

Supervision 2

Michaelmas Term 2026 · Michael Lee

●●●●● difficulty ★ optional

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	6 hours (+ 20 min for the optional questions)
OPTIONAL	★ marks a question, or a part of one, you may skip

L2 AND L3 DEFINITIONS

These questions refer to the L2 and L3 languages. For your convenience, these definitions are collated in the **Language Definitions** sheet. For clarity, my definitions differ notationally from the course's.

1 Function basics

TIME 55 min

The aim of this section is to get you thinking about functions and associated concepts, including bound/free variables, substitution, and recursion.

QUESTION 1

CONCEPTUAL · 10 MIN

●●●●● BASIC

Consider the OCaml functions `fun (x:int) -> x+0` and `fun (x:int) -> x`.

- Explain by what definition they could be considered equal, and what definition they could be considered unequal.
- Discuss, with reference to the two functions, why “reducing to the same (syntactic) value” is not a good notion of equality in a language with functions as values.

QUESTION 2

PRACTICAL · 5 MIN

●●●●● BASIC

What are the free variables of the following?

- $x + ((\text{fun } y : \text{int} \rightarrow z) 2)$
- $\text{fun } y : \text{int} \rightarrow \text{fun } y : \text{int} \rightarrow \text{fun } y : \text{int} \rightarrow y$

QUESTION 3

PRACTICAL · 10 MIN

● ● ● ● ● BASIC

Perform the following substitutions

- (a) $\{\text{fun } x : \text{int} \rightarrow y/z\}(\text{fun } y : \text{int} \rightarrow z y)$
- (b) $\{\text{fun } x : \text{int} \rightarrow x/x\}(\text{fun } y : \text{int} \rightarrow x y)$
- (c) $\{\text{fun } x : \text{int} \rightarrow x/x\}(\text{fun } x : \text{int} \rightarrow x x)$

QUESTION 4

CONCEPTUAL · 25 MIN

● ● ● ● ● CORE

Consider the following OCaml function

```
1 let rec triangular n =
2   if n = 0 then 0
3   else n + triangular (n-1)
```

OCaml

Now consider the following function

```
1 let rec triangular_gen f n =
2   if n = 0 then 0
3   else n + f (n-1)
```

OCaml

- (a) Construct a function g such that $\text{triangular_gen } g = \text{triangular}$. Here, $=$ is a semantic notion: for all n of type int , $\text{triangular_gen } g \ n$ should have the same behaviour as $\text{triangular } n$.
- (b) Given a function f , x is a *fixed point* of f if $f x = x$. State the relationship between (the meaning of) triangular_gen and (the meaning of) triangular .
- (c) Define tri_0 as follows

```
1 let tri0 = failwith "bottom"
```

OCaml

On what values does tri_0 agree with triangle ?

- (d) Define tri_1 as follows

```
1 let tri_1 = triangular_gen tri_0
```

OCaml

On what values does tri_1 agree with triangle ?

- (e) Define tri_2 as follows

```
1 let tri_2 = triangular_gen tri_1
```

On what values does `tri_2` agree with `triangle`?

- (f) State a relationship between `triangular` and `tri_n`, where `n` is a natural number. Is there an `n` for which `triangular = tri_n`?

QUESTION 5

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Use ω to denote a non-terminating program. Use L2 notation for functions and applications. Write a program that tests if a language implements call-by-value or call-by-name reduction semantics.

2 Functions, in L2

TIME 1 hour 25 min

This section aims to build on the general understanding of functions established in the previous section, focusing on functions as they are defined in L2. The questions in this section thus aim to build an understanding of why the L2 rules are defined the way they are.

QUESTION 6

PRACTICAL · 35 MIN

● ● ● ● ● CORE

- Translate the `triangular` function in question 4 into L2, mirroring the implementation as closely as possible.
- Using the L2 operational semantics, prove that your translation reduces to 3 when applied to 2. You should not have used state in part (a): feel free to ignore the store `s`, rather than carrying it around.
- Explain, in words, and with reference to question 4, particularly its part (b), what the (R-LETREC FUN) operational semantics rule does.

QUESTION 7

PRACTICAL · 15 MIN

● ● ● ● ● CORE

Prove that `fun x : int → (x + ((fun y : bool → if y then 3 else 4) true))` is well-typed, by giving a typing derivation.

QUESTION 8

CONCEPTUAL · 15 MIN

● ● ● ● ● CORE

- Define the notion of closed, well-typed expression as a property of typing judgements of the form $\Sigma; \Gamma \vdash e : T$. Use $\cdot$ to refer to an empty context.

HINT Locations should be allowed to be free, but variables should not.

- (b) Re-state the theorems of Progress and Type Preservation for L2, but using Γ to refer **exclusively** to variables x, y , and Σ to refer **exclusively** to locations $\ell_0, \ell_1, \dots$. You should use $\Sigma \vdash s$ to mean that s is well-typed under Σ ,

$$\Sigma \vdash s \iff \forall \ell \in \text{dom}(\Sigma). (\Sigma; \cdot \vdash \ell : T \text{ ref} \implies \Sigma; \cdot \vdash s(\ell) : T)$$

and $\Sigma; \Gamma \vdash \langle e, s \rangle$ to mean $\Sigma \vdash s$ and $\Sigma; \Gamma \vdash e : T$.

QUESTION 9

DESIGN · 20 MIN

● ● ● ● ● CORE

Give a grammar for types, and typing rules for functions and application, that allow only first-order functions (functions cannot accept other functions as arguments, nor return functions), and prohibit partial applications (functions must take all their arguments at once, like in Python).

3 L2 Metatheory

TIME 1 hour 50 min

The aim of this section is twofold. First, to practice inductive proofs on a more complicated language. Second, to highlight the importance of the Substitution Lemma, and the assumptions it relies on.

QUESTION 10

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Explain how the Progress theorem handles the case $x + 1$ where $x : \text{int} \in \Gamma$.

QUESTION 11

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

- (a) State the Substitution Lemma. Remember that we use $\Sigma; \Gamma \vdash e : T$ for well-typed term.
- (b) Using only the Substitution Lemma and the fact that $\cdot; \cdot \vdash 1 : \text{int}$ (so ignoring all L2 typing rules),
- (i) Assuming $\cdot; x : \text{int} \vdash x + 0 : \text{int}$, show that $\cdot; \cdot \vdash 1 + 0 : \text{int}$
 - (ii) Assuming $\cdot; y : \text{int}, x : \text{int} \vdash x + 0 : \text{int}$, show that $\cdot; y : \text{int} \vdash 1 + 0 : \text{int}$
 - (iii) Assuming $\cdot; y : \text{int}, x : \text{int} \vdash x + y : \text{int}$, show that $\cdot; y : \text{int} \vdash y + y : \text{int}$
- (c) Can you ever encounter case (b)(iii) in L2? Why, or why not?

QUESTION 12

PROOF · 35 MIN

● ● ● ● ● CORE

Prove the following cases for the Substitution Lemma for L2:

- (a) $\frac{\Gamma(y) = T}{\Sigma; \Gamma \vdash y : T}$ (T-VAR)

$$(b) \frac{\Sigma; \Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Sigma; \Gamma \vdash e_2 : T_1}{\Sigma; \Gamma \vdash e_1 e_2 : T_2} \text{ (T-APP)}$$

$$(c) \frac{\Sigma; \Gamma, y : T_1 \vdash e_0 : T_2}{\Sigma; \Gamma \vdash \text{fun } y : T_1 \rightarrow e_0 : T_1 \rightarrow T_2} \text{ (T-FUN)}$$

What assumptions did you have to make about valid operations on the context Γ ? Why do those assumptions hold for L2?

QUESTION 13

PROOF · 30 MIN

● ● ● ● ● CORE

Prove Type Preservation for L2 for the following cases

$$(a) \frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle e_1 e_2, s \rangle \rightsquigarrow \langle e'_1 e_2, s' \rangle} \text{ (R-APP1)}$$

$$(b) \frac{}{\langle (\text{fun } x : T_1 \rightarrow e_1) v, s \rangle \rightsquigarrow \langle e_1[v/x], s \rangle} \text{ (R-FUN)}$$

$$(c) \frac{}{\langle \text{let rec } f : T_1 \rightarrow T_2 = \text{fun } x : T_1 \rightarrow e_1 \text{ in } e_2, s \rangle \rightsquigarrow \langle e_2[\text{fun } x : T_1 \rightarrow \text{let rec } f : T_1 \rightarrow T_2 = \text{fun } x : T_1 \rightarrow e_1 \text{ in } e_1/f], s \rangle} \text{ (R-LETRECFUN)}$$

Your proof of Type Preservation should use the Substitution Lemma: please highlight where it is used.

QUESTION 14

CONCEPTUAL · 20 MIN

● ● ● ● ● ADVANCED

This question is designed to make you think about what induction can, and *can't* do for you. Consider a simple language, comprising naturals, variables, non-recursive functions, and application, whose typing rules and operational semantics rules are identical to those given for their counterparts in L2 (except that we ignore the store). All programs in this language terminate. Explain why you cannot prove this fact using only the induction principle over well-typed expressions, where the property Φ to be proved is

$$\Phi(e) \triangleq e \text{ terminates}$$

with reference to the rule

$$\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 e_2 : T_2}$$

(The statement can be proved, but requires a more powerful technique than induction: you will learn this technique if you take Part II Types.)

PICKED UP IN

PART II Types · logical relations

4 Products, sums, & Curry–Howard

TIME 55 min (+ 20 min)

QUESTION 15

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

In words, what can you say about the following pairs of L2 programs? *This is a discussion question.*

- (a) `(fst (v1, v2), snd (v1, v2))` and `(v1, v2)`
- (b) `fun x → f x` and `f`, where `f` is a value of function type

QUESTION 16

DESIGN · 20 MIN

● ● ● ● ● CORE

In OCaml, tuples can be *deconstructed*, in a syntax that looks roughly like

```
let (x, y) = (1, 2) in x+y
```

In abstract syntax, this corresponds to an expression former

```
destruct (x, y) = e1 in e2
```

- (a) Design typing and evaluation rules for this expression former.
- (b) What can you say about the behaviour of

```
(fst (v1, v2), snd (v1, v2))    destruct (x, y) = (v1, v2) in (x, y)
```

How are they the same, and how are they different?

QUESTION 17

DESIGN · 25 MIN

● ● ● ● ● CORE

Labelled variant types are a generalization of sum types, just as records are a generalization of products. Design abstract syntax, typing rules and evaluation rules for labelled variants, analogously to the way in which records generalise products. What do labelled variant types look like in OCaml?

★ QUESTION 18

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

The Curry Howard correspondence states that there is a correspondence between programs and proofs, and types and propositions.

- (a) What is a program of type `T` a proof of?

HINT The answer is simpler than you might think.

- (b) Consider the different semantic meaning of “prove” in the following statements:
 - (i) Prove that there is an L2 function which calculates the gcd of 2 numbers.
 - (ii) Prove that, for any program, either it terminates, or it does not.

HINT If you asked someone to complete Part (i), what would you expect in response? Would you have the same expectation if you asked someone to complete Part (ii)? What if you applied your expectation in Part (i) to Part (ii)?

5 Mutable references and store typing

TIME 55 min

This set of questions is designed to help you understand the complexities and subtleties around store typing.

NOTATION

Recall that I deviate from the notation in the course; the deviations are set out in the **Notation** section of the **Language Definitions** sheet.

QUESTION 19

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Consider the definition of Well-Typed Store given in the L3 typing rules of the **Language Definitions** sheet, repeated here:

- i. A store s is well-typed under assumptions Σ , written $\Sigma \vdash s$, if, for all $\ell \in \text{dom}(\Sigma)$, if $\Sigma; \cdot \vdash \ell : T$ ref, then $\Sigma; \cdot \vdash s(\ell) : T$.

Consider simplifying this definition to the following:

- i. A store s is well-typed under assumptions Σ , written $\Sigma \vdash s$, if, for all $\ell \in \text{dom}(\Sigma)$, if $\Sigma; \cdot \vdash \ell : T$ ref, then $\cdot; \cdot \vdash s(\ell) : T$.

Give an example store s that is well-typed under the first definition, but ill-typed under the second.

QUESTION 20

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Consider the statement of Type Preservation:

- i. Assume $\Sigma; \Gamma \vdash e : T$ and $\Sigma \vdash s$. Assume further that $\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle$. Then, there exists Σ' , disjoint to Σ , such that $\Sigma, \Sigma'; \Gamma \vdash e' : T$ and $\Sigma, \Sigma' \vdash s'$.

Consider simplifying this statement to:

- i. Assume $\Sigma; \Gamma \vdash e : T$ and $\Sigma \vdash s$. Assume further that $\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle$. Then $\Sigma; \Gamma \vdash e' : T$ and $\Sigma \vdash s'$.

Give a counter-example that shows this simplified statement is false.

QUESTION 21

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Does L3 have the Determinacy property? Why, or why not?

QUESTION 22

CONCEPTUAL · 30 MIN

● ● ● ● ● ADVANCED

This is a question about the expressiveness of languages with higher-order references

For readability, in this question, I use OCaml syntax (but, other than the `failwith`, the program encodes cleanly in L3). Consider the following program

```
1  let landin (f: (int -> int) -> (int -> int)) =
2    let stored = ref (fun _ -> failwith "bottom") in
3    let recursive = fun x ->
4      let next = f (!stored) in
5      stored := next; next x in
6    stored := recursive; recursive
```

OCaml

- (a) Explain, with reference to an example of your choosing, what the program computes, and how it computes it.

HINT You might want to pick `triangular_gen` from question 4.

- (b) Explain the behaviour of `landin id`, where `id` is the identity function `fun x -> x`
- (c) Explain if the `failwith` is ever called, why, or why not
- (d) Explain what this program implies about the `let rec` construct in L3.

PICKED UP IN

PART II Types · *programming with effects*