Programming Languages (CS 328)
Computer Science and Engineering
Indian Institute of Technology Bombay

Final examination
2000-04-22
(Open notes)

1. In a strictly functional language without locations and assignment, can the expression `(if` E_1 E_1 E_2`)` always be be transformed into `(let ((I` E_1`)) (if I I` E_2`))` for any `I` while preserving the meaning of the program? Justify or give a counterexample. (1 points)

   The transformation is not valid if `I` is free in E_2.

```
(let ((x 4))
    (if (> p q) (> p q) x))
```

```
(let ((x 4))
    (let ((x (> p q)))
        (if x x x) ) )
```

   (No partial credit.)

2. Is the currying transformation from `(lambda (I1 I2)` E`)` to `(lambda (I1) (lambda (I2)` E`))` valid under both static and dynamic scoping? If not give an example program where the transformation changes the computed result. (2 points)

   The transformation is not valid under dynamic scoping;

```
    (let ((x 2))
        ( (lambda (x y) (+ x y)) 1 x ) )
```

   evaluates to `(+ 1 2) = 3` whereas

```
    (let ((x 2))
        ( ( (lambda (x) (lambda (y) (+ x y))) 1 ) x ) )
```

   evaluates to `(+ 1 1) = 2`. (One point for yes/no, one for example.)

3. Suppose `(pairof` T_1 T_2`)` is a type constructor for heterogeneous pairs. Give a type T that makes the following two types structurally equivalent:

   - `(recof s (pairof int (pairof bool s)))`
   - `(pairof int` T`)`.

   Here `(recof s (... s ...))` is a recursive type constructor, e.g. a binary tree may be typed as `(recof tree (pairof tree tree))`. (4 points)

   The first type is an infinite cascade of alternating `int` and `bool`, starting with `int`. For this to match the second type expression, T must therefore be `(recof u (pairof bool (pairof int u)))`. (One point is student understands recursive types, remaning three for correct answer.)

4. Implement a λ-macro `DEC` which decrements positive Church numerals without using recursion, pairs, or their syntactic equivalents. (5 points)

5. We have defined macros `HALF`, `DEC` and `EVEN?`. Using these, we are interested in determining if the binary representation of a number (given as a Church numeral) contains an even or odd number of ones. If we could use recursion directly, we could write this as:

```
EVEN-ONES? ≡ (λ n (IF (ZERO? n)
                      TRUE
                      (IF (EVEN? n)
                          (EVEN-ONES? (HALF n))
                          (ODD-ONES?  (DEC  n)) ) ) )
ODD-ONES?  ≡ (λ n (IF (ZERO? n)
                      FALSE
                      (IF (EVEN? n)
                          (ODD-ONES?  (HALF n))
                          (EVEN-ONES? (DEC  n)) ) ) )
```

We have seen in class that defining

FACT' $\equiv$ (λ f (λ n (IF (ZERO? n) $\bar{1}$ (TIMES n ((f f) (DEC n))))))

and calling `((FACT' FACT') `$\bar{n}$`)` enabled us to implement recursion. Use a similar technique to eliminate recursion from the above procedures. (6 points)

6. In Scheme, `(cons el li)` returns a list formed by pushing an element `el` on to the head of a list `li`. Thus `(car (cons el li)) = el` and `(cdr (cons el li)) = li`.

   Here is a recursive definition of `append` which attaches element `el` to the *tail* of list `li`.

```
(define append (el li)
    (if (null? li)
        (list el) ;; sorry for this bug!
        (cons (car li) (append el (cdr li)) ) ) )
```

   And here is a recursive definition of `reverse` which reverses the contents of a list.

```
(define reverse (li)
    (if (null? li) li (append (car li) (reverse (cdr li)))))
```

   (a) Explain why `(append (car li) (reverse (cdr li)))` cannot be replaced by `(cons (reverse (cdr li)) (car li))`.
   The semantics of `cons` are such that `(cons '(a b) '(c)) = ((a b) c)`, whereas the necessary behavior is to produce `(a b c)`.

   (b) Convert `reverse` to tail-recursive form.

```
(define reverse-tail (in ni)
    (if (null? in) ni
        (reverse-tail (cdr in) (cons (car in) ni)) ) )
(define reverse (in) (reverse-tail in ()))
```

   For example,

```
    (reverse-tail (a b c) ())
=   (reverse-tail (b c) (a))
=   (reverse-tail (c) (b a))
=   (reverse-tail () (c b a)) = (c b a)
```

   (c) Convert `append` to tail-recursive form.

```
(define append-tail (el li) (reverse (cons el (reverse-tail li)) ),
```

   where `reverse` is tail-recursive as above. Note that although there are two calls to `reverse`, the stack space required is always bounded by a constant.

   To save time, you are advised not to perform a formal CPS conversion but to arrive at the result by designing your own customized 'accumulator' or 'continuation'. (1+3+3=7 points)

7. Evaluate the following expressions illustrating the `call/cc` construct. (1+2+2+4=9 points)

   (a) `(+ 1 (call/cc (lambda (k) (+ 2 (k 3)))))`
   The expression evaluates to `(+ 1 3) = 4`.

   (b) Suppose we first `(define r (ref #f))` in the top level store and environment. Next we evaluate

```
(+ 1 (call/cc (lambda (k) (begin (:= r k) (+ 2 (k 3)))) ) )
```

   What is the result?
   The answer is still `4`, but now `r` holds a reference to the continuation

(lambda (kt) (lambda v (kt (+ 1 v)))), where kt has been closed using the top-level continuation.

(c) *After* evaluating the above expression, what will (r 5) evaluate to? How about (+ 3 (r 5))?

It is easy to see that the first answer is (+ 1 5) = 6. Interestingly, the second answer is *not* 9 but *also* 6. This is because r has already been closed with respect to the top-level continuation, so the addition of 3 above has no effect!

(d) Suppose first we evaluate the following two pieces of code:

```
]=> (define foo (ref #f))
]=> (let (( counter (ref 0) ))
      (begin
        (call/cc (lambda (k) (:= foo k)))
        (:= counter (+ 1 (^ counter)))
        counter
      ) )
```

The second expression evaluates to 1.

What will happen if we now repeatedly call (foo 32)?

We will get answers 2, 3, 4, ...

8. The following program EXP(x,n) computes x^n and stores it in y.

$k \leftarrow n$;
$y \leftarrow 1$;
$z \leftarrow x$;
while $k \neq 0$ **do**
  if ODD?(k) **then**
    $k \leftarrow k - 1$;
    $y = y \times z$; // sorry about this!
  else
    $k \leftarrow k/2$;
    $z \leftarrow z \times z$;
  end if
end while

(a) Find the loop invariant P for the while-loop.

$$P \equiv (yx^k = x^n) \tag{1}$$

(b) Show that

$$\{n \geq 0 \wedge x \neq 0\} \quad \texttt{EXP}(x, n) \quad \{y = x^n\}.$$

You must show all steps except perhaps trivial arithmetic inferences.

(4+4=8 points)

9. We augment the FLK! language with an assert expression, of the form (assert E_a E_b). E_a must evaluate to a *Boolean*; if this is true, the meaning of (assert E_a E_b) is the meaning of the body E_b. However, if it is false, the computation immediately stops with the error message "Failed assertion", and no store mutations expressed in E_b should be executed. In either case, E_a must not make any modifications to the store; if such an attempt is made, the message "Illegal assertion" should be printed and the program halted. (1+3+4=8 points)

(a) Are any changes needed in the signature of *ExpCont*? If so describe the new *ExpCont*.

(b) Are any changes needed in the signature of $\mathcal{E}$? If so describe the new $\mathcal{E}$.

(c) Write $\mathcal{E}$ for the (assert E_a E_b) expression, and any other expression required, using standard semantics.