Computer Science & Engineering Department
I. I. T. Kharagpur

Principles of Programming Languages: CS40032

Elective

Assignment – 8: Interactive File Editor *Marks: 25*

Assign Date: *12th April, 2019* Submit Date: *18th April, 2019*

---

1. The following set of questions are based on the Interactive File Editor.

   - The language is an interactive file editor. A file is a list of records, where the domain of records is taken as primitive. The file editor makes use of two levels of store: - the primary store is a component holding the file edited upon by the user, and the secondary store is a system of text files indexed by their names.

   Abstract Syntax:

   - Consider the entities as:
     $P \in Program\_session$, $S \in Command\_sequence$, $C \in Command$, $R \in Record$, $B \in Boolean\_expr$
     $I \in Identifier$
     $P ::=$ **edit** I **cr** S
     $S ::= C$ **cr** S | **quit**
     $C ::=$ **newfile** | **moveforward** | **moveback** | **insert** R | **delete**

   How does the File Editor work:

   - The edited files are values from the $Openfile$ domain
   - An opened file $r_1, r_2, \cdots, r_{last}$ is represented by two lists of text records; the lists break the file open in the middle: $\underline{r_{i-1}...r_2r_1}$ $\underline{r_ir_{i+1}...r_{last}}$ where r_i is the *current* record of the opened file
   - $newfile$ represents a file with no records
   - *copyin* takes a file from the file system and organizes it as:
     $\underline{\quad\quad}$ $\underline{r_1r_2...r_{last}}$ where r_1 is the *current* record of the opened file
   - The $forwards$ operation makes the record following the current record the new current record. Pictorially, for:
     $\underline{r_{i-1}...r_2r_1}$ $\underline{r_ir_{i+1}...r_{last}}$ a forwards move produces: $\underline{r_ir_{i-1}...r_2r_1}$ $\underline{r_{i+1}...r_{last}}$
   - *backwards* performs the reverse operation
   - $insert$ places a record r before the current record; an insertion of record r' produces: $\underline{r_{i-1}...r_2r_1}$ $\underline{r'r_ir_{i+1}...r_{last}}$ Hence the current record is r'
   - *backspace* removes the record before the current record
   - The final three operations test whether a) the first record in the file is current (at_first_record), b) the last record in the file is current (at_last_record), or if c) the file is empty ($isempty$)

   Semantic Algebra:

   - *Truth Values:*
     Domain: $t \in Tr$ Operations: $true, false : Tr = \mathcal{B}$, $and : Tr \times Tr \to Tr$
   - *Identifiers:* Domain: $i \in Id = Identifier$
   - *Text records:* Domain: $r \in Record$

- *Text file:* Domain: $f \in File = Record^*$
- *File System:* Domain: $s \in File\_system = Id \to File$
  Operations: $access : Id \times File\_system \to File$ where $access = \lambda(i, s).s(i)$,
  $update : Id \times File \times File\_system \to File\_system$ where $update = \lambda(i, f, s).[i \mapsto f]s$
- *Open file:*
  Domain: $p \in Openfile = Record^* \times Record^*$
  Operations:
  $newfile : Openfile$ where $newfile = (nil, nil)$,
  $copyin : File \to Openfile$ where $copyin = \lambda f.(nil, f)$
  $copyout : Openfile \to File$ where $copyout = fix\ F = F(copyout)$
  Functional $F : (Openfile \to File_\perp) \to (Openfile \to File_\perp)$:
  $F = \lambda f.\lambda(front, back).null\ front \to back\ []\ f((tl\ front), ((hd\ front)\ cons\ back))$
  $forwards : Openfile \to Openfile$ where
  $forwards = \lambda(front, back).null\ back \to (front, back)[]\ ((hd\ back)\ cons\ front, (tl\ back))$
  $backwards : Openfile \to Openfile$ where
  $backwards = \lambda(front, back).null\ front \to (front, back)[]\ (tl\ front, (hd\ front)\ cons\ back)$
  $append : Record \times Openfile \to Openfile$ where
  $append = \lambda(r, (front, back)).((front), (back)\ cons\ r)$
  $delete : Openfile \to Openfile$ where
  $delete = \lambda(front, back).(front, (null\ back \to back\ []\ (tl\ back)))$
- *Open file* (Contd.)
  Operations:
  $at\_first\_record : Openfile \to Tr$, $at\_first\_record = \lambda(front, back).null\ front$
  $at\_last\_record : Openfile \to Tr$, $at\_last\_record = \lambda(front, back).null\ back \to true[]\ (null\ (tl\ back)$
  $true\ []\ false)$
  $isempty :\ Openfile \to Tr$, $isempty = \lambda(front, back).(null\ front)\ and\ (null\ back)$
- *Character String*
  Domain: $String$ = the strings formed from the elements of $\mathcal{C}$
  (including an *error* string)
  Operations:
  $A, B, C, ..., Z : String$
  $empty : String$,$error : String$,$concat : String \times String \to String$,$length : String \to Nat$
  $substr : String \times Nat \times Nat \to String$
- *Output terminal log*
  Domain: $l \in Log = String^*$

Valuation Function:

- $\mathbf{P} : Program\_session \to File\_system \to (Log \times File\_system)$
  $\mathbf{P}[[\mathbf{edit}\ I\ \mathbf{cr}\ S]]$
  $= \lambda s.let\ p = copyin(access([[I]], s))\ in$
  $(\text{"}edit\ I\text{"} cons\ fst(\mathbf{S}[[S]]p),$
  $update([[I]], copyout(snd(\mathbf{S}[[S]]p)), s))$
- $\mathbf{S} : Command\_sequence \to Openfile \to (Log \times Openfile)$
  $\mathbf{S}[[C\ \mathbf{cr}\ S]]$
  $= \lambda p.let\ (l', p') = \mathbf{C}[[C]]p\ in$
  $((l' cons\ fst(\mathbf{S}[[S]]p')), snd(\mathbf{S}[[S]]p'))$
  $\mathbf{S}[[\mathbf{quit}]] = \lambda p.(\text{"}quit\text{"}\ cons\ nil, p)$
  - The **S** function collects the log messages into a list, **S**[[**quit**]] builds the very end of this list

- $\mathbf{C} : Command \rightarrow Openfile \rightarrow (String \times Openfile)$
  $\mathbf{C}[[\mathbf{newfile}]] = \lambda p.(\text{"}newfile\text{"}, newfile)$

  $\mathbf{C}[[\mathbf{moveforward}]]$
  $= \lambda p.let\ (k', p') = isempty(p) \rightarrow (\text{"}error : file\ is\ empty\text{"}, p)[]\ (at\_last\_record(p) \rightarrow$
  $(\text{"}error : at\ back\ already\text{"}, p)$
  $[]\ (\text{""}, forwards(p)))in\ (\text{"}moveforward\text{"}\ concat\ k', p'))$
  $\mathbf{C}[[\mathbf{moveback}]]$
  $= \lambda p.let\ (k', p') = isempty(p) \rightarrow (\text{"}error : file\ is\ empty\text{"}, p)[]\ (at\_first\_record(p) \rightarrow$
  $(\text{"}error : at\ front\ already\text{"}, p))$
  $[]\ (\text{""}, backwards(p))in\ (\text{"}moveback\text{"}\ concat\ k', p')$

Using the above abstract syntax, semantic algebra and valuation functions write the valuation functions for :

(a) $\mathbf{C}[[\mathbf{append}]]$ **[3]**

Evaluate the given expression: **[9]**

$\mathbf{S}[[\mathbf{insert}\ r\ \mathbf{cr\ insert}\ s\ \mathbf{moveback\ cr\ delete\ cr\ quit}]](newfile)$

2. Expand the following recursively defined functions. Perform upto the third unfolding, and the i^{th} case(if it can be generalised). An example is given below:

$Q = \lambda(g).\lambda(n).n\ equals\ zero \rightarrow one\ []\ g(n\ plus\ one)$
$graph(Q^0(\phi)) = \{\ \}$
$graph(Q^1(\phi)) = \{(zero,\ one)\}$
$graph(Q^2(\phi)) = \{(zero, one)\}$
$graph(Q^i(\phi)) = \{(zero, one)\}$

(a) $f = \lambda(x).x\ equals\ zero \rightarrow g(zero)\ []\ f(g(x\ minus\ one))\ plus\ two$ **[3]**
$g = \lambda(y).y\ equals\ zero \rightarrow zero\ []\ y\ plus\ f(y\ minus\ one)$

(b) $C = \lambda(f).\lambda(x).x\ lessthan\ two \rightarrow one\ []\ (f(x\ minus\ one)\ plus\ f(x\ minus\ two))$ **[3]**

3. Consider a recursive definition $h : Nat \rightarrow Nat$ as: **[7]**
$h = \lambda n.(n\ mod\ two)\ equals\ zero \rightarrow zero$
$[](n\ mod\ three)\ equals\ zero \rightarrow one$
$[](h(h(n\ minus\ one)\ mult\ h(n\ plus\ two))$

Compute the first 9 finite unfoldings for h.