Imperative Languages II

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

An Interactive File Editor

- File = list of records.
- Primary store holds currently edited file.
- Secondary store holds files indexed by their 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.

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

- *newfile* represents file with no records.

  $\boxed{}\ \boxed{}$

- *copyin* reads file from file system

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

- *copyout* writes file back to file system.
- *forwards* steps one record ahead.

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

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

- *backwards* steps one record back.
- *insert* places record behind current one.

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

- *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*)

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

$\mathbf{P}[\![\textbf{edit}\ \mathsf{A}\ \textbf{cr moveback cr delete cr quit}]\!]s_0$
$= (\text{"edit A"}\ \textit{cons}$
$\quad \textit{fst}(\mathbf{S}[\![\textbf{moveback cr delete cr quit}]\!]p_0),$
$\quad \textit{update}([\![\mathsf{A}]\!],$
$\quad\quad \textit{copyout}(\ \textit{snd}(\mathbf{S}[\![\textbf{moveback}$
$\quad\quad\quad \textbf{cr delete cr quit}]\!]p_0),\ s_0))$
$\quad\quad \text{where } p_0 = \textit{copyin}(\textit{access}\ (\ [\![\mathsf{A}]\!],\ s_0)))$
$= \ldots$
$= (\text{"edit A"}\ \textit{cons}\ \text{"moveback error:}$
$\quad\quad \text{at front already"}$
$\quad\quad \textit{cons}\ \textit{fst}(\mathbf{S}[\![\textbf{delete cr quit}]\!]p_0),$
$\quad \textit{update}([\![\mathsf{A}]\!],$
$\quad\quad \textit{copyout}(\textit{snd}(\mathbf{S}[\![\textbf{delete cr quit}]\!]p_0))))$
$= (\text{"edit A moveback error:}$
$\quad\quad \text{at front already delete quit"},$
$\quad \textit{update}([\![\mathsf{A}]\!],\ \textit{copyout}(p_1),\ s_0)$
$\quad \text{where } p_1 = \textit{delete}(p_0))$

Copyout function

Interactive text editor

copyout: Openfile $\rightarrow$ $\text{File}_\perp$
copyout = λ(front, back). null front $\rightarrow$ back
    [] copyout((tl front), ((hd front) cons back))

- Functional C, copyout = fix C.
- With i unfoldings, list pairs of length $i - 1$ can be appended.
- Codomain is lifted because least fixed point semantics requires that codomain of any recursively defined function be pointed.

  ispointed($A \rightarrow B$) = ispointed(B)
  min($A \rightarrow B$) = λa.min(B)

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"

Storable-value = Tr + Nat
Store = Id → Storable-value

Errvalue = Unit
Expressible-value =
Storable-value + Errvalue

Valuation Functions

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

  store, input and output buffers

- *Post-State* $=$ *OK* $+$ *Err*

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

- **P**: Program $\rightarrow$ *Store* $\rightarrow$ *Input* $\rightarrow$ *Post-state*$_\perp$

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

- **C**: Command $\rightarrow$ *State* $\rightarrow$ *Post-state*$_\perp$

  Command takes state and returns a new state.

- **E**: Expression $\rightarrow$ *Store* $\rightarrow$ *Expressible-value*

  Expression takes store and returns a value.

Composition of States

$$\mathbf{C}[[\mathsf{C}_1;\ \mathsf{C}_2]] = \mathbf{C}[[\mathsf{C}_1]]\ \textit{check-cmd}\ \mathbf{C}[[\mathsf{C}_2]]$$

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

$$\text{f: Nat}_\perp \rightarrow \text{Nat}_\perp$$
$$\text{f} = \lambda x.\ \text{zero}$$
$$\text{f}(\perp) = \text{zero}$$

$$\mathbf{E}[[\text{e}]] = \perp$$
$$\text{f}(\mathbf{E}[[\text{e}]]) \rightarrow ?$$

Simplification of argument may require infinite number of steps!

Non-strict Store Updates

May store operate on improper values?

- *Store* $=$ *Id* $\rightarrow$ *Nat*$_\perp$

  Improper values may be stored.

- *update: Id* $\rightarrow$ *Nat*$_\perp$ $\rightarrow$ *Store* $\rightarrow$ *Store*
  - *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 use.
  - 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 → *Store*$_\bot$ → $Nat_\bot$
K[[**begin** C **return** E]] =
  $\underline{\lambda} s$. **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

$$s_2 = \textit{update}\ [[\mathsf{Y}]]\ (\mathbf{E}[[\mathsf{X}+1]]s_1)\ s_1$$
$$s_3 = \textit{update}\ [[\mathsf{X}]]\ (\mathbf{E}[[4]]s_2)\ s_2$$

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

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

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

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

$\mathbf{E}[[\textbf{begin}\ \mathsf{C}\ \textbf{return}\ \mathsf{E}]] =$
$\quad \underline{\lambda} s.$ 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: $D \times D \rightarrow D$
Domain $s \in \mathit{Store} = \mathit{Id} \rightarrow D$

C: Command $\rightarrow \mathit{Store}_\perp \rightarrow \mathit{Store}_\perp$
$\mathbf{C}[[C_1; C_2]] = \underline{\lambda} s.\ \mathit{join}\ (\mathrm{C}[[C_1]]s)\ (\mathbf{C}[[C_2]]s)$

join: $\mathit{Store}_\perp \rightarrow \mathit{Store}_\perp \rightarrow \mathit{Store}_\perp$
$\mathit{join} = \underline{\lambda} s_1.\underline{\lambda} s_2.\ (\lambda i.s_1(i)\ \mathit{combine}\ s_2(i))$

Parallel but non-interfering parallelism.