration ; Command end
              | raise Identifier
```

- Semantic Domains

Cont = Store $\rightarrow$ Store

Bindable = *value* Value + *contin* Cont + …

Exception Handling: Commands

- *execute:* `Command` → Environ → Cont → Cont

execute ⟦**begin** *D* **;** *C* **end**⟧ *e c* =
  let *e′* = *elaborate* ⟦*D*⟧ *e c* **in**
  execute ⟦ *C* ⟧ *overlay*(*e′*, *e*) *c*

Termination model: continuation to be used by local handlers Is continuation of entire block

After body, take continuation after the **end**

Exception Handling: Commands (cont.)

$$execute\ [\![\ \texttt{raise}\ I\ ]\!]\ e\ c\ =$$

$$\quad \textbf{let}\ \ contin\ c' = find(e, I)\ \ \textbf{in}$$

$$\quad c'$$

- Discard normal continuation c
- Extract continuation for I from environment of *current* block —block containing the **`raise`**
- Handler at c' repairs current Store
- Handler continuation bound to exception name in declaration semantics

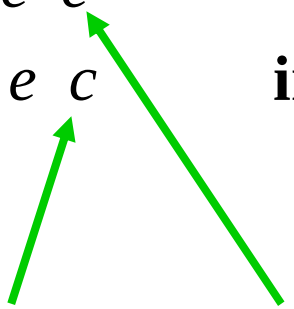
Exception Handling: Declarations

- *elaborate:* `Declaration` → Environ → Cont → Environ

elaborate ⟦**on** *I* **do** *C*⟧ *e c* =

let *c′* = *execute* ⟦ *C* ⟧ *e c* **in**

c′



- Continuation after handler is same as that which was passed to declaration

Exception Handling: Declarations (cont.)

- *elaborate:* `Declaration` → Environ → Cont → Environ

Continuation will be that from block end

$$elaborate\ [\![D_1\ ;\ D_2]\!]\ e\ c\ =$$

$$\textbf{let}\ e_1 = elaborate\ [\![D_1]\!]\ e\ c\ \ \textbf{in}$$

$$\textbf{let}\ e_2 = elaborate\ [\![D_2]\!]\ overlay\ (e_1, e)\ c\ \ \textbf{in}$$

$$overlay\ (e_2,\ overlay\ (e_1, e))$$

- Continuation passed is that after block end