CS421 Fall 2010 Midterm 1

Thursday, October 7, 2009

Name:	
NetID:	

- You have **75 minutes** to complete this exam.
- This is a **closed-book** exam. You are allowed one 3inch by 5inch card of notes prepared by yourself. This card is **not to be shared**. All other materials, besides pens, pencils and erasers, are to be away.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, you may seek clarification from myself or one of the TAs. You must use a whisper, or write your question out. Speaking out aloud is not allowed.
- Including this cover sheet and rules at the end, there are 9 sheets, 17 pages to the exam, including one blank page for workspace. The exam is printed double sided. Please verify that you have all 17 pages.
- Please write your name and NetID in the spaces above, and also at the top of every sheet.

Problems	Possible Points	Points Earned
1	3	
2	8	
3	11	
4	7	
5	9	
6	10	
7	7	
8	13	
9	13	
10	19	
PreTotal	100	
Extra Credit	10	
PostTotal	110	

CS 421 Midterm 1 **Name:**_____

1. (3 pts total) Suppose that the following code is input one line at a time into OCaml:

```
let h = 5;;
let y = 3.14;;
let cyl_vol r h  = y *.  r *. h;;
let y = 4.0;;
let a = cyl_vol 2.0 1.0;;
let b = cyl_vol (1.0, 1.0);;
let c = cyl_vol  3.0;;
```

For each of **a**, **b**, and **c**, either give what is returned, or give the reason why nothing is returned.

a. (1 pt) Tell what is returned, if anything, for **a**:

Soluiton: 6.28

b. (1 pt) Tell what is returned, if anything, for **b**:

Solution: type error: **cyl_vol** should first be applied to an argument of type **int**, but here is applied to An argument of type **int * int**

c. (1 pt) Tell what is returned, if anything, for **c**:

Solution: A function that computes **fun h -> 3.14 *. 3.0 *. h**

2. (8 pts total) Consider the following OCaml code:

let f = (fun x -> (fun y -> y && x)) in let g = f true in f (g true) (g false)

Below is the same code augmented by having almost every program point instrumented with a print statement. Assuming OCaml's current order of execution, evaluating the argument before evaluating the function in an application, write the sequence of characters printed by the following code (not including the type information for the declaration):

```
let ps s = print_string s;;
let f = (ps "a"; (fun x -> (ps "b"; (fun y -> ((ps "c"; y) && (ps "d"; x)))))) in
let g = ((ps "e"; f) (ps "f"; true)) in
(ps "g";
 (((ps "h"; f)
   ((ps "i"; g) (ps "k"; true))
  )
   (ps "m";
    ((ps "n"; g) (ps "p"; false))
   )
 )
);;
```

Solution: afebgmpnckicdhbc

CS 421 Midterm 1 **Name:**_____

3. (11 pts total) Consider the following OCaml code

```
let x = 5;;
let a = x + 4;;
let f x y = x + y + a;;
let h z  = f (z + 4);;
let a =
  let h x = f a (h 3 x) in
   h (2 * x);;
```

Describe the final environment that results from the execution of the above code if execution is begun in an empty environment. Your answer should be written as a set of bindings of variables to values, with only those bindings visible at the end of the execution present. Your answer should be a precise mathematical answer, with a precise description of values involved in the environment. The update operator (+) and abbreviations should not be used.

Solution: {x → 5,
f → <x → fun y -> x + y + a, {x →5, a → 9}>,
h → <z → f (z + 4), { f → <x → fun y -> x + y + a, {x →5, a → 9}>,
a → 9; x → 5>}>.
a → 44}

4. (7 pts) Write a function **unzip : (’a * ‘b) list -> ’a list * ‘b list** that, when given a list of pairs, returns a pair of lists such that the first list consists of first element of each pair in the given list, and the second list consists of the second element in each pair of the given list. The order of the elements in the returned list should be the same as the order of the corresponding elements in the given list. A sample execution is as follows:

 # let rec unzip l = …
val unzip : ('a * 'b) list -> 'a list * 'b list = <fun>
 # unzip [(1,2); (3,4); (5,6)];;
- : int list * int list = ([1; 3; 5], [2; 4; 6])

Solution:

```
let rec unzip l =
    match l with [] -> ([],[])
     | (a,b) :: rest ->
       (match unzip rest with (flist, slist) -> (a::flist, b::slist));;
```

CS 421 Midterm 1 **Name:**_____

5. (9 pts total)
   a. (5 pts) Write a function **rev_compose: (‘a -> ‘a) list -> ‘a -> ‘a** such that **rev_compose flist** returns the result of composing the functions in **flist** in reverse (left to right) order. The result of applying **rev_compose** to the empty list of functions should be the identity function (that takes an element to itself). The function is required to use (only) tail recursion (no other form of recursion). You may start your code in a manner different from shown below and you may use auxiliary functions of your own so long as the only form of recursion used is tail recursion, but you may **not** use any library functions. Executing your code should give the following behavior:

```
# let rec rev_compose flist x = ... ;;
val rev_compose : ('a -> 'a) list -> 'a -> 'a = <fun>
# rev_compose [(fun x -> 2 * x); (fun y -> y - 1)] 5;;
- : int = 9
```

Solution:

```
let rec rev_compose flist x =
  match flist with [] -> x
   | f::fs -> rev_compose fs (f x)
```

   b. (4 pts) Rewrite **rev_compose** as described above, using

List.fold_left : : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

   but no other library functions and no explicit recursion.

Solution:

```
let rev_compose flist x =
  List.fold_left (fun acc_value -> fun f -> f acc_value) x flist
```

6. (10 pts) Write a function **sum_squarek** that is a complete Continuation Passing Style transformation of the following code:

```
let rec sum_square n =
  if n <= 0 then 0 else (sum_square (n – 1)) + ( n * n);;
```

You may treat **+, *, -, ,<=** all as built-in operators that do not need to be converted to CPS. Your function should have the following type:

sum_squarek : int -> (int -> 'a) -> 'a

Solution:

```
let rec sum_squarek n k =
  if n<= 0 then k 0
  else sum_squarek (n – 1) (fun ss -> k(ss + (n * n)));;
```

CS 421 Midterm 1 **Name:**_____

7. (7 pts). Create an OCaml recursive data type to describe paths on a grid from a fixed start location. A path is either a horizontal move of **n** blocks (positive to the right and negative to the left) together with a remaining path, or it is a vertical move of **m** blocks (positive is up and negative down), again together with a remaining path, or it is the end of the path. Both **n** and **m** are integers.

Soluiton: type path = End | Horizontal of int * path | Vertical of int * path

8. (13 pts total) Consider the following Ocaml recursive data type:

```
type  prop = Atom of string
           | Not of prop
           | Or of prop * prop
           | And of prop * prop
           | Implies of prop * prop
```

This OCaml data type **prop** represents propositional formulae built with connectives of the same name. Write a function **eImp : prop -> prop** that eliminates all instances of implication from a proposition using the fact that $A \Rightarrow B \equiv (\neg A) \vee B$. You may use recursion, auxiliary functions, and library functions from OCaml freely.

Solution:

```
let rec eImp p =
    match p with Atom _ -> p
     | Not a -> Not(eImp a)
     |  Or(a, b) -> Or(eImp a, eImp b)
     |  And(a, b) -> And(eImp a, eImp b)
     |  Implies(a, b) -> Or(Not(eImp a), eImp b);;
```

CS 421 Midterm 1 **Name:**_____

9. (13 pts total) Give a type derivation for the following type judgment:

{x:bool, y:bool } |- (let f = fun x -> (x > 7) && y in if f 2 then x else y) : bool

You may use the attached sheet of typing rules. Label every use of a rule with the rule used. You may abbreviate, but you must define your abbreviations. You may find it useful to break your derivation into pieces. If you do, give names to your pieces, which you may then use in describing the whole.

Solution:

Let V stand for the variable rule, C stand for the constants rule, PO stand for the primitive operations rule, AR stand for the arithmetic relations rule, LC for logical connectives, ITE for the if then else rule, F stand for the function rule, APP for the application rule, L for the let, and LR for the let-rec rule.

Let Γ_1 = **{x:bool, y:bool }**
Let Γ_2 = **{x:bool, y:bool, f: int -> bool}**
Let Γ_3 = **{x:int, y:bool }**

```
----------V    ----------C
Γ3 |- x : int      Γ3 |- 7: int
-------------------------------AR ----------V   ---------------------V  ------------ C
        Γ3 |- (x > 7) : bool   Γ3 |- y :  bool   Γ2 |- f : int  ->bool   Γ2 |-  2 : int
        ---------------------------------------- LC  ---------------------------------APP  ------------V      ---------------V
        Γ3 |- (x > 7)  && y :  bool                    Γ2 |- f 2 : bool            Γ2 |- x : bool     Γ2 |- y  : bool
        ------------------------------------------------F  ------------------------------------------------------------------ITE
         Γ1 |- (fun x -> (x > 7)  && y): int -> bool              Γ2 |- (if f 2 then x else y ) : bool
        ------------------------------------------------------------------------------------------------------------------------L
          {x:bool, y:bool } |- (let f = fun x -> (x > 7)  && y in if f 2 then x else y ) : bool
```

10. (19 pts) Give a type inference for the following type judgment:

{ } |- (let rec f = fun x -> f (x – 1) in f 3 > 0) : α

You may use the attached sheet of typing rules. Label every use of a rule with the rule used. You may abbreviate, but you must define your abbreviations. You may find it useful to break your inference into pieces. If you do, give names to your pieces, which you may then use in describing the whole. Your constraints should be given in a single set of equations. All constraints directly necessary for the validity of the type inference from the rules must be present. Duplicates and identities (x = x) may be removed. Orientation of equations does not matter.

Solution:

Let V stand for the variable rule, C stand for the constants rule, PO stand for the primitive operations rule, AR stand for the arithmetic relations rule, LC for logical connectives, ITE for the if then else rule, F stand for the function rule, APP for the application rule, L for the let, and LR for the let-rec rule.

```
                         ---------------------V --------------------C
                         {x:γ, f:β} |- x : int  {x:γ, f:β} |- 1 : int
--------------------V    ----------------------------------------PO  ----------------------V --------------C
{x:γ, f:β} |- f: ε-> δ   {x:γ, f:β} |- (x – 1) : ε                   {f:β} |-  f : ϕ -> int    {f:β} |-  3: ϕ
----------------------------------------------------APP              -------------------------------------APP  ----------------C
             {x:γ, f:β} |- (f (x – 1)) : δ                               {f:β} |-  f 3: int                   {f:β} |-  0  : int
             ----------------------------------F                         -------------------------------------------------------AR
             {f:β} |- (fun x -> f (x – 1)) : β                                      {f:β} |-  (f 3 > 0 ) : α
             ---------------------------------------------------------------------------------------------LR
                                       { } |- (let rec f = fun x -> f (x – 1) in f 3 > 0 ) : α
```

Constraints: { **β = γ -> δ, β = ε-> δ, γ = int, ε= int, α = bool, β = ϕ -> int, ϕ = int}**

CS 421 Midterm 1 **Name:_____**

11. Extra Credit: (10 pts) Write a function **lastk** that is a complete Continuation Passing Style transformation of the following code:

```
let last p l =
  let rec last_aux p l x =
      match l with [] -> x
      | y::ys -> last_aux p ys (if p y then Some y else x)
  in last_aux p l None;;
```

You may treat **Some** as a built-in operator that does not need to be converted to CPS. All procedures must be put in CPS form. Your function should have the following type:

lastk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> ('a option -> 'b) -> 'b

Solution:

```
let lastk pk l k =
   let rec last_aux pk l x k =
    match l with [] -> k x
     | y::ys -> pk y
       (fun b -> let a = if b then Some y else x in last_aux pk ys a k)
in last_aux pk l None k;;
```

CS 421 Midterm 1 **Name:**_____

Rules for type derivations:

Constants:

$$\frac{}{\Gamma \vdash n : \text{int}} \quad \text{(assuming } n \text{ is an integer constant)}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

Variables:

$$\frac{}{\Gamma \vdash x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Primitive operators ($\oplus \in \{ +, -, *, \text{mod}, \ldots\}$):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}}$$

Relations ($\sim \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \sim e_2 : \text{bool}}$$

Connectives :

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \text{bool}} \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \,||\, e_2 : \text{bool}}$$

If_then_else rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

Function rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

Let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad [x : \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

Let Rec rule:

$$\frac{[x: \tau_1] + \Gamma \vdash e_1 : \tau_1 \quad [x: \tau_1] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$