Name: Abhinav R. Shripad Entry: 2022CS11596 1

Indian Institute of Technology Delhi
Department of Computer Science and Engineering

COL226 Programming Languages Minor Test
February 29, 2024 120 minutes Maximum Marks: 80

Q0 (4)	Q1 (6)	Q2 (4)	Q3 (12)	Q4 (10)	Q5 (8)	Q6 (10)	Q7 (12)	Q8 (6)	Q9 (8)	Total (80)

Open notes. Write your name, entry number and group at the top of <u>each sheet</u> 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.
Attestation: I agree to abide by the Honour Code of IIT Delhi.

Signature: Abhinav

Q0 (4 marks) **Function Space**. Let $A, B \subseteq \mathcal{U}$ be subsets of universal set $\mathcal{U}$. We define function space

$$[A \rightharpoonup B] = \{f \in \mathcal{U} \rightharpoonup \mathcal{U} \mid \text{if } a \in A \text{ and } (a,b) \in graph(f), \text{ then } b \in B\}$$

i.e, all those functions which, given *any* argument $a \in A$, whenever they return a result $b = f(a)$, it is guaranteed that $b \in B$. Suppose (i) $A' \subseteq A$ and (ii) $B \subseteq B'$. Note the two assumptions (i) and (ii) are not in the same direction of inclusion. **Show that** $[A \rightharpoonup B] \subseteq [A' \rightharpoonup B']$.

[Hint: Take *any* $f \in [A \rightharpoonup B]$ and show that $f \in [A' \rightharpoonup B']$, using the definition of function space.]

PROOF. let $f \in [A \to B]$, $\to$ if $a \in A$ and $(a,b) \in graph(f)$ $\longrightarrow b \in B$ ...... (I)

Consider arbitrary $x \in A'$, since $A' \subseteq A \to x \in A$,
If $(x,y) \in graph(f)$ for some $y \to y \in B$ (from (I))
Since $B \subseteq B' \to y \in B'$. $\to x \in A'$ and $(x,y) \in graph(f)$
$\to y \in B'$

Since f and x are arbitrary, $[A \to B] \subseteq [A' \to B']$

Q1 (2+4 = 6 marks) **Σ-homomorphisms**. Let Σ be any signature, and let $\mathcal{A} = \langle A, \ldots \rangle$, $\mathcal{B} = \langle B, \ldots \rangle$, $\mathcal{C} = \langle C, \ldots \rangle$ be Σ-algebras. Recall that a function $h : A \to B$ is a Σ-homomorphism from $\mathcal{A}$ to $\mathcal{B}$ if for each k-ary symbol $f \in \Sigma$ ($k \geq 0$), for all $a_1, \ldots, a_k \in A$, $h(f_{\mathcal{A}}(a_1, \ldots, a_k)) = f_{\mathcal{B}}(h(a_1), \ldots, h(a_k))$.

(a) Show that the identity function $id_A : A \to A$ is a Σ-homomorphism.

case 1:- Consider a 0-ary symbol, say c_A in A $\to id_A(c_A) = c_A$ (id is Identity)
$= c_A \in B = A$

case 2:- Consider a k-ary symbol, $k > 0$ say f, arbitrary $a_1, a_2 \cdots a_k \in A$
$id_A(f_A(a_1, a_2 \cdots a_k)) = f_A(a_1, a_2 \cdots a_k) = f(id_A(a_1), id_A(a_2) \cdots)$

$\to$ 1st equality because id is identity (applied on f)
$\to$ 2nd because id is ~~identity~~, applied on a_i
$\to$ Since c, f are arbitrary
$\to$ homomorphism

(b) Let $h_1 : A \to B$ and $h_2 : B \to C$ be Σ-homomorphisms from $\mathcal{A}$ to $\mathcal{B}$ and from $\mathcal{B}$ to $\mathcal{C}$ respectively. Then, $h_1; h_2 : A \to C$, defined as $(h_1; h_2)(a) = h_2(h_1(a))$, is Σ-homomorphism from $\mathcal{A}$ to $\mathcal{C}$.

Consider a 0-ary Symbol $c_A \in A$,
$\to \underbrace{h_2(h_1(c_A)) = h_2(c_B)}_{h_1 \text{ is } \Sigma \text{homomorphism } A \to B} = c_C$ $\hookrightarrow h_2$ is homomorphism $B \to C$

Consider a k-ary symbol $f_A \in A$, arbitrary $a_1, a_2 \cdots a_k \in A$
$\to h_2(h_1(f_A(a_1, a_2 \cdots a_k))) = h_2(f_B(h_1(a_1), \cdots h_1(a_k)))$ (h_1 is Σ homo... $A \to B$)
$= f_C(h_2(h_1(a_1)), \cdots h_2(h_1(a_k)))$ $\hookrightarrow h_2$ is Σ homo $B \to C$

since c, f, a_i are arbitrary, $h_1; h_2$ is Σ homo $A \to C$

Q2 (4 marks) **Enumerated Types.**

Many languages allow users to define *enumerated types*, that is a type consisting of a finite number $n > 0$ of distinct symbolic values represented by *constructors*. For example,

```
type primary_colour = Red | Blue | Green
```

The general form is: **type** *enum* $=$ $C_1 \mid \ldots \mid C_n$ where each constructor C_i in the enumerated type *enum* is a canonical value/answer. Associated with the enumerated type is a case-analysis expression (like **match** in OCaml or **switch** in C),

$$\textbf{case } e_0 \textbf{ of } C_1 \Rightarrow e_1 \mid \ldots \mid C_n \Rightarrow e_n \textbf{ esac}$$

which first evaluates/calculates the value/answer of expression e_0 which is of type *enum*, and if it calculates to constructor C_i, then returns the value/answer of the corresponding expression e_i. All the expressions e_i $(1 \leq i \leq n)$ should be of the same type.

Provide the reference denotational semantics for type *enum*. First, we extend the set of Values such that $\{C_1, \ldots, C_n\} \subseteq \mathbf{V}$. Now complete the definition for the *eval* function for expressions involving type *enum* appropriately. Mention any required side conditions.

$$eval[\![C_i]\!]\rho \;=\; \cancel{eval[[e_i]]\rho}$$

$$eval[\![\textbf{case } e_0 \textbf{ of } C_1 \Rightarrow e_1 \mid \ldots \mid C_n \Rightarrow e_n \textbf{ esac}]\!]\rho \;=\; eval[[e_i]]\rho \text{ if } eval[[e_0]]\rho = eval[[C_i]]\rho \;\; \cancel{= eval[[}$$

Q3 (4+8 = 12 marks) **Typing Rules for Enumerated Types.**

(a) Complete the type-checking rules for expressions involving type *enum*:

$$\frac{T \vdash e_0 : \tau}{\Gamma \vdash C_i : \tau} \qquad \frac{T \vdash e_1 : \tau_1 \quad T \vdash e_2 : \tau_1 \quad \cdots \quad T \vdash e_3 \vdash \tau_1}{\Gamma \vdash \textbf{case } e_0 \textbf{ of } C_1 \Rightarrow e_1 \mid \ldots \mid C_n \Rightarrow e_n \textbf{ esac} : \tau_1}$$

(b) Extend the Prolog program predicate `hastype`$(Gamma, E, T)$ by translating the rules above. You should use Prolog terms `c1`, ..., to represent the constructors C_i; and the Prolog term `case`$(E0, L)$, where L is a list of pairs $(\mathtt{c}_i, E_i)$, where $(1 \leq i \leq n)$, to represent the case analysis expression. Let the enumerate type *enum* be represented by the Prolog atom `enum`. [Note: You may wish to define a predicate `allhavetype`$(Gamma, L, T)$ to check that all the E_i in L have the *same* type T. Only provide the new clauses; do not reproduce the earlier clauses for other kinds of expressions.]

```
hastype(Gamma, case(E0, L), T) :- hastype(Gamma, E0, enum),
                                   allhavetype(Gamama, L, T).

allhavetype(Gamma, [(c, E) | R], T) :- hastype(Gamma, E, T),
                                        hastype(Gamma, c, enum),
                                        hastype(Gamma, R, T).

allhavetype(Gamma, [], T) :- !.
```

Name: Abhinav Entry: 2022CS11596 3

Q4 (4+6=10 marks) **Operational Semantics for Enumerated Types**

(a) From the informal description and denotational semantics, provide Big-step (Natural) Operational semantic rules for the constructors of an enumerated type and for case-analysis expressions:

$$\frac{}{\gamma \vdash C_i \Longrightarrow_e A} \qquad \frac{\gamma \vdash e_0 \Longrightarrow C_i,\ \gamma \vdash e_i \Longrightarrow B}{\gamma \vdash \textbf{case } e_0 \textbf{ of } C_1 \Rightarrow e_1 \mid \ldots \mid C_n \Rightarrow e_n \textbf{ esac} \Longrightarrow_e B}$$

(b) Extend the Prolog program calculate($Table, E, A$), by translating the above big-step rules. Again, use Prolog terms c1, ..., to represent the constructors; and the Prolog term case($E0, L$), where L is a list of pairs (c_i, E_i) ($1 \leq i \leq n$) to represent the case analysis expression. Only provide the new clauses; do not reproduce the old clauses. You may use the following predicate that, given a constructor C_i, returns the matching expression E_i from the list L:

```
find(Ci, [(Ci, Ei)|_], Ei) :- !.
find(Ci, [(Cj, Ej)| L1], Ei) :- find(Ci, L1, Ei).
find(Ci, [], Ei) :- fail.
```

calculate(Table, case(E0, L), A) :- find(E0, L, B),
calculate(Table, B, A).

Q5 (8 marks) **Type Preservation** Using the typing and big-step rules you have provided in Q3 and Q4, show that the Type Preservation Theorem

For all type assumptions Γ and tables γ such that γ is type-consistent with Γ, for all expressions e, for all types τ, for all answers a: if $\Gamma \vdash e : \tau$ and $\gamma \vdash e \Longrightarrow_e a$, then $\Gamma \vdash a : \tau$

continues to hold. You only need to indicate the new cases, whether base cases or induction cases.

New Base cases

$e \equiv$ Case / enum

Induction Hypothesis: (Same as before.)

New Induction cases:

if $e \equiv$ Case(E_0, L), assume that $\Gamma \vdash e : \tau$, $\gamma \vdash e \Rightarrow a$

$\exists$ exist a pair $(c, a) \in L$

such that $E_0 = c$, thus $\Gamma \vdash e \Rightarrow a$,

Since γ is assumed to be type consistent with Γ

$\Gamma \vdash e : \tau$, $\gamma \vdash e \Rightarrow a \rightarrow \Gamma \vdash a : \tau$

Name: Abhinav Entry: 2022CS11596

Q6 (2+1+4+5=10 marks) **Stack Machine.**

(a) Create new opcodes for the Stack machine for *loading* the constructors of an enumerated type, and for conditional *case analysis*, and extend the definition of type opcode (only write the new opcodes).

`type opcode = ....` | CASE of ~~answer list~~ (enum * answer) list

(b) Mention the *new* forms of answers for such expressions:

$a \in Answer ::= \ldots$ | Answer list

(c) Then present rules for *compiling the new expressions* into op-code sequences.

$compile(C_i) =$

$compile(\textbf{case}\ e_0\ \textbf{of}\ C_1 \Rightarrow e_1 \mid \ldots \mid C_n \Rightarrow e_n\ \textbf{esac}) =$ ~~(compile e0) @ [list map compile~~ CASE[

(compile e0)@[CASE](con (C1, compile e1), (C2, compile e2) ......)

(d) Finally present the new stack machine *execution rules for only these two new op-codes* by extending the program stkmc:

```
let rec stkmc g s c = match s, c with
:    :    :    :    :    :    (* older cases elided *)
```

|(Enum E0) :: s', CASE[(C1, compile e1) ....] :: c'

~~→ stkmc g (find (E0, [C1, compile e1] ...))~~

→ stkmc g s' (find(E0, (C1, compile e1) . (......))) :: c'

Q7 (2+3+3+2+2=12 marks) **Definitions and Pattern matching.** OCaml permits the left hand sides of definitions to be tuple patterns instead of merely variables (in fact, they can be even more complex patterns). For example, one can write `let (x,y) = (3*17, not true);` which binds variable `x` to the answer `51` and `y` to the answer `false`. The generalised form of simple definitions is now **def** $p \triangleq e$, where

$$p \in Pat ::= x \mid (x_1, \ldots, x_n) \quad \text{where all the } x_i \text{ are distinct.}$$

(a) Extend the function *dv* of defined variables, to include the new definition form (mention any required conditions):

$dv(\textbf{def}\ (x_1, \ldots, x_n) \triangleq e) =$ dv(x1) U dv(x2) .... U dv(xn)

(b) Complete the static semantic (***typing***) rule for this new kind of definition, assuming the variables in the patterns are all distinct.

$$\frac{T \vdash def(x_1) :> T_1 \quad T \vdash def(x_2) :> T_2 \cdots\cdots}{\Gamma \vdash \mathbf{def}\ (x_1, \ldots, x_n) \triangleq e :> T_1 \times T_2 \times T_3 \cdots \times T_n}$$

(c) Provide Big-step (Natural) dynamic semantics (calculation rules) for these new form of definitions.

$$\frac{\gamma \vdash def(x_1) \approx\!> \gamma_1 \cdots\cdots \gamma \vdash def(x_n) \approx\!> \gamma_n}{\gamma \vdash \mathbf{def}\ (x_1, \ldots, x_n) \triangleq e \approx\!> \gamma_1 \times \gamma_2 \times \gamma_3 \cdots \gamma_n}$$

(d) How can one encode the projection operator $\mathbf{proj}_i^{(n)}$ which extracts the i^{th} component of an n-tuple expression using this generalized form of pattern matched definitions? (Assume $1 \leq i \leq n$)

$proj^1_n(x_1, x_2 \cdots x_n) = x_1$

$proj^i_n(x_1, x_2 \cdots x_n) = proj^{i-1}_{n-1}(x_2, x_3 \cdots\cdots x_n)$

(e) How can one encode parallel definition ($\mathbf{def}\ x \triangleq e_1 \parallel \mathbf{def}\ y \triangleq e_2$) using this tuple-definition form?

$def\ (x, y) \triangleq (e_1, e_2)$

Q8 (6 marks) **Generalised Substitution Lemma.** Recall that the function $eval[\![e]\!]\rho$ is nothing but $\widehat{\rho}(e)$, i.e., the application to expression e of the valuation ρ's unique homomorphic extension (UHE). Likewise, substitution defined as `subst` σ e is $\widehat{\sigma}(e)$, i.e., the UHE of σ applied to e. Let Σ be any signature, and consider the following diagram.

$$\begin{tikzcd} \mathcal{T}_\Sigma(\mathcal{X}) \arrow[r, "\hat{\sigma}"] & \mathcal{T}_\Sigma(\mathcal{X}) \arrow[r, "\hat{\rho}"] & \mathcal{A} \\ \mathcal{X} \arrow[ur, "\sigma"'] & \mathcal{X} \arrow[ur, "\rho"'] & \end{tikzcd}$$

where $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{A}$ are Σ-algebras. We can talk of the *Kleisli composition* of σ with ρ as the $\mathcal{A}$-valuation $\rho_1 = (\sigma; \widehat{\rho}) : \mathcal{X} \to \mathcal{A}$, i.e., $\rho_1(x) = \widehat{\rho}(\sigma(x))$ for any variable $x \in \mathcal{X}$. Prove for any expression e that

$$eval[\![\ subst\ \sigma\ e\ ]\!]\rho \ = \ eval[\![\ e\ ]\!]\rho_1$$

[Hint: Use the Unique Homomorphic Extension Theorem and the results from Q1.]

We use induction on height of e, Base case $ht(e) = 0$

$\rightarrow e \equiv x \in X \rightarrow subst\ \sigma\ e = \underbrace{\hat{\sigma}(e)}_{\text{given}} = \sigma(e)$ (Extension of σ)

$\rightarrow eval[[subst\ \sigma\ e]]\rho = \hat{\rho}(\underbrace{\hat{\sigma}(e)}_{\text{shown above}}) = \hat{\rho}(\sigma(e)) = eval[[e]]\rho_1$

$\rightarrow e = {}^{\circ}c$ in $Tree_\Sigma(X) \rightarrow subst\ \sigma\ e = e$ (e has no variable to be substituted)

and $\hat{\rho_1}(c) = c_A$ (extension of ρ_1)

$\rightarrow eval[subst\ \sigma\ e]]\rho = eval[[e]]\rho_1$

Inductive Hypothesis, assume true for height $= 0, 1, 2 \cdots k$

for height$(e) = k+1$, e must be some k-ary symbol f in Σ, $(k > 0)$

$\rightarrow \hat{\rho}(\hat{\sigma}(f_A(a_1, a_2 \cdots a_k)) \underbrace{=}_{\hat{\sigma} \text{ is homomorphism}} \hat{\rho}(f_A(\hat{\sigma}(a_1), \hat{\sigma}(a_2) \cdots)) = \hat{\rho}(\sigma(f_A(\cdots))$

$\rightarrow$ Thus $eval[[subst\ \sigma\ e]]\rho = eval[[e]]\rho_1$

Name: ~~D~~ Abhinav R. Sripad Entry: 2022CS11596

Q9 (3+5=8 marks) **Grammars**.

(a) Consider the following grammar for binary numerals.

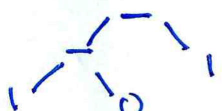
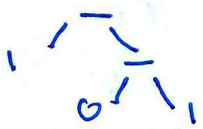
```
<bn> ::= 0 | 1 | <bn> <bn>
```

(i) Identify the start, terminal, and non-terminal symbols in this grammar.

Start → 0, 1 , Terminal → 0, 1 , Non-Terminal = —

(ii) Is this grammar unambiguous? Justify your answer.

Ambigious → consider ~~100~~ 1 _ 0 _ 1

→  and  2 different AST for same expression, ambigious

(b) Recall that we can extend the syntax of expressions to include definitions, i.e., let-expressions of the form `let def ... in` e `ni`. Provide an *unambiguous* context-free grammar (CFG) for expressions *qualified by* definitions. You will need to consider the following cases:

- Simple definitions of the form `def` $x = e$, where x is an identifier (an alphanumeric string starting with a small letter)
- Sequential composition of definitions: d_1 ; d_2
- Parallel composition of definitions: $d_1 \| d_2$ [Note that by convention sequential composition binds tighter than parallel composition, and otherwise explicit parentheses "(" and ")" are used]
- Definitions qualified with a local definition: `local` d_1 `in` d_2 `ni`

Assume that you have an unambiguous CFG for generating expressions. Write out the grammar production rules corresponding to the only the above cases for *definitions*.

Start → S → D | L

D → def X = S | D; def X = S | D || def X = S

L → local S in S ni

X → [a-z] Y

Y → ε | [a-z] Y | [A-Z] Y | [0-9] Y