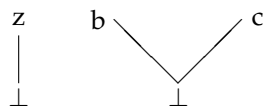
MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

1999 Midterm & Solutions

Problem 1: Short Answer Questions [15 points]

a. Let D and E be the following CPOs:



D E

How many elements are in the set of

(i) [3 points] total functions from D to E?

(ii) [4 points] continuous functions from D to E?

(i) There are 3^2 or 9 total functions from D to E.

(ii) For finite CPOs, the set of continuous functions is equal to the set of monotonic functions. The entire domain D is a chain. This means any monotonic function must map the domain D to a chain in E. There are three chains in E: $\{\perp\}$, $\{a, \perp\}$ and $\{b, \perp\}$. There is one way to monotonically map to the first chain, and there are two ways to monotonically map to the other two. This means there are five monotonic functions from D to E, or five continuous functions from D to E.

b. Evaluate each of the following expressions with the appropriate model.

(i) [2 points] Static Scoping, call by value

```
(let  ((a 1))
  (let ((f (lambda (x y) (/ x a))))
    (let ((a 0))
      (f 4 3))))
```

Solution:
4

(ii) [2 points] Dynamic Scoping, call by value

```
(let  ((a 1))
  (let ((f (lambda (x y) (/ x a))))
    (let ((a 0))
      (f 4 3))))
```

Solution:
Error, Division by 0

(iii) [2 points] Dynamic Scoping, call by name

```
(let  ((a 1))
  (let ((f (lambda (x y) (/ x a))))
    (let ((a 2))
      (f 4 (let ((a 0)) (f 2 2))))))
```

Solution:
2

(iv) [2 points] Static Scoping, call by value

```
(let  ((a 1))
  (let ((f (lambda (x y) (/ x a))))
    (let ((a 2))
      (f 4 (let ((a 0)) (f 2 2))))))
```

Solution:
4

Problem 2: Operational Semantics [35 points]

After completing 6.821, you get a job working for Ben Bitdiddle, whose company sells commercial implementations of PostFix (without `dup`). After several focus-group studies, Ben has decided that PostFix needs a macro facility. Below is Ben's sketch of the informal semantics of the facility.

Macros are specified as part of a PostFix program, as follows:

```
(program ((I_1 V_1) ... (I_n V_n)) Q)
```

I_1 through I_n are identifiers. Each macro (I_i V_i) creates a command, I_i, that pushes the value V_i (which can be an integer or a command sequence) onto the stack. It is illegal to give macros the names of existing PostFix commands, or to use an identifier more than once in a list of macros. The behavior of programs that do so is undefined.

Here are some examples Ben has come up with:

`(program ((inc (1 add))) (0 inc exec inc exec))` $\Rightarrow$ `2`
`(program ((A 1) (B (2 mul))) (A B exec))` $\Rightarrow$ `2`
`(program ((A 1) (B (2 mul))) (A C exec))` $\Rightarrow$ *`error`* (undefined macro C)
`(program ((A 1) (B (C mul)) (C 2)) (A B exec))` $\Rightarrow$ `2`
`(program ((A pop)) (1 A))` ; *`Ill-formed program`* (The body of a macro must be a value, it cannot be a command.)

Ben started writing an SOS for PostFix with macros, but had to go make a presentation for some venture capitalists. It is your job to complete the SOS.

New (or modified) domains:

$$\begin{aligned} I &\in \text{Identifier} \\ M &\in \text{Macro-list} = (\text{Identifier} \times \text{Value})^* \\ C &\in \text{Command}_{\text{PostFix+macros}} = \text{Command}_{\text{PostFix}} + \text{Identifier} \\ P &\in \text{Program} = \text{Commands} \times \text{Macro-list} \\ \text{Definition} &= \text{Value} \\ \text{Value} &= \text{Intlit} + \text{Commands} \end{aligned}$$

Ben's notes describe a helper function, lookup, that you can use to figure out what identifiers have been defined as. The function has the signature:

$lookup$: Identifier $\times$ Macro-list $\rightarrow$ Definition

If lookup is given an identifier and a macro list, it returns a value if the identifier is defined to be that value in the macro list. Otherwise lookup gets stuck.

a. [10 points] Ben's notes begin the SOS for PostFix+macros as follows:

$$\begin{aligned} \mathcal{C} &= \text{Commands} \times \text{Stack} \times \text{Macro-list} \\ \Rightarrow &: \frac{\langle Q,\ S\rangle \Rightarrow_P \langle Q',\ S'\rangle}{\langle Q,\ S,\ M\rangle \Rightarrow \langle Q',\ S',\ M\rangle} \\ &\quad \text{where } \Rightarrow_P \text{ is the original transition relation for PostFix} \end{aligned}$$

Complete the transition relation for PostFix+macros. Your completed SOS should handle the first four of Ben's examples. It should **not** handle the ill-formed program. Model errors in the SOS as stuck states (and do not handle them explicitly).

Solution:

$$\begin{aligned} \Rightarrow &: \\ &\langle I.Q,\ S,\ M\rangle \Rightarrow \langle Q,\ V.S,\ M\rangle \\ &\text{where } V = (lookup\ \ I\ \ M) \end{aligned}$$

b. [10 points] Louis Reasoner finds out that you have completed an SOS for PostFix+macros. He sees that your SOS handles macros that depend on other macros. He wants to launch a new advertising campaign with the slogan: "Guaranteed to terminate: PostFix with mutually recursive macros!" Show that Louis' new campaign is a bad idea by writing a nonterminating program in PostFix+macros (without `dup`).

Solution: A simple nonterminating program is:

```
(program ((I (I exec))) (I exec))
```

c. When Ben returns from his presentation, he finds out you've written a nonterminating program in PostFix+macros. He decides to restrict the language so nonterminating programs are no longer possible. **Ben's restriction is that the body (or value) of a macro cannot use any macros.** Ben wants you to prove that this restricted language terminates.

(i) [10 points] Extend the PostFix energy function so that it assigns an energy to configurations that include macros. Fill in the definitions of the functions $\mathcal{E}_{\text{com}}[\![C, M]\!]$, $\mathcal{E}_{\text{seq}}[\![Q, M]\!]$ and $\mathcal{E}_{\text{stack}}[\![S, M]\!]$ and use these functions to define your extended energy function. *(Hint: If a command pushes a value, V, onto the stack, how does the energy of the configuration change?)*

$$
\begin{aligned}
\mathcal{E}_{\text{com}}[\![C, M]\!] &= 1 \text{ (C is not an identifier or an executable sequence)} \\
\mathcal{E}_{\text{com}}[\![I, M]\!] &= \\
\mathcal{E}_{\text{seq}}[\![[]_{\text{Command}}, M]\!] &= 0 \\
\mathcal{E}_{\text{seq}}[\![C.Q, M]\!] &= \\
\mathcal{E}_{\text{stack}}[\![[]_{\text{Value}}, M]\!] &= 0 \\
\mathcal{E}_{\text{stack}}[\![V.S, M]\!] &=
\end{aligned}
$$

Solution: Completing the energy function we have:

$$
\begin{aligned}
\mathcal{E}_{\text{com}}[\![I, M]\!] &= \mathcal{E}_{\text{com}}[\![V, M]\!] \\
&\quad \text{where } V = (\mathit{lookup}\ \ I\ \ M) \\
&\quad \text{This is all that is necessary because if an undefined I is ever encountered,} \\
&\quad \text{the program will immediately get stuck (and terminate).} \\
\mathcal{E}_{\text{seq}}[\![C.Q, M]\!] &= 1 + \mathcal{E}_{\text{com}}[\![C, M]\!] + \mathcal{E}_{\text{seq}}[\![Q, M]\!] \\
\mathcal{E}_{\text{stack}}[\![V.S, M]\!] &= \mathcal{E}_{\text{com}}[\![V, M]\!] + \mathcal{E}_{\text{stack}}[\![S, M]\!] \\
\mathcal{E}_{\text{config}}[\![\langle Q\ ,\ S\ ,\ M\rangle]\!] &= \mathcal{E}_{\text{seq}}[\![Q, M]\!] + \mathcal{E}_{\text{stack}}[\![S, M]\!]
\end{aligned}
$$

(ii) [5 points] Use the extended energy function (for the restricted form of PostFix+macros) to show that executing a macro decreases the energy of a configuration. Since it is possible to show all the other commands decrease the energy of a configuration (by adapting the termination proof for PostFix without macros), this will show that the restricted form of PostFix+macros terminates.

Solution:

$$
\begin{aligned}
\langle I.Q,\ S,\ M\rangle &\Rightarrow \langle Q,\ V.S,\ M\rangle \\
&\quad \text{where } V = (\mathit{lookup}\ \ I\ \ M) \\
\mathcal{E}_{\text{config}}[\![\langle I.Q\ ,\ S\ ,\ M\rangle]\!] &= \mathcal{E}_{\text{seq}}[\![I.Q, M]\!] + \mathcal{E}_{\text{stack}}[\![S, M]\!]
\end{aligned}
$$

$$
\begin{aligned}
&= 1 + \mathcal{E}_{\mathrm{com}}[\![I, M]\!] + \mathcal{E}_{\mathrm{seq}}[\![Q, M]\!] + \mathcal{E}_{\mathrm{stack}}[\![S, M]\!] \\
&= 1 + \mathcal{E}_{\mathrm{com}}[\![V, M]\!] + \mathcal{E}_{\mathrm{seq}}[\![Q, M]\!] + \mathcal{E}_{\mathrm{stack}}[\![S, M]\!] \\
&= 1 + \mathcal{E}_{\mathrm{seq}}[\![Q, M]\!] + \mathcal{E}_{\mathrm{stack}}[\![V.S, M]\!] \\
&= 1 + \mathcal{E}_{\mathrm{config}}[\![\langle Q\,,\ V.S\,,\ M\rangle]\!]
\end{aligned}
$$

Thus, after executing a macro, the energy of a configuration decreases by one unit, proving Post-Fix with macros terminates.

Problem 3: Control [50 points]

Ben Bitdiddle, whose company also sells FLK!, has decided to improve FLK! with new facilities for exception handling. Ben's idea is to allow users to store exceptions in variables. Other exception facilities are also included, in the form of `handle` and `exception`. Ben also adds `let-x` to allow either an exception or a value to be bound to an identifier:

```
(exception E)  ; creates an exception with value E
(handle E1 E2) ; handler procedure E1 handles exceptions
               ; from E2. If E2 does not raise an exception,
               ; E2 is the value of the HANDLE. If E2 raises
               ; an exception, the value of the HANDLE is the
               ; result of calling the handler procedure E1 on
               ; the value of the exception.
(let-x I E1 E2) ; binds an exception or value from E1
               ; to I in E2.
```

For example:

```
(handle (lambda (x) (+ 1 x))
   (let-x a (exception 4)
        (let ((b 2))
             (+ a b))))
⇒ 5
```

Any attempt to reference a variable that is bound to an exception will result in the immediate raising of the exception. Thus, the above example will evaluate to 5 because the reference to `a` in `(+ a b)` will cause an exception to be raised.

Ben had begun to develop a new semantics for FLK!, but was called away to work for a new Internet startup that was about to have an IPO event. You find on Ben's desk the following sheet of paper:

$$\mathcal{E} : \text{Exp} \to \text{Environment} \to \text{Computation}$$

$$\begin{aligned}
\text{Computation} &= \text{ExceptionCont} \to \text{ExpCont} \to \text{CmdCont} \\
\text{CmdCont} &= \text{Store} \to \text{Answer} \\
k \in \text{ExpCont} &= \text{Value} \to \text{CmdCont} \\
w \in \text{ExceptionCont} &= \text{ExpCont}
\end{aligned}$$

$$\begin{aligned}
\text{Procedure} &= \text{Value} \to \text{Computation} \\
\delta \in \text{Denotable} &= \text{Value} + \text{Exception} \\
\text{Exception} &= \text{Value}
\end{aligned}$$

$\mathcal{E}[\![\texttt{L}]\!] = \lambda ewk\,.\ (k\ \mathcal{L}[\![\texttt{L}]\!])$
$\mathcal{E}[\![\texttt{(handle E1 E2)}]\!] = \lambda ewk\,.\ (\mathcal{E}[\![\texttt{E1}]\!]\ e\ w\ (\textit{test-procedure}\ \lambda p\,.\,(\mathcal{E}[\![\texttt{E2}]\!]\ e\ (\lambda v\,.\,(p\ v\ w\ k))\ k)))$
$\mathcal{E}[\![\texttt{(proc I E)}]\!] = \lambda ewk\,.\ (k\ (\text{Procedure} \mapsto \text{Value}\ (\lambda vw'k'\,.\,(\mathcal{E}[\![\texttt{E}]\!]\ [\texttt{I} : (\text{Value} \mapsto \text{Denotable}\ v)]e\ w'\ k'))))$
$\mathcal{E}[\![\texttt{(call E1 E2)}]\!] = \lambda ewk\,.\ \mathcal{E}[\![\texttt{E1}]\!]\ e\ w\ (\textit{test-procedure}\ \lambda p\,.\,\mathcal{E}[\![\texttt{E2}]\!]e\ w\ (\lambda v\,.\,(p\ v\ w\ k)))$

a. [10 points] Ben wants to add a new kernel form `(exception? I)` that tests whether or not an identifier `I` is bound to an exception and returns a boolean. Alyssa insists that the form can be desugared into the other exception-handling forms, but does not have time to provide the desugaring. Prove Alyssa correct by writing a desugaring for `(exception? I)`.

$\mathcal{D}[\![\texttt{(exception? I)}]\!] =$

Solution:

$\mathcal{D}[\![\texttt{(exception? I)}]\!] =$ `(handle (lambda (x) #t) (begin I #f))`

b. [10 points] Write the meaning function clause for `(exception E)`.

Solution:

$\mathcal{E}[\![\texttt{(exception E)}]\!] = \lambda ewk\,.\ (\mathcal{E}[\![\texttt{E}]\!]\ e\ w\ w)$

c. [10 points] Write the meaning function clause for `I`.

Solution:

$\mathcal{E}[\![\texttt{I}]\!] = \lambda ewk\,.$ **matching** $(lookup\ e\ I)$
$\triangleright$ (Denotable$\mapsto$Binding d) $[\![$ **matching** d
$\triangleright$ (Value$\mapsto$Denotable v) $[\![\ (k\ v)$
$\triangleright$ (Exception$\mapsto$Denotable v) $[\![\ (w\ v)$
endmatching
$\triangleright\ else\ [\![\ \lambda s\,.$ (Error$\mapsto$Answer $error$)
endmatching

d. [10 points] Write the meaning function clause for `(let-x I E1 E2)`.

Solution:

$\mathcal{E}[\![\texttt{(let-x I E1 E2)}]\!] =$
$\lambda ewk\,.\ (\ \mathcal{E}[\![\texttt{E1}]\!]\ \texttt{e}\ (\lambda v\,.\ (\mathcal{E}[\![\texttt{E2}]\!]\ [I : (\text{Exception}\mapsto\text{Denotable}\ v)]e\ w\ k))$
$(\lambda v\,.\ (\mathcal{E}[\![\texttt{E2}]\!]\ [I : (\text{Value}\mapsto\text{Denotable}\ v)]e\ w\ k)))$

While scrounging through Ben's notes, you noticed Bolt Cola spilled all over a very important document. After sponging off most of the goop, you notice it's a memo from Alyssa P. Hacker about a brand new command called `rec-handle`. It reads the following:

```
Dear Ben,
I figured out how you can get ahead of the competition.  All you need is a
recursive handler.  Something like:


        (rec-handle E1 E2)  ;  handler procedure E1 handles exceptions
                            ;  from E2. If E2 does not raise an exception,
                            ;  E2 is the value of REC-HANDLE. In addition, any
                            ;  exceptions raised during execution of E1
                            ;  are HANDLED by E1.

Here's an example:

(rec-handle (lambda (x)
    (if (= x 0) 1
        (exception (- x 1))))
    (exception 5))


⇒1
```

`Here, I whipped up the meaning function for you.`

$\mathcal{E}[\![$`(handle E1 E2)`$]\!] = \lambda e w_1 k$.
$(\mathcal{E}[\![$`E1`$]\!]\ e\ w_1\ (test\text{-}procedure\ \lambda p$. `let` w_2 `be` $BOLTCOLA$ `in` $(\mathcal{E}[\![$`E2`$]\!]\ e\ (\lambda v\ .\ (p\ v\ w_2\ k)\ k))))$

Unfortunately, the critical part of the function was too smeared with Cola. It's your job to help Ben by fixing Alyssa's memo.

e. [10 points] Fill in the BOLTCOLA splotch for `rec-handle`.

Solution: $\mathbf{fix}_{ExceptionCont}\ \lambda w_3\ .\ \lambda v\ .\ (p\ v\ w_3\ k)$