6.821 Programming Languages
Fall 2002

Handout

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

1999 Final Examination and Solutions

1. Final Examination

There are *five* problems on this examination. ***Make sure you don't skip over any of a problem's parts!*** They are followed by an appendix that contains reference material from the course notes. The appendix contains no problems; it is just a handy reference.

You will have three hours in which to work the problems. Some problems are easier than others: read all problems before beginning to work, and use your time wisely!

This examination is open-book: you may use whatever reference books or papers you have brought to the exam. The number of points awarded for each problem is placed in brackets next to the problem number. There are 100 points total on the exam.

Do all written work in your examination booklet – we will not collect the examination handout itself; you will only be graded for what appears in your examination booklet. It will be to your advantage to show your work – we will award partial credit for incorrect solutions that make use of the right techniques.

If you feel rushed, be sure to write a brief statement indicating the key idea you expect to use in your solutions. We understand time pressure, but we can't read your mind.

This examination has text printed on only one side of each page. Rather than flipping back and forth between pages, you may find it helpful to rip pages out of the exam so that you can look at more than one page at the same time.

Contents

The figures in the Appendix are very similar to the ones in the course notes. Some bugs have been fixed, and some figures have been simplified to remove parts inessential for this exam. You will not be marked down if you use the corresponding figures in the course notes instead of the appendices.

Problem 1: Short Answer [18 points]

a. Explicit Polymorphism [6 points]

   Rewrite the following programs in Scheme/XSP with the most flexible type possible.

   (i) [3 points]

```
(lambda (f g x)
 (if (f x)
     (g x)
     (g (g x))))
```

   (ii) [3 points]

```
(lambda (f)
  (if (f #t)
      (f (f 1))
      (f 2)))
```

b. Type Reconstruction [12 points]

   Determine the reconstructed types of the following Scheme/R expressions. If none exists, explain why.

   (i) [3 points]

```
(lambda (f x)
   (if (f #t)
       (f x)
       2))
```

   (ii) [3 points]

```
(let ((f (lambda (g x) (g (g x)))))
  (if (f not #t)
      (f (lambda (x) (+ x 1)) 1)
      2))
```

   (iii) [3 points]

```
(let ((f (lambda (x) x)))
  (cons (f 1)
        (cons (f #t)
              (null))))
```

   (iv) [3 points]

```
(lambda (g) (g g))
```

Problem 2: Operational Semantics [22 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON EXTENDING THE POSTFIX GRAMMAR IN APPENDIX A AND THE POSTFIX OPERATIONAL SEMANTICS GIVEN IN APPENDIX B.

Intel has released the new 747 chip, a chip so big it can handle mind boggling amounts of number crunching. The good programmers over at Microsquish are excited because this means there is now a computer fast enough to run their dream programming language, TwoStack PostFix. In TwoStack PostFix, a program is a tuple of two command sequences, one per stack. For instance, to have one stack add the numbers 1 and 3 and have the other stack multiply 4 * 5, use the program

```
<[1 3 add],[4 5 mul]>
```

The meaning of a TwoStack program is also a tuple, reporting the top of each stack at the end of the program. In the previous case, `<4,20>`.

Keep in mind that the stacks are represented as completely separate entities at different locations in memory. Note that we will model errors as stuck states. For example, the program

```
<[5 1 mul],[4 0 div]>
```

should get stuck. It is also a stuck state if one stack runs out of commands before the other. For instance,

```
<[5 1 mul 3 add],[4 1 div]>
```

should get stuck right after the transition which performs the div command. Note that matching commands are executed simultaneously- that is, the `5` and `4` are pushed at the same time and the `mul` and the `div` are executed at the same time.

Finally, Executives at Microsquish would like to implement a `talk` command allowing the two stacks to communicate. For now, do not worry about the transition rule for the `talk` command, but know that it requires the following domain updates:

$$\begin{array}{rcl}
C & \in & \text{Command}_{\text{PostFix+Talk}} \\
C & ::= & \text{... existing PostFix commands ...} \\
 & | & \texttt{talk} \\
Q & \in & \text{Commands}_{\text{PostFix+Talk}} = \text{Command}_{\text{PostFix+Talk}}{}^{*} \\
V & \in & \text{Value} = \text{IntLit} + \text{Commands}_{\text{PostFix+Talk}}
\end{array}$$

In addition, the transition relation for PostFix+Talk is the same as the relation for PostFix except it is updated to work with the new domains. It currently results in a stuck state for the `talk` command.

The important domains for TwoStack are as follows:

$$\begin{array}{rcl}
P & \in & \text{Program}_{\text{TwoStack}} = \text{Commands}_{\text{TwoStack}} \\
 & & \text{Commands}_{\text{TwoStack}} = \text{Commands}_{\text{PostFix+Talk}} \times \text{Commands}_{\text{PostFix+Talk}} \\
A & \in & \text{Answer}_{\text{TwoStack}} = \text{Answer} \times \text{Answer}
\end{array}$$

a. [12 points] Louis Reasoner is given the job of defining the Operational Semantics of PostFix+Talk. He decides on the following configuration:

$$\mathcal{C}_{\text{TwoStack}} \quad = \quad \text{Commands}_{\text{TwoStack}} \times \mathsf{Stack} \times \mathsf{Stack}$$

However, he needs your help to define the rest of the 5-tuple. Define the Input, Ouput and Transition functions in terms of the PostFix+Talk functions. Use subscripting to express which domains the functions are over. For instance, $\Rightarrow_{PostFix+Talk}$ is the transition function for PostFix+Talk and $\Rightarrow_{TwoStack}$ is the transition function for TwoStack PostFix.

Be sure to also define the set of final configurations of TwoStack Postfix, but do not worry about reporting errors for stuck states.

It's not very exciting having two stacks run in parallel unless they can communicate. So, Louis Reasoner decides to define the `talk` command. If `talk` is at the front of both command sequences, the top value on each stack is copied to the top of the other stack. For instance,

```
<[5 1 mul talk add],[4 1 div talk mul]>
```

should return `<9,20>`

b. [6 points] Extend the transition function for TwoStack to include the `talk` command.

c. [4 points] Mark An-treason (who is also working at Microsquish since his company was bought out) is worried that programs with `talk` may not terminate. If programs in TwoStack PostFix always terminate, set Mark's fears at rest by explaining how you would modify the energy proof to show this. If not, give an example program which does not terminate.

Problem 3: Denotational Semantics [24 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON MODIFYING THE FLK! STANDARD SEMANTICS GIVEN IN APPENDIX C.

Ben Bitdiddle has been called in to assist in the development of a new feature for FL!. This feature will allow procedures to call themselves recursively without having to use **rec** or **letrec**. Ben proposes adding a new form **(self** E**)** to FLK!. The form **(self** E**)** calls the containing procedure with an actual parameter that is the result of evaluating E.

The FLK! expression grammar is changed by the addition of the form **(self** E**)**:

$$\begin{array}{lll} E & ::= & \texttt{... existing FLK! forms ..} \\ & | & \texttt{(self } E\texttt{)} \end{array}$$

Here is an example of the use of **(self** E**)** written in FL! (which would be desugared into FLK! before execution):

```
(let ((fact (lambda (n) (if (= n 0) 1 (* n (self (- n 1))))))
     (fact 4))
```

⇒24

Ben further specifies that when **(self** E**)** is used outside of a procedure it causes the program to terminate immediately with a value that is the result of evaluating E.

Ben begins describing the denotational semantics of the **self** form by modifying the signature of the meaning function, $\mathcal{E}$. His new signature is:

$$\begin{array}{rcl} \mathcal{E} & : & \text{Exp} \rightarrow \text{Environment} \rightarrow \text{SelfProc} \rightarrow \text{ExpCont} \rightarrow \text{CmdCont} \\ \text{SelfProc} & = & \text{Procedure} \end{array}$$

Ben asks you to complete the denotational description of the **self** form because he is going to a conference to give a speech on why FL! is the language of the future.

a. [4 points] Give the revised meaning function $\mathcal{TL}[\![E]\!]$.

b. [4 points] What is the revised $\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!]$?

c. [4 points] What is $\mathcal{E}[\![\texttt{(self } E\texttt{)}]\!]$?

d. [4 points] What is the revised $\mathcal{E}[\![\texttt{(proc } I\ E\texttt{)}]\!]$?

e. [4 points] Prove that $\mathcal{TL}[\![\texttt{(self (self 1))}]\!]$ in your semantics means $(\text{Value}\mapsto\text{Expressible}\ (\text{Int}\mapsto\text{Value}\ 1))$.

f. [4 points] Use your semantics to show **(proc x (self 1))** evaluates to a procedure that, no matter what input it is called with, loops forever.

Problem 4: Type Reconstruction [20 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON EXTENDING THE TYPING RULES GIVEN IN APPENDIX D AND THE TYPE RECONSTRUCTION ALGORITHM GIVEN IN APPENDIX E.

Alyssa P. Hacker has been asked to extend the type system for SCHEME/R to handle **(label** *I E***)** and **(jump** E_1 E_2**)**. As introduced in class and in the book, **(label** *I E***)** establishes a control point called *I* in *E*, and **(jump** E_1 E_2**)** jumps to the control point that is the value of E_1 and gives it the value obtained by evaluating E_2.

For example:

```
(label out
   (if (= x 0)
       (jump out 0)
       (/ y x)))
```

Alyssa has added a control point type to the type grammar for SCHEME/R as follows:

```
    T  ::=  ... existing SCHEME/R types ...
         |  (control-point T)
```

In the example above, the control point **out** would have type (control-point int). It is possible to have control point type errors in SCHEME/R. For example, a `label` expression must have the same type regardless of whether a jump is encountered, making the following expression not well-typed:

```
(label out
   (if (= x 0)
       (jump out #f)
       (/ y x)))
```

Your job is to complete the implementation of typed control points.

a. [5 points] Give the new typing rules in SCHEME/R for **(label** *I E***)** and **(jump** E_1 E_2**)**.

b. [5 points] Give the type reconstruction algorithm clause for **(label** *I E***)**.

c. [5 points] Give the type reconstruction algorithm clause for **(jump** E_1 E_2**)**.

d. [5 points] Give the reconstructed type of the following expression, or give the input parameters to the procedure that fails during the reconstruction:

```
(let ((x (label y y)))
     (jump x (label z z)))
```

Problem 5: Compilers [16 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON META CPS CONVERSION ALGORITHM GIVEN IN APPENDIX F.

Ben Bitdiddle is a consummate compiler hacker, and has been asked by JavaHard to analyze their new Scheme compiler. JavaHard has recently realized that Java will not win in the marketplace against Scheme, and has adopted the 6.821 Scheme compiler as the cornerstone of their crash development effort.
The following code was produced by the compiler after the `desugar`, `globals/wrap`, `cps-convert`, `closures/flat-all`, and `lift-convert` phases of compilation.

```
(program
 (define
  .lambda24.
  (lambda
   (.closure11. .t5.)
   (call-closure .t5. 1 (primop closure-ref .closure11. 1))))
 (define
  .lambda21.
  (lambda
   (.closure14. f g .k1.)
   (call-closure .k1. (primop closure .lambda22. g f))))
 (define
  .lambda22.
  (lambda
   (.closure13. x .k2.)
   (call-closure
    (primop closure-ref .closure13. 1)
    x
    (primop closure .lambda23. (primop closure-ref .closure13. 2) .k2.))))
 (define
  .lambda23.
  (lambda
   (.closure12. .t3.)
   (call-closure
    (primop closure-ref .closure12. 1)
    .t3.
    (primop closure-ref .closure12. 2))))
 (define .lambda20. (lambda (.closure17. x) x))
 (define
  .lambda19.
  (lambda
   (.closure16. a .k9.)
   (let ((.t10. (primop not a))) (call-closure .k9. .t10.))))
 (define
  .lambda18.
  (lambda
   (.closure15. a .k7.)
   (let ((.t8. (primop integer? a))) (call-closure .k7. .t8.))))
 (define *top* (primop closure .lambda20.))
 (define not (primop closure .lambda19.))
 (define integer? (primop closure .lambda18.))
 (let
  ((compose (primop closure .lambda21.)))
  (call-closure compose not integer? (primop closure .lambda24. *top*))))
```

a. [6 points] What source code resulted in the incompletely compiled code above?

JavaHard soon discovers another problem with the compiler. The compiler they are using cannot handle the **begin** form because the `cps-convert` phase does not include a rule for translating **begin**.

b. [4 points] Ben looked in the 6.821 book and could not find the meta-cps rule for begin. What is $\mathcal{MCPS}[\![$`(begin` $E1$ $E2$`)`$]\!]$?

JavaHard has decided to not let programmers access control features such as cwcc, label, and jump to simplify the analysis of programs. Ben suggests to JavaHard that they include a region identifier in every procedure type:

$$T \quad ::= \quad ...|(\rightarrow \quad (T^*) \; T \; R)$$

Just as regions are assigned to cells, every procedure will be assigned a new region identifier except when two procedures are aliased together.

A procedure in E can be explicitly freed if all of the following conditions are met: (1) the procedure is in region R, (2) region R is not in the type of any free variables of E, and (3) region R is not in the type of E.

c. [4 points] Excited about explicit freedom, Ben invents a new primitive called **(%procedure-free x)** that frees the procedure represented by value x. In the following example, the lambda expression **(lambda (y) y)** is bound to x and freed:

```
(let ((x (lambda (y) y)))
   (%procedure-free x))
```

Let E be an expression that contains a non-nested lambda expression P. From effect analysis, we know that P's value is no longer needed after E completes. Thus, Ben would like to use **(%procedure-free x)** to free the procedure value corresponding to P. Help Ben by writing a translation function for E that will free the value of P and return the value of E. Use [v/P]E to substitute the variable v for lambda expression P in E. Assume that your translation occurs before MCPS conversion and that the variable v does not appear in E.

$\mathcal{T}[\![E]\!]$=`(let ((v P))`

```
    ...fill in text here...
       )
```

d. [2 points] Ben revises his compiler to call this primitive to free all closures using your translation rule (assuming it works), and notes that certain programs slow down as a consequence. Ben had thought that reducing the work that the garbage collector had to do would make programs run faster. What is a possible explanation for this behavior?

Appendix A: PostFix Grammar

$P \in$ Program
$Q \in$ Commands
$C \in$ Command
$A \in$ Arithmetic-operator $=$ {`add`, `sub`, `mul`, `div`}
$R \in$ Relational-operator $=$ {`lt`, `eq`, `gt`}
$N \in$ Intlit $=$ {`...`, `-2`, `-1`, `0`, `1`, `2`, `...`}

P	::=	`(`Q`)`	[Program]
Q	::=	C^*	[Command-sequence]
C	::=	N	[Integer-literal]
	|	`pop`	[Pop]
	|	`swap`	[Swap]
	|	A	[Arithmetic-op]
	|	R	[Relational-op]
	|	`sel`	[Select]
	|	`exec`	[Execute]
	|	`(`Q`)`	[Executable-sequence]

Figure 1: The S-Expression Grammar for PostFix

Appendix B: PostFix SOS from Chapter 3

$\mathcal{C}, \mathcal{F}, \mathcal{I}$ and $\mathcal{O}$ for our PostFix SOS are given by:

$\mathcal{C} = \text{Commands} \times \text{Stack}$

$\mathcal{F} = \{[\,]_{\text{Command}}\} \times \text{Stack}$

$$
\begin{array}{l}
\mathcal{I} : \text{Program} \rightarrow \mathcal{C} \\
= \lambda P\ .\ \textbf{matching}\ P \\
\qquad \triangleright\ \texttt{(}Q\texttt{)}\ \|\ \langle Q, [\,]_{\text{Value}} \rangle \\
\qquad \textbf{endmatching}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{O} : \mathcal{F} \rightarrow \text{Answer} \\
= \lambda \langle [\,]_{\text{Command}}, S \rangle\ .\ \textbf{matching}\ S \\
\qquad\qquad\qquad\quad \triangleright\ V\,.\,S'\ \|\ (\text{Value} \mapsto \text{Answer}\ V) \\
\qquad\qquad\qquad\quad \triangleright\ [\,]\ \|\ (\text{Error} \mapsto \text{Answer}\ \texttt{error}) \\
\qquad\qquad\qquad\quad \textbf{endmatching}
\end{array}
$$

$$
\boxed{
\begin{array}{cr}
\langle N\,.\,Q,\ S \rangle \Rightarrow \langle Q,\ N\,.\,S \rangle & [\textit{numeral}] \\[6pt]
\langle \texttt{(}Q_{exec}\texttt{)}\,.\,Q_{rest},\ S \rangle \Rightarrow \langle Q_{rest},\ Q_{exec}\,.\,S \rangle & [\textit{executable}] \\[6pt]
\langle \texttt{pop}\,.\,Q,\ V_{top}\,.\,S \rangle \Rightarrow \langle Q,\ S \rangle & [\textit{pop}] \\[6pt]
\langle \texttt{swap}\,.\,Q,\ V_1\,.\,V_2\,.\,S \rangle \Rightarrow \langle Q,\ V_2\,.\,V_1\,.\,S \rangle & [\textit{swap}] \\[6pt]
\langle \texttt{sel}\,.\,Q_{rest},\ V_{false}\,.\,V_{true}\,.\,\texttt{0}\,.\,S \rangle \Rightarrow \langle Q_{rest},\ V_{false}\,.\,S \rangle & [\textit{sel-false}] \\[6pt]
\begin{array}{c}\langle \texttt{sel}\,.\,Q_{rest},\ V_{false}\,.\,V_{true}\,.\,N_{test}\,.\,S \rangle \Rightarrow \langle Q_{rest},\ V_{true}\,.\,S \rangle \\ \text{where } N_{test} \not\equiv \texttt{0}\end{array} & [\textit{sel-true}] \\[6pt]
\langle \texttt{exec}\,.\,Q_{rest},\ Q_{exec}\,.\,S \rangle \Rightarrow \langle Q_{exec} @ Q_{rest},\ S \rangle & [\textit{execute}] \\[6pt]
\begin{array}{c}\langle A\,.\,Q,\ N_1\,.\,N_2\,.\,S \rangle \Rightarrow \langle Q,\ N_{result}\,.\,S \rangle \\ \text{where } N_{result} \equiv (\textit{calculate}\ A\ N_2\ N_1) \\ \text{and}\quad \neg\,((A \equiv \texttt{div}) \wedge (N_1 \equiv \texttt{0}))\end{array} & [\textit{arithop}] \\[6pt]
\begin{array}{c}\langle R\,.\,Q,\ N_1\,.\,N_2\,.\,S \rangle \Rightarrow \langle Q,\ \texttt{1}\,.\,S \rangle \\ \text{where } (\textit{compare}\ R\ N_2\ N_1)\end{array} & [\textit{relop-true}] \\[6pt]
\begin{array}{c}\langle R\,.\,Q,\ N_1\,.\,N_2\,.\,S \rangle \Rightarrow \langle Q,\ \texttt{0}\,.\,S \rangle \\ \text{where } \neg\,(\textit{compare}\ R\ N_2\ N_1)\end{array} & [\textit{relop-false}]
\end{array}
}
$$

Figure 2: Rewrite rules defining the transition relation for PostFix.

Appendix C: Standard Semantics of FLK!

$$
\begin{array}{rcl}
v \in \text{Value} & = & \text{Unit + Bool + Int + Sym + Pair + Procedure + Location} \\
k \in \text{Expcont} & = & \text{Value} \rightarrow \text{Cmdcont} \\
\gamma \in \text{Cmdcont} & = & \text{Store} \rightarrow \text{Expressible} \\
\text{Expressible} & = & (\text{Value} + \text{Error})_\perp \\
\text{Error} & = & \text{Sym} \\
p \in \text{Procedure} & = & \text{Denotable} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont} \\
d \in \text{Denotable} & = & \text{Value} \\
e \in \text{Environment} & = & \text{Identifier} \rightarrow \text{Binding} \\
\beta \in \text{Binding} & = & (\text{Denotable} + \text{Unbound})_\perp \\
\text{Unbound} & = & \{\mathit{unbound}\} \\
s \in \text{Store} & = & \text{Location} \rightarrow \text{Assignment} \\
l \in \text{Location} & = & \text{Nat} \\
\alpha \in \text{Assignment} & = & (\text{Storable} + \text{Unassigned})_\perp \\
\sigma \in \text{Storable} & = & \text{Value} \\
\text{Unassigned} & = & \{\mathit{unassigned}\}
\end{array}
$$

top-level-cont : Expcont
$= \lambda v\,.\ \lambda s\,.\ (\text{Value} \mapsto \text{Expressible}\ v)$

error-cont : Error → Cmdcont
$= \lambda y\,.\ \lambda s\,.\ (\text{Error} \mapsto \text{Expressible}\ y)$

empty-env : Environment $= \lambda I\,.\ (\text{Unbound} \mapsto \text{Binding}\ \mathit{unbound})$

test-boolean : (Bool → Cmdcont) → Expcont

```
= λf . (λv . matching v
             ▷ (Bool ↦Value  b) ‖ (f  b)
             ▷ else (error-cont non-boolean)
             endmatching )
```

Similarly for:
test-procedure : (Procedure → Cmdcont) → Expcont
test-location : (Location → Cmdcont) → Expcont
etc.

ensure-bound : Binding → Expcont → Cmdcont

```
= λβk . matching β
        ▷ (Denotable ↦Binding  v) ‖ (k  v)
        ▷ (Unbound ↦Binding  unbound) ‖ (error-cont unbound-variable)
        endmatching
```

Similarly for:
ensure-assigned : Assignment → Expcont → Cmdcont

Figure 3: Semantic algebras for standard semantics of strict CBV FLK!.

```
$\mathit{same\text{-}location?}$ : Location → Location → Bool $= \lambda l_1 l_2 \,.\, (l_1 =_{\text{Nat}} l_2)$
$\mathit{next\text{-}location}$ : Location → Location $= \lambda l \,.\, (l +_{\text{Nat}} 1)$

$\mathit{empty\text{-}store}$ : Store $= \lambda l \,.$ (Unassigned $\mapsto$Assignment $\mathit{unassigned}$)

$\mathit{fetch}$ : Location → Store → Assignment $= \lambda l s \,.\, (s\ l)$

$\mathit{assign}$ : Location → Storable → Store → Store
 $= \lambda l_1 \sigma s \,.\ \lambda l_2 \,.$ **if** ($\mathit{same\text{-}location?}\ l_1\ l_2$)
                     **then** (Storable $\mapsto$Assignment $\sigma$)
                     **else** ($\mathit{fetch}\ l_2\ s$)

$\mathit{fresh\text{-}loc}$ : Store → Location $= \lambda s \,.$ ($\mathit{first\text{-}fresh}\ s\ 0$)

$\mathit{first\text{-}fresh}$ : Store → Location → Location
 $= \lambda s l \,.$ **matching** ($\mathit{fetch}\ l\ s$)
          ▷ (Unassigned $\mapsto$Assignment $\mathit{unassigned}$) $\|$ $l$
          ▷ **else** ($\mathit{first\text{-}fresh}\ s$ ($\mathit{next\text{-}location}\ l$))
          **endmatching**

$\mathit{lookup}$ : Environment → Identifier → Binding $= \lambda e I \,.\ (e\ I)$
```

Figure 4: Store helper functions for standard semantics of strict CBV FLK!.

$\mathcal{TL} : \mathrm{Exp} \to \mathrm{Expressible}$
$\mathcal{E} : \mathrm{Exp} \to \mathrm{Environment} \to \mathrm{Expcont} \to \mathrm{Cmdcont}$
$\mathcal{L} : \mathrm{Lit} \to \mathrm{Value}$ *; Defined as usual*

$$\mathcal{TL}[\![E]\!] = \mathcal{E}[\![E]\!]\ \textit{empty-env}\ \textit{top-level-cont}\ \textit{empty-store}$$

$$\mathcal{E}[\![L]\!] = \lambda ek\,.\ k\ \mathcal{L}[\![L]\!]$$

$$\mathcal{E}[\![I]\!] = \lambda ek\,.\ \textit{ensure-bound}\ (\textit{lookup}\ e\ I)\ k$$

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda ek\,.\ k\ (\mathrm{Procedure} \mapsto \mathrm{Value}\ (\lambda dk'\,.\ \mathcal{E}[\![E]\!]\ [I : d]e\ k'))$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\textit{test-procedure}\ (\lambda p\,.\ \mathcal{E}[\![E_2]\!]\ e\ (\lambda v\,.\ p\ v\ k)))$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] = \\ &\quad \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\textit{test-boolean}\ (\lambda b\,.\ \textbf{if}\ b\ \textbf{then}\ \mathcal{E}[\![E_2]\!]\ e\ k\ \textbf{else}\ \mathcal{E}[\![E_3]\!]\ e\ k))\end{aligned}$$

$$\mathcal{E}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\,.\ \mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\,.\ k\ (\mathrm{Pair} \mapsto \mathrm{Value}\ \langle v_1,\ v_2\rangle)))$$

$$\begin{aligned}\mathcal{E}[\![\texttt{(cell}\ E\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E]\!]\ e\ \ (\lambda vs\,.\ k\ &(\mathrm{Location} \mapsto \mathrm{Value}\ (\textit{fresh-loc}\ s)) \\ &(\textit{assign}\ (\textit{fresh-loc}\ s)\ v\ s))\end{aligned}$$

$$\mathcal{E}[\![\texttt{(begin}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_{\mathrm{ignore}}\,.\ \mathcal{E}[\![E_2]\!]\ e\ k)$$

$$\mathcal{E}[\![\texttt{(primop cell-ref}\ E\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E]\!]\ e\ (\textit{test-location}\ (\lambda ls\,.\ \textit{ensure-assigned}\ (\textit{fetch}\ l\ s)\ k\ s))$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{(primop cell-set!}\ E_1\ E_2\texttt{)}]\!] \\ &\quad = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\textit{test-location}\ (\lambda l\,.\ \mathcal{E}[\![E_2]\!]\ e\ (\lambda vs\,.\ k\ (\mathrm{Unit} \mapsto \mathrm{Value}\ \textit{unit})\ (\textit{assign}\ l\ v\ s))))\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![\texttt{(rec}\ I\ E\texttt{)}]\!] = \lambda eks\,.\ \mathrm{let}\ f = {}&\mathbf{fix}_{\mathrm{Expressible}}\ (\lambda a\,.\ \mathcal{E}[\![E]\!]\ [I : (\textit{extract-value}\ \mathrm{a})]\ e\ \mathrm{top\text{-}level\text{-}cont}\ s) \\ &\textbf{matching}\ f \\ &\triangleright (\mathrm{Value} \mapsto \mathrm{Expressible}\ v)\ [\!]\ \mathcal{E}[\![E]\!]\ [I : v]\ \mathrm{e}\ k\ s \\ &\triangleright\ \textbf{else}\ f \\ &\textbf{endmatching}\end{aligned}$$

$$\begin{aligned}&\textit{extract-value} : \mathrm{Expressible} \to \mathrm{Binding} \\ &= \lambda a\,.\ \ \textbf{matching}\ \mathrm{a} \\ &\qquad\quad \triangleright (\mathrm{Value} \mapsto \mathrm{Expressible}\ v)\ [\!]\ (\mathrm{Denotable} \mapsto \mathrm{Binding}\ v) \\ &\qquad\quad \triangleright\ \textbf{else}\ \bot_{\mathrm{Binding}} \\ &\qquad\quad \textbf{endmatching}\end{aligned}$$

Figure 5: Valuation clauses for standard semantics of strict CBV FLK!.

Appendix D: Typing Rules for SCHEME/R

$$\vdash \texttt{\#u} : \texttt{unit} \qquad [\textit{unit}]$$

$$\vdash B : \texttt{bool} \qquad [\textit{bool}]$$

$$\vdash N : \texttt{int} \qquad [\textit{int}]$$

$$\vdash \texttt{(symbol } I\texttt{)} : \texttt{sym} \qquad [\textit{symbol}]$$

$$[\ldots, I{:}T, \ldots] \vdash I : T \qquad [\textit{var}]$$

$$[\ldots, I{:}\texttt{(generic (}I_1 \texttt{ ... } I_n\texttt{) } T_{body}\texttt{)}, \ldots] \vdash I : (\forall i \ \ [T_i/I_i])T_{body} \qquad [\textit{genvar}]$$

$$\frac{A \vdash E_{test} : \texttt{bool} \ ; \ A \vdash E_{con} : T \ ; \ A \vdash E_{alt} : T}{A \vdash \texttt{(if } E_{test} \ E_{con} \ E_{alt}\texttt{)} : T} \qquad [\textit{if}]$$

$$\frac{A[I_1{:}T_1\texttt{, ..., } I_n{:}T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda (}I_1 \texttt{ ... } I_n\texttt{) } E_{body}\texttt{)} : \texttt{(-> (}T_1 \texttt{ ... } T_n\texttt{) } T_{body}\texttt{)}} \qquad [\lambda]$$

$$\frac{\begin{array}{c} A \vdash E_{rator} : \texttt{(-> (}T_1 \texttt{ ... } T_n\texttt{) } T_{body}\texttt{)} \\ \forall i \,.\, (A \vdash E_i : T_i) \end{array}}{A \vdash \texttt{(}E_{rator} \ E_1 \texttt{ ... } E_n\texttt{)} : T_{body}} \qquad [\textit{apply}]$$

$$\frac{\begin{array}{c} \forall i \,.\, (A \vdash E_i : T_i) \\ A[I_1{:}\mathit{Gen}\texttt{(}T_1\texttt{, } A\texttt{), ..., } I_n{:}\mathit{Gen}\texttt{(}T_n\texttt{, } A\texttt{)}] \vdash E_{body} : T_{body} \end{array}}{A \vdash \texttt{(let ((}I_1 \ E_1\texttt{) ... (}I_n \ E_n\texttt{)) } E_{body}\texttt{)} : T_{body}} \qquad [\textit{let}]$$

$$\frac{\begin{array}{c} \forall i \,.\, (A[I_1{:}T_1\texttt{, ..., } I_n{:}T_n] \vdash E_i : T_i) \\ A[I_1{:}\mathit{Gen}\texttt{(}T_1\texttt{, } A\texttt{), ... } I_n{:}\mathit{Gen}\texttt{(}T_n\texttt{, } A\texttt{)}] \vdash E_{body} : T_{body} \end{array}}{A \vdash \texttt{(letrec ((}I_1 \ E_1\texttt{) ... (}I_n \ E_n\texttt{)) } E_{body}\texttt{)} : T_{body}} \qquad [\textit{letrec}]$$

$$\frac{\forall i \,.\, (A \vdash E_i : T_i)}{A \vdash \texttt{(record (}I_1 \ E_1\texttt{) ... (}I_n \ E_n\texttt{))} : \texttt{(recordof (}I_1 \ T_1\texttt{) ... (}I_n \ T_n\texttt{))}} \qquad [\textit{record}]$$

$$\frac{\begin{array}{c} A \vdash E_r : \texttt{(recordof (}I_1 \ T_1\texttt{) ... (}I_n \ T_n\texttt{))} \\ A[I_1{:}T_1\texttt{, ..., } I_n{:}T_n] \vdash E_b : T \end{array}}{A \vdash \texttt{(with (}I_1 \texttt{ ... } I_n\texttt{) } E_r \ E_b\texttt{)} : T} \qquad [\textit{with}]$$

$$\frac{A \vdash \texttt{(letrec ((}I_1 \ E_1\texttt{) ... (}I_n \ E_n\texttt{)) } E_{body}\texttt{)} : T}{A \vdash \texttt{(program (define } I_1 \ E_1\texttt{) ... (define } I_n \ E_n\texttt{) } E_{body}\texttt{)} : T} \qquad [\textit{program}]$$

$$\mathit{Gen}(T, A) = \texttt{(generic (}I_1 \texttt{ ... } I_n\texttt{) } T\texttt{)}, \text{where } \{I_i\} = \mathit{FTV}\ (T) - \mathit{FTE}(A)$$

Appendix E: Type Reconstruction Algorithm for SCHEME/R

$R[\![\texttt{\#u}]\!]\ A\ S = \langle\texttt{unit}, S\rangle$

$R[\![B]\!]\ A\ S = \langle\texttt{bool}, S\rangle$

$R[\![N]\!]\ A\ S = \langle\texttt{int}, S\rangle$

$R[\![\texttt{(symbol}\ I\texttt{)}]\!]\ A\ S = \langle\texttt{sym}, S\rangle$

$R[\![I]\!]\ A[I : T]\ S = \langle T, S\rangle$

$R[\![I]\!]\ A[I : \texttt{(generic (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}]\ S = \langle T[?v_i/I_i], S\rangle$ ($?v_i$ are new)

$R[\![I]\!]\ A\ S = \text{fail}$ (when I is unbound)

```
$R[\![\texttt{(if}\ E_t\ E_c\ E_a\texttt{)}]\!]\ A\ S =$ let $\langle T_t, S_t\rangle = R[\![E_t]\!]A\ S$
    in let $S'_t = U(T_t, \texttt{bool}, S_t)$
        in let $\langle T_c, S_c\rangle = R[\![E_c]\!]A\ S'_t$
            in let $\langle T_a, S_a\rangle = R[\![E_a]\!]A\ S_c$
                in let $S'_a = U(T_c, T_a, S_a)$
                    in $\langle T_a, S'_a\rangle$
```

```
$R[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_b\texttt{)}]\!]\ A\ S =$ let $\langle T_b, S_b\rangle = R[\![E_b]\!]A[I_i : ?v_i]\ S$
    in $\langle\texttt{(-> (}?v_1\ \ldots\ ?v_n\texttt{)}\ T_b\texttt{)}, S_b\rangle$ ($?v_i$ are new)
```

```
$R[\![\texttt{(}E_0\ E_1\ \ldots\ E_n\texttt{)}]\!]\ A\ S =$ let $\langle T_0, S_0\rangle = R[\![E_0]\!]A\ S$
    in ...
     let $\langle T_n, S_n\rangle = R[\![E_n]\!]A\ S_{n-1}$
     in let $S_f = U(T_0, \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ ?v_f\texttt{)}, S_n)$
         in $\langle ?v_f, S_f\rangle$ ($?v_f$ is new)
```

```
$R[\![\texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_b\texttt{)}]\!]\ A\ S =$ let $\langle T_1, S_1\rangle = R[\![E_1]\!]A\ S$
    in ...
     let $\langle T_n, S_n\rangle = R[\![E_n]\!]A\ S_{n-1}$
     in $R[\![E_b]\!]A[I_i : \mathit{Rgen}(T_i, A, S_n)]S_n$
```

```
$R[\![\texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_b\texttt{)}]\!]\ A\ S =$ let $A_1 = A[I_i : ?v_i]$ ($?v_i$ are new)
    in let $\langle T_1, S_1\rangle = R[\![E_1]\!]A_1\ S$
        in ...
         let $\langle T_n, S_n\rangle = R[\![E_n]\!]A_1\ S_{n-1}$
         in let $S_b = U(?v_i, T_i, S_n)$
             in $R[\![E_b]\!]A[I_i : \mathit{Rgen}(T_i, A, S_b)]\ S_b$
```

```
$R[\![\texttt{(record (}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}]\!]\ A\ S =$ let $\langle T_1, S_1\rangle = R[\![E_1]\!]A\ S$
    in ...
     let $\langle T_n, S_n\rangle = R[\![E_n]\!]A\ S_{n-1}$
     in $\langle\texttt{(recordof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}, S_n\rangle$
```

```
$R[\![\texttt{(with (}I_1\ \ldots\ I_n\texttt{)}\ E_r\ E_b\texttt{)}]\!]\ A\ S =$ let $\langle T_r, S_r\rangle = R[\![E_r]\!]A\ S$
    in let $S_b = U(T_r, \texttt{(recordof (}I_1\ ?v_i\texttt{)}\ \ldots\ \texttt{(}I_n\ ?v_n\texttt{))}, S_r)$ ($?v_i$ are new)
        in $R[\![E_b]\!]A[I_i : ?v_i]\ S_b$
```

$$\mathit{Rgen}(T, A, S) = \mathit{Gen}((S\ T), (\mathit{subst\text{-}in\text{-}type\text{-}env}\ S\ A))$$

Appendix F: Meta-CPS Conversion Rules

In the following rules, grey mathematical notation (like λv) and square brackets [ ] are used for "meta-application", which is evaluated *as part of meta-CPS conversion.* Code in `BLACK TYPEWRITER FONT` is part of the output program; meta-CPS conversion does *not* evaluate any of this code. Therefore, you can think of meta-CPS-converting an expression E as rewriting $\mathcal{MCPS}[\![E]\!]$ until no grey is left.

$$\begin{array}{rcl} \text{E} & \in & \text{Exp} \\ \text{m} & \in & \text{Meta-Continuation} \ = \ \text{Exp} \rightarrow \text{Exp} \end{array}$$

$\textit{meta-cont} \rightarrow \textit{exp}$: (Exp → Exp) → Exp = $[\lambda m\,.\,\texttt{(LAMBDA (t)}[m\ \texttt{t}]\texttt{)}]$
$\textit{exp} \rightarrow \textit{meta-cont}$: Exp → (Exp → Exp) = $[\lambda E\,.\,[\lambda V\,.\,\texttt{(CALL}\ E\ \ V\texttt{)}]]$

meta-cont→exp $[\lambda V\,.\,\texttt{(CALL K}\ V\texttt{)}] = K$

$\mathcal{MCPS}$: Exp → Meta-Continuation → Exp

$\mathcal{MCPS}[\![I]\!] = [\lambda m\,.\,[m\ \ I]]$

$\mathcal{MCPS}[\![L]\!] = [\lambda m\,.\,[m\ \ L]]$

$$\begin{array}{l} \mathcal{MCPS}[\![\texttt{(LAMBDA (}I_1\ \texttt{...}\ I_n\texttt{)}\ E\texttt{)}]\!] \\ \quad = [\lambda m\,.\,[m\ \ \texttt{(LAMBDA (}I_1\ \texttt{...}\ I_n\ \texttt{.K}i\texttt{.)} \\ \qquad\qquad\qquad [\mathcal{MCPS}[\![E]\!]\ \ [\textit{exp} \rightarrow \textit{meta-cont}\ \ \texttt{.K}i\texttt{.}]]\texttt{)}]] \end{array}$$

$$\begin{array}{l} \mathcal{MCPS}[\![\texttt{(CALL}\ E_1\ \ E_2\texttt{)}]\!] \\ \quad = [\lambda m\,.\,[\mathcal{MCPS}[\![E_1]\!]\ \ [\lambda v_1\,. \\ \qquad\qquad\qquad [\mathcal{MCPS}[\![E_2]\!]\ \ [\lambda v_2\,. \\ \qquad\qquad\qquad\qquad \texttt{(CALL}\ v_1\ \ v_2\ \ [\textit{meta-cont} \rightarrow \textit{exp}\ \ \ m]\texttt{)}]]]]] \end{array}$$

$$\begin{array}{l} \mathcal{MCPS}[\![\texttt{(PRIMOP}\ P\ \ E_1\ \ E_2\texttt{)}]\!] \\ \quad = [\lambda m\,.\,[\mathcal{MCPS}[\![E_1]\!]\ \ [\lambda v_1\,. \\ \qquad\qquad\qquad [\mathcal{MCPS}[\![E_2]\!]\ \ [\lambda v_2\,. \\ \qquad\qquad\qquad\qquad \texttt{(LET ((.T}i\texttt{. (PRIMOP}\ P\ \ v_1\ \ v_2\texttt{)))} \\ \qquad\qquad\qquad\qquad\quad [m\ \ \ \texttt{.T}i\texttt{.}]\texttt{)}]]]]] \end{array}$$

$$\begin{array}{l} \mathcal{MCPS}[\![\texttt{(IF}\ E_c\ \ E_t\ \ E_f\texttt{)}]\!] \\ \quad = [\lambda m\,.\,[\mathcal{MCPS}[\![E_c]\!]\ \ [\lambda v_1\,. \\ \qquad\qquad\qquad \texttt{(LET ((K}\ \ [\textit{meta-cont} \rightarrow \textit{exp}\ \ m]\texttt{))} \\ \qquad\qquad\qquad\quad \texttt{(IF}\ v_1 \\ \qquad\qquad\qquad\qquad [\mathcal{MCPS}[\![E_t]\!]\ \ [\textit{exp} \rightarrow \textit{meta-cont}\ \ \texttt{K}]] \\ \qquad\qquad\qquad\qquad [\mathcal{MCPS}[\![E_f]\!]\ \ [\textit{exp} \rightarrow \textit{meta-cont}\ \ \texttt{K}]]\texttt{))}]]] \end{array}$$

$$\begin{array}{l} \mathcal{MCPS}[\![\texttt{(LET ((}I\ \ E_{\text{def}}\texttt{))}\ \ E_{\text{body}}\texttt{)}]\!] \\ \quad = [\lambda m\,.\,[\mathcal{MCPS}[\![E_{\text{def}}]\!]\ \ [\lambda v\,. \\ \qquad\qquad\qquad \texttt{(LET ((I}\ \ v\texttt{))}\ \ [\mathcal{MCPS}[\![E_{\text{body}}]\!]\ \ m]\texttt{)}]]] \end{array}$$

2. Final Examination Solutions

Problem 1: Short Answer

a. Explicit Polymorphism

(i)
```
(plambda (t)
  (lambda ((f (-> (t) bool)) (g (-> (t) t) (x t)))
    (if (f x)
        (g x)
        (g (g x)))))
```
(ii)
```
(lambda ((f (poly (t) (-> (t) t))))
  (if ((proj f bool) #t)
      ((proj f int) ((proj f int) 1))
      ((proj f int) 2)))
```

b. Type Reconstruction

(i) No reconstructed type exists. There is no first class polymorphism allowed in Scheme/R.

(ii) int

(iii) No reconstructed type exists. No heterogeneous lists in Scheme/R.

(iv) No reconstructed type exists. Hindley-Milner cannot reconstruct the type of an identifier involved in a self application.

Problem 2: Operational Semantics

a.

$$
\begin{aligned}
\mathcal{I}_{\text{TwoStack}} &: \text{Program}_{\text{TwoStack}} \rightarrow \mathcal{C}_{\text{TwoStack}} \\
\mathcal{I}_{\text{TwoStack}} &= \lambda\langle Q_1,\ Q_2\rangle\ .\ \langle\langle Q_1,\ Q_2\rangle,\ []_{\text{Value}},\ []_{\text{Value}}\rangle \\
\mathcal{F}_{\text{TwoStack}} &= \{[]_{\text{Command}_{\text{PostFix+Talk}}}\} \times \{[]_{\text{Command}_{\text{PostFix+Talk}}}\} \times \mathsf{Stack} \times \mathsf{Stack} \\
\mathcal{O}_{\text{TwoStack}} &: \mathcal{F} \rightarrow \text{Answer}_{\text{TwoStack}} \\
\mathcal{O}_{\text{TwoStack}} &= \lambda\langle\langle []_{\text{Command}_{\text{PostFix+Talk}}},\ []_{\text{Command}_{\text{PostFix+Talk}}}\rangle,\ S_1,\ S_2\rangle\ . \\
&\quad \langle \mathcal{O}_{\text{PostFix}}\langle []_{\text{Command}_{\text{PostFix+Talk}}},\ S_1\rangle,\ \mathcal{O}_{\text{PostFix}}\langle []_{\text{Command}_{\text{PostFix+Talk}}},\ S_2\rangle\rangle \\
\Rightarrow_{\text{TwoStack}} &: \\
&\frac{\langle Q_1,\ S_1\rangle \Rightarrow_{\text{PostFix+Talk}} \langle Q_1',\ S_1'\rangle \wedge \langle Q_2,\ S_2\rangle \Rightarrow_{\text{PostFix+Talk}} \langle Q_2',\ S_2'\rangle}{\langle\langle Q_1,\ Q_2\rangle,\ S_1,\ S_2\rangle \Rightarrow_{\text{TwoStack}} \langle\langle Q_1',\ Q_2'\rangle,\ S_1',\ S_2'\rangle}
\end{aligned}
$$

b.

$$\langle\langle \texttt{talk}.Q_1,\ \texttt{talk}.Q_2\rangle,\ V_1.S_1,\ V_2.S_2\rangle \quad \Rightarrow_{\text{TwoStack}} \quad \langle\langle Q_1,\ Q_2\rangle,\ V_2.V_1.S_1,\ V_1.V_2.S_2\rangle$$

c. The following program does not terminate:

```
<((talk exec) talk exec),((talk exec) talk exec)>
```

Problem 3: Denotational Semantics

a. $\mathcal{TL}[\![E]\!] = (\mathcal{E}[\![E]\!]\ \textit{empty-env}\ (\lambda dk\,.\,(\textit{top-level-cont}\ d))\ \textit{top-level-cont}\ \textit{empty-store})$

b. $\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e p_1 k\,.\ (\mathcal{E}[\![E_1]\!]\ e\ p_1\ (\textit{test-procedure}\ (\lambda p_2\,.\,(\mathcal{E}[\![E_2]\!]\ e\ p_1\ (\lambda v\,.\,(p_2\ v\ k))))))$

c. $\mathcal{E}[\![\texttt{(self } E\texttt{)}]\!] = \lambda epk\,.\ (\mathcal{E}[\![\mathrm{E}]\!]\ e\ p\ \lambda v\,.\,(p\ v\ k))$

d. $\mathcal{E}[\![\texttt{(proc } I\ E\texttt{)}]\!] =$
$\lambda e p_1 k_1\,.\ (k_1\ (\mathrm{Procedure}{\mapsto}\mathrm{Value}\ (\mathbf{fix}_{Procedure}(\lambda p_2\,.\,(\lambda d k_2\,.\,(\mathcal{E}[\![E]\!]\ [I:d]e\ p_2\ k_2))))))$

e.

$$\begin{aligned}
& \mathcal{TL}[\![\texttt{(self (self 1))}]\!] = \\
= \ & (\mathcal{E}[\![\texttt{(self (self 1))}]\!]\ \textit{empty-env}\ (\lambda dk\,.\,(\textit{top-level-cont}\ d))\ \textit{top-level-cont}\ \textit{empty-store}) \\
= \ & (\mathcal{E}[\![\texttt{(self 1)}]\!]\ \textit{empty-env}\ (\lambda dk\,.\,(\textit{top-level-cont}\ d))\ \textit{top-level-cont}\ \textit{empty-store}) \\
& \text{because } \lambda v\,.\,(\lambda dk\,.\,(\textit{top-level-cont}\ d)\ v\ k) = \textit{top-level-cont} \\
= \ & (\mathcal{E}[\![1]\!]\ \textit{empty-env}\ (\lambda dk\,.\,(\textit{top-level-cont}\ d))\ \textit{top-level-cont}\ \textit{empty-store}) \\
= \ & (\textit{top-level-cont}\ \mathcal{L}[\![1]\!]) \\
= \ & (\mathrm{Value}{\mapsto}\mathrm{Expressible}\ (\mathrm{Int}{\mapsto}\mathrm{Value}\ 1))
\end{aligned}$$

f.

$$\begin{aligned}
& \mathcal{E}[\![\texttt{(proc x (self 1))}]\!] \\
= \ & \lambda e p_1 k_1\,.\,(k_1\ \ (\mathrm{Procedure}{\mapsto}\mathrm{Value}\ (\mathbf{fix}_{Procedure}\ \ \lambda p_2\,.\,(\lambda d k_2\,.\,(\mathcal{E}[\![\texttt{(self 1)}]\!]\ [\texttt{x}:d]e\ \ p_2\ \ k_2))))) \\
= \ & \lambda e p_1 k_1\,.\,(k_1\ \ (\mathrm{Procedure}{\mapsto}\mathrm{Value}\ (\mathbf{fix}_{Procedure}\ \ \lambda p_2\,.\,(\lambda d k_2\,.\,(\mathcal{E}1)\ \ [\texttt{x}:d]e\ \ p_2\ \ \lambda v\,.\,(p_2\ \ v\ \ k_2)))))) \\
= \ & \lambda e p_1 k_1\,.\,(k_1\ \ (\mathrm{Procedure}{\mapsto}\mathrm{Value}\ (\mathbf{fix}_{Procedure}\ \ \lambda p_2\,.\,(\lambda d k_2\,.\,(\lambda v\,.\,(p_2\ \ v\ \ k_2)\ \ \mathcal{L}[\![1]\!]))))) \\
= \ & \lambda e p_1 k_1\,.\,(k_1\ \ (\mathrm{Procedure}{\mapsto}\mathrm{Value}\ (\mathbf{fix}_{Procedure}\ \ \lambda p_2\,.\,(\lambda d k_2\,.\,(p_2\ \ \mathcal{L}[\![1]\!]\ \ k_2))))) \\
& \text{but } \bot_{Procedure} \text{ is a fixed point of } \lambda p_2\,.\,(\lambda d k_2\,.\,(p_2\ \ \mathcal{L}[\![1]\!]\ \ k_2))
\end{aligned}$$

It should be clear that $\bot_{Procedure}$ must be the procedure-generating function's least fixed point. This means that the procedural value that the expression **(proc x (self 1))** computes is $\bot_{Procedure}$, a procedure that given any denotable and expression continuation returns $\bot_{CmdCont}$ i.e. a procedure that loops forever regardless of its input. Since this procedural value is the value computed by **(proc x (self 1))**, we have completed the demonstration.

Problem 4: Type Reconstruction

a.

$$\frac{A[I\!:\!\texttt{(control-point } T\texttt{)}] \vdash E : T}{A \vdash \texttt{(label } I\ E\texttt{)} : T} \quad [\textit{label}]$$

$$\frac{\begin{array}{c} A \vdash E_1 : \texttt{(control-point } T\texttt{)} \\ A \vdash E_2 : T \end{array}}{A \vdash \texttt{(jump } E_1\ E_2\texttt{)} : T_{any}} \quad [\textit{jump}]$$

b. $R[\![\texttt{(label } I\ E\texttt{)}]\!]\ A\ S = \ \ \mathbf{let}\ \langle T_1, S_1\rangle = R[\![E]\!]A[I:\texttt{(control-point } ?v_1\texttt{)}]\ S$
$\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \langle T_1, U(T_1, ?v_1, S_1)\rangle$

where $?v_1$ is a fresh type variable

c. $R[\![$`(jump` E_1 E_2`)`$]\!]\ A\ S =$ **let** $\langle T_1, S_1\rangle = R[\![E_1]\!] A\ S$
   in **let** $\langle T_2, S_2\rangle = R[\![E_2]\!] A\ S_1$
   in $\langle ?v_1, U(T_1,$ `(control-point` T_2`)`$, S_2)\rangle$

   where $?v_1$ is a fresh type variable

d. Unification fails while reconstructing the type of **(label y y)**. The unification call that fails is U(`(control-point` $?v_1$`)`,$?v_1$,S_1).

Problem 5: Compilers

a.
```
   (let ((compose (lambda (f g)(lambda (x)(f (g x))))))
        ((compose not integer?) 1))
```

b. $\mathcal{MCPS}[\![$`(begin` $E1$ $E2$`)`$]\!]$ m =
   (begin ($\mathcal{MCPS}[\![E1]\!]$ (λV . V)) ($\mathcal{MCPS}[\![E2]\!]$ m))

c. $\mathcal{T}[\![E]\!]$=

```
(let ((v P))
  (let ((r [v/P]E))
    (begin
     (%procedure-free v)
     r)))
```

d. One possible explanation is that the programs do not exhaust memory, and thus the garbage collector is never called. In that case, explicitly freeing unused closures is extra work that has no benefit. Another possibility is that in some programs, explicitly freeing closures releases enough memory so the garbage collector is no longer invoked, leading to storage fragmentation that slows the program down.