Programming Languages (CS 329) **Final exam**
Computer Science and Engineering **Books and personal notes permitted**
Indian Institute of Technology Bombay **2001-11-20**

This exam consists of pages 1 through 4. Credit for each question is marked up alongside and the total is given at the end. Use these credits for time management. It is advisable to provide some verbal intuition, because this may earn you partial credit in case the final answer is wrong. The answer to each question must start from a fresh page and be contiguous in your answer book. Clarifications provided during the exam and partial answers to some questions are shown in color.

1. In a language where exceptions are looked up in a dynamic environment and result in termination (as against resumption) of the try block, what is the value of the following expression? [2]

```
(catch (err (y) (+ y 200))
  (let f = (proc (x) (+ 1000 (throw err x)))
    (catch (err (z) (+ 500 z))
      (call f 4) )))
```

The answer is 504. No partial credit.

2. Reconstruct the types of the following expressions, or indicate that the type cannot be reconstructed within the inference-based or procedural methods that we have studied:

   (a) `(proc (g f) (proc (x) (call g (call f x))))` [2]

   $$(\texttt{?b} \to \texttt{?c}) \times (\texttt{?a} \to \texttt{?b}) \quad \to \quad (\texttt{?a} \to \texttt{?c})$$

   Roughly speaking, the steps are to assign free type variable `?a` to `x`, `?b` to `(call f x)`, then `?b`→`?c` to `g`. No partial credit. Mild variants such as obtained by replacing $\times$ with $\to$ are ok.

   (b)
   ```
   (let id = (prox (x) x)
     (call (call id id) 1) )
   ```
   [3]

   The type of this expression *can* be reconstructed, to `int`, obviously. The type of `id` is instantiated in two ways: the operand `id` has type `int` → `int`, whereas the operator `id` has type (`int` → `int`) → (`int` → `int`). One mark each for typing the two `id`s correctly, and one mark for typing the whole expression as `int`. No credit if the claim is that the expression cannot be typed.

3. We will design standard semantics for the `callcc` construct.

   (a) Complete the following equation: [4]

   $$\mathcal{E}[\![\texttt{(callcc } E\texttt{)}]\!] \quad = \quad \lambda u.\lambda k.\Big(\mathcal{E}[\![E]\!]u(\lambda p\underline{\qquad})\Big).$$

   $$\mathcal{E}[\![\texttt{(callcc } E\texttt{)}]\!] \quad = \quad \lambda u.\lambda k.\left(\mathcal{E}[\![E]\!]\ u\ \left(\lambda p\ \boxed{(p\ (\lambda d.\lambda j.\ (k\ d))\ \ k)}\right)\right).$$

   (b) Using your denotation, evaluate [2]

   ```
   (+ 1 (callcc (proc (c) (+ 2 (c 3)))))
   ```

and

```
(+ 1 (callcc (proc (c) (+ 2 3))))
```

The answers are 4 and 6. No partial credit, and no credit if part (a) is wrong. It is easy to guess the answers from the lectures on CPS conversion.

Note that this question is *not* about CPS-conversion.

4. Consider the usual object and method constructs we have defined in class:

$$
\begin{array}{rcl}
E & \texttt{::=} & \ldots \\
 & | & \texttt{(object (method M (I}_a\texttt{)}\ E_b\texttt{)}^*\texttt{)} \\
 & | & \texttt{(callm}\ E_{\mathrm{obj}}\ \texttt{(M}_1\texttt{...M}_n\texttt{) M}_j\ E_{\mathrm{arg}}\texttt{)}
\end{array}
$$

Here `callm` invokes method $\texttt{M}_j \in \{\texttt{M}_1, \ldots, \texttt{M}_n\}$ on E_{obj} which is expected to evaluate to an object with (at least) the n methods $\{\texttt{M}_1, \ldots, \texttt{M}_n\}$ defined for it.

Note that methods may refer to other methods. E.g. the following code

```
(callm
  (object
    (method value (i) (callm this (value inc) inc i))
    (method inc   (j) (+ j 1))
  ) (value inc) value 4 )
```

should evaluate to 5. `this` is a reserved keyword and is instantiated properly by the interpreter $\mathcal{E}$. This question is not about evaluation but about type inferencing, so you don't have to worry about `this`.

Apart from existing type expressions involving `listof` etc., we will introduce a type expression

$$\texttt{(objectof (M (}T_a\texttt{)}\ T_b\texttt{)}^*\texttt{)}$$

(a) Extend the reconstruction function $\mathcal{R}$ to process the `object` construct properly. (Hint: Pay special attention to the specification that methods may call other methods in the same object.) $\boxed{4}$

We illustrate the rule with an object having two methods, extending this to any number of methods is straightforward.

$$
\begin{array}{l}
\mathcal{R}[\![\texttt{(object (method M}_1\ \texttt{(I}_1\texttt{)}\ E_1\texttt{) (method M}_2\ \texttt{(I}_2\texttt{)}\ E_2\texttt{))}]\!]\ A\ S_0 = \\
\quad \text{let } A' = A[\texttt{self:(objectof (M}_1\ \texttt{(}?v_1\texttt{)}\ ?v_1'\texttt{) (M}_2\ \texttt{(}?v_2\texttt{)}\ ?v_2'\texttt{))}] \qquad \bullet \\
\qquad \text{let } \langle T_1, S_1\rangle = \mathcal{R}[\![E_1]\!]\ A'[\texttt{I}_1 : ?v_1]\ S_0 \\
\qquad\quad \text{let } \langle T_2, S_2\rangle = \mathcal{R}[\![E_2]\!]\ A'[\texttt{I}_2 : ?v_2]\ S_1 \\
\qquad\qquad \text{let } S_f = \mathcal{U}\left(\{?v_1' = T_1, ?v_2' = T_2\}, S_2\right) \\
\qquad\qquad\quad \langle \texttt{(objectof (M}_1\ \texttt{(}?v_1\texttt{)}\ ?v_1'\texttt{) (M}_2\ \texttt{(}?v_2\texttt{)}\ v_2'\texttt{))}, S_f\rangle
\end{array}
$$

All v_i and v_i' are freshly generated. Note how the line marked with '•' introduces a type binding for `self` to support methods calling each other. Two marks for this binding and the unification, two for the rest. No other partial marks.

(b) Write the reconstruction rule for `callm`. $\boxed{4}$

$$
\begin{array}{l}
\mathcal{R}[\![\texttt{(callm}\ E_{\mathrm{obj}}\ \texttt{(M}_1\ \texttt{M}_2\texttt{) M}_j\ E_{\mathrm{arg}}\texttt{)}]\!]\ A\ S = \\
\quad \text{let } \langle T_{\mathrm{obj}}, S_{\mathrm{obj}}\rangle = \mathcal{R}[\![E_{\mathrm{obj}}]\!]\ A\ S \\
\qquad \text{let } S'_{\mathrm{obj}} = \mathcal{U}(\{T_{\mathrm{obj}} = \texttt{(objectof (M}_1\ \texttt{(}?v_1\texttt{)}\ ?v_1'\texttt{) (M}_2\ \texttt{(}?v_2\texttt{)}\ v_2'\texttt{))}\}, S_{\mathrm{obj}}) \\
\qquad\quad \text{let } \langle T_{\mathrm{arg}}, S_{\mathrm{arg}}\rangle = \mathcal{R}[\![E_{\mathrm{arg}}]\!]\ A\ S'_{\mathrm{obj}} \\
\qquad\qquad \text{let } S_f = \mathcal{U}(\{T_{\mathrm{arg}} = ?v_j\}, S_{\mathrm{arg}}) \\
\qquad\qquad\quad \langle ?v_j', S_f\rangle
\end{array}
$$

One mark each for (the substance of) the last four lines.

(c) (Extra credit) Why did we need to extend the syntax of `callm` to make the method list explicit? $\boxed{2}$

5. We wish to add a **return** construct to FL of the form `(return` E`)` for use inside the body of `proc`s, with the usual connotation of abandoning the rest of the computation in the body of the `proc` and returning from the procedure with the value of E.

(a) Do the signatures of $\mathcal{E}$ and *Proc* need to change? If so, give the new signatures. $\boxed{2}$

Yes, the problem is easiest to solve by changing the signatures. The signature of $\mathcal{E}$ is changed from

$$\begin{array}{rl} & \text{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{ExpCont} \rightarrow \mathit{Store} \rightarrow \mathit{ExpVal} \\ \text{to} & \text{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{ExpCont} \underline{\rightarrow \mathit{ReturnCont}} \rightarrow \mathit{Store} \rightarrow \mathit{ExpVal}, \\ \text{where} & \mathit{ReturnCont} \quad = \quad \mathit{ExpCont}. \end{array}$$

We also change *Proc* from

$$\begin{array}{rl} & \mathit{DenotVal} \rightarrow \mathit{ExpCont} \rightarrow \mathit{Store} \rightarrow \mathit{ExpVal} \\ \text{to} & \mathit{DenotVal} \rightarrow \mathit{ExpCont} \underline{\rightarrow \mathit{ReturnCont}} \rightarrow \mathit{Store} \rightarrow \mathit{ExpVal} \end{array}$$

The idea is to preserve the call-time continuation of a procedure as we go in and evaluate the body, and to bail out using this *ReturnCont* if a **return** construct is found. We will name *ReturnCont* variables r. One mark for each, no partial marks.

(b) Give the standard denotational semantics for `(proc (I)` E`)`. $\boxed{3}$

$$\mathcal{E}[\![\texttt{(proc (I) } E\texttt{)}]\!] \quad = \quad \lambda u.\lambda k_1.\lambda r_1.\left(k_1 \left(\lambda d.\lambda k_2.\lambda r_2.\, \mathcal{E}[\![E]\!] \boxed{\frac{I \rightarrow d}{u}}\, k_2\, k_2\right)\right)$$

If two continuations are used, the arguments r_1 and r_2 are discarded, and k_2 used as the return continuation for E, we give 2 marks.

(c) Give the standard denotational semantics for `(call` E_1 E_2`)`. $\boxed{2}$

$$\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] \quad = \quad \lambda u.\lambda k_1.\lambda r_1.\ (\mathcal{E}[\![E_1]\!]\ u\ (\lambda p.\ (\mathcal{E}[\![E_2]\!]\ u\ (\lambda d.\ (p\ d\ k_1\ \underline{r_1}))\ r_1))\ r_1)$$

The underlined r_1 is discarded, so anything else should also be fine.

(d) Give the standard denotational semantics for `(return` E`)`. $\boxed{2}$

$$\mathcal{E}[\![\texttt{(return } E\texttt{)}]\!] \quad = \quad \lambda u.\lambda k_1.\lambda r_1.\ (\mathcal{E}[\![E]\!]\ u\ r_1\ r_1)$$

To get credit you must use r_1 as both the *ExpCont* and the *ReturnCont* for E.

6. Write a Prolog program to print all permutations of an input list by filling in the blanks below. (Underscore represents an unnamed variable which unifies with everything but never introduces a binding.) Assume that the input list has no duplicate elements. All 1- and 2-letter symbols are distinct variables. $\boxed{4}$

```
remove(X,[X|Xs],Xs).
remove(X,[Y|Ys],[Y|Zs]) :- remove(X,Ys,Zs).
permuter(Xs, Ys) :- permuter1(Xs, Ys, Ys).
permuter1([], [], []).
permuter1([X|Xs], Ys1, [_|Zs]):- permuter1(Xs, Ys, Zs), remove(X, Ys1, Ys).
```

Is `permuter` reversible w.r.t. `Xs` and `Ys`? Justify your answer. $\boxed{2}$

Yes, it is reversible. Show with a small example why.

7. Euclid's algorithm for finding the GCD of two numbers is well known.

Input: $n > m > 0$
$n_0 \leftarrow n$
$m_0 \leftarrow m$
while $m > 0$ **do**
    $q \leftarrow \lfloor n/m \rfloor$
    $r \leftarrow n - qm$
    $n \leftarrow m$
    $m \leftarrow r$
    Invariant: $\text{GCD}(m, n) = \text{GCD}(m_0, n_0)$
end while
Output: n

(a) Find a loop invariant which will help you prove that the above program correctly computes the GCD of m and n. For convenience, you may introduce new program variables and simple statements which do not change the correct operation of the program. $\boxed{3}$

See the code augmentation and invariant added to the code. Two marks for inserting the above lines (or something equivalent) and one mark for proving the invariant using standard arithmetic.

(b) Also propose a loop variant which is useful for proving termination. $\boxed{1}$

m is a suitable loop variant. No partial credit.

$\boxed{\textbf{Total: 40}}$