MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

2000 Midterm

Problem 1: Short Answer [20 points]

Evaluate the following expressions in the given models.

```
(let ((a 1)
      (g (lambda (x) 25)))
  (let ((f (lambda (y) (if (< a y)
                           (g (/ 2 0))
                           ((lambda (x) 15) (g g))))))
     (let ((a 4)
           (y 5)
           (g (lambda (x) (x x))))
       (f 2)))))
```

a. [5 points] static scoping, call by value

Solution: error (divide by 0)

b. [5 points] dynamic scoping, call by value

Solution: $\perp$ (infinite loop)

c. [5 points] static scoping, call by name

Solution: 25

d. [5 points] dynamic scoping, call by name

Solution: 15

Problem 2: Operational Semantics: Postfix + {sdup} [20 points]

Alyssa P. Hacker extended the PostFix language with a new command called `sdup`: smart `dup`. This allows us to compute $square(x) = x^2$ without hurting the termination property of PostFix programs. The informal semantics for `sdup` is as follows: duplicate the top of the stack if it is a number or a command sequence that doesn't contain `sdup`; otherwise, report an error.

Formally, the operational semantics has been extended with the following two transition rules:

$$\langle \texttt{sdup} \,.\, Q_{rest},\ \ N \,.\, S\rangle \Rightarrow \langle Q_{rest},\ \ N \,.\, N \,.\, S\rangle \qquad [\textit{sdup-numeral}]$$

$$\langle \texttt{sdup} \,.\, Q_{rest},\ \ Q \,.\, S\rangle \Rightarrow \langle Q_{rest},\ \ Q \,.\, Q \,.\, S\rangle \quad \text{where } \neg(contains\_sdup\ Q) \qquad [\textit{sdup-sequence}]$$

$contains\_sdup : Command^* \rightarrow$ Bool is a helper function that takes a sequence of commands and checks whether it contains `sdup` or not (yes, $contains\_sdup$ handles even nested sequences of commands)

As a new graduate student in Alyssa's AHRG (Advanced Hacking Research Group), you were assigned to give a proof that all $\text{PostFix} + \{\texttt{sdup}\}$ programs terminate. However, you are not alone! Alyssa already took care of most of the mathematical weaponry:

Consider the product domain $P = Nat \times Nat$ (as usual, Nat is the set of natural numbers, starting with 0). On this domain, we define the relation $<_P$ as follows:

Definition 1 (lexicographic order) $\langle a_1, b_1\rangle <_P \langle a_2, b_2\rangle$ *iff:*

a. $a_1 < a_2$ *or*

b. $a_1 = a_2$ *and* $b_1 < b_2$.

E.g. $\langle 3, 10000\rangle <_P \langle 4, 0\rangle$, $\langle 5, 2\rangle <_P \langle 5, 3\rangle$.

Definition 2 *A* strictly decreasing chain *in P is a finite or infinite sequence of elements* $p_1, p_2, \ldots$ *such that* $p_i \in P, \forall i$ *and* $p_{i+1} <_P p_i, \forall i$.

After a long struggle, Alyssa proved the following lemma for you:

Lemma 1 *There is no infinite strictly decreasing chain in* P.

Give a rigorous proof that each $\text{PostFix} + \{\texttt{sdup}\}$ program terminates by using a cleverly defined energy function $\mathcal{ES}_{config}$. *Hint:* Each transition of Postfix reduces the energy function $\mathcal{E}_{\text{config}}$ you saw in class. Try to see what is reduced by the two new rules, and how you can combine these two things into a single energy function.

Note: If you need to use some helper functions that are intuitively easy to describe but tedious to define (e.g. $contains\_sdup$), just give an informal description of them.

Grading scheme:

- [10 points] $\mathcal{ES}_{config}$;
- [10 points] Termination proof.

Solution:

Consider the following energy function:

$$\mathcal{ES}_{config} : \mathcal{C} \rightarrow Nat \times Nat = \lambda\langle Q, S\rangle \,.\, \langle sdup\_count[\![\langle Q, S\rangle]\!], \mathcal{E}_{\text{config}}[\![\langle Q, S\rangle]\!]\rangle$$

where $sdup\_count$ is a helper function that computes the number of times `sdup` appears in a configuration and $\mathcal{E}_{\text{config}}$ is the energy function shown in class.

Let's first prove that for any transition $c_{old} \Rightarrow c_{new}$, $\mathcal{ES}_{config}[\![c_{new}]\!] <_P \mathcal{ES}_{config}[\![c_{old}]\!]$.

Old transitions: None of them introduces new `sdup` commands but they all strictly decrease $\mathcal{E}_{\text{config}}$. So, the first component of $\mathcal{ES}_{config}$ doesn't increase and the second one strictly decreases which implies $\mathcal{ES}_{config}[\![c_{new}]\!] <_P \mathcal{ES}_{config}[\![c_{old}]\!]$.
New transitions: Each of the new `sdup` related rules "consumes" exactly one `sdup`: this is clearly true for [*dup-numeral*] and [*dup-sequence*] doesn't duplicate sequences containing `sdup`. So the first component of $\mathcal{ES}_{config}$ is strictly decreased by these transitions which implies that no matter what happens with the second component (note that [*dup-sequence*] might actually increase it), $\mathcal{ES}_{config}[\![c_{new}]\!] <_P \mathcal{ES}_{config}[\![c_{old}]\!]$ for the new transitions too.

Suppose now for the sake of contradiction that there is some $\mathrm{PostFix} + \{\texttt{sdup}\}$ program with an infinite execution $c_1 \Rightarrow c_2 \Rightarrow c_3 \Rightarrow \ldots$. This implies $\mathcal{ES}_{config}[\![c_2]\!] <_P \mathcal{ES}_{config}[\![c_1]\!], \mathcal{ES}_{config}[\![c_3]\!] <_P \mathcal{ES}_{config}[\![c_2]\!], \ldots$ and we've just constructed an infinite strictly decreasing chain in P! Contradiction with Lemma 1.

Problem 3: State: FLK! + {undo-once!} [30 points]

Ben Bitdiddle introduced a new `undo-once!` instruction to roll the store back one operation at a time. Informally speaking, `undo-once!` undoes the last store operation (`cell` or `cell-set!`). If there is no store operation to undo, `undo-once!` does nothing.

E	::=	...	[Classic FLK! expressions]
	|	`(undo-once!)`	[Undo last store operation]

Initially, Ben thought of modifying the meaning function to use a stack of stores (as it did in the fall-98 midterm), but the implementors refused to work on such an idea and threatened to quite Ben's company *en masse*. So, Ben had to turn to a more efficient idea: maintain the current store and a stack of undo functions. An undo function takes a store and reverses a specific store operation (one done with `cell` or `cell-set!`) to obtain the store before the operation.

Pursuing this idea, Ben modified the Cmdcont semantic domain and the top level function as follows:

$$\text{Cmdcont} = \text{Store} \rightarrow \text{StoreTransformStack} \rightarrow \text{Expressible}$$
$$h \in \text{StoreTransformStack} = \text{StoreTransform}^*$$
$$t \in \text{StoreTransform} = \text{Store} \rightarrow \text{Store}$$
$$\mathcal{TL}[\![E]\!] = (\ \mathcal{E}[\![E]\!]\ \textit{empty-env}\ \ \textit{top-level-cont}\ \ \textit{empty-store}\ \ [\,]_{\text{StoreTransform}}\ )$$

As each store operation (`cell` or `cell-set!`) consists of assigning a Storable to a Location, it can be reversed by putting the old Assignment into that Location. Ben even wrote the following undo function producer for you:

$$\textit{make-undofun} : \text{Location} \rightarrow \text{Assignment} \rightarrow \text{StoreTransform}$$
$$= \lambda l \alpha\ .\ \lambda s\ .\ (\textit{assign}'\ \ l\ \ \alpha\ \ s)$$

assign′ is a function similar to *assign* which allows us to assign even *unassigned*:

$$\textit{assign}' : \text{Location} \rightarrow \text{Assignment} \rightarrow \text{Store} \rightarrow \text{Store}$$
$$= \lambda l_1 \alpha s\ .\ \lambda l_2\ .\ \textbf{if}\ (\textit{same-location?}\ \ l_1\ \ l_2)\ \ \textbf{then}\ \alpha\ \ \textbf{else}\ (\textit{fetch}\ \ l_2\ \ s)\ \ \textbf{fi}$$

If a store operation modified location l, the undo function for it can be obtained by calling *make-undofun* on l and the old assignment for l. **All the undo functions that you write in this problem must be obtained by calling *make-undofun* with the appropriate arguments.**

Now, guess what?, Ben went away to deliver a better Internet and grab some more billions, and you were assigned to finish his job.

a. [10 points] Write the meaning function clause for $\mathcal{E}[\![$`(undo-once!)`$]\!]$.

Solution:

$$\mathcal{E}[\![\texttt{(undo-once!)}]\!] =$$
$$\lambda eksh\ .\ \textbf{matching}\ h$$
$$\triangleright\ t\ .\ h_{\text{rest}}\ [\!]\ \ (k\ \ (\text{Unit} \mapsto \text{Value}\ \ \textit{unit})\ \ (t\ \ s)\ \ h_{\text{rest}})$$
$$\triangleright\ [\,]_{\text{StoreTransform}}\ [\!]\ \ (k\ \ (\text{Unit} \mapsto \text{Value}\ \ \textit{unit})\ \ s\ \ h)$$
$$\textbf{endmatching}$$

We specially treat the case of an enpty stack of undo functions: when there is nothing to undo, `undo-once!` does nothing.

b. [10 points] Write a revised version for $\mathcal{E}[\![$`(primop cell-set!` $E_1\ \ E_2$`)`$]\!]$.

Solution:

$$\mathcal{E}[\![\texttt{(primop cell-set! } E_1\ E_2\texttt{)}]\!] = $$
$$\lambda ek\,.\ (\mathcal{E}[\![E_1]\!]\ \ e\ (\textit{test-location}\ \ (\lambda l\,.\ (\mathcal{E}[\![E_2]\!]\ \ e\ (\lambda vsh\,.\ (k\ \ (\text{Unit} \mapsto \text{Value}\ \ \textit{unit})\ (\textit{assign}\ \ l\ \ v\ \ s\ )\ (\textit{make-undofun}\ \ l\ \ (\textit{fetch}\ \ l\ \ s)\ ) \,.\, h\ )))))))$$

The store that is passed to k is, as previously, the store obtained by assigning v to location l; we add to the head of the stack of store transformers an undo function that restores the old assignment for l.

c. [10 points] Write a revised version for $\mathcal{E}[\![\texttt{(cell } E\texttt{)}]\!]$. *Note:* we want to be able to undo even cell creation operations. That is, the following program must end with an error:

```
(let ((c (cell 0)))
   (begin
      (undo-once!)
      (primop cell-ref c)))
```

Solution:

$$\mathcal{E}[\![\texttt{(cell } E\texttt{)}]\!] = $$
$$\lambda ek\,.\ (\mathcal{E}[\![E]\!]\ \ e\ (\lambda vsh\,.\ ((\lambda l\,.\ (k\ \ (\text{Location} \mapsto \text{Value}\ \ l)\ (\textit{assign}\ \ l\ \ v\ \ s)\ (\textit{make-undofun}\ \ l\ \ (\text{Unassigned} \mapsto \text{Assignment}\ \ \textit{unassigned}))\,.\,h\ ))\ (\textit{fresh-loc}\ \ s)\ ))))$$

Undoing a cell allocation is done by assigning back *unassigned* to the cell location l. Now, that cell is free to be allocated again! Calling $(\lambda l\,.\ \ldots)$ on $(\textit{fresh-loc}\ \ s)$ is just a trick to avoid us writing $(\textit{fresh-loc}\ \ s)$ three times (it's like the desugaring for `let` in FL).

Problem 4: Denotational Semantics: Control [30 points]

Sam Antics of eFLK.com wants to cash in on the election year media bonanza by introducing a new feature into standard FLK!:

```
(elect E_pres E_vp)  ; evaluates to E_pres unless impeach
                     ; is evaluated within E_pres, in which
                     ; case evaluates to E_vp. If impeach is
                     ; evaluated within E_vp, signals an error.

(reelect)            ; if evaluated within E_pres of (elect E_pres E_vp),
                     ; goes back to the beginning of elect.
                     ; otherwise, signals an error.

(impeach)            ; if evaluated within E_pres of (elect E_pres E_vp),
                     ; causes the expression to evaluate to E_vp.
                     ; otherwise, signals an error.
```

For example:

```
(let ((scandals (primop cell 0)))
   (elect (if (< (primop cell-ref scandals) 5)
              (begin (primop cell-set! (+ (primop cell-ref scandals) 1))
                     (reelect))
              (impeach))
          (* (primop cell-ref scandals) 2)))
⇒ 10
```

You are hired by eFLK.com to modify the standard denotational semantics of FLK! to produce *FLK! 2000 Presidential Edition (TM)*. To get you started, Sam tells you that he has added the following domains:

$$r \in \text{Prescont} = \text{Cmdcont}$$
$$i \in \text{Vpcont} = \text{Cmdcont}$$

He also changed the signature of the meaning function:

$$\mathcal{E} : \text{Exp} \to \text{Environment} \to \text{Prescont} \to \text{Vpcont} \to \text{Expcont} \to \text{Cmdcont}$$

a. [9 points] give the meaning function for `(elect E_pres E_vp)`.

Solution:

$$\mathcal{E}[\![(\texttt{elect } \texttt{E}_{\texttt{pres}} \ \texttt{E}_{\texttt{vp}})]\!] =$$
$$\lambda erik \, . \, (\mathbf{fix}_{\text{Cmdcont}} \, (\lambda r_1 \, . \, \mathcal{E}[\![E_{pres}]\!] \, e \, r_1 \, (\lambda s \, . \, \mathcal{E}[\![E_{vp}]\!] \, \text{e} \, (\text{error-cont cannot-reelect-vp}) \, (\text{error-cont cannot-impeach-vp}) \, k) \, k))$$

b. [7 points] give the meaning function for `(reelect)`.

Solution:

$$\mathcal{E}[\![(\texttt{reelect})]\!] = \lambda erik \, . \, r$$

c. [7 points] give the meaning function for (`impeach`).

Solution:

$\mathcal{E}[\![(\texttt{impeach})]\!] = \lambda erik\ .\ i$

d. [7 points] using the meaning functions you defined, show that (`elect (reelect) 1`) is equivalent to $\perp$.

Solution:

$$\mathcal{E}[\![(\texttt{elect (reelect) 1})]\!] =$$
$$\lambda erik\ .\ (\textbf{fix}_{\text{Cmdcont}}\ (\lambda r_1\ .\ \mathcal{E}[\![(reelect)]\!]\ e\ r_1\ (\lambda s\ .\ \mathcal{E}[\![1]\!]\ \text{e}$$
$$\text{(error-cont cannot-reelect-vp)}$$
$$\text{(error-cont cannot-impeach-vp)}\ k)\ k))$$

$\Rightarrow$

$$\mathcal{E}[\![(\texttt{elect (reelect) 1})]\!] =$$
$$\lambda erik\ .\ (\textbf{fix}_{\text{Cmdcont}}\ (\lambda r_1\ .\ (\lambda erik\ .\ r)\ e\ r_1\ (\lambda s\ .\ \mathcal{E}[\![1]\!]\ \text{e}$$
$$\text{(error-cont cannot-reelect-vp)}$$
$$\text{(error-cont cannot-impeach-vp)}\ k)\ k))$$

$\Rightarrow$

$$\mathcal{E}[\![(\texttt{elect (reelect) 1})]\!] = \lambda erik\ .\ (\textbf{fix}_{\text{Cmdcont}}\ (\lambda r_1\ .\ r_1))$$

$\Rightarrow$

$$\mathcal{E}[\![(\texttt{elect (reelect) 1})]\!] = \perp$$