Major Exam

Graded

Student

Abhinav Shripad

Total Points

55 / 100 pts

Question 1

(no title)

Resolved 6 / 7 pts

- ✔ **+ 1 pt** empty? (create ()) = true
- ✔ **+ 1 pt** empty? (add (x,y,) t) = false
- ✔ **+ 1 pt** present? x (create()) = false
- ✔ **+ 1 pt** present? x (add (x,y,) t) = true
- ✔ **+ 1 pt** lookup x (add (x,y) t)) = y

+ 2 pts add (x_1, y_1) (add (x_2, y_2) t) =
add (x_2, y_2) (add (x_1, y_1) t)

+ 1 pt BONUS: add (x_1, y_1) (add (x_1, y_1) t) =
add (x_1, y_1) t

− 1 pt Law that make the Abstract Theory inconsistent or wrong

+ 0 pts Blank/Incorrect

+ 1 pt Point adjustment

Regrade Request **Submitted on: May 11**

Sir i wrote empty?(add(x,y) t) = false. It is at the last line of my answer.

closing

Reviewed on: May 11

Question 2

(no title)

5 / 5 pts

- ✔ **+ 1 pt** a) MGU does not exist
- ✔ **+ 1.5 pts** a) Correct reasoning
- ✔ **+ 2.5 pts** b) MGU: Z -> h(X, a), Y-> h(b, X)
- **+ 0 pts** Incorrect/Not attempted

Question 3

(no title)

8.5 / 15 pts

- ✔ **+ 0 pts** Blank, Incomplete or incorrect
- **+ 2 pts** Part a: Step 1: On *L*, run program *gcc* (in native code) on input *p2c.c* (written in C) to get output *p2c.o* (*p2c* compiler in native code)
- **+ 2 pts** Part a: Step 2: On *L*, run program *p2c.o* (in native code) on input file *foo.p* (written in Pascal) to get output *foo.c* (program *foo* in C)
- **+ 1.5 pts** Part a: Step 3: on *L*: run program *gcc* (in native code) on input file *foo.c* (in C) to get output *foo.o* (program foo in native code)
- **+ 0.5 pts** Part a: Step 4: On L: run program foo.o (in native code) on input file *input* to send output to file *output*
- ✔ **+ 4 pts** Part b: Step 1: On *W*, run BI_W (in Wintel native code) on inputs *jbc.b* (in B byte code) and secondary input file *jc.j* (written in *j*) to get output *jc.b* (in B byte code)
- ✔ **+ 4 pts** Part b: Step 2: On *W*, run BI_W (in Wintel native code) on inputs *jc.b* (in B byte code) and secondary input file *jc.j* (written in *j*) to get output *jc.m* (in Macos native code)
- **+ 1 pt** Part b: Step 3: Transfer and install file *jc.m* onto machine *M*; you can now compile any *j* program on machine *M*.
- **+ 0.5 pts** part a wrong, part b ok. missing last step...token deduction.

Question 4

(no title) **10** / 10 pts

+ 0 pts Incorrect

✔ **+ 1 pt** $\Gamma \Vdash \text{skip}\checkmark$

Assignment

✔ **+ 0.5 pts** $\Gamma \vdash e : \tau_1$

✔ **+ 0.5 pts** $\Gamma \vdash x : \tau_2$

✔ **+ 1 pt** $\tau_1 \preceq \tau_2$

Seq

✔ **+ 1 pt** $\Gamma \Vdash c_1\checkmark$

✔ **+ 1 pt** $\Gamma \Vdash c_2\checkmark$

Conditional

✔ **+ 0.5 pts** $\Gamma \vdash e : \tau$

✔ **+ 1 pt** $\tau \preceq \text{bool}$

✔ **+ 1.5 pts** $\Gamma \Vdash c_1\checkmark$ and $\Gamma \Vdash c_2\checkmark$

Looping

✔ **+ 0.5 pts** $\Gamma \vdash e : \tau$

✔ **+ 1 pt** $\tau \preceq \text{bool}$

✔ **+ 0.5 pts** $\Gamma \Vdash c_1\checkmark$

Question 5

(no title)

9.5 / 10 pts

+ 0 pts Incorrect

✔ **+ 2 pts** Principle of abstraction

Condition false

✔ **+ 1 pt** $l\sigma \vdash e_1 \leq e_2 \implies \mathrm{F}$

✔ **+ 1 pt** $l \vdash \sigma - [\text{foreach } i \text{ from } e_1 \text{ to } e_2 \text{ do } c \text{ od}] \to \sigma$

Condition true

✔ **+ 1 pt** $l\sigma \vdash (e_1 \leq e_2) \implies \mathrm{T}$

✔ **+ 1.5 pts** $l\sigma \vdash e_1 \implies a$

✔ **+ 1.5 pts** $l \vdash \sigma[l(i) \mapsto a] - [c] \to \sigma'$

✔ **+ 1 pt** $l \vdash \sigma' - [\text{foreach } i \text{ from } e_1 + 1 \text{ to } e_2 \text{ do } c \text{ od}] \to \sigma''$

✔ **+ 1 pt** $l \vdash \sigma - [\text{foreach } i \text{ from } e_1 \text{ to } e_2 \text{ do } c \text{ od}] \to \sigma''$

− 0.5 pts Docking half point for not considering the value taken on by i in the fourth premise for rule 2.

Question 6

(no title) — **4** / 12 pts

+ 0 pts Incorrect

Type signature rule: Premise looks like $\Gamma, \vec{x} : \vec{\tau}, \vec{y} : \vec{\mu}, p : \vec{\tau}\ \ \mathbf{proc} \Vdash c\checkmark$

- ✔ **+ 1.5 pts** $\vec{x} : \vec{\tau}$
- ✔ **+ 1.5 pts** $\vec{y} : \vec{\mu}$
- **+ 2 pts** $p : \vec{\tau}\ \ \mathbf{proc}$
- **+ 2 pts** $\Vdash c\checkmark$
- **+ 1 pt** $\Gamma \vdash p : \vec{\tau}\ \ \mathbf{proc}$

Typing rule for call

- **+ 1 pt** $\Gamma \vdash p : \vec{\tau}\ \ \mathbf{proc}$
- **+ 1 pt** $\Gamma \vdash e_i : \mu_i$ for every i
- **+ 1 pt** $\mu_i \preceq \tau_i$ for every i

- ✔ **+ 1 pt** No change if parameter-passing mechanism is call-by-name

Question 7

(no title) — **6** / 12 pts

- **+ 0 pts** Blank, Wrong, incomplete
- **+ 2 pts** Allocation of fresh locations: $l_1 \notin range(\ell) \cup dom(\sigma) \cup range(\ell_1)$
- **+ 1 pt** Use bindings ℓ_1 from closure of procedure (procedure definition-time)
- ✔ **+ 2 pts** and augment with binding of formal parameter to newly allocated location $[y \mapsto l_1]$
- ✔ **+ 2 pts** Augment call-time store with new location initialised to actual parameter's store value $\sigma[l_1 \mapsto \sigma(l)]$
  (Note: other correct ways of defining initial value are marked ok)
- ✔ **+ 2 pts** Execute body of procedure to yield a modified store $-[\ c\ ]\rightarrow\ \ \sigma'$
- **+ 2 pts** In conclusion: copy back results in parameter location to argument location $\sigma'[l \mapsto \sigma'(l_1)]$
  (Note: $\sigma[\ldots]$ instead of $\sigma'[\ldots]$) will not be penalised
- **+ 1 pt** and deallocate all new storage -- $\sigma'[l \mapsto \sigma'(l_1)]|_{dom(\sigma)}$

Mostly incorrect. No specification of fresh allocation of location for y. Does not use binding \ell_1 from closure. Copyback not correctly specified. No deallocation.

Question 8

(no title)

Resolved **5** / 12 pts

+ 0 pts Incorrect

(a)

✔ **+ 0.5 pts** No.

+ 1 pt Enough for $range(l) \subseteq dom(\sigma)$

(b)

+ 1.5 pts Yes

✔ **+ 1.5 pts** Multiple variables might end up referring to the same location unintentionally

✔ **+ 1.5 pts** Other variable values in the procedure scope might be changed by accident

+ 1.5 pts Overwriting might happen if some $l_i \in range(l) \cap dom(\sigma)$. No problem otherwise.

✔ **+ 1.5 pts** Space is wasted carrying around parameters which ought to be deallocated

(g)

+ 0.5 pts No consequence

+ 1 pt Addresses in $dom(\sigma) \setminus range(l)$ cannot be accessed from the procedure

✔ **+ 1.5 pts** Restricting σ' to $dom(\sigma)$, namely $\sigma' \upharpoonright dom(\sigma)$ since we are clearing the space for the parameters once the procedure call is finished (and ℓ does not change from before to after the call).

− 1.5 pts Deducting half point for imprecise explanation for (f). Docking 1 points for frivolous regrade request.

Regrade Request **Submitted on: May 11**

(e), Sir I wrote overwrite might occur to few global variables. I did not write for li belonging to intersection of range of l and domain of rho.

That's not a precise answer. Docking 4 points for frivolous regrade request.

Reviewed on: May 11

Question 9

(no title)

1 / 17 pts

+ 0 pts Blank, incomplete or incorrect

✔ **+ 1 pt** 9a1 (Base Case): `int` $\preceq$ `real`

+ 2 pts 9a2 (Function Types antimonotone in domain, monotone in range):
$\tau_1 \to \tau_2 \ \preceq \ \tau_3 \to \tau_4$ if $\tau_3 \preceq \tau_1$ and $\tau_2 \preceq \tau_4$.

+ 4 pts 9a3 (Record fields) Correctly states for each $id'_i : \tau'_i \ (1 \leq i \leq m)$ in $\{id'_1 : \tau'_1, \ldots, id'_m : \tau'_m\}$, there exists $j \ (1 \leq j \leq n)$ such that $id'_i = id_j$ and $id_j : \tau_j$ in $\{id_1 : \tau_1, \ldots, id_n : \tau_n\}$

+ 2 pts (9a3 contd) and moreover states $\tau_j \preceq \tau'_i$.

+ 5 pts (9b1: x's type must have a method f)
$\Gamma \vdash x : \tau_0$ and $\tau_0 \preceq \{f : \tau_1 \times \ldots \times \tau_n \to \tau\}$

+ 3 pts (9b2: each argument e_i must be of a subtype of the expected arguments of f)
For each $i \ (1 \leq i \leq n) : \Gamma \vdash e_i : \tau'_i$ and $\tau'_i \preceq \tau_i$.

almost all wrong.

Name: Abhinav Rajesh Shripad Entry: 2022CS11596

Indian Institute of Technology Delhi
Department of Computer Science and Engineering

COL226	Programming Languages	Major Exam
May the Fourth, 2024	120 minutes	Maximum Marks: 100

Q1 (7)	Q2 (5)	Q3 (15)	Q4 (10)	Q5 (10)	Q6 (12)	Q7 (12)	Q8(12)	Q9(17)	Total (100)

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.
Attestation: I agree to abide by the Honour Code of IIT Delhi.

Signature: Abhinav

Q1 (7 marks) **Abstract Types** Consider an *abstract type* $('a,'b)$ `table`, which has as entries *key-value pairs* of the form (k, v), where $k : 'a$ and $v : 'b$. Consider the following operations on this abstract type:

- *create* : `unit` $\rightarrow$ $('a,'b)$ `table` creates an empty table.
- *add* : $('a,'b)$ $\rightarrow$ $('a,'b)$ `table` $\rightarrow$ $('a,'b)$ `table` adds a given entry to a given table.
- *empty?* : $('a,'b)$ `table` $\rightarrow$ `bool` says if a given table is empty or not.
- *present?* : $'a \rightarrow ('a,'b)$ `table` $\rightarrow$ `bool` says if there is some entry in the table for a given key.
- *lookup* : $'a \rightarrow ('a,'b)$ `table` $\rightarrow 'b$ returns a value associated with a given key if there is an entry with that key in the table.

Write a *minimal* **set of axioms** in terms of these operations to specify the $('a,'b)$ `table` abstract type.

empty? (create ()) = True

present? x (add (x,y) t) = True

present? x (create ()) = False

lookup x (add (x,y) t) = y

present? x (add (w,v) t) = present x t

lookup x (add (w,v) t) = lookup x t

~~lookup x (create ())~~ empty? (add(x,y) t) = False

Q2 (5 marks) **Unification**. For each of the following pairs of terms, what is the *most general unifier* if it exists, or else state why it does not exist:

(a) $f(h(a, Y), X)$ and $f(X, h(b, Y))$.

DO NOT EXIST, a → b not possible both constant

(b) $f(h(X, a), Y)$ and $f(Z, h(b, X))$.

{ Y → h(b,X) . Z → h(X,a) }

Q3 (6+9=15 marks) **Compilation.** *Byte code* is code that is interpreted on a *virtual machine*, whereas *native code* is machine code that runs directly on a physical machine, and *source code* is program code written in (usually) a high-level programming language. Remember that a program can run on a machine only if it is in that machine's native code. You will have to clearly describe the steps to install and run the desired software, using sentences of the form: **"On machine __, run the program __ (in file __, in language __) on input(s) __ (in file(s) __) to get output __ (in file __)."**

(a) Suppose you have a Linux machine (`L`) on which `gcc`, a native-code C compiler, is installed and runs, i.e., is in native code format. Now you have downloaded onto your machine a file `p2c.c` containing the C-language source code of *p2c*, a Pascal-to-C compiler (written in C). **Give the necessary steps** you will have to take to be able to run on your machine the Pascal program which is in file `foo.p` on the input data in file `input` and send output to file `output`.

Step 1.

on machine L, run the program p2C (in file p2C-c in language C) on inputs foo (in file foo-p) to get out temp (in file temp-c)

Step 2.

Step 2: on machine L, run the program ~~gcc~~ temp (in file temp-c), in language C) on inputs input (in file input) to get output output (in file output)

(b) Suppose I have a *Macos* machine M and a *Wintel* machine W, as well as the following software objects:

i. The source code, in file `jc.j`, written in language j, of a compiler jc; jc is designed to translate programs written in language j, to produce *native code* for a *Macos*-machine;

ii. A compiler jbc (in file `jbc.b`) already compiled into *byte code* language B; when run, jbc translates programs in language j to programs in byte code B;

iii. A byte-code B-interpreter BI_W which runs on a *Wintel*-machine; this interpreter is already in native code of the *Wintel*-machine.

Indicate the steps by which I can produce a *native code compiler* for language j programs, which runs on *Macos*-machines (and produces native *Macos*-machine code). Name the outputs of each step appropriately.

Step 1.

Step 2.

	Machine	run program file	input files	output
Step 1	Wintel	BIw	jbc-b, jc-j	jc-b
Step 2	wintel	BIw	jc-b, jc-j	sc-a
Step 3	Macos	~~sc-a~~		Compiler

$$\frac{l\sigma \vdash e_1 \Rightarrow a_1,\ l\sigma \vdash e_2 \Rightarrow a_2,\ l\sigma \vdash (a_1 \leq a_2) \Rightarrow T,\ l\sigma \vdash \sigma\text{-[}c\text{]}\rightarrow \sigma_1 \quad l \vdash \sigma_1 \text{ (for each } i \text{ from } (e_1+1) \text{ to } e_2 \text{ do } c) \longrightarrow \sigma_2}{l \vdash \sigma \text{ (for each } i \text{ from } e \text{ to } e_2 \text{ do } c) \longrightarrow \sigma_2}$$

2022CS11586

Q4 (10 marks) **Type-checking Imperative Programs**. Assume we have defined the type-checking relation "$\Gamma \vdash e : \tau$" which checks that, under type assumptions Γ, the expression e has type τ. Also assume that we have a (reflexive, transitive) subtype relation $\tau_1 \preceq \tau_2$ between certain types, such as `int` $\preceq$ `real`. Our objective is to write a type checker for the commands of a simple imperative language by **inductively specifying a relation** "$\Gamma \Vdash c \checkmark$", that checks that the command c has no type error, given type assumptions Γ on the identifiers that appear in the command.

$$c \in Comm ::= \textbf{skip} \mid x := e \mid c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} \mid \textbf{while } e \textbf{ do } c_1 \textbf{ od}$$

Informally, an assignment has no type error if the expression e is of a type that can be automatically converted into that of the identifier x; sequences of commands have no type error if each of the commands has no type error; the conditional and iterative constructs have no type error if the test expression e can be type-converted into the type `bool` and the commands in the branches or the body have no type error.

$$\frac{}{\Gamma \Vdash \text{skip} \checkmark}\ \text{Rule Skip} \qquad \frac{\Gamma \Vdash c_1 \checkmark \quad \Gamma \Vdash c_2 \checkmark}{\Gamma \Vdash c_1; c_2 \checkmark}\ \text{Rule Seq}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash x : \tau_2 \quad \tau_1 \leq \tau_2}{\Gamma \Vdash x := e \checkmark}\ \text{Rule Assign}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \text{bool} \quad \Gamma \Vdash c_1 \checkmark \quad \Gamma \Vdash c_2 \checkmark}{\Gamma \Vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ fi} \checkmark}\ \text{Rule ifelse}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \leq \text{bool} \quad \Gamma \Vdash c_1 \checkmark}{\Gamma \Vdash \text{while } e \text{ do } c_1 \text{ od} \checkmark}\ \text{(Rule while)}$$

Q5 (2+8 = 10 marks) **Command Equivalence**. In mathematics, we have expressions such as $\sum_{i=a}^{i=b} f(i)$ and $\prod_{i=a}^{i=b} f(i)$. By **which principle** should we consider a special iterator command

$$\textbf{foreach } i \textbf{ from } a \textbf{ to } b \textbf{ do } c \textbf{ od}$$

where c is a command in which i appears as an integer variable (but is not explicitly assigned to in c)?

Principle of Abstraction.

Provide Big-step semantics for the definite iteration **foreach** i **from** e_1 **to** e_2 **do** c **od**, which is equivalent in behaviour to

$$i := e_1;\ j := e_2;\ \textbf{while } i \leq j \textbf{ do } c;\ i := i + 1 \textbf{ od}$$

assuming that j is a fresh variable, which is deallocated immediately after the loop.

$$\frac{l\sigma \vdash e_1 \Rightarrow a_1 \quad l\sigma \vdash e_2 \Rightarrow a_2 \quad l\sigma \vdash (a_1 \leq a_2) \Rightarrow F}{\ell \vdash \sigma \text{ –[} \textbf{foreach } i \textbf{ from } e_1 \textbf{ to } e_2 \textbf{ do } c \textbf{ od]}\rightarrow \sigma}$$

~~$l\sigma \vdash e_1 \Rightarrow a_1$, $l\sigma \vdash e_2 \Rightarrow a_2$, $l\sigma \vdash (a_1 \leq a_2) \Rightarrow T$, $l \vdash \sigma$ (foreach i from e_1+1 to~~

$\ell \vdash \sigma$ –[**foreach** i **from** e_1 **to** e_2 **do** c **od**]→ ________

At the top |

⊛ $\tau \Vdash x_i := e_i$ ✓ $\forall i$, $\tau[\vec{x} : \vec{\tau}] \vdash y_i : \tau'_i$
$\tau[\vec{x} : \vec{\tau}, \vec{y} : \vec{\tau}'] \Vdash c$ ✓

$\tau \Vdash p(e_1, e_2 \cdots)$ ✓

Name: Entry: $\boxed{2022CS11556}$ 4

Q6 (12 marks) **Procedure Call.** Consider a (possibly recursive) imperative procedure definition:

> procedure $p(x_1 : \tau_1, \ldots, x_n : \tau_n)$;
> var $y_1 : \tau'_1, \ldots, y_m : \tau'_m$;
> begin c end;

A procedure definition has a *type signature* of the form $\Gamma \vdash p : \tau$ **proc**, which indicates that the procedure definition is well-formed under the type assumptions Γ on the global variables and (prior and current) procedure definitions, and that the defined procedure takes parameters of type τ. (Remember that the procedure may be recursively defined.) Extending the type checker specification of Q4, **present a type signature rule** for the above procedure p (you may use the abbreviation $\vec{x} : \vec{\tau}$ instead of $x_1 : \tau_1, \ldots, x_n : \tau_n$):

$$\frac{\tau \vdash \vec{x} : \vec{\tau}_1 \quad \tau[\vec{x} : \vec{\tau}_1] \vdash \vec{y} : \vec{\tau}_2 \quad \tau[\tau_3] \vdash c : \tau_4}{\Gamma \vdash p : \vec{\tau}_1 \to \tau_4}$$

where $\tau_3 = [\vec{x} : \vec{\tau}_1][\vec{y} : \vec{\tau}_2]$;

Also **present the typing rule** for *procedure call* $p(e_1, \ldots, e_n)$ being a well-formed command:

$$\frac{\uparrow \text{ ⊛ Above}}{\Gamma \Vdash p(e_1, \ldots, e_n) \checkmark}$$

Will these rules be different if the parameter-passing mechanism is call-by-name? $\boxed{\text{No}}$

Q7 (12 marks) **Parameter passing mechanisms.** The *call-by-reference* parameter passing mechanism is necessary for writing procedures (such as swap) in which the procedure is intended to change the contents of its arguments. However, this mechanism is not a "clean abstraction" in that it suffers from the problem that any assignment to a reference argument immediate changes the corresponding global variable *before the completion of the execution of the procedure.* A better variant of this method is *call-by-value-result*, in which the contents of the argument variables are copied into *fresh* storage in the called procedure, the assignments are made to these locations, and finally, just before exit, the contents of these new locations are copied back into the argument variables.

Suppose p is a procedure with a single call-by-value-result parameter y and whose body is command c. Assume the analysis phase of the compiler creates the environment β that *binds* the procedure p to a closure, i.e., $\beta(p) = «\lambda y.c, \ell_1»$, where ℓ_1 accounts for the location-bindings for "global" variables in c.
Provide big-step operational semantics for calling p with (global) variable x as argument in value-result mode, where ℓ binds variables to locations, and σ is the store at the time of the procedure call.

$$\frac{\ell, \sigma \vdash x \Rightarrow a_1, \quad \ell[y \to l_2] \vdash \sigma[l_2 \to a_1] -[c]\to \sigma', \quad \sigma'[l_2 \to a_2]}{\ell \vdash \sigma -[\,p(x)\,]\to \sigma[l_1 \to a_2]} \quad \beta(p) = «\lambda y.c, \ell_1», \quad \ell(x) = l$$

Q8 (12 marks) **Allocation schemes in procedure call.** Consider the operational semantics rule for call-by-value parameter passing for the procedure in Q5:

$$\frac{\ell, \sigma \vdash e_1 \Longrightarrow a_1 \;\ldots\; \ell, \sigma \vdash e_n \Longrightarrow a_n \qquad \ell[\{x_i \mapsto l_i\}_{i=1}^n] \vdash \sigma[\{l_i \mapsto a_i\}_{i=1}^n] -[\textbf{var } \vec{y} : \vec{\tau}'; \textbf{ begin } c \textbf{ end}]\to \sigma'}{\ell \vdash \sigma -[p(e_1, \ldots, e_n)]\to \sigma'|_{dom(\sigma)}}$$

where locations $l_1, \ldots, l_n$ are all distinct locations and fresh with respect to $range(\ell) \cup dom(\sigma)$.
Answer the following questions in **very few words**:

(a) In this rule, do we require $range(\ell) = dom(\sigma)$?

No, some location can not ~~be~~ contain of any variable (garbage value)

(b) According to this rule, can the allocation of $l_1, \ldots, l_n$ be on the heap?

No, at the end of procedure, $l_1 l_2 \cdots l_n$ are restored, must be on stack.

(c) What can happen if locations $l_1, \ldots, l_n$ are not all distinct?

~~2 variables~~ 2 different identifier pointing to same address

(d) What can happen if locations $l_1, \ldots, l_n$ are not all fresh with respect to (disjoint from) $range(\ell)$?

we can change global variables, some variable points to address /location of a global variable

(e) What can happen if locations $l_1, \ldots, l_n$ are not all fresh with respect to (disjoint from) $dom(\sigma)$?

we overwrite a few global variable

(f) What will happen if in the target store in the rule's conclusion, we wrote σ' instead of $\sigma'|_{dom(\sigma)}$?

local variables will be accessible from outside the block

(g) What would be the consequence if instead of $\sigma'|_{dom(\sigma)}$, we wrote $\sigma'|_{range(\ell)}$?

few variables defined inside the block would be visible outside

(h) Which parts of the conclusion of the rule indicate we can use stack allocation for the parameters?

$\sigma'|_{dom(\sigma)}$, ie we restore the previous states (ie popping the stack)

Q9 (9+8=17 marks) **Subtyping in OO Languages.** Let *TypeExp* represent type expressions, with τ as a typical type expression, as given by the following abstract grammar:

$$\tau ::= int \mid real \mid bool \mid \tau_1 \to \tau_2 \mid \{id_1 : \tau_1, \ldots, id_n : \tau_n\}$$

Here the notation $\{id_1 : \tau_1, \ldots, id_n : \tau_n\}$ denotes a "record" or "struct" or "object", with id_i is an identifier denoting a *field* or *member* (including possibly a method/function) of type τ_i.

Suppose also that the type system includes the concept of a reflexive, transitive *subtyping relation* induced by (a) $int \preceq real$; (b) *record subtyping*, where a record with additional (extra) fields is considered to belong to a subtype of the record-type to which the original record belongs; also, each named field in the record-supertype should be of a supertype of the type of the identically-named field in the record-subtype; (c) *function-space or arrow subtyping* based on the idea that if a function (method) is type correct for a supertype argument, it will work correctly on an argument belonging to a subtype as well, and that a returned value belonging to a subtype is acceptable wherever a value belonging to a supertype is expected. (Recall the question in the minor).

(a) **Present an inductive definition of the relation** $\tau_1 \preceq \tau_2$ using inference rules, to indicate τ_1 is a subtype of τ_2. You need to provide rules for when one arrow-type $\tau_1 \to \tau_2$ is a subtype of another arrow-type $\tau_3 \to \tau_4$, and when one record type $\{id_1 : \tau_1, \ldots, id_n : \tau_n\}$ is a subtype of another $\{id'_1 : \tau'_1, \ldots, id'_m : \tau'_n\}$.
Note: $\tau \preceq \tau$ for all τ (reflexivity), as well as if $\tau_1 \preceq \tau_2$ and $\tau_2 \preceq \tau_3$ then $\tau_1 \preceq \tau_3$ (transitivity).
Base case(s):

~~int ≤ real~~ $int \leq real, \quad \tau \leq \tau \ \forall \tau$

Function (Arrow) Types subtyping case(s):

$\tau_1 \leq \tau_2, \ \tau_3 \leq \tau_4 \longleftrightarrow (\tau_1 \to \tau_3) \leq (\tau_2 \to \tau_4)$

Record (object) Subtyping case(s) (Note that the fields of one record type may not appear in the same order in the other):

See below

$\cancel{\Gamma \vdash \{id_1 : \tau_1, \dots, id_i : \tau_i, \dots, id_j : \tau_j \dots\}}$

$$\cancel{\frac{\Gamma \vdash \{\dots id_i : \tau_i \dots, id_j : \tau_j\} : \tau_1 \quad \Gamma \vdash \{\dots id_j : \tau_j \dots, id_i : \tau_i\} : \tau_2}{\Gamma \vdash \tau_1 \preceq \tau_2}}$$

~~(Rule Re-arrange)~~

$$\cancel{\frac{\Gamma \vdash \{id_1 : \tau_{11}, id_2 : \tau_{12}, \dots\} : \tau_1 \quad \Gamma \vdash \{id_1 : \tau_{21}, id_2 : \tau_{22} \dots\} : \tau_2 \qquad \tau_{1i} \preceq \tau_{2i} \ \forall i \ 1 \le i \le n}{\tau_1 \preceq \tau_2}}$$

(b) Now, possibly using the relation $\preceq$, **present a typing rule** for a method call in an object-oriented language:

$$\frac{\qquad\qquad\qquad}{\Gamma \vdash x.f(e_1, \dots, e_n) : \tau}$$

(Hint: Object x must belong to a subtype of a type that contains a method f that can accept arguments $e_1, \dots, e_n$ and return a value that can be converted to type τ)

(a)

$$\frac{\Gamma \vdash \{\vec{id} : \vec{\tau}\} : \tau_A \ , \quad \Gamma \vdash \{\vec{id} : \vec{\tau_1}, id_{n+1} : \tau_{n+1}\} : \tau_B}{\Gamma \vdash \tau_B \preceq \tau_A}$$

(B has one more attribute than A, B is child class of A)

(b)

$$\frac{\Gamma \vdash x : \tau_x \qquad \tau_x . f : \tau_1}{\Gamma \vdash x.f(e_1, e_2 \dots e_n) : \tau_1}$$

(τ_x type has a method f of type τ_1)

$$\frac{\Gamma \vdash x : \tau_x \ , \ \Gamma \vdash y : \tau_y \quad \tau_x \preceq \tau_y \quad \Gamma \vdash y.f(e_1 e_2 \dots) \vdash \tau_1}{\Gamma \vdash x.f(e_1, e_2 \dots) : \tau_1}$$