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:** $D[(\text{proc } I \ E)] = (\text{proc } I \ D[(\text{begin (primop cell-set! (cell } I \ I) \ E))])$

For all other $E$, $D[E]$ 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:** $D = \text{integers, with} \leq \text{as the ordering}; E = \text{bool (unordered)}$

c. [3 points] Consider a new construct $\text{smart-compose}$ to FL/R. $(\text{smart-compose } f \ g)$ has the following behavior: it evaluates to whichever of $(\text{compose } f \ g)$ or $(\text{compose } g \ f)$ 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 $\text{smart-compose}$ is reconstructible?

**Solution:** Type reconstruction could fail if both $(\text{compose } f \ g)$ and $(\text{compose } g \ f)$ type check (for instance, if $f$ has type $\alpha \rightarrow \beta$ and $g$ has type $\beta \rightarrow \alpha$, for some concrete $\alpha$ and $\beta$). An expression that uses $(\text{smart-compose } f \ g)$ might require it to have either $\alpha \rightarrow \alpha$ or $\beta \rightarrow \beta$, 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 copying 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:**

- $g: \text{int} \rightarrow \text{int}$
- $f: (\text{int} \rightarrow \text{int}) \times \text{int} \rightarrow (\text{int} \rightarrow \text{int})$
- overall: $((\text{int} \rightarrow \text{int}) \times \text{int} \rightarrow (\text{int} \rightarrow \text{int})) \times (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$
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 FLAVA! expression

```flava
(\x.3)
  (foo (\q. (+ x y)))
(\y. x)
  (z (\x.15) (\q. (z q))))
(\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.

```flava
(\x. (\x. (+ x 1)))
(\y. (\x. y))
(\y. y)
  (\x. (\x. (\x)) x))
  (x 5)))
```

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

**Solution:** 6. Note that (\y. (\x. (\x)) 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, (\y. (\x. (\x)) y)) ... has no effect.
Furthermore, (y (\x. (\x)) x) is evaluated outside the scope of the last (\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:

\[
(+(\text{cwcc} \ (\text{lambda} \ (\text{exit}) \ 100)) \\
(\text{cwcc} \ (\text{lambda} \ (\text{exit}) \ (+2\ (\text{exit}4))))))
\]
\[\Rightarrow 104\]

Solution:

\[
R[\langle \text{cwcc} \ p \rangle] \ A \ S = \\
\text{let } \langle T_r, S_1 \rangle = R[E] \ A \ S \\
\text{in let } S_2 = U(T_r, (-\rightarrow (\rightarrow ?v_r \ ?v_{\text{ignore}}) ?v_r)) \\
\text{in } \langle ?v_r, S_2 \rangle \quad (?v_r \text{ and } ?v_{\text{ignore}} \text{ are new})
\]
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 \((-\rightarrow p (T_1 \ldots T_n) T_b)\), 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

\[
\vdash N : \text{int} \% 1 \\
\vdash B : \text{bool} \% 1
\]

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 \textbf{if}.

Solution:

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

\textbf{[if]}

b. [3 points] Give a type/effect rule for \textbf{lambda}.

Solution:

\[
\frac{A \vdash E_r : (-\rightarrow p_l (T_1 \ldots T_n) T_{\text{body}}) \% p_r}{A \vdash (\lambda (l_1 \ldots l_n) E_r) : \left(\rightarrow p_B (T_1 \ldots T_n) T_B\right) \% 1}
\]

\textbf{[λ]}

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:

\[
\frac{A \vdash E_r : (-\rightarrow p_l (T_1 \ldots T_n) T_{\text{body}}) \% p_r \quad \forall i . \left( A \vdash E_i : T_i \% p_i \right) }{A \vdash (E_r E_1 \ldots E_n) : T_{\text{body}} \% p_r \cdot p_1 \cdot \ldots \cdot p_n}
\]

\textbf{[apply]}

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

\[
\frac{A \vdash E_r : (-\rightarrow p_l (T_1' \ldots T_n') T_{\text{body}}) \% p_r \quad \forall i . \left( A \vdash E_i : T_i' \% p_i \right) \quad \forall i . \left( T_i \sqsubseteq T_i' \right) }{A \vdash (E_r E_1 \ldots E_n) : T_{\text{body}} \% p_r \cdot p_1 \cdot \ldots \cdot p_n}
\]

\textbf{[apply]}
d. [3 points] Give a subtyping rule for procedures by revising the $\rightarrow\sqsubseteq$ rule of Appendix B (page 18).

Solution:

$$\forall i \ (T_i \sqsubseteq S_i) \quad S_{\text{body}} \sqsubseteq T_{\text{body}} \quad p_s \geq p_t$$

$$\vdash (\rightarrow p_s \ (S_1 \ldots S_n) \ S_{\text{body}}) \sqsubseteq (\rightarrow p_t \ (T_1 \ldots T_n) \ T_{\text{body}})$$
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
\[
\begin{align*}
A \vdash E_1 : T \& S_1 & \quad A \vdash E_2 : T \& S_2 \\
\hline
A \vdash (\text{strange } E_1 E_2) : T \& S_1 & \quad \text{AtoN}(S_2)
\end{align*}
\]

where \( \text{strange} \) is taken elementwise and \( \text{AtoN} \) is defined below.

Procedure types have the following form: \((-> \text{Ids} \ S \ (T_1 \ldots T_n) T\)). “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, “\( \text{AtoM}(\text{NtoA}(S)) \)”.

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

For example,
\[
\text{MtoN}(\{ a : M, b : A, c : N, d : M \}) = \{ a : N, b : A, c : N, d : N \}.
\]

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

Solution: \( A \sqsupset M \sqsupset N \)

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

Solution:
\[
\begin{align*}
A \vdash E_1 : \text{bool} & \quad A \vdash E_2 : T \& S_2 & \quad A \vdash E_3 : T \& S_3 \\
\hline
A \vdash (\text{if } E_1 E_2 E_3) : T \& S_1 & \quad \text{AtoM}(S_2) \quad \text{AtoM}(S_3)
\end{align*}
\]
c. [4 points] Give the typing rule for $\lambda$.

**Solution:**

$$A[I_1:T_1, \ldots, I_n:T_n] \vdash E_B : T_B \& S_B$$

$$A \vdash (\lambda (I_1 T_1) \ldots (I_n T_n)) E_B : (\to (I_1 \ldots I_n)) S_B (T_1 \ldots T_n) T_B \& \text{AllN}(S_B)$$

\[\lambda\]

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

**Solution:**

Here is another way to write the same thing:

$$A \vdash E_{rator} : (\to (I_1 \ldots I_n) S_{body} (T'_1 \ldots T'_n) T_{body}) \& S_{rator}$$

$$\forall i. (A \vdash E_i : T_i \& S_i)$$

$$\forall i. (T_i \subseteq T'_i)$$

$$A \vdash (E_{rator} E_1 \ldots E_n) : T_{body} \& S_r (S_b \{I_1, \ldots, I_n\}) \begin{cases} 
\text{if } S_b(i) = A, \text{ then } S_i \\
\text{if } S_b(i) = N, \text{ then } \text{allN}(S_i) \\
\text{if } S_b(i) = M, \text{ then } AtoM(S_i) \\
\end{cases}$$

[apply]

e. [3 points] Give the typing rule for (application of) $\ast$.

**Solution:**

$$\forall i. (A \vdash E_i : \text{int} \& S_i)$$

$$A \vdash (* E_1 \ldots E_n) : \text{int} \& \text{ all } i. S_i$$

[+]
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 \( n \)th parent environment of the current one. For example, consider the following DHS-inserted code:

\[
\begin{align*}
\text{(up-env 0 letter)} & \quad \text{equivalent to letter; returns "s"} \\
\text{(up-env 1 letter)} & \quad \text{returns "Dear sweetheart ...", which is the value of letter} \\
& \quad \text{in the parent environment (in which mail-agent is also bound)} \\
\text{(up-env 2 letter)} & \quad \text{also returns "Dear sweetheart ...", which is the value of letter} \\
& \quad \text{in the outermost environment} \\
\text{(up-env 3 letter)} & \quad \text{issues an “unbound identifier” error}
\end{align*}
\]

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.)

\[
\begin{align*}
\text{(let ((n 11))}
& \quad \text{(letrec ((fact (let ((n 9))}
& \quad \quad \text{(lambda (n)}
& \quad \quad \quad \text{(print (up-env 1 n))}
& \quad \quad \quad \text{(print (up-env 2 n))}
& \quad \quad \quad \text{(if (= n 0)}
& \quad \quad \quad \quad \quad \text{1})
& \quad \quad \quad \quad \quad \text{(* n (fact (- n 1)))))})
& \quad \quad \text{(let ((n 7))}
& \quad \quad \quad \text{(fact 5))}))
\end{align*}
\]

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?

\textbf{Solution}: 9, 11, 9, 11

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

\textbf{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:

- \( \text{make-nenv} : \text{Environment} \times \text{nEnvironment}_\perp \rightarrow \text{nEnvironment} \)
- \( \text{nenv-env} : \text{nEnvironment} \rightarrow \text{Environment} \)
- \( \text{nenv-parent} : \text{nEnvironment} \rightarrow \text{nEnvironment}_\perp \)

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 \( \text{nlookup} \), which looks up an identifier:

\( \text{nlookup} : (\text{nEnvironment} \rightarrow \text{Identifier} \rightarrow \text{Integer}) \rightarrow \text{Binding} \)

You may use \( \text{lookup} \) 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

\[
\text{nlookup} : \text{nEnvironment} \rightarrow \text{Identifier} \rightarrow \text{Binding} \\
= \lambda \alpha \text{In} . \text{if } n > 0 \text{ then } (\text{lookup} \ (\text{nenv-parent} \ \alpha) \ I \ (n - 1)) \text{ else let } l = (\text{lookup} \ (\text{nenv-env} \ \alpha) \ I) \text{ in if } l \neq \text{unbound} \text{ then } l \text{ else let } p = (\text{nenv-parent} \ \alpha) \text{ in if } p = \perp \text{ then } \text{unbound} \text{ else } (\text{lookup} \ p \ I \ 0)
\]

The solution that does copy environments is

\[
\text{nlookup} : \text{nEnvironment} \rightarrow \text{Identifier} \rightarrow \text{Binding} \\
= \lambda \alpha \text{In} . \text{if } n > 0 \text{ then } (\text{lookup} \ (\text{nenv-parent} \ \alpha) \ I \ (n - 1)) \text{ else } (\text{lookup} \ (\text{nenv-env} \ \alpha) \ I)
\]

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

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

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

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

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

**Solution:** This solution works with the non-environment-copying definition of \( \text{nlookup} \):

\[
\mathcal{E}[(\text{proc} \ I \ E)] = \lambda ek . \text{(Procedure} \rightarrow \text{Value} \ (\lambda dk' . \mathcal{E}[E] (\text{make-nenv} \ {} [I : d] e) k'))
\]

This solution works with the environment-copying definition of \( \text{nlookup} \):

\[
\mathcal{E}[(\text{proc} \ I \ E)] = \lambda ek . \text{(Procedure} \rightarrow \text{Value} \ (\lambda dk' . \mathcal{E}[E] (\text{make-nenv} (\text{nenv-env} e)[I : d] e) k'))
\]
(v) [3 points] What is $E[(\text{call } E_1 \ E_2)]$?

**Solution:** $E[(\text{call } E_1 \ E_2)] = \lambda ek . E[E_1] e (\text{test-proc } (\lambda p . E[E_1] e (\lambda v . (p w k))))$

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-l-n, the rule for D (in terms of a helper desugaring $D'$), and one rule from the helper desugaring $D'$:

fresh-l-n : Identifier $\times$ Integer $\rightarrow$ Identifier

fresh-l-n returns the same unique fresh identifier for each invocation with the same arguments.

\[
D[E] = D'[E] 0 \quad \text{for all } E \\
D'[I \ n] = I
\]

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 $D'[E]n$ 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:**

\[
D'[\text{up-env } n_1 I] \ n = \text{fresh-l-n}(I, n - n_1) \\
D'[\text{proc } I E] \ n = (\text{proc } I (\text{call } (\text{proc fresh-l-n}(I, n) D'[E] \ n + 1) I))
\]

A slightly less elegant solution is

\[
D'[\text{up-env } n_1 I] \ n = \text{fresh-l-n}(I, n - n_1) \\
D'[\text{proc } I E] \ n = (\text{proc fresh-l-n}(I, n)) (\text{proc } (I \ D'[E] \ n + 1)) \\
D'[\text{call } E_1 E_2] \ n = (\text{call } (\text{proc } I_f) (\text{call } (\text{call } (D'[E_1] \ n) I_f) I_f)) (D'[E_2] \ n)
\]

where $I_f$ is fresh
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:

\[
\begin{align*}
\text{(let ((f (lambda (b) E)))} \\
&\quad \text{(primop + (call f 1) (call f 2)))}
\end{align*}
\]

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:

\[
\begin{align*}
\text{(let ((f (\%closure (lambda (\cdot closure 73. b) E closed)) a)))} \\
&\quad \text{(primop + (call-closure f 1) (call-closure f 2)))}
\end{align*}
\]

Here, \(E_{closed}\) is a version of \(E\) in which every reference to \(a\) has been replaced by \((\%closure-ref \cdot closure 73. 1)\). (Of course, closure conversion also affects \(\textit{call}\), \(\textit{lambda}\), and other variable references within \(E_{closed}\).) The \textit{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:

\[
\begin{align*}
\text{(primop + \left[\frac{1}{b}\right]E \left[\frac{2}{b}\right]E)}
\end{align*}
\]

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 \textit{increase} the space used by the program.

\textbf{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:

\[
\begin{align*}
\text{(let ((f' (lambda (a b) E)))} \\
&\quad \text{(primop + (call f' a 1) (call f' a 2)))}
\end{align*}
\]

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 \textit{call} of \(f\) into a \textit{call} of \(f'\), adding \(a\) as an extra argument to the call. After closure conversion, this expression becomes:

\[
\begin{align*}
\text{(let ((f' (%closure (lambda (\cdot closure 37. a b) E_{closed}) a)))} \\
&\quad \text{(primop + (call-closure f' a 1) (call-closure f' a 2)))}
\end{align*}
\]

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 \textit{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:
\[
(\text{let } (I_{\text{fun}} \ (\text{lambda} \ (I_1 \ldots I_n) \ E_{\text{fun}})))
\]

into

\[
(\text{let } (I_{\text{fun}}' \ (\text{lambda} \ (J_1 \ldots J_k \ I_1 \ldots I_n) \ E_{\text{fun}})))
\]

where:

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

“Of course,” Bud notes, “it may be necessary to rename the bound variables of \(E_{\text{let}}\) 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 \(E_{\text{let}}\) such that Bud’s transformation does not preserve the semantics of the following program. (Recall that the language is functional.)

\[
(\text{let } ((f \ (\text{lambda} \ (b) \ (\text{primop} + \ a \ b))))
\]

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

\[
(\text{let } ((f \ (\text{lambda} \ (b) \ (\text{primop} + \ a \ b))))
\]

\[
(\text{call} \ (\text{if} \ #\ t \ f \ f) \ 17))
\]

After Bud’s transformation this becomes:

\[
(\text{let } ((f' \ (\text{lambda} \ (a \ b) \ (\text{primop} + \ a \ b))))
\]

\[
(\text{call} \ (\text{if} \ #\ t \ f \ f) \ 17))
\]

Because the definition of \(f\) has been removed by the transformation, \(f\) has become a free variable. An even simpler example is:

\[
(\text{let } ((f \ (\text{lambda} \ (b) \ (\text{primop} + \ a \ b))))
\]

\[
f)
\]

“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 \(E_{\text{let}}\) such that:

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

\[
(\text{let } ((f \ (\text{lambda} \ (b) \ (\text{primop} + \ a \ b))))
\]

\[
E_{\text{let}}
\]
• AND the transformed expression allocates more \textit{space for closures} than the original expression.

(Recall that the language is functional.)

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

\begin{verbatim}
(let ((f (lambda (b) (primop + a b))))
  (list (lambda () (call f 1)) (lambda () (call f 2))))
\end{verbatim}

Here \textit{a} will only appear in the closure for \textit{f}.

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

\begin{verbatim}
(let ((f (lambda (a b) (primop + a b))))
  (list (lambda () (call f a 1)) (lambda () (call f a 2))))
\end{verbatim}

Now \textit{a} no longer need appear in the closure for \textit{f}, but it must appear in the closures of both anonymous functions.
Appendix A: Standard Semantics of FLK!

\[
\begin{align*}
  v &\in \text{Value} = \text{Unit} + \text{Bool} + \text{Int} + \text{Sym} + \text{Pair} + \text{Procedure} + \text{Location} \\
  k &\in \text{Expcont} = \text{Value} \rightarrow \text{Cmdcont} \\
  \gamma &\in \text{Cmdcont} = \text{Store} \rightarrow \text{Expressible} \\
  \text{Expressible} &= (\text{Value} + \text{Error})_\bot \\
  \text{Error} &= \text{Sym} \\
  p &\in \text{Procedure} = \text{Denotable} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont} \\
  d &\in \text{Denotable} = \text{Value} \\
  e &\in \text{Environment} = \text{Identifier} \rightarrow \text{Binding} \\
  \beta &\in \text{Binding} = (\text{Denotable} + \text{Unbound})_\bot \\
  \text{Unbound} &= \{\text{unbound}\} \\
  s &\in \text{Store} = \text{Location} \rightarrow \text{Assignment} \\
  l &\in \text{Location} = \text{Nat} \\
  \alpha &\in \text{Assignment} = (\text{Storable} + \text{Unassigned})_\bot \\
  \sigma &\in \text{Storable} = \text{Value} \\
  \text{Unassigned} &= \{\text{unassigned}\} \\
  \text{top-level-cont} : \text{Expcont} \\
  &= \lambda v . \lambda s . (\text{Value} \mapsto \text{Expressible} v) \\
  \text{error-cont} : \text{Error} \rightarrow \text{Cmdcont} \\
  &= \lambda y . \lambda s . (\text{Error} \mapsto \text{Expressible} y) \\
  \text{empty-env} : \text{Environment} = \lambda I . (\text{Unbound} \mapsto \text{Binding unbound}) \\
  \text{test-boolean} : (\text{Bool} \rightarrow \text{Cmdcont}) \rightarrow \text{Expcont} \\
  &= \lambda f . (\lambda v . \text{matching} v \triangleright (\text{Bool} \mapsto \text{Value} b) \uplus (f b) \\
  &\triangleright \text{else} (\text{error-cont} \text{ non-boolean}) \\
  &\text{endmatching}) \\
\end{align*}
\]

Similarly for:
\[
\begin{align*}
  \text{test-procedure} : (\text{Procedure} \rightarrow \text{Cmdcont}) \rightarrow \text{Expcont} \\
  \text{test-location} : (\text{Location} \rightarrow \text{Cmdcont}) \rightarrow \text{Expcont}
\end{align*}
\]

etc.

\[
\begin{align*}
  \text{ensure-bound} : \text{Binding} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont} \\
  &= \lambda \beta k . \text{matching} \beta \triangleright (\text{Denotable} \mapsto \text{Binding} v) \uplus (k v) \\
  &\triangleright (\text{Unbound} \mapsto \text{Binding unbound}) \uplus (\text{error-cont} \text{ unbound-variable}) \\
  &\text{endmatching} \\
\end{align*}
\]

Similarly for:
\[
\begin{align*}
  \text{ensure-assigned} : \text{Assignment} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont}
\end{align*}
\]

Figure 1: Semantic algebras for standard semantics of strict CBV FLK!
\[
\text{same-location?} : \text{Location} \rightarrow \text{Location} \rightarrow \text{Bool} = \lambda l_1 l_2 . (l_1 =_{\text{Nat}} l_2)
\]
\[
\text{next-location} : \text{Location} \rightarrow \text{Location} = \lambda l . (l +_{\text{Nat}} 1)
\]
\[
\text{empty-store} : \text{Store} = \lambda l . (\text{Unassigned} \mapsto \text{Assignment unassigned})
\]
\[
\text{fetch} : \text{Location} \rightarrow \text{Store} \rightarrow \text{Assignment} = \lambda l s . (s l)
\]
\[
\text{assign} : \text{Location} \rightarrow \text{Storable} \rightarrow \text{Store} \rightarrow \text{Store} = \lambda l \sigma s . \lambda l_2 . \begin{cases} 
\text{Storable} \mapsto \text{Assignment} \sigma & \text{if (same-location? } l_1 l_2) \\
\text{else} & \text{fetch } l_2 s
\end{cases}
\]
\[
\text{fresh-loc} : \text{Store} \rightarrow \text{Location} = \lambda s . (\text{first-fresh } s 0)
\]
\[
\text{first-fresh} : \text{Store} \rightarrow \text{Location} \rightarrow \text{Location} = \lambda s l . \begin{cases} 
\text{matching} (\text{fetch } l s) & \text{if (Unassigned} \mapsto \text{Assignment unassigned}) \parallel l \\
\text{else} & \text{first-fresh } s (\text{next-location } l)
\end{cases}
\end{cases}
\]
\[
\text{lookup} : \text{Environment} \rightarrow \text{Identifier} \rightarrow \text{Binding} = \lambda e I . (e I)
\]

Figure 2: Store helper functions for standard semantics of strict CBV FLK.
$\mathcal{L} : \text{Exp} \to \text{Expressible}$

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

$L : \text{Lit} \to \text{Value}; \text{Defined as usual}$

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

$\mathcal{E}[L] = \lambda e k. \mathcal{E}[E]$

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

$\mathcal{E}[(\text{proc } I \ E)] = \lambda e k. \text{ Procedure } \to \text{Value} (\lambda d . \mathcal{E}[E] \text{ I : d} e k')$

$\mathcal{E}[(\text{call } E_1 \ E_2)] = \lambda e \mathcal{E}[E_1] \text{ e (test-procedure} (\lambda p . \mathcal{E}[E_2] \text{ e (} \lambda v . p v k))$

$\mathcal{E}[(\text{if } E_1 \ E_2 \ E_3)] = \lambda e \mathcal{E}[E_1] \text{ e (test-boolean} \text{ (} \lambda b \text{ if then } \mathcal{E}[E_2] \text{ e k else } \mathcal{E}[E_3] \text{ e k))}$

$\mathcal{E}[(\text{pair } E_1 \ E_2)] = \lambda e \mathcal{E}[E_1] \text{ e (} \lambda v_1 . \mathcal{E}[E_2] \text{ e (} \lambda v_2 . k \text{ (Pair } \to \text{Value} \langle v_1 , v_2 \rangle))$

$\mathcal{E}[(\text{cell } E)] = \lambda e \mathcal{E}[E] \text{ e (} \lambda v s . k \text{ (Location } \to \text{Value} \text{ (fresh-loc } s))$

$\text{ (assign (fresh-loc } s) v s))$

$\mathcal{E}[(\text{begin } E_1 \ E_2)] = \lambda e \mathcal{E}[E_1] \text{ e (} \lambda v \text{ ignore . } \mathcal{E}[E_2] \text{ e k))}$

$\mathcal{E}[(\text{primop cell-ref } E)] = \lambda e \mathcal{E}[E] \text{ e (} \text{test-location } \text{ (} \lambda l s . \text{ ensure-assigned (} \text{fetch } l s) k s))$

$\mathcal{E}[(\text{primop cell-set! } E_1 \ E_2)] = \lambda e \mathcal{E}[E_1] \text{ e (} \text{test-location } \text{ (} \lambda l . \mathcal{E}[E_2] \text{ e (} \lambda v s . k \text{ (Unit } \to \text{Value} \text{ unit}) \text{ (assign } l v s))))$

$\mathcal{E}[(\text{rec } I \ E)] = \lambda e k s . \text{ let } f = \text{fixExpressible} (\lambda a . \mathcal{E}[E] \text{ I : (extract-value a}) \text{ e top-level-cont s) matching f}$

$\text{ endmatching}$

$\text{extract-value : Expressible } \to \text{ Binding}$

$\text{ = } \lambda a . \text{ matching a}$

$\text{ endmatching}$

Figure 3: Valuation clauses for standard semantics of strict CBV FLK!
Appendix B: Typing Rules for SCHEME/XSP

SCHEME/X Rules

\[ \vdash N : \text{int} \]

\[ \vdash B : \text{bool} \]

\[ \vdash S : \text{string} \]

\[ \vdash (\text{symbol } I) : \text{sym} \]

\[ A[I : T] \vdash I : T \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash \text{begin } E_1 \ldots E_n : T_n \]

\[ \vdash E : T \]

\[ \vdash \text{the } T E : T \]

\[ \vdash E_1 : \text{bool} ; \vdash E_2 : T ; \vdash E_3 : T \]

\[ \vdash (\text{if } E_1 E_2 E_3) : T \]

\[ \vdash A[I_1 : T_1, \ldots, I_n : T_n] \vdash E_B : T_B \]

\[ \vdash (\text{lambda } ((I_1 : T_1) \ldots (I_n : T_n)) E_B) : (\rightarrow (T_1 \ldots T_n) T_B) \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash (\text{let } ((I_1 : E_1) \ldots (I_n : E_n)) E_B) : T_B \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash (\text{letrec } ((I_1 : T_1) \ldots (I_n : T_n)) E_B) : T_B \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash (\text{tlet } ((I_1 : T_1) \ldots (I_n : T_n)) E_B) : T_B \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash (\text{record } ((I_1 : E_1) \ldots (I_n : E_n)) : (\text{recordof } (I_1 : T_1) \ldots (I_n : T_n))) \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash (\text{select } I E) : T \]

\[ \vdash E : T_E ; T = (\text{oneof } \ldots (I T_E) \ldots) \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \vdash (\text{oneof } (I_1 : T_1) \ldots (I_n : T_n)) \]

\[ \forall i \exists j ((I_i = I_{tag_j}) \land (A[I_{val_i} : T_i] \vdash E_j : T)) \]

\[ \vdash (\text{tagcase1 } E_{\text{disc}} (I_{tag_1} I_{val_1} E_1) \ldots (I_{tag_n} I_{val_n} E_n)) : T \]

\[ \forall i (A \vdash E_i : T_i) \]

\[ \forall i \exists j ((I_i = I_{tag_j}) \land (A[I_{val_i} : T_i] \vdash E_j : T)) \]

\[ \vdash (\text{tagcase2 } E_{\text{disc}} (I_{tag_1} I_{val_1} E_1) \ldots (I_{tag_n} I_{val_n} E_n) (\text{else } E_{\text{default}})) : T \]
Rules Introduced by SCHEME/XS to Handle Subtyping

\[
\begin{align*}
T & \sqsubseteq T & \text{[reflexive-\sqsubseteq]} \\
T_1 & \sqsubseteq T_2 ; T_2 & \sqsubseteq T_3 & \text{[transitive-\sqsubseteq]} \\
T_1 & \sqsubseteq T_3 & \\
(T_1 & \sqsubseteq T_2) \\
(T_2 & \sqsubseteq T_1) & \\
T_1 & \equiv T_2 & \text{[\equiv]} \\
\forall i \exists j \ ((J_i = J_j) \land (S_j \sqsubseteq T_i)) & \\
\text{(recordof} (J_1 S_1) \ldots (J_m S_m)) & \sqsubseteq \text{(recordof} (I_1 T_1) \ldots (I_n T_n)) & \text{[recordof-\sqsubseteq]} \\
\forall j \exists i \ ((J_j = I_i) \land (S_i \sqsubseteq S_j)) & \\
\text{(oneof} (J_1 S_1) \ldots (J_m S_m)) & \sqsubseteq \text{(oneof} (I_1 T_1) \ldots (I_n T_n)) & \text{[oneof-\sqsubseteq]} \\
\forall i (T_i \sqsubseteq S_i) ; S_{\text{body}} & \sqsubseteq T_{\text{body}} & \\
\forall T ((T/I_1)T_1 \sqsubseteq [T/I_2]T_2) & \\
\text{(recof} I_1 T_1) & \sqsubseteq \text{(recof} I_2 T_2) & \text{[recof-\sqsubseteq]} \\
A \vdash \text{E}rator : (\Rightarrow (T_1 \ldots T_n) T_{\text{body}}) \\
\forall i ((A \vdash E_i : S_i) \land (S_i \sqsubseteq T_i)) & \\
A \vdash (\text{E}rator E_1 \ldots E_n) : T_{\text{body}} & \\
A \vdash E : S \\
S & \sqsubseteq T & \\
A \vdash (\text{the} T E) : T & \text{[the-inclusion]} \\
\end{align*}
\]

Rules Introduced by SCHEME/XSP to Handle Polymorphism

\[
\begin{align*}
A \vdash E : & T; \\
\forall i (I_i \not\in (\text{FTV} \ (\text{Free-Ids}[E])) A) & \\
A \vdash (\text{plambda} (I_1 \ldots I_n) E) : (\text{poly} (I_1 \ldots I_n) T) & \text{[p\lambda]} \\
A \vdash E : (\text{poly} (I_1 \ldots I_n) T_k) & \\
A \vdash (\text{proj} E T_1 \ldots T_n) : (\forall i [T_i/I_i]) T_E & \text{[project]} \\
(\forall i [I_i/J_i]) S \sqsubseteq T, \forall i (I_i \not\in \text{Free-Ids}[S]) & \\
(\text{poly} (I_1 \ldots I_n) S) \sqsubseteq (\text{poly} (I_1 \ldots I_n) T) & \text{[poly-\sqsubseteq]} \\
\text{recof Equivalence} & \\
\text{(recof} I T) & \equiv (\text{recof} I T/I) T & \\
\end{align*}
\]
Appendix C: Typing Rules for SCHEME/R

\[ \vdash \#u : \text{unit} \] \hspace{1cm} [\text{unit}]

\[ \vdash B : \text{bool} \] \hspace{1cm} [\text{bool}]

\[ \vdash N : \text{int} \] \hspace{1cm} [\text{int}]

\[ \vdash (\text{symbol } I) : \text{sym} \] \hspace{1cm} [\text{symbol}]

\[ [..., I : T, ...] \vdash I : T \] \hspace{1cm} [\text{var}]

\[ [..., I : (\text{generic } (I_1, ..., I_n) : \text{T}_{\text{body}}), ...] \vdash I : (\forall [T_i / I_i]) \text{T}_{\text{body}} \] \hspace{1cm} [\text{genvar}]

\[ \begin{align*}
A &\vdash E_{\text{test}} : \text{bool} & A &\vdash E_{\text{con}} : T & A &\vdash E_{\text{alt}} : T \\
\vdash (\text{if } E_{\text{test}} E_{\text{con}} E_{\text{alt}}) : T
\end{align*} \] \hspace{1cm} [\text{if}]

\[ \begin{align*}
A &\vdash \text{lambda } (I_1, ..., I_n : T) \vdash E_{\text{body}} : \text{T}_{\text{body}} \\
\vdash \text{lambda } (I_1, ..., I_n : T) \vdash (\rightarrow (T_1, ..., T_n) : \text{T}_{\text{body}})
\end{align*} \] \hspace{1cm} [\text{\lambda}]

\[ \begin{align*}
A &\vdash E_{\text{rator}} : (\rightarrow (T_1, ..., T_n) : \text{T}_{\text{body}}) \\
\forall i . (A &\vdash E_i : T_i) \\
A &\vdash (E_{\text{rator}} E_1, ..., E_n) : \text{T}_{\text{body}}
\end{align*} \] \hspace{1cm} [\text{apply}]

\[ \begin{align*}
\forall i . (A &\vdash E_i : T_i) \\
A &\vdash \text{letrec } ((I_1, E_1), ..., (I_n, E_n)) : \text{E}_{\text{body}} : \text{T}_{\text{body}} \\
A &\vdash (\text{let rec } ((I_1, E_1), ..., (I_n, E_n)) : \text{E}_{\text{body}}) : \text{T}_{\text{body}}
\end{align*} \] \hspace{1cm} [\text{let}]

\[ \begin{align*}
\forall i . (A &\vdash E_i : T_i) \\
A &\vdash \text{letrec } ((I_1, E_1), ..., (I_n, E_n)) : \text{E}_{\text{body}} : \text{T}_{\text{body}} \\
A &\vdash (\text{let rec } ((I_1, E_1), ..., (I_n, E_n)) : \text{E}_{\text{body}}) : \text{T}_{\text{body}}
\end{align*} \] \hspace{1cm} [\text{letrec}]

\[ \begin{align*}
\forall i . (A &\vdash E_i : T_i) \\
A &\vdash \text{record } ((I_1, E_1), ..., (I_n, E_n)) : \text{(recordof } (I_1, T_1), ..., (I_n, T_n))
\end{align*} \] \hspace{1cm} [\text{record}]

\[ \begin{align*}
A &\vdash E_r : \text{(recordof } (I_1, T_1), ..., (I_n, T_n)) \\
A &\vdash (\text{record } (I_1, T_1), ..., (I_n, T_n)) E_r : T \\
A &\vdash (\text{with } (I_1, ... I_n) E_r E_b) : T
\end{align*} \] \hspace{1cm} [\text{with}]

\[ \begin{align*}
A &\vdash (\text{program } (\text{define } I_1 E_1), ..., (\text{define } I_n E_n)) : \text{E}_{\text{body}} : T
\end{align*} \] \hspace{1cm} [\text{program}]

\[ \text{Gen}(T, A) = (\text{generic } (I_1, ..., I_n) T), \text{where } \{I_i\} = \text{FTV}(T) - \text{FTE}(A) \]
Appendix D: Type Reconstruction Algorithm for SCHEME/R

\[ R[\#u] \] \( A \ S = \langle \text{unit}, S \rangle \)
\[ R[\text{if}] \] \( A \ S = \langle \text{bool}, S \rangle \)
\[ R[\text{let}] \] \( A \ S = \langle \text{int}, S \rangle \)
\[ R[\text{letrec}] \] \( A \ S = \langle \text{sym}, S \rangle \)

\[ R[I] \] \( A[I : T] \) \( S = \langle T, S \rangle \)
\[ R[I] \] \( A[I : (\text{generic } (I_1 \ldots I_n) \ T)] \) \( S = \langle T[?v_i/I_i], S \rangle \) (\( ?v_i \) are new)
\[ R[I] \] \( A \ S = \text{fail} \quad \text{(when } I \text{ is unbound)} \)

\[ R[(\text{if } E_1 \ E_c \ E_a)] \] \( A \ S = \langle \text{unit}, S \rangle \)
\[ \text{in} \ \text{let} \ S' = U(T_2, \text{bool}, S_t) \]
\[ \text{in} \ \text{let} \ (T_c, S_c) = R[E_a] \ A \ S'_t \]
\[ \text{in} \ \text{let} \ (T_a, S_a) = R[E_c] \ A \ S_c \]
\[ \text{in} \ \text{let} \ S'_a = U(T_c, T_a, S_a) \]
\[ \text{in} \ (T_a, S'_a) \]

\[ R[(\text{lambda } (I_1 \ldots I_n) \ E_b)] \] \( A \ S = \langle \text{unit}, S \rangle \)
\[ \text{in} \ \text{let} \ (T_b, S_b) = R[E_b] \ A[I_1 : ?v_i] \ S \]
\[ \text{in} \ (\langle \rightarrow (?v_1 \ldots ?v_n) \ T_b \rangle, S_b) \] (\( ?v_i \) are new)

\[ R[(E_0 \ E_1 \ldots E_n)] \] \( A \ S = \langle \text{unit}, S \rangle \)
\[ \text{in} \ldots \]
\[ \text{let} \ (T_n, S_n) = R[E_n] \ A \ S_{n-1} \]
\[ \text{in} \ \text{let} \ S_f = U(T_0, \langle \rightarrow (T_1 \ldots T_n) \ ?v_f \rangle, S_n) \]
\[ \text{in} \ (\langle ?v_f, S_f \rangle) \] (\( ?v_f \) is new)

\[ R[(\text{let } ((I_1 \ E_1) \ldots (I_n \ E_n)) \ E_b)] \] \( A \ S = \langle \text{unit}, S \rangle \)
\[ \text{in} \ldots \]
\[ \text{let} \ (T_1, S_1) = R[E_1] \ A \ S \]
\[ \text{in} \ldots \]
\[ \text{let} \ (T_n, S_n) = R[E_n] \ A \ S_{n-1} \]
\[ \text{in} \ \text{let} \ S_b = U(?v_i, T_i, S_i) \]
\[ \text{in} \ R[E_b] \ A[\langle ?v_i : \text{Rgen}(T_i, A, S_n) \rangle, S_n] \]

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

\[ R[(\text{record } (I_1 \ E_1) \ldots (I_n \ E_n))] \] \( A \ S = \langle \text{unit}, S \rangle \)
\[ \text{in} \ldots \]
\[ \text{let} \ (T_n, S_n) = R[E_n] \ A \ S_{n-1} \]
\[ \text{in} \ (\langle \text{recordof } (I_1 \ T_i) \ldots (I_n \ T_n) \rangle, S_n) \]

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

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