Indian Institute of Technology Delhi
Department of Computer Science and Engineering

COL226 | Programming Languages | Minor I
February 5, 2018 | 60 minutes | Maximum Marks: 60

Open notes. Write your name, entry number and group at the top of each sheet in the blanks provided. Answer all questions in the space provided, in blue or black ink (no pencils, no red pens). Budget your time according to the marks. Do rough work on separate sheets.

Q0. (8 marks) **Functional Programming.** I hope you have been programming in OCaml. Consider the following data type for finitely-branching trees where every node contains an integer and has finitely many subtrees (the number of subtrees is not necessarily bounded by any $k \geq 0$; lists are used since there isn't a fixed number of subtrees).

```
type int_tree = Node of int * (int_tree list);;
let add (x,y) = x+y;;
```

Write an OCaml program sumtree to compute the sum of all the integers appearing in the nodes of a given tree, preferably using functions such as map and foldl or foldr, and the given function add.

```
(* sumtree: int_tree -> int *)
```

let rec sumtree i ~~$~~ = match i with
Node (n, []) -> n ~~+$~~
| Node (~~x::xs~~ n, x::xs) -> ~~(~~ n + sumtree x) ~~$~~
+ (sumtree ~~(0, xs)~~ (Node 0,xs) ~~$~~

5/8 creates new nodes

(works after a fashion)

Q1. (8 marks) **Type-checking.** Consider the following OCaml program

```
exception UnequalLength;;
let rec zipf f l1 l2 = match (l1, l2) with
                       ([ ], [ ] ) -> [ ]
                       | ([ ], y::ys) -> raise UnequalLength
                       | (x::xs, [ ]) -> raise UnequalLength
                       | (x::xs, y::ys) -> (f(x,y))::(zipf f xs ys);;
```

What is the type of the function zipf? (('b*'c) -> 'd) -> 'b list -> 'c list -> 'd list
(Show your working)

f : α
l_1 : β => 'b list
l_2 : γ => 'c list
output : δ => 'd list

7/8

∴ zipf => $\alpha \to \beta \to \gamma \to \delta$
=> α -> 'b list -> 'c list -> 'd list

f => ('b * 'c) -> 'd. Why?

∴ zipf => (('b * 'c) -> 'd) -> 'b list -> 'c list -> 'd list

Name: Siddhant Shingi Entry: 2016CS10310 Gp: 1

3/10

Q2. (2+8 marks) **Representational Invariants**. It is common to use lists to represent sets (essentially by forgetting the order of elements and the number of copies of an element). Consider the *representational invariant property* stated informally as: "list l represents a set S if whenever $a \in S$ then there is a *unique* suffix list of l (trailing part of l), the first element of which is a." Clearly list representations of sets need not be unique, since both [1;3;2] and [3; 1; 2] are legal representations of $\{1,2,3\}$.

1. How is the empty set {} represented? Argue in one sentence that your representation satisfies the above property.

1

empty set { } can be represntd as [ ]
here the suffix will also be [ ]
So it satisfies the above relation. ✗

2. Given any valid representations list l_1 for set S_1 and list l_2 for set S_2, define an OCaml function union (assuming the elements of S_1, S_2 are the same type), and show that your program satisfies the above representational property:

```
(* union: 'a list -> 'a list -> 'a list *)
let rec union (l1, l2) = match (l1 , l2) with
  ([],[]) -> []
  (x::xs, []) -> union (xs, []) @ [x]
  ([], y::ys) -> union ([], ys) @ [y]
  (x::xs, y) -> union (xs, y) @ [x]
```

2 ✗ horrible code.

⇒ This program satisfies the above representation.
Case ① both list L_1 and L_2 are not empty then it adds only element of x into the union list. and here the new union list follows the representation. as [x] is added to the suffix of the union list.
Case ② L_1 is empty then y is added and here as well y is added at suffix.
Case ③ L_2 is empty then x is added at suffix. ✗

Q3. (4+6+8 marks) **Interpreters and compilers.** Consider extending our language of expressions with *pairs* and the two *projection functions* (giving the first and second element of a pair respectively). The abstract syntax may be coded in OCaml as follows

```
type exp = ... | Pair of exp * exp | First of exp | Second of exp;;
```

1. Consider the mathematical semantics, where expressions are mapped to values in a set *value*, implemented in OCaml as follows:

```
type value = Intval of int | PairVal of value * value;;
```

Extend the definition of the function *eval* defined in class to deal with the (three) new kinds of expressions:

Name: Siddhant Shingi Entry: 2016CS10310 Gp: 1


```
(* eval:  (string -> value) -> exp -> value *)
let rec eval rho e = match e with
      Const(n) -> Intval(n)
      ...
      | Pair(e1, e2) -> PairVal( eval rho e1, eval rho e2)
      | First e1 -> Intval( eval rho e1)
      | Second e1 -> Intval (eval rho e1)
      ;;
```

✓ 2

✗

10/18

2. Let *Answer* (with metavariable a ranging over its typical elements) be the subset of Exp comprising those forms that we wish calculation to return.
Extend the specification of the "*calculates*" relation $\gamma \vdash e \Longrightarrow a \subseteq Table \times Exp \times Answer$:

$$\frac{}{\gamma \vdash \mathsf{Pair}(e_1, e_2) \Longrightarrow \mathrm{PAIR}(a_1, a_2)}$$

✗ What are a_1, a_2?

$$\frac{}{\gamma \vdash \mathsf{First}\ e \Longrightarrow a_1} \qquad \frac{}{\gamma \vdash \mathsf{Second}\ e \Longrightarrow a_2}$$

3. We now extend the type opcode to include opcodes for creating pairs, and performing the two projections (First and Second):

```
type opcode = .... | PAIR | FIRST | SECOND;;
```

Extend the functions compile and execute:

```
(* compile:  exp -> opcode list  *)
let rec  compile  e = match e with
...
| Pair(e1, e2) -> (compile e1) @ (compile e2) @ [PAIR]
| First e1 -> (compile e1) @ [FIRST]
| Second e1 -> (compile e2) @ [SECOND]
;;
```

✓ 4

```
(* execute:  stack*table*(opcode list) -> answer *)
let rec execute (s,gamma, c) = match (s,c) with
  ....
| (n1::n2::s, PAIR::c') -> execute ((n1,n2)::s, gamma, c)
| ((n1,n2)::s, FIRST::c') -> execute (n1::s, gamma, c)
| ((n1,n2)::s, SECOND::c') -> execute (n2::s, gamma, c)
;;
```

✓ 4

Q4. (16 marks) **Substitutions and Σ-homomorphisms.** The following lemma, which is a special case of composition of Σ-homomorphisms, relates meaning-giving functions and substitutions, and how they can be interchanged.

Let Σ be a signature, and $\mathcal{T}_\Sigma(\mathcal{X})$ be the set of terms (or trees). Suppose we are given $u \in \mathcal{T}_\Sigma(\mathcal{X})$. Let $\sigma = \{ \mathtt{x} \mapsto u \}$ be the substitution that maps variable x to the given u (every other variable is mapped to itself). Now let $\mathcal{A} = \langle A, \ldots \rangle$ be any Σ-algebra, and let $\rho \in \mathcal{X} \to A$ be an $\mathcal{A}$-assignment. Suppose

Name: Siddhant Shingi Entry: 2016CS10310 Gp: 1

$\hat{\rho}(u) = a \in A$. Let $\rho_1 = \rho[\mathtt{X} \mapsto a]$, that is the A-assignment function that is identical to ρ everywhere, except that for variable X it returns a.

Prove by *structural induction* that for all $t \in \mathcal{T}_\Sigma(\mathcal{X})$:

$$\hat{\rho}(\mathtt{subst}\ \sigma\ t) = \hat{\rho}_1(t)$$

Base case $ht(t) = 0$: t is of the form $c \in \Sigma^{(0)}$.

$\hat{\rho}(\mathtt{subst}\ \sigma\ c) =$ $\hat{\rho}(c)$ (using the defination of subst σ t) subst σ ~~t~~ t is applied for substit of variables

1

and $\hat{\rho}_1$ ~~(c) is~~ · mapping of $\hat{\rho}_1$ is identical to mapping ρ everywhere except at x $\therefore$ $\hat{\rho}(c) = \hat{\rho}_1(c)$ but wrong reason

Base case $ht(t) = 0$: t is (a node labelled by) a variable.

Subcase t is variable X: $\hat{\rho}(\mathtt{subst}\ \sigma\ \mathtt{X}) =$ $\hat{\rho}(\text{subst } \sigma\ x)$

$= \hat{\rho}(u) = a \in A$

and $\hat{\rho}_1(x) = a \in A$ (defination of ρ_1)

3 $\therefore$ $\hat{\rho}$(~~t~~ subst σ x) $= \hat{\rho}_1(x)$

9/16

Subcase t is variable $\mathtt{Y} \not\equiv \mathtt{X}$: $\hat{\rho}(\mathtt{subst}\ \sigma\ \mathtt{Y}) =$ $\hat{\rho}(y)$ (every other variable is mapped to itself in subst;

2

~~∴~~ and ~~similarly~~ $\hat{\rho}_1(y) = \hat{\rho}(y)$ (defination of ρ_1)

Induction Hypothesis: Assume that

for any height of term say $ht(t) > 0$

$\hat{\rho}$(subst σ ~~x~~ t) $= \hat{\rho}_1(t)$

always holds.

If you've assumed this, what's there to prove?

X

Induction Case $ht(t) = m + 1$: t is of the form $f(t_1, \ldots, t_k)$.

$\hat{\rho}(\mathtt{subst}\ \sigma\ f(t_1, \ldots, t_k)) =$ ~~$\hat{\rho}$(St f(subst σ t_1,~~

In subst σ ~~f(t~~ t

$f(t_1, \ldots, t_k)$ is mapped to f(subst σ t_1, ...

subst σ t_k).

$\therefore$ $\hat{\rho}$(subst σ $f(t_1 \ldots, t_k)$) $=$ $\hat{\rho}$(f(subst σ t_1, . . . , subst σ t_k))

~~and using IH subst σ t_1~~

$=$ f_A($\hat{\rho}$(subst σ t_1), . . . , $\hat{\rho}$(subst σ t_k) Why?

$=$ f_A($\hat{\rho}_1(t_1)$, . . . $\hat{\rho}_1(t_k)$) Why?

3

$=$ $\hat{\rho}_1$($f(t_1, \ldots, t_k)$)

$\therefore$ $\hat{\rho}$(subst σ t) $= \hat{\rho}_1(t)$