6.821 Programming Languages Handout
Fall 2002

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Final Examination Solutions 2002

Problem 1: Short Answer [29 points]

a. [4 points] Give a desugaring from a FLAVAR! program P to a FLAVAR! program P' that has the following property: P evaluated with call-by-value semantics has the same behavior as P' when evaluated with call-by-name semantics.

Solution: $\mathcal{D}[\![$`(proc` I E`)`$]\!]$ = `(proc` I $\mathcal{D}[\![$`(begin (primop cell-set! (cell` I`)` I`)` E`)`$]\!]$`)`
For all other E, $\mathcal{D}[\![E]\!]$ operates by structural induction.

b. [3 points] Give two domains D and E such that the number of set-theoretic functions from D to E is infinite, but the number of continuous functions is finite.

Solution: D = integers, with $\leq$ as the ordering; E = bool (unordered)

c. [3 points] Consider a new construct `smart-compose` to FL/R. `(smart-compose f g)` has the following behavior: it evaluates to whichever of `(compose f g)` or `(compose g f)` type checks. If both type-check, it chooses one of them arbitrarily. Why could type reconstruction using Appendix D fail even when a type for `smart-compose` is reconstructible?

Solution: Type reconstruction could fail if both `(compose f g)` and `(compose g f)` type check (for instance, if `f` has type $\alpha \to \beta$ and `g` has type $\beta \to \alpha$, for some concrete α and β). An expression that uses `(smart-compose f g)` might require it to have either $\alpha \to \alpha$ or $\beta \to \beta$, but type reconstruction might choose the other type. Correcting this problem would require adding backtracking to the algorithm.

d. [2 points] What is a broken heart, and what is it used for?

Solution: A broken heart is a forwarding pointer, used in cpoying garbage collection. After an object has been moved, it is replaced by the broken heart, so that references to it can be updated to use the new copy.

e. [2 points] Give the reconstructed type of the following expression.

```
(lambda (f g)
  (f (f g 2)
     (g 2)))
```

Solution:

```
g: int -> int
f: (int -> int) x int -> (int -> int)
overall: ((int -> int) x int -> (int -> int)) x (int -> int) -> (int -> int)
```

f. [2 points] What is the purpose of closure conversion?

Solution: It enables lifting by eliminating free variables and making each procedure self-contained, so that the procedure definition may appear anywhere in the program.

g. [2 points] Consider the following FLAVAR! expression

```
(let ((x 3)
      (foo (lambda (q) (+ x y)))
  (letrec ((y x)
           (z (let ((x 15)) (lambda (q) (z q)))))
     (let ((x 16)
           (y x))
        (foo z))))
```

(i) [1 points] What is its value, or what error occurs, under call-by-value static scoping?

Solution: Error: y is not bound

(ii) [1 points] What is its value, or what error occurs, its value under call-by-value dynamic scoping?

Solution: 19

h. [2 points] Give the value of the following expression, or state what error occurs.

```
(let ((x (lambda (x) (+ x 1))))
  (let ((y (lambda (y) (y x))))
    (let ((y (let ((y y)) y)))
      (let ((x (y (lambda (x) x))))
        (x 5)))))
```

(i) [1 points] When evaluated under call-by-value static scoping.

Solution: 6. Note that `(let ((y (let ((y y)) y))) ...)` has no effect on the evaluation of the body.

(ii) [1 points] When evaluated under call-by-value dynamic scoping.

Solution: 6, under identical control flow to that of the static case.
Again, `(let ((y (let ((y y)) y))) ...)` has no effect.
Furthermore, `(y (lambda (x) x))` is evaluated outside the scope of the last `(let ((x ...)) ...)`.

i. [3 points] Louis Reasoner has an application in which at least half of memory is garbage each time the garbage collector is invoked. He concludes that the use of two spaces is wasteful: he can modify the garbage collector to work in place. Explain why Louis's proposal does not work.

Solution: One problem is that it may be impossible to relocate objects, obviating some advantages of a copying garbage collector. For instance, there may be an object that is larger than any contiguous free chunk of memory.

j. [6 points] Give a type reconstruction algorithm rule for "call with current continuation" (`cwcc`). You may find referring to Appendix D helpful.

Here is an example of `cwcc`:

```
(+ (cwcc (lambda (exit) 100))
   (cwcc (lambda (exit) (+ 2 (exit 4)))))
⇒ 104
```

Solution:

```
$R[\![$(cwcc $p$)$]\!]\ A\ S$ =
  let $\langle T_e, S_1\rangle = R[\![E]\!]\ A\ S$
  in  let $S_2 = U(T_e,$ (-> (-> ($?v_r$) $?v_{\text{ignore}}$) $?v_r$))
      in $\langle ?v_r, S_2\rangle$            ($?v_r$ and $?v_{\text{ignore}}$ are new)
```

Problem 2: Probabilistic Types [12 points]

Sensing he won't have time to finish his SCHEME/R killer app before Christmas, Ben Bitdiddle buys a subroutine from Subroutines 'R' Us. While Ben's program was completely deterministic, the subroutine he bought is randomized. The fine print states that the subroutine returns the correct answer $99\frac{44}{100}\%$ of the time. Ben worries about the correctness of his program, since he uses this subroutine many times. But before he gets to answer his own question, his mind wanders off to another subroutine he purchased.

To deal with this problem, you will create a type/effect system. The new type system has judgments like the following:

$$A \vdash E : T \,\%\, P,$$

which is pronounced, "in the type environment A, expression E has type T and probability of correctness at least P." The types in this new type system are all inherited from SCHEME/R, with one difference: a function type now contains a probability of correctness. We write this as `(->` p `(`$T_1 \ldots T_n$`)` T_b`)`, which means that when this procedure is applied to arguments $a_1 \ldots a_n$, it evaluates to the correct value with probability p.

SCHEME/R always operates correctly and deterministically; however, it is possible to code randomized algorithms using it. When you use a randomized algorithm, you add a probabilistic type for it (specified by the manufacturer) to the typing environment; then, you use the type/effect system to compute the probability that your whole program operates correctly.

The following type/effect judgments

$$\vdash N : \texttt{int} \,\%\, 1$$
$$\vdash B : \texttt{bool} \,\%\, 1$$

indicate that numeric and boolean literals are guaranteed to be correct. You should assume that if any subexpression evaluates incorrectly, then the whole program evaluates incorrectly: subsequent wrong answers never cancel out previous ones.

a. [3 points] Give a type/effect rule for `if`.

Solution:

$$\frac{A \vdash E : \texttt{bool} \,\%\, p \;;\; A \vdash E_1 : T \,\%\, p_1 \;;\; A \vdash E_2 : T \,\%\, p_2}{A \vdash \texttt{(if } E \; E_1 \; E_2\texttt{)} : T \,\%\, p \cdot \min(p_1, p_2)} \quad [\textit{if}]$$

b. [3 points] Give a type/effect rule for `lambda`.

Solution:

$$\frac{A[I_1{:}T_1,\ \ldots,\ I_n{:}T_n] \vdash E_B : T_B \,\%\, p_B}{A \vdash \texttt{(lambda (}I_1 \ \ldots \ I_n\texttt{)} \ E_B\texttt{)} : \texttt{(->} \, p_B \, (T_1 \ldots T_n) T_B\texttt{)} \,\%\, 1} \quad [\lambda]$$

Note that there are no % annotations in the typing environment; they are not necessary there.

c. [3 points] Give a type/effect rule for application.

Solution:

$$\frac{\begin{array}{c} A \vdash E_{\text{rator}} : \texttt{(->} \ p_l \ \texttt{(}T_1 \ \ldots \ T_n\texttt{)} \ T_{body}\texttt{)} \,\%\, p_r \\ \forall i \,.\, (A \vdash E_i : T_i \,\%\, p_i) \end{array}}{A \vdash \texttt{(}E_{\text{rator}} \ E_1 \ \ldots \ E_n\texttt{)} : T_{body} \,\%\, p_r \cdot p_l \cdot p_1 \cdots p_n} \quad [\textit{apply}]$$

If you wished to support subtyping, you could instead write:

$$\frac{\begin{array}{c} A \vdash E_{\text{rator}} : \texttt{(->} \ p_l \ \texttt{(}T'_1 \ \ldots \ T'_n\texttt{)} \ T_{body}\texttt{)} \,\%\, p_r \\ \forall i \,.\, (A \vdash E_i : T_i \,\%\, p_i) \\ \forall i \,.\, (T_i \sqsubseteq T'_i) \end{array}}{A \vdash \texttt{(}E_{\text{rator}} \ E_1 \ \ldots \ E_n\texttt{)} : T_{body} \,\%\, p_r \cdot p_l \cdot p_1 \cdots p_n} \quad [\textit{apply}]$$

d. [3 points] Give a subtyping rule for procedures by revising the `->`-$\sqsubseteq$ rule of Appendix B (page 18).

Solution:

$$\frac{\begin{array}{c}\forall i\ (T_i \sqsubseteq S_i) \\ S_{body} \sqsubseteq T_{body} \\ p_s \geq p_t\end{array}}{\texttt{(->}\ p_s\ \texttt{(}S_1\ \ldots S_n\texttt{)}\ S_{body}\texttt{)} \sqsubseteq \texttt{(->}\ p_t\ \texttt{(}T_1\ \ldots T_n\texttt{)}\ T_{body}\texttt{)}} \quad [\texttt{->}\text{-}\sqsubseteq]$$

Problem 3: Strictness Effects [17 points]

As presented in class, effect systems indicate what side effects may result from evaluation of an expression. In a lazy language, a natural extension to this notion is whether a particular argument to a procedure is ever evaluated.

In this problem, you will write type reconstruction rules that determine whether a particular expression evaluates a given argument. There are three possibilities for each argument: the procedure may always evaluate the argument, the procedure may never evaluate the argument, or the procedure may sometimes evaluate the argument; we represent these three values as A (for always), N (for never), and M (for maybe).

Typing judgments have the form

$$A \vdash E : T \,\&\, S\,,$$

where S maps variables to strictness values. (You may assume that S always contains an entry for every variable that is ever used in the program, and furthermore that all variables are uniquely named. In other words, you don't have to worry about scoping and variable capture.)

For example, a reasonable judgment in a program with 4 variables a, b, c, and d might be

$$A \vdash E : T \,\&\, \{a : M, b : A, c : N, d : M\}\,.$$

and a reasonable typing rule might be

$$\frac{A \vdash E_1 : T \,\&\, S_1 \qquad A \vdash E_2 : T \,\&\, S_2}{A \vdash \texttt{(strange } E_1\ E_2\texttt{)} : T \,\&\, S_1 \quad \mathit{AtoN}(S_2)} \quad [\mathit{strange}]$$

where is taken elementwise and *AtoN* is defined below.

Procedure types have the following form: `(->` Ids S `(`T_1 `...` T_n`)` T`)`. "Ids" is a list of the procedure's parameter names. S is the strictness of the procedure (over all program variables, as usual).

Your task is to extend the typing rules of Appendix C to also compute a strictness environment.

You may find one or more of the following auxiliary procedures to be helpful. However, you are not permitted to cascade them (nor do you need to!); don't write, for example, "*AtoM*(*NtoA*(S))".

allA returns a new mapping in which every variable is bound to A.
allN returns a new mapping in which every variable is bound to N.
allM returns a new mapping in which every variable is bound to M.
AtoN returns a new mapping in which bindings to A have been replaced by N.
AtoM returns a new mapping in which bindings to A have been replaced by M.
NtoA returns a new mapping in which bindings to N have been replaced by A.
NtoM returns a new mapping in which bindings to N have been replaced by M.
MtoA returns a new mapping in which bindings to M have been replaced by A.
MtoN returns a new mapping in which bindings to M have been replaced by N.

For example,

$$\mathit{MtoN}(\{a : M, b : A, c : N, d : M\}) = \{a : N, b : A, c : N, d : N\}\,.$$

a. [2 points] Draw the domain whose elements are $\{A, N, M\}$: draw the elements, and draw a line for each non-transitive $\sqsubset$ relationship, with the greater element higher on the page. (Hint: No element is greater than A.)

   Solution: $A \sqsupseteq M \sqsupseteq N$

b. [4 points] Give the typing rule for `if`.

   Solution:

$$\frac{A \vdash E_1 : \texttt{bool} \,\&\, S_1 \ ; \ A \vdash E_2 : T \,\&\, S_2 \ ; \ A \vdash E_3 : T \,\&\, S_3}{A \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : T \ \&\ S_1 \quad \mathit{AtoM}(S_2) \quad \mathit{AtoM}(S_3)} \quad [\mathit{if}]$$

c. [4 points] Give the typing rule for `lambda`.

Solution:

$$\frac{A[I_1 : T_1\texttt{,}\ \ldots\texttt{,}\ I_n : T_n] \vdash E_B\ \texttt{:}\ T_B\ \texttt{\&}\ S_B}{A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}\ E_B\texttt{)}\ \texttt{:}\ \texttt{(-> (}I_1 \ldots I_n\texttt{)}\ S_B\ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_B\texttt{)}\ \texttt{\&}\ \mathit{AllN}(S_B)}\ [\lambda]$$

d. [4 points] Give the typing rule for application.

Solution:

Here is another way to write the same thing:

$$\frac{\begin{array}{c} A \vdash E_{rator}\ \texttt{:}\ \texttt{(-> (}I_1 \ldots I_n\texttt{)}\ S_{body}\ \texttt{(}T'_1\ \ldots\ T'_n\texttt{)}\ T_{body}\texttt{)}\ \texttt{\&}\ S_{rator} \\ \forall i\ .\ (A \vdash E_i\ \texttt{:}\ T_i\ \texttt{\&}\ S_i) \\ \forall i\ .\ (T_i \sqsubseteq T'_i) \end{array}}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)}\ \texttt{:}\ T_{body}\ \texttt{\&}\ S_r \quad (S_b \setminus \{I_1, \ldots, I_n\}) \quad {}_{i=1}^{n} \begin{cases} \text{if } S_b(i) = A, \text{ then } S_i \\ \text{if } S_b(i) = N, \text{ then } allN(S_i) \\ \text{if } S_b(i) = M, \text{ then } AtoM(S_i)) \end{cases}}\ [\mathit{apply}]$$

e. [3 points] Give the typing rule for (application of) +.

Solution:

$$\frac{\forall i\ .\ (A \vdash E_i\ \texttt{:}\ \texttt{int}\ \texttt{\&}\ S_i)}{A \vdash \texttt{(+}\ E_1\ \ldots E_n\texttt{)}\ \texttt{:}\ \texttt{int}\ \texttt{\&}\ {}_{i=1}^{n} S_i}\ [+]$$

Problem 4: Nosy FLK! [25 points]

The Department of Homeland Security has adopted FLK! as its official programming language, for two reasons. First, FLK! has no foreign function interface, and the DHS disapproves of anything foreign. Second, and more importantly, FLK! has no "private" declaration. The DHS reasons that programmers who have done nothing wrong should have nothing to hide, and the DHS insists on access to any data at any time for any purpose. Imagine the DHS's disappointment when they realize that shadowing can make a variable binding inaccessible! For example, consider the following code that you might wish to execute:

```
;; Send a love letter
(let ((letter "Dear sweetheart,  ...  With love, Snookums"))
  (let ((mail-agent (mail-sending-program letter)))
    (let ((letter "s"))    ; The "s" key sends a message
      ;; DHS INSERTS CODE HERE
      (apply-keystroke letter mail-agent))))
```

It is a serious problem that, at the specified point in the program, spy agencies cannot read the text of your letter. Accordingly, DHS hires Ben Bitdiddle to add a new construct, `up-env`, to FLK!; the new language is called "Nosy FLK!". `up-env` permits access to shadowed variables. It takes a number n and an identifier, and it looks up the identifier in the nth parent environment of the current one. For example, consider the following DHS-inserted code:

`(up-env 0 letter)`	equivalent to `letter`; returns `"s"`
`(up-env 1 letter)`	returns `"Dear sweetheart ..."`, which is the value of `letter` in the parent environment (in which `mail-agent` is also bound)
`(up-env 2 letter)`	also returns `"Dear sweetheart ..."`, which is the value of `letter` in the outermost environment
`(up-env 3 letter)`	issues an "unbound identifier" error

With the work only half-finished, DHS's hidden microphones overhear Ben Bitdiddle talking about "sexps," and he is taken away as a suspected pornographer. In order to demonstrate your patriotism, you agree to complete work on this language extension.

a. [4 points] Consider the following code. (We use syntactic sugar such as `let` for convenience and clarity.)

```
(let ((n 11))
  (letrec ((fact (let ((n 9))
                   (lambda (n)
                     (print (up-env 1 n))
                     (print (up-env 2 n))
                     (if (= n 0)
                         1
                         (* n (fact (- n 1))))))))
    (let ((n 7))
      (fact 5))))
```

The first two executions of `fact` print a total of 4 values.

(i) [2 points] Assuming static scoping, what are the first four values printed?

Solution: 9, 11, 9, 11

(ii) [2 points] Assuming dynamic scoping, what are the first four values printed?

Solution: 7, 11, 5, 7

b. [15 points]

We modify the denotational semantics of Appendix A to support the new `up-env` form by adding the following new domain:

$$\text{nEnvironment} = \text{Environment} \times \text{nEnvironment}_\perp$$

We also have the following constructor and destructor helper functions:

make-nenv : Environment × $\text{nEnvironment}_\perp$ → nEnvironment
nenv-env : nEnvironment → Environment
nenv-parent : nEnvironment → $\text{nEnvironment}_\perp$

With these domains in hand, one can extend the denotational semantics to handle `up-env`. Note that your new definitions for $\mathcal{E}$ will take an nEnvironment instead of an Environment.

(i) [3 points] Define *nlookup*, which looks up an identifier:
nlookup : (nEnvironment → Identifier → Integer) → Binding
You may use *lookup* as a subroutine.

Solution: There are two possible solutions that permit answering the remainder of the problem. The more efficient solution, which does not copy environments, is
nlookup : nEnvironment → Identifier → Binding

```
= $\lambda\alpha I n$ . if $n > 0$
            then $(lookup\ (nenv\text{-}parent\ \alpha)\ I\ (n-1))$
            else let l = $(lookup\ (nenv\text{-}env\ \alpha)\ I)$
                in if $l \neq$ unbound
                  then $l$
                  else let $p = (nenv\text{-}parent\ \alpha)$
                      in if $p = \perp$
                        then unbound
                        else $(lookup\ p\ I\ 0)$
```

The solution that does copy environments is
nlookup : nEnvironment → Identifier → Binding

```
= $\lambda\alpha I n$ . if $n > 0$
            then $(lookup\ (nenv\text{-}parent\ \alpha)\ I\ (n-1))$
            else $(lookup\ (nenv\text{-}env\ \alpha)\ I)$
```

(ii) [3 points] What is $\mathcal{E}[\![I]\!]$?

Solution: $\mathcal{E}[\![I]\!] = \lambda ek\ .\ (\textit{ensure-bound}\ (\textit{nlookup}\ e\ I\ 0)\ k)$

(iii) [3 points] What is $\mathcal{E}[\![\texttt{(up-env}\ E\ I\texttt{)}]\!]$?

Solution: $\mathcal{E}[\![\texttt{(up-env}\ E\ I\texttt{)}]\!] = \lambda ek\ .\ \mathcal{E}[\![E]\!]\ e\ (\lambda v\ .\ (\textit{with-integer}\ (\lambda n\ .\ (\textit{ensure-bound}\ (\textit{nlookup}\ e\ I\ n)\ k))))$

(iv) [3 points] What is $\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!]$?

Solution: This solution works with the non-environment-copying definition of *nlookup*:

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda ek\ .\ (k\ (\text{Procedure} \mapsto \text{Value}\ \ (\lambda d k'\ .\ \mathcal{E}[\![E]\!]\ (\textit{make-nenv}\ \{\}[I:d]\ e)\ k'))$$

This solution works with the environment-copying definition of *nlookup*:

$$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda ek\ .\ (k\ (\text{Procedure} \mapsto \text{Value}\ \ (\lambda d k'\ .\ \mathcal{E}[\![E]\!]\ (\textit{make-nenv}\ (\textit{nenv-env}\ e)[I:d]\ e)\ k'))$$

(v) [3 points] What is $\mathcal{E}[\![\texttt{(call}\ E_1\ \ E_2\texttt{)}]\!]$?

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

Modify the denotational semantics of Appendix A to support the new `up-env` form. Show any changes that you must make to the denotational semantics; do not mention any domains or rules that remain unchanged.

c. [6 points] Before being dragged away, Ben began to work on a desugaring from Nosy FLK! to FLK!. Unfortunately, most of Ben's notes were confiscated; all that remains is one auxiliary function *fresh-I-n*, the rule for D (in terms of a helper desugaring $\mathcal{D}'$), and one rule from the helper desugaring $\mathcal{D}'$:

fresh-I-n : Identifier $\times$ Integer $\rightarrow$ Identifier
fresh-I-n returns the same unique fresh identifier for each invocation with the same arguments.

$$\begin{aligned} \mathcal{D}[\![E]\!] &= \mathcal{D}'[\![E]\!]\ 0 \qquad \text{for all } E \\ \mathcal{D}'[\![I]\!]\ n &= I \end{aligned}$$

Furthermore, you find a comment that his preliminary solution is quite constrained: the numeric argument to `up-env` must be a literal number; the number must indicate an environment that explicitly binds the identifier; the desugaring assumes static scoping; and the desugaring fails in the presence of side effects. (You may provide a solution better than Ben's, but you are not required to. You must reuse the two existing rules, however.)

Write the rules for the auxiliary desugaring $\mathcal{D}'[\![E]\!]n$ that helps to convert Nosy FLK! to FLK!. (Don't introduce any additional helper functions.) Omit any rules that merely implement the structural induction by applying the desugaring to each subexpression, with the same arguments as were used for the full expression. Assume that both Nosy FLK! and FLK! use call-by-value semantics.

Solution:

$$\begin{aligned} \mathcal{D}'[\![\texttt{(up-env}\ n_1\ \ I\texttt{)}]\!]\ n &= \textit{fresh-I-n}(I, n - n_1) \\ \mathcal{D}'[\![\texttt{(proc}\ I\ \ E\texttt{)}]\!]\ n &= \texttt{(proc}\ I\ \texttt{(call (proc}\ \textit{fresh-I-n}(I,n)\ \ \mathcal{D}'[\![E]\!]\ \ n+1\texttt{)}\ \ I\texttt{))} \end{aligned}$$

A slightly less elegant solution is

$$\begin{aligned} \mathcal{D}'[\![\texttt{(up-env}\ n_1\ \ I\texttt{)}]\!]\ n &= \textit{fresh-I-n}(I, n - n_1) \\ \mathcal{D}'[\![\texttt{(proc}\ I\ \ E\texttt{)}]\!]\ n &= \texttt{(proc (}\textit{fresh-I-n}(I,n)\texttt{)) (proc (I)}\ \mathcal{D}'[\![E]\!]\ \ n+1\texttt{))} \\ \mathcal{D}'[\![\texttt{(call}\ E_1\ \ E_2\texttt{)}]\!]\ n &= \texttt{(call (proc (}I_f\texttt{) (call (call (}\mathcal{D}'[\![E_1]\!]\ \ n\texttt{)}\ I_f\texttt{)}\ I_f\texttt{)) (}\mathcal{D}'[\![E_2]\!]\ \ n\texttt{))} \\ &\quad \text{where } I_f \text{ is fresh} \end{aligned}$$

Problem 5: Pragmatics [17 points]

Space-Savers, Inc., is working on reducing the run-time space requirements of compiled programs written in a **functional** subset of SCHEME. The company is particularly interested an expression with the following form:

```
(let ((f (lambda (b) E)))
  (primop + (call f 1) (call f 2)))
```

where `a` is the only free variable of `(lambda (b) E)`. (You should assume that `a` is bound somewhere in the surrounding context of this example and all subsequent examples.)

The company uses a compiler that is composed of a series of source-to-source transforms. The closure conversion phase of their compiler generates flat closures for all `lambda` expressions. Recall that a flat closure is a tuple containing a code component and the values of all free variables referenced by the code component. Closure conversion produces an expression in which closures are explicitly allocated with the `closure` primop and the values of free variables are explicitly extracted from closures by the `closure-ref` primop:

```
(let ((f (%closure (lambda (.closure.73. b) E_closed)
                   a)))
  (primop + (call-closure f 1) (call-closure f 2)))
```

Here, E_{closed} is a version of E in which every reference to `a` has been replaced by `(%closure-ref .closure.73. 1)`. (Of course, closure conversion also affects `call`s, `lambda`s, and other variable references within E_{closed}.) The `call-closure` construct applies the code component of a closure to the specified arguments, supplying the closure itself as an implicit first argument.

New employee Bud Lojack, recalling a recent 6.821 problem set on partial evaluation, suggests inlining and reducing the calls to `f` in the original example to yield:

```
(primop + [1/b]E [2/b]E)
```

Bud says, "Before my nifty transformation, the program had to allocate a closure for `f`. But the transformed program doesn't even have a definition for `f`, so no closure for `f` is ever allocated! So my transformation reduces the space used by the program."

a. [5 points] Name one way in which Bud's transformation can *increase* the space used by the program.

   Solution: If the expression E is large, then in the example above, the transformed expression is about twice as large as the original expression. Thus the space consumed by the code of the program may increase.

Seasoned hacker Chloe Jour has an idea for reducing the space used by the program. She transforms the sample expression into:

```
(let ((f′ (lambda (a b) E)))
  (primop + (call f′ a 1) (call f′ a 2)))
```

Chloe has transformed the function `f` into the function `f′`, which has the same body as `f`, but takes the free variable `a` of `f` as an explicit argument. She has changed each `call` of `f` into a call of `f′`, adding `a` as an extra argument to the `call`. After closure conversion, this expression becomes:

```
(let ((f′ (%closure (lambda (.closure.37. a b) E_closed′))))
  (primop + (call-closure f′ a 1) (call-closure f′ a 2)))
```

She says, "Notice that the variable `a` is not free in `f′`, so the closure associated with `f′` is smaller than the closure associated with `f`. Therefore my transformation reduces the amount of space *allocated for closures* during execution of the program."

Bud is excited by Chloe's suggestion. He decides to transform every subexpression of his program of the form:

```
(let (($I_{fun}$ (lambda ($I_1$ ... $I_n$) $E_{fun}$)))
   $E_{let}$)
```

into

```
(let (($I_{fun}'$ (lambda ($J_1$ ... $J_k$ $I_1$ ... $I_n$) $E_{fun}$)))
   $E_{let}'$)
```

where:

- $J_1 \ldots J_k$ are the free variables of `(lambda (`I_1 `...` I_n`)` E_{fun}`)`, and
- E_{let}' has been obtained from E_{let} by transforming each occurrence `(call` I_{fun} E_1 `...` E_n`)` into `(call` I_{fun}' J_1 `...` J_k E_1 `...` E_n`)`.

"Of course," Bud notes, "it may be necessary to rename the bound variables of E_{let} in order to avoid capture of the variables $J_1 \ldots J_k$."

He shows his idea to Chloe, who has two objections (even after Bud fixes the potential name capture problem). "First," she says, "this transformation is not semantics-preserving in general."

b. [6 points] Give an expression E_{let} such that Bud's transformation does not preserve the semantics of the following program. (Recall that the language is functional.)

```
(let ((f (lambda (b) (primop + a b))))
  $E_{let}$)
```

Solution: Bud's transformation does not preserve semantics when I_{fun} appears in a non-operator position. For example:

```
(let ((f (lambda (b) (primop + a b))))
  (call (if #t f f) 17))
```

After Bud's transformation this becomes:

```
(let ((f' (lambda (a b) (primop + a b))))
  (call (if #t f f) 17))
```

Because the definition of `f` has been removed by the transformation, `f` has become a free variable.

An even simpler example is:

```
(let ((f (lambda (b) (primop + a b))))
  f)
```

"Second," says Chloe, "even when your transformation *is* semantics-preserving, it may actually *increase* the total amount of memory allocated for closures during the evaluation of a program."

c. [6 points] Give an expression E_{let} such that:

- The semantics of the following expression is preserved by Bud's transformation:

```
(let ((f (lambda (b) (primop + a b))))
  $E_{let}$)
```

- AND the transformed expression allocates more *space for closures* than the original expression.

(Recall that the language is functional.)

Solution: The problem is that while we may reduce the size of the closure for `f`, we can increase the size of closures for functions that call `f`. For example:

```
(let ((f (lambda (b) (primop + a b))))
  (list (lambda () (call f 1)) (lambda () (call f 2))))
```

Here `a` will only appear in the closure for `f`.

After Bud's transformation, we have the following expression:

```
(let ((f (lambda (a b) (primop + a b))))
  (list (lambda () (call f a 1)) (lambda () (call f a 2)))))
```

Now `a` no longer need appear in the closure for `f`, but it must appear in the closures of *both* anonymous functions.

Appendix A: Standard Semantics of FLK!

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

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

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

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

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

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

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

$\textit{first-fresh}$ : Store → Location → Location
 $= \lambda s l \,.$ matching $(\textit{fetch}\ l\ s)$
          $\triangleright$ $(\text{Unassigned} \mapsto \text{Assignment}\ \textit{unassigned})$ $[\!]$ $l$
          $\triangleright$ else $(\textit{first-fresh}\ s\ (\textit{next-location}\ l))$
          endmatching

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

Figure 2: Store helper functions for standard semantics of strict CBV FLK!.

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

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

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

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

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

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

$\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] =$
  $\lambda ek\,.\;\mathcal{E}[\![E_1]\!]\ e$ (*test-boolean* (λb . **if** b **then** $\mathcal{E}[\![E_2]\!]\ e\ k$ **else** $\mathcal{E}[\![E_3]\!]\ e\ k$))

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

$\mathcal{E}[\![\texttt{(cell}\ E\texttt{)}]\!] = \lambda ek\,.\;\mathcal{E}[\![E]\!]\ e\ \ (\lambda vs\,.\;k$ (Location $\mapsto$Value (*fresh-loc* s))
  (*assign* (*fresh-loc* s) v s))

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

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

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

```
E[[(rec I E)]] = λeks . let f = fix_Expressible (λa . E[[E]] [I : (extract-value a)] e top-level-cont s)
                        matching f
                        ▷ (Value ↦Expressible v) ‖ E[[E]] [I : v] e k s
                        ▷ else f
                        endmatching
```

extract-value : Expressible $\rightarrow$ Binding

```
 = λa . matching a
        ▷ (Value ↦Expressible v) ‖ (Denotable ↦Binding v)
        ▷ else ⊥_Binding
        endmatching
```

Figure 3: Valuation clauses for standard semantics of strict CBV FLK!.

Appendix B: Typing Rules for SCHEME/XSP

SCHEME/X Rules

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

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

$$\vdash S : \texttt{string} \qquad [\textit{string}]$$

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

$$A[I\!:\!T] \vdash I : T \qquad [\textit{var}]$$

$$\frac{\forall i\ (A \vdash E_i : T_i)}{A \vdash \texttt{(begin } E_1\ \ldots\ E_n\texttt{)} : T_n} \qquad [\textit{begin}]$$

$$\frac{A \vdash E : T}{A \vdash \texttt{(the } T\ E\texttt{)} : T} \qquad [\textit{the}]$$

$$\frac{A \vdash E_1 : \texttt{bool}\ ;\ A \vdash E_2 : T\ ;\ A \vdash E_3 : T}{A \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : T} \qquad [\textit{if}]$$

$$\frac{A[I_1\!:\!T_1,\ \ldots,\ I_n\!:\!T_n] \vdash E_B : T_B}{A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}\ E_B\texttt{)} : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_B\texttt{)}} \qquad [\lambda]$$

$$\frac{\begin{array}{c} A \vdash E_P : \texttt{(-> (}T_1\ \ldots\ T_n\texttt{)}\ T_B\texttt{)} \\ \forall i\ (A \vdash E_i : T_i) \end{array}}{A \vdash \texttt{(}E_P\ E_1\ \ldots\ E_n\texttt{)} : T_B} \qquad [\textit{call}]$$

$$\frac{\begin{array}{c} \forall i\ (A \vdash E_i : T_i) \\ A[I_1\!:\!T_1,\ \ldots,\ I_n\!:\!T_n] \vdash E_B : T_B \end{array}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_B\texttt{)} : T_B} \qquad [\textit{let}]$$

$$\frac{\begin{array}{c} A' = A[I_1\!:\!T_1,\ \ldots,\ I_n\!:\!T_n] \\ \forall i\ (A' \vdash E_i : T_i) \\ A' \vdash E_B : T_B \end{array}}{A \vdash \texttt{(letrec ((}I_1\ T_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_1\ E_n\texttt{))}\ E_B\texttt{)} : T_B} \qquad [\textit{letrec}]$$

$$\frac{A \vdash (\forall i\ \ [T_i/I_i])E_{body} : T_{body}}{A \vdash \texttt{(tlet ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [\textit{tlet}]$$

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

$$\frac{A \vdash E : \texttt{(recordof}\ \ldots\ \texttt{(}I\ T\texttt{)}\ \ldots\texttt{)}}{A \vdash \texttt{(select } I\ E\texttt{)} : T} \qquad [\textit{select}]$$

$$\frac{A \vdash E : T_E\ ;\ T = \texttt{(oneof}\ \ldots\ \texttt{(}I\ T_E\texttt{)}\ \ldots\texttt{)}}{A \vdash \texttt{(one } T\ I\ E\texttt{)} : T} \qquad [\textit{one}]$$

$$\frac{\begin{array}{c} A \vdash E_{disc} : \texttt{(oneof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))} \\ \forall i\ .\ \exists j\ .\ ((I_i = I_{tag_j}) \wedge (A[I_{val_j}\!:\!T_i] \vdash E_j : T)) \end{array}}{A \vdash \texttt{(tagcase } E_{disc}\ \texttt{(}I_{tag_1}\ I_{val_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}I_{tag_n}\ I_{val_n}\ E_n\texttt{))} : T} \qquad [\textit{tagcase1}]$$

$$\frac{\begin{array}{c} A \vdash E_{disc} : \texttt{(oneof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))} \\ \forall i\ \mid\ (\exists j\ .\ (I_i = I_{tag_j}))\ .\ A[I_{val_j}\!:\!T_i] \vdash E_j : T \\ A \vdash E_{default} : T \end{array}}{A \vdash \texttt{(tagcase } E_{disc}\ \texttt{(}I_{tag_1}\ I_{val_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}I_{tag_n}\ I_{val_n}\ E_n\texttt{) (else } E_{default}\texttt{))} : T} \qquad [\textit{tagcase2}]$$

Rules Introduced by SCHEME/XS to Handle Subtyping

$$T \sqsubseteq T \qquad [\textit{reflexive-}\sqsubseteq]$$

$$\frac{T_1 \sqsubseteq T_2 \ ; \ T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3} \qquad [\textit{transitive-}\sqsubseteq]$$

$$\frac{\begin{array}{c}(T_1 \sqsubseteq T_2)\\(T_2 \sqsubseteq T_1)\end{array}}{T_1 \equiv T_2} \qquad [\equiv]$$

$$\frac{\forall i \, \exists j \, ((I_i = J_j) \wedge (S_j \sqsubseteq T_i))}{\texttt{(recordof (}J_1\ S_1\texttt{)} \ldots \texttt{(}J_m\ S_m\texttt{))} \sqsubseteq \texttt{(recordof (}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}} \qquad [\textit{recordof-}\sqsubseteq]$$

$$\frac{\forall j \, \exists i \, ((J_j = I_i) \wedge (S_j \sqsubseteq S_i))}{\texttt{(oneof (}J_1\ S_1\texttt{)} \ldots \texttt{(}J_m\ S_m\texttt{))} \sqsubseteq \texttt{(oneof (}I_1\ T_1\texttt{)} \ldots \texttt{(}I_n\ T_n\texttt{))}} \qquad [\textit{oneof-}\sqsubseteq]$$

$$\frac{\forall i \, (T_i \sqsubseteq S_i) \ ; \ S_{body} \sqsubseteq T_{body}}{\texttt{(-> (}S_1 \ \ldots S_n\texttt{)}\ S_{body}\texttt{)} \sqsubseteq \texttt{(-> (}T_1 \ \ldots T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\texttt{->-}\sqsubseteq]$$

$$\frac{\forall T \, ([T/I_1]T_1 \sqsubseteq [T/I_2]T_2)}{\texttt{(recof}\ I_1\ T_1\texttt{)} \sqsubseteq \texttt{(recof}\ I_2\ T_2\texttt{)}} \qquad [\textit{recof-}\sqsubseteq]$$

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

$$\frac{\begin{array}{c}A \vdash E : S\\ S \sqsubseteq T\end{array}}{A \vdash \texttt{(the}\ T\ E\texttt{)} : T} \qquad [\textit{the-inclusion}]$$

Rules Introduced by SCHEME/XSP to Handle Polymorphism

$$\frac{\begin{array}{c}A \vdash E : T;\\ \forall i \, (I_i \notin (FTV \ \ (\textit{Free-Ids}[\![E]\!])A))\end{array}}{A \vdash \texttt{(plambda (}I_1 \ \ldots \ I_n\texttt{)}\ E\texttt{)} : \texttt{(poly (}I_1 \ \ldots \ I_n\texttt{)}\ T\texttt{)}} \qquad [p\lambda]$$

$$\frac{A \vdash E : \texttt{(poly (}I_1 \ \ldots \ I_n\texttt{)}\ T_E\texttt{)}}{A \vdash \texttt{(proj}\ E\ T_1 \ \ldots \ T_n\texttt{)} : (\forall i \ \ [T_i/I_i]) \ \ T_E} \qquad [\textit{project}]$$

$$\frac{(\forall i \ \ [I_i/J_i]) \ \ S \sqsubseteq T, \ \ \forall i \, (I_i \notin \textit{Free-Ids}[\![S]\!])}{\texttt{(poly (}J_1 \ \ldots \ J_n\texttt{)}\ S\texttt{)} \sqsubseteq \texttt{(poly (}I_1 \ \ldots \ I_n\texttt{)}\ T\texttt{)}} \qquad [\textit{poly-}\sqsubseteq]$$

recof Equivalence

$$\text{(recof} \ \ I \ \ T) \equiv [(recof \ \ I \ \ T)/I]T$$

Appendix C: Typing Rules for SCHEME/R

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix D: 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)

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

```
R[[(if E_t E_c E_a)]] A S =  let ⟨T_t, S_t⟩ = R[[E_t]] A S
                             in  let S'_t = U(T_t, bool, S_t)
                                 in  let ⟨T_c, S_c⟩ = R[[E_c]] A S'_t
                                     in  let ⟨T_a, S_a⟩ = R[[E_a]] A S_c
                                         in  let S'_a = U(T_c, T_a, S_a)
                                             in ⟨T_a, S'_a⟩
```

```
R[[(lambda (I_1 ... I_n) E_b)]] A S =  let ⟨T_b, S_b⟩ = R[[E_b]] A[I_i : ?v_i] S
                                       in ⟨(-> (?v_1 ... ?v_n) T_b), S_b⟩    (?v_i are new)
```

```
R[[(E_0 E_1 ... E_n)]] A S =  let ⟨T_0, S_0⟩ = R[[E_0]] A S
                              in ...
                               let ⟨T_n, S_n⟩ = R[[E_n]] A S_{n-1}
                               in  let S_f = U(T_0, (-> (T_1 ... T_n) ?v_f), S_n)
                                   in ⟨?v_f, S_f⟩    (?v_f is new)
```

```
R[[(let ((I_1 E_1) ... (I_n E_n)) E_b)]] A S =  let ⟨T_1, S_1⟩ = R[[E_1]] A S
                                                in ...
                                                 let ⟨T_n, S_n⟩ = R[[E_n]] A S_{n-1}
                                                 in R[[E_b]] A[I_i : Rgen(T_i, A, S_n)] S_n
```

```
R[[(letrec ((I_1 E_1) ... (I_n E_n)) E_b)]] A S =  let A_1 = A[I_i : ?v_i]    (?v_i are new)
                                                   in  let ⟨T_1, S_1⟩ = R[[E_1]] A_1 S
                                                       in ...
                                                        let ⟨T_n, S_n⟩ = 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 : Rgen(T_i, A, S_b)] S_b
```

```
R[[(record (I_1 E_1) ... (I_n E_n))]] A S =  let ⟨T_1, S_1⟩ = R[[E_1]] A S
                                             in ...
                                              let ⟨T_n, S_n⟩ = R[[E_n]] A S_{n-1}
                                              in ⟨(recordof (I_1 T_1) ... (I_n T_n)), S_n⟩
```

```
R[[(with (I_1 ... I_n) E_r E_b)]] A S =  let ⟨T_r, S_r⟩ = R[[E_r]] A S
                                         in  let S_b = U(T_r, (recordof (I_1 ?v_i) ... (I_n ?v_n)), S_r)    (?v_i are new)
                                             in R[[E_b]] A[I_i : ?v_i] S_b
```

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