

Supervision 3

Michaelmas Term 2026 · Michael Lee

●●●●● difficulty · ★ optional

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	6 hours 10 min (+ 30 min for the optional questions)
OPTIONAL	★ marks a question, or a part of one, you may skip

1 Evaluation Contexts

TIME 1 hour

QUESTION 1

CONCEPTUAL · 10 MIN

●●●●● BASIC

Assume a left-to-right evaluation order as L1 does. Consider the OCaml expression

```
1  (); 1 + (2 + 3*4); ()
```

OCaml

- (a) Give the evaluation context E in which $3*4$ is evaluated.
- (b) Give the evaluation context E in which $2 + 3*4$ is evaluated.

QUESTION 2

CONCEPTUAL · 5 MIN

●●●●● BASIC

Continuing question 1, draw stacks representing each of the evaluation contexts that you have identified.

Explain what filling the hole in the context corresponds to on the stack.

QUESTION 3

CONCEPTUAL · 15 MIN

●●●●● CORE

Rewrite the following function such that the evaluation contexts are explicitly maintained. Hitting the base case should fill the hole in the context.

```

1  let rec sum = function
2    | [] -> 0
3    | x::xs -> x + sum xs

```

HINT Might you have learnt a technique for making the stack explicit in another course?

QUESTION 4

DESIGN · 30 MIN

●●●●● CORE

Consider the Dyn language, defined as follows:

Dyn SYNTAX

Let n range over integers, b over booleans, and x over variables. The form **pop**(e) is internal syntax: it arises during evaluation, and is not written by the programmer.

EXPRESSIONS

$$e ::= x \mid n \mid b \mid e_1 + e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{fun } x:T \rightarrow e \mid e_1 e_2 \mid \text{pop}(e)$$

VALUES

$$v ::= n \mid b \mid \text{fun } x:T \rightarrow e$$

EVALUATION CONTEXTS

$$E ::= \text{let } x = [-] \text{ in } e_2 \mid [-] + e_2 \mid v_1 + [-] \mid \text{if } [-] \text{ then } e_2 \text{ else } e_3 \mid [-] e_2 \mid v_1 [-] \mid \text{pop}([-])$$

ENVIRONMENTS

$$\rho ::= \cdot \mid \rho, x = v$$

Dyn OPERATIONAL SEMANTICS

$$\begin{array}{c}
\frac{\langle \rho; e \rangle \rightsquigarrow \langle \rho'; e' \rangle}{\langle \rho; E[e] \rangle \rightsquigarrow \langle \rho'; E[e'] \rangle} \text{ (R-CTX)} \qquad \frac{x \notin \text{dom}(\rho_1)}{\langle \rho_0, x = v, \rho_1; x \rangle \rightsquigarrow \langle \rho_0, x = v, \rho_1; v \rangle} \text{ (R-VAR)} \\
\\
\frac{n = n_1 \pm n_2}{\langle \rho; n_1 + n_2 \rangle \rightsquigarrow \langle \rho; n \rangle} \text{ (R-OP+)} \qquad \frac{}{\langle \rho; \text{let } x = v \text{ in } e \rangle \rightsquigarrow \langle \rho, x = v; \text{pop}(e) \rangle} \text{ (R-LET)} \\
\\
\frac{}{\langle \rho; \text{if true then } e_2 \text{ else } e_3 \rangle \rightsquigarrow \langle \rho; e_2 \rangle} \text{ (R-IF1)} \\
\frac{}{\langle \rho; \text{if false then } e_2 \text{ else } e_3 \rangle \rightsquigarrow \langle \rho; e_3 \rangle} \text{ (R-IF2)} \\
\\
\frac{}{\langle \rho; (\text{fun } x:T \rightarrow e) v \rangle \rightsquigarrow \langle \rho, x = v; \text{pop}(e) \rangle} \text{ (R-APP)} \\
\\
\frac{}{\langle \rho, x = v'; \text{pop}(v) \rangle \rightsquigarrow \langle \rho; v \rangle} \text{ (R-POP)}
\end{array}$$

(a) Construct a program whose behaviour is unexpected.

HINT Search for static and dynamic scoping. You may find the [course notes](#) for the now-retired *Concepts in Programming Languages* course helpful.

(b) Suggest a modification to the semantics that fixes this problem.

HINT You should keep the semantics mostly as is. I am not looking for “rewrite the semantics in the style of L2”.

2 Exceptions

TIME 1 hour

This section guides you through adding exceptions to a programming language.

Let X be a metavariable ranging over a set of pre-declared exceptions, for example.

$$X = \{\text{KeyNotFoundError}, \text{IndexOutOfBoundsError}, \dots\}$$

Let h be a grammar of **exception handlers**, defined as follows:

$$h ::= X(x) \mapsto e \mid h; X(x) \mapsto e$$

For example, assume **KeyNotFoundError** raises an integer payload. The handler

$$\text{KeyNotFoundError}(x) \mapsto x + 1$$

takes the integer raised by **KeyNotFoundError** and returns the integer one greater than it. Having added exception handlers to the language, the **raise** and **try** expression formers allow exceptions to be raised and handled, respectively

$$\text{raise } X(e) \quad \text{try } e \text{ with } h$$

For example,

$$\text{try } 2 \text{ with KeyNotFoundError}(x) \mapsto x + 1$$

should reduce to 2,

$$\text{try raise KeyNotFoundError}(2) \text{ with KeyNotFoundError}(x) \mapsto x + 1$$

should reduce to 3,

$$(\text{try raise KeyNotFoundError}(2) \text{ with KeyNotFoundError}(x) \mapsto x + 1) + 1$$

should reduce to 4.

QUESTION 5

DESIGN · 20 MIN

● ● ● ● ● CORE

Add operational semantics rules for **raise** and **try**.

QUESTION 6

DESIGN · 20 MIN

● ● ● ● ● CORE

Add typing rules for **raise** and **try**. Assume that for each declared exception X , you can look up the type of its payload in advance in a pre-declared set Δ . For example,

$$\Delta(\text{KeyNotFoundError}) = \text{int}$$

QUESTION 7

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

State, but do not prove, the Type Safety theorem for this language.

QUESTION 8

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Explain, with reference to your typing rule for **try** e **with** h whether your language has Type Safety, and why.

3 Subtyping

TIME 1 hour 50 min

QUESTION 9

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Question inspired by Dr. Jeremy Yallop. Give an infinitely long proof that $\{p : \text{int}, q : \text{int}\} <: \{p : \text{int}\}$

QUESTION 10

PRACTICAL · 15 MIN

● ● ● ● ● CORE

Consider the following record

$$\{p = 1, q = 2\}$$

In L3 extended with subtyping,

- (a) Give a type derivation that shows the record has type $\{p : \text{int}, q : \text{int}\}$ (you can ignore Σ if you wish, using the simpler type derivation $\Gamma \vdash e : T$)
- (b) Give a type derivation that shows the record has type $\{q : \text{int}\}$
- (c) Without using your answer from question 9, give an infinitely long type derivation

QUESTION 11

PRACTICAL · 25 MIN

● ● ● ● ● CORE

For each of the following, either give a type derivation or explain why it is untypable. Since none of the examples use the store, feel free to use the simpler typing rule that ignores the store, $\Gamma \vdash e : T$. Assume an empty typing context $\Gamma = \cdot$. In parts (b) to (d), you are free to choose the type T under which to type the expressions

- (a) $\{p = \{p = \{p = 3\}\}\}$ is well-typed with type $\{p : \{\}\}$
- (b) **fun** $x : \{p : \text{bool}, q : \{p : \text{int}, q : \text{bool}\}\} \rightarrow x.p.q$

- (c) $\text{fun } f : \{p : \text{int}\} \rightarrow \text{int} \rightarrow (f\{q = 3\}) + (f\{p = 4\})$
- (d) $\text{fun } f : \{p : \text{int}\} \rightarrow \text{int} \rightarrow (f\{q = 3, p = 2\}) + (f\{p = 4\})$

QUESTION 12

PROOF · 20 MIN

● ● ● ● ● CORE

Attempt to prove the (APP) case of Type Preservation with subtyping, justifying (but not proving) the lemmas that you rely upon. Explain why the proof of the (APP) case is more complicated in the presence of subtyping.

QUESTION 13

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

For each of the following rules, give an example program that is typeable with that rule but gets stuck at runtime:

- (a)
$$\frac{T <: T'}{T \text{ ref } <: T' \text{ ref}}$$
- (b)
$$\frac{T' <: T}{T \text{ ref } <: T' \text{ ref}}$$

QUESTION 14

CONCEPTUAL · 25 MIN

● ● ● ● ● CORE

- (a) What relationship can you establish between the set of expressions with type $\{p : \text{int}\}$ and the set of expressions with type $\{p : \text{int}, q : \text{bool}\}$
- (b) Characterise the set of types T such that the following expression is well-typed:

$$\text{fun } x : T \rightarrow f\ x + g\ x$$

where f has type $\{p : \text{int}\}$ and g has type $\{q : \text{bool}\}$. Therefore, what is the largest/least-constrained type T such that the expression above is well-typed?

- (c) Characterise the set of types T such that the following expression is well-typed:

$$\text{if } e \text{ then } e_1 \text{ else } e_2 : T$$

where e_1 has type $\{p : \text{int}, q : \text{bool}, r : \text{bool}\}$ and e_2 has type $\{p : \text{int}, q : \text{bool}, s : \text{bool}\}$. Therefore, what is the smallest/most-constrained type T that can be given to the expression above?

- (d) Given two types, T_1 and T_2 , the join $T_1 \sqcup T_2$ is the smallest type such that $T_1 <: T_1 \sqcup T_2$ and $T_2 <: T_1 \sqcup T_2$. The meet $T_1 \sqcap T_2$ is the largest type such that $T_1 \sqcap T_2 <: T_1$ and $T_1 \sqcap T_2 <: T_2$.
- (i) Rephrase your answers to parts b and c using the language of joins and meets
- (ii) If any two types have a join and a meet, then the types form a structure known as a lattice. Explain why requiring that types have a lattice structure might make type inference easier.

4 Concurrency

TIME 35 min (+ 30 min)

QUESTION 15

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

By drawing a picture, or otherwise, explain the relationship between the thread-local semantics and the global semantics. Your conceptual model should illustrate how two threads communicate, for example, by writing from and reading to the same store, or by locking and unlocking shared locks.

QUESTION 16

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

How are the locks in the lectures different from the mutexes you've learnt about in concurrent and distributed systems?

HINT Is Thread B allowed to unlock a lock held by Thread A?

PICKED UP IN

PART IB Concurrent and Distributed Systems · *mutual exclusion*

QUESTION 17

DESIGN · 15 MIN (+ 30 MIN)

● ● ● ● ● ADVANCED

In this question, we will gradually extend the L1 multi-threaded semantics to resemble POSIX threads. Thus, each extension should build on the previous extensions:

- (a) Each thread should have their own thread ID, which should not change.
- (b) Each thread should be able to execute a **fork**(e) instruction, of type unit, that creates a new thread, with an ID, distinct from all other threads
- (c) Adjust the semantics of locks such that only the lock-holder can unlock the lock. For example, if Thread A locks lock m , then no other thread can unlock m
- ★ (d) Let z be a variable ranging over some fixed, pre-declared set of binary semaphores. Design semantics for **wait**(z) and **signal**(z), as in Concurrent and Distributed Systems.
- ★ (e) A **barrier** allows threads to wait until some number of threads have reached a pre-agreed "meeting point" (the barrier), before continuing. For example, a barrier that synchronises two threads allows both threads to wait for each other to reach the barrier, before either is allowed to continue. Add to L1 a **barrier**(b) instruction. b is a variable ranging over the name of a barrier, which is taken from some set of pre-declared barriers. Use $\beta(b)$ to look-up how many threads b synchronises. For example, if $\beta(b) = 2$, then the barrier synchronises two threads.

5 Contextual equivalence

TIME 1 hour 45 min

QUESTION 18

CONCEPTUAL · 15 MIN

● ● ● ● ● CORE

Taking L2 as an example, how does the notion of program context C differ from the context E (slide 187)? Give a program context that is not an evaluation context. Is it possible for an evaluation context to not be a program context?

QUESTION 19

PROOF · 25 MIN

● ● ● ● ● CORE

Prove the **if** $[-]$ **then** e_1 **else** e_2 case of the congruence theorem

QUESTION 20

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

The definition for semantic equivalence for L1 broadly translates into “two L1 programs are equivalent if they reduce to the same value”. Explain, with reference to an example, why this definition does not work for L2 and L3.

HINT Consider **fun** $x : \text{int} \rightarrow x + 0$ and **fun** $x : \text{int} \rightarrow x$. Are these semantically equivalent by the L1 definition?

QUESTION 21

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

A *ground type* is, informally speaking, a “simple” type, like `int` or `bool`. Importantly, function types, like $T_1 \rightarrow T_2$, are not ground types. How would you change the definition of semantic equivalence for L1 to avoid the problem faced in question 20?

QUESTION 22

PROOF · 30 MIN

● ● ● ● ● ADVANCED

Consider the L1 programs e_1 and e_2 such that $\Sigma_1 \vdash e_1 : \text{unit}$, and $\Sigma_2 \vdash e_2 : \text{unit}$. Prove that, if Σ_1 is disjoint from Σ_2 (no location appears both in Σ_1 and in Σ_2), then

$$e_1; e_2 \cong_{\Sigma}^{\text{unit}} e_2; e_1$$

where $\Sigma = \Sigma_1 \cup \Sigma_2$

PICKED UP IN

PART II Hoare Logic and Model Checking · *separating conjunction*

QUESTION 23

PROOF · 15 MIN

● ● ● ● ● CORE

Prove that the L3 programs `let x = ref 0 in x := 0` and `let x = ref 0 in x := 1` are not contextually equivalent.
