6. Imperative Languages II

An Interactive File Editor

- File = list of records.
- Primary store holds currently edited file.
- Secondary store holds files indexed by names.
- User issues commands to operate on files and records.
- Output log echoes input commands and reports errors.

Opened File

- Pair of record lists with "current" record.

  $\boxed{r_{i-1} \ldots r_2 r_1}\ \boxed{\underline{r_i} r_{i+1} \ldots r_{last}}$

- *newfile* represents file with no records.

  $\boxed{}\ \boxed{}$

- *copyin* reads file from file system

  $\boxed{}\ \boxed{\underline{r_1} r_2 \ldots r_{last}}$

- *copyout* writes file back to file system.

- *forwards* steps one record ahead.

  $\boxed{r_{i-1} \ldots r_2 r_1}\ \boxed{\underline{r_i} r_{i+1} \ldots r_{last}} \Rightarrow$
  $\boxed{r_i r_{i-1} \ldots r_2 r_1}\ \boxed{\underline{r_{i+1}} \ldots r_{last}}$

- *backwards* steps one record back.

- *insert* places record behind the current one.

  $\boxed{r_i \ldots r_2 r_1}\ \boxed{\underline{r'} r_{i+1} \ldots r_{last}}$

- *delete* removes the current record.

Valuation Functions

- **C**: Program-session $\rightarrow$ *File-system* $\rightarrow$ (*Log* $\times$ *File-system*)

  Program-session takes a file system and produces a log file and an updated file system.

- **S**: Command-sequence $\rightarrow$ *Openfile* $\rightarrow$ (*Log* $\times$ *Openfile*)

  Command-sequence takes an open file and produces a log and an updated file.

- **C**: Command $\rightarrow$ *Openfile* $\rightarrow$ (*String* $\times$ *Openfile*)

  Command takes an open file and produces a log message and an updated file.

Error Messages

C[[**delete**]](*newfile*)
= let $(k', p') =$ *isempty*(*newfile*) $\rightarrow$
("error: file is empty", *newfile*)
[] ("", *delete*(*newfile*))
in ("delete" *concat* k', p')
= let $(k', p') =$ ("error: file is empty", *newfile*)
in ("delete" *concat* k', p')
= ("delete" *concat* "error: file is empty", *newfile*)
= ("delete error: file is empty", *newfile*)

S[[C **cr** S]]

1. Evaluate **C**[[C]]p to obtain next log entry l' plus updated file p'.
2. Cons l' to log list and pass p' to **S**[[S]].
3. Evaluate **S**[[S]]p' to obtain meaning of rest of program i.e. rest of log output plus final version of file.

Collecting Log

P[[**edit** A **cr moveback cr delete cr quit**]]s_0
= (“edit A” *cons*
  fst(**S**[[**moveback cr delete cr quit**]]p_0),
  update([[A]],
    copyout(*snd*(**S**[[**moveback**
     cr delete cr quit]]p_0), s_0))
    where p_0 = *copyin*(*access* ([[A]], s_0)))
= . . .
= (“edit A” *cons* “moveback error:
    at front already”
    cons fst(**S**[[**delete cr quit**]]p_0),
  update([[A]],
    copyout(*snd*(**S**[[**delete cr quit**]]p_0))))
= (“edit A moveback error:
    at front already delete quit”,
  update([[A]], *copyout*(p_1), s_0)
  where p_1 = *delete*(p_0)

A Dynamically Typed Language with Input and Output

- Variable may take on values from different data types.
- Run-time type checking required.
- Input and output included.

Introduction of "type tags"

$$\mathit{Storable\text{-}value} = \mathit{Tr} + \mathit{Nat}$$
$$\mathit{Store} = \mathit{Id} \to \mathit{Storable\text{-}value}$$

$$\mathit{Errvalue} = \mathit{Unit}$$
$$\mathit{Expressible\text{-}value} = \mathit{Storable\text{-}value} + \mathit{Errvalue}$$

Valuation Functions

- $State = Store \times Input \times Output$

  store, input and output buffers

- $Post\text{-}State = OK + Err$

  $OK = State$, $Err = State$
  successful evaluation or type error

- **P**: Program $\rightarrow$ $Store \rightarrow Input \rightarrow Post\text{-}state_{\perp}$

  Program takes input store and input buffer and returns a new state

- **C**: Command $\rightarrow$ $State \rightarrow Post\text{-}state_{\perp}$

  Command takes state and returns a new state.

- **E**: Expression $\rightarrow$ $Store \rightarrow Expressible\text{-}value$

  Expression takes store and returns a value.

Composition of States

$\mathbf{C}[[\mathrm{C}_1;\ \mathrm{C}_2]] = \mathbf{C}[[\mathrm{C}_1]]$ *check-cmd* $\mathbf{C}[[\mathrm{C}_2]]$

Command = *State* $\rightarrow$ *Post-state*$_\perp$
check-cmd: Command $\times$ *Command* $\rightarrow$ *Command*
h_1 *check-cmd* h_2 =
    $\lambda a.$ let $z = h_1(a)$ in
    cases z of
        isOK$(s,\ i,\ o) \rightarrow h_2(s, i, o)$
        isErr$(s,\ i,\ o) \rightarrow z$
    end

1. Give current state a to $\mathbf{C}[[\mathrm{C}_1]]$ producing a post-state $z = \mathbf{C}[[\mathrm{C}_2]]a'$.
2. If z is a proper state a' and if the state component is OK, produce $\mathbf{C}[[\mathrm{C}_2]]a'$
3. If z is errnoneous, $\mathbf{C}[[\mathrm{C}_2]]$ is ignored and z is the result.

Similar for check of expression results.

Error Handling

- Algebra operations abort normal evaluation when type error occurs.

  Representation of low-level (e.g. hardware-level) fault detection and branching mechanisms.

- Machine action: on fault, branch out of the program.
- Semantics: on fault, branch out of function expression

  Propagation of type errors yields same result as machine action.

Altering the Properties of Stores

1. Store critical to evaluation of a phrase.
2. Only one copy of store exists during execution.
3. Store serves as means of communication between phrases.

- Typical features of a store in sequential programming languages.
- What happens if we relax each of these restrictions?

Delayed Evaluation

How to rewrite a function application $f(e)$?

- Call-by-value simplification
  - *e evaluated before f is executed;*
  - *Safe method if f is strict.*
- Call-by-name simplification
  - *f executed with non-evaluated e;*
  - *Safe method also if f is non-strict.*

f: $Nat_\perp \rightarrow Nat_\perp$
$f = \lambda x.$ zero
$f(\perp) =$ zero

E[[e]] $= \perp$
f(**E**[[e]]) $\Rightarrow$?

Simplification of argument may require infinite number of steps!

Non-strict Store Updates

May store operate on improper values?

- $\mathit{Store} = \mathit{Id} \to \mathit{Nat}_\perp$

  Improper values may be stored.

- *update:* $\mathit{Id} \to \mathit{Nat}_\perp \to \mathit{Store} \to \mathit{Store}$
  - $\mathit{update} = \lambda i.\lambda n.\lambda s.[i \mapsto n]s$
  - (*update* [[I]] (**E**[[E]]s) s) *is defined even in the "loop forever situation"* **E**[[E]]$s = \perp$.
  - *Unevaluated expressions may be stored in* s.

- **E**[[E]]s needs not be evaluated until used.
  - *Delayed (lazy) evaluation.*
  - *Value must be determined with respect to the store that was active when [[E]] was saved.*

Example

begin
X := 0
Y := X+1
X := 4
return Y

K: Block $\to$ $Store_\perp \to Nat_\perp$
K[[**begin** C **return** E]] =
$\lambda s.$ **E**[[E]] (**C**[[C]]s)

K[[**begin** X:=0; Y:= X+1; X:=4 **return** Y]]s_0
= **E**[[Y]] (**C**[[X:=0; Y:= X+1; X:=4]]s_0)
= **E**[[Y]] (**C**[[Y:= X+1; X:=4]] (**C**[[X:=0]]s_0))
= **E**[[Y]] (**C**[[Y:= X+1; X:=4]]
(*update* [[X]] (**E**[[0]]s_0) s_0))
= **E**[[Y]] (**C**[[Y:= X+1; X:=4]] s_1)

s_1 = (**E**[[0]]s_0) *needs not be simplified!*

Example

s_2 = *update* [[Y]] (**E**[[X+1]]s_1) s_1
s_3 = *update* [[X]] (**E**[[4]]s_2) s_2

E[[Y]] (**C**[[Y:= X+1; X:=4]] s_1)
= **E**[[Y]]s_3
= *access* [[Y]] s_3
= **E**[[X+1]]s_1
= **E**[[X]]s_1 *plus one*
= (*access* [[X]] s_1) *plus one*
= **E**[[0]]s_0 *plus one*
= *zero plus one*
= *one*

- Evaluation **E**[[X]]s_1 required.
- Old store s_1 must be retained.
- **E**[[X]]s_3 would be *five*!

Non-Strict Command Execution

- Carry delayed evaluation up to level of commands.
- Make **C**, **E**, and **K** non-strict in their store arguments.
- Only those commands need to be evaluated that have an effect on the output of a program!

Example

begin
    X:=0;
    diverge;
    X:=2
return X+1

K[[**begin** X:=0; **diverge**; X:=2 **return** X+1]]s_0
= **E**[[X+1]] (**C**[[X:=0; **diverge** X:=2]]s_0)
= **E**[[X+1]] (**C**[[X:=2]](**C**[[**diverge**; X:=2]]s_0))
= **E**[[X+1]](**C**[[X:=2]]s_1)
= **E**[[X+1]](*update* [[X]] (**E**[[2]]s_1) s_1)
= **E**[[X+1]]([[[X]] $\mapsto$ (**E**[[2]]s_1)]s_1)
= **E**[[X]]([[[X]] $\mapsto$ (**E**[[2]]s_1)]s_1) *plus one*
= **E**[[2]]s_1 *plus one*
= *two plus one*
= *three*

s_1 = **C**[[X:=0; **diverge**]]s_0 *needs not be simplified!*

Retaining Multiple Stores

$\mathbf{E}[[\mathsf{E}_1 + \mathsf{E}_2]] = \lambda \mathsf{s}.\mathbf{E}[[\mathsf{E}_1]]s \textit{ plus } \mathbf{E}[[\mathsf{E}_2]]s$

- **E** *uses store* s *in "read only" mode.*
- *No second store required for implementation.*

$\mathbf{E}[[$**begin** C **return** E$]] =$
$\underline{\lambda}s.$ let $s' = \mathbf{C}[[\mathsf{C}]]s$ in $\mathbf{E}[[\mathsf{E}]]s'$

- *Side effects in expression evaluation possible.*
- *Multiple stores required for implementation.*

$\mathbf{E}[[\mathsf{E}_1 + \mathsf{E}_2]] = \lambda s.$ let $(v', s') = \mathbf{E}[[\mathsf{E}_1]]s$
$(v'', s'') = \mathbf{E}[[\mathsf{E}_2]]s'$
in $(v' \textit{ plus } v'', s'')$
$\mathbf{E}[[$**begin** C **return** E$]] =$
$\underline{\lambda}s.$ let $s' = \mathbf{C}[[\mathsf{C}]]s$ in $(\mathbf{E}[[\mathsf{E}]]s',\ s')$

- *Local updates remain in global store.*
- *Expression evaluation returns new store.*

Non-Communicating Commands

- Store facilitates the building up of side effects that lead to some final value.
- Command advances computation by reading and modifying the values left in the store by previous commands.
- Otherwise, communication breaks down and language loses sequential flavor.

combine: $D \times D \to D$
Domain $s \in Store = Id \to D$

C: Command $\to Store_\perp \to Store_\perp$
$\mathbf{C}[[C_1;\ C_2]] = \underline{\lambda} s.\ join\ (\mathbf{C}[[C_1]]s)\ (\mathbf{C}[[C_2]]s)$

join: $Store_\perp \to Store_\perp \to Store_\perp$
$join = \underline{\lambda} s_1 . \underline{\lambda} s_2 .\ (\lambda i . s_1(i)\ combine\ s_2(i))$

Parallel but non-interfering parallelism.