CIS 624 Programming Languages

Spring 1999

Final (100 pts)

Name (please print): ....

Id: ....

1	CPS	10 pts	
2	Jumps in CPS	15 pts	
3	Jumps using call/cc	15 pts	
4	Inheritance	10 pts	
5	Types	35 pts	
6	Higher-Order Functions	15 pts	
	Total	100 pts	

1 Continuation-Passing Style

Convert the following function to continuation-passing style. You may assume that the functions `null?`, `add1`, `max`, `car`, and `cdr` are simple primitives that do not need a continuation argument:

```
(define path
  (lambda (tree)
    (if (null? tree)
        0
        (add1 (max (path (car tree))
                   (path (cdr tree)))))))
```

2 Jumps in Continuation-Passing Style

Here is a portion of an interpreter written in continuation-passing style. The interpreter is missing two clauses for *labels* and *jumps*. The evaluation of an expression of the form `label(labName body)` introduces `labName` as the name of a label to which we can jump during the evaluation of `body`. The evaluation of an expression `jump(labName exp)` proceeds as follows. First we evaluate `exp` yielding some value v. Then if `labName` was introduced by a label expression `label(labName` ..). then we return v as the value of that expression.

For example, the evaluation of:

```
(label z ((lambda (x) (add1 (jump z x))) 3))
```

produces 3.

Complete the missing two clauses in the interpreter. You may assume the existence of functions `extend` and `lookup` that behave as follows:

```
(lookup (extend env x v) x) = x
(lookup (extend env x v) y) = (lookup env y)     if x not equal to y
```

```
(define interp
  (lambda (exp env k)
    (variant-case exp
      [num (n) (k n)]
      [var (x) (k (lookup env x))]
      [lambda (x body) (k (make-closure x body env))]
      [app (e1 e2)
        (interp e1 env (lambda (fun)
        (interp e2 env (lambda (arg)
        (apply-closure fun arg k)))))]
      [label (labName body)









      [jump (labName exp)










      )))
```

3 Jumps using call/cc

Rewrite the interpreter from the last problem using call/cc instead of continuation-passing style. You can assume the existence of the same `extend` and `lookup` functions.

```
(define interp
  (lambda (exp env)
    (variant-case exp
      [num (n) n]
      [var (x) (lookup env x)]
      [lambda (x body) (make-closure x body env)]
      [app (e1 e2)
           (apply-closure (interp e1 env) (interp e2 env))]
      [label (labName body)









      [jump (labName exp)










)))
```

4 Inheritance

What does the following program print?

```
class A {
    int i = 100;
    int m (int n) {
        if (n == 1) return i;
        else return i + m(n-1);
    }
}

class B extends A {
    int i = 1000;
    int m (int n) {
        if (n == 0) return i;
        else return i + super.m(n);
    }
}

class TestInheritance {
    public static void main (String[] args) {
        A x = new B();
        System.out.println(x.m(1));
        System.out.println(x.m(2));
        System.out.println(x.m(3));
        System.out.println(x.m(50));
    }
}
```

5 Types

Let n range over the set of integers and x range over an unspecified set of variables. The syntax of the simply typed λ-calculus is:

$$\begin{array}{lrcl} \text{(Terms)} & M & ::= & n \mid x \mid \lambda x.M \mid MM \\ \text{(Types)} & \tau & ::= & \mathsf{int} \mid \tau \to \tau \end{array}$$

The typing rules are:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$$

- Write a type derivation for the term $((\lambda x.x)\ 3)$.

- The combinator K is defined as:

$$K = \lambda x.\lambda y.x$$

Without actually building the type derivation it is apparent that K has the type $(\tau_1 \to (\tau_2 \to \tau_1))$. Indeed K is a function that takes an argument x of type τ_1, and returns a function $(\lambda y.x)$ which when given an argument y of type τ_2 returns x which was assumed to be of type τ_1.

Give types to the S and I combinators defined below (you do *not* have to show the type derivation):

$$\begin{array}{rcl} S & = & \lambda f.\lambda g.\lambda x.((f\ x)\ (g\ x)) \\ I & = & \lambda x.x \end{array}$$

- What is the type of the expression $((SK)K)$?

- A **let**-expression has the form (**let** $x = M$ **in** N) and the usual semantics. Extend the type system with one rule for **let**-expressions.

- A **letrec**-expression has the form (**letrec** $x = M$ **in** N) and the usual semantics. Extend the type system with one rule for **letrec**-expressions.

- We know that the Y-combinator satisfies the equation:

$$YM = M(YM)$$

Assume that M has type $(\tau \to \tau)$, what type should Y have?

6 Higher-Order Functions

Instead of writing in the convoluted continuation-passing style, one can write in the so-called continuation monad. This is equivalent but looks much more readable.

For example, consider the familiar *factorial* function:

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (let ((v (fact (sub1 n))))
          (* n v)))))
```

Using the continuation monad the function becomes:

```
(define newFactCPS
  (lambda (n)
    (if (zero? n)
        (returnCPS 1)
        (letCPS ((v (newFactCPS (sub1 n))))
          (returnCPS (* n v))))))
```

The only changes to the code were to replace every `let` by `letCPS` and add explicit calls to `returnsCPS`. These new constructs are defined such that:

```
(returnCPS M) == (lambda (k) (k M))
(letCPS ((v M)) N) == (lambda (k) (M (lambda (v) ((N v) k))))
```

Given the following equivalence:

```
   (if M (lambda (k) N) (lambda (k) L))
== (lambda (k) (if M N L))               if k does not occur in M
```

prove that `newFactCPS` is equivalent to the following usual continuation-passing style version of `fact`:

```
(define factCPS
  (lambda (n)
     (lambda (k)
        (if (zero? n)
            (k 1)
            ((factCPS (sub1 n)) (lambda (v) ((n v) k)))))))
```