

Supervision 1

Michaelmas Term 2026 · Michael Lee

●●●●● difficulty · ★ optional

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	6 hours (+ 35 min for the optional questions)
OPTIONAL	★ marks a question, or a part of one, you may skip

L1 DEFINITIONS

Several of these questions refer to the L1 language. For your convenience, these definitions are collated in the **Language Definitions** sheet. For clarity, my definitions differ notationally from the course's.

1 Syntax and Semantics

TIME 35 min

This set of exercises is meant to emphasise the difference between syntax (what you write) and semantics (what you mean). The questions are meant for discussion.

QUESTION 1

CONCEPTUAL · 10 MIN

●●●●● BASIC

In what ways are the following statements true? In what ways are they not true?

- (a) $1 + 2 = 2 + 1$
- (b) $111 = 111$

QUESTION 2

CONCEPTUAL · 10 MIN

●●●●● BASIC

Is the OCaml `+` operator the “same” as the mathematical `+`? If they are different, give an example where they would behave differently.

QUESTION 3

CONCEPTUAL · 15 MIN

● ● ● ● ● CORE

Given two OCaml expressions, e_1 and e_2 , is it true that if $e_1 + e_2$ evaluates to v , then $e_2 + e_1$ evaluates to v ? If untrue, what restriction on e_1 and e_2 would make it true? Why might “evaluating to the same value” not be a good way of equating expressions?

2 Small-step operational semantics

PREREQUISITES Lecture 2 · TIME 1 hour 45 min

This course is not about semantics as a whole, but about (small-step) operational semantics. This set of questions will aim to expose you to (a) reasoning about operational semantics, and (b) designing operational semantics for some given syntax.

QUESTION 4

PRACTICAL · 25 MIN

● ● ● ● ● CORE

(a) Consider the following OCaml data type

```
1 type expr = Int of int | Plus of expr * expr | Mul of expr * expr
```

OCaml

Now consider the following partially complete evaluator (or interpreter)

```
1 let rec eval e = match e with
2   | Int(n) -> Int(n)
3   | e       -> eval (step e)
```

OCaml

Write an OCaml function, `step`, such that `eval (Mul(Plus(Int(1), Int(2)), Int(3)))` evaluates to `Int(9)`.

(b) Now, consider the following programming language, *Arith*, whose syntax is written as a Backus-Naur Form (BNF) grammar.

$$e ::= n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 * e_2$$

Give a left-to-right small-step operational semantics for this grammar.

(c) Compare your answers to parts (a) and (b).

QUESTION 5

PRACTICAL · 30 MIN

● ● ● ● ● CORE

Consider the L1 configuration

$$\langle (\ell_0 := 7); (\ell_1 := (!\ell_0 + 2)), \{\ell_0 \mapsto 0, \ell_1 \mapsto 0\} \rangle$$

(a) Give the full reduction sequence for this configuration.

(b) Prove that your answer to (a) is correct.

HINT Each step can be proven correct with a derivation tree.

QUESTION 6

DESIGN · 15 MIN

● ● ● ● ● CORE

As explained in the notes, the L1 implementation does not match the semantics. This is because the L1 implementation uses (OCaml) integers, which are bounded (31 or 63 bits). However, the L1 semantics uses unbounded integers. Change the L1 semantics to match the implementation.

QUESTION 7

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Assume we considered a subset of L1 without locations ℓ , and without expression formers for assignment $\ell := e$, and dereferencing $!\ell$. How would you change the definition of Config and the transition relation $\leadsto$ in response?

QUESTION 8

DESIGN · 15 MIN

● ● ● ● ● CORE

Recall our toy language Arith.

- (a) Extend the grammar of Arith **at least** with an e_1/e_2 expression former, representing integer division (`//` in Python). You can add more expression formers as needed.
- (b) Extend the operational semantics to give Div the expected meaning.

QUESTION 9

DESIGN · 15 MIN

● ● ● ● ● CORE

Extend L1 with an expression former **ref** e , such that **ref** v creates a new heap location, ℓ , that is initialised to v . Does your new language still have determinacy? Why, or why not?

3 Types

PREREQUISITES Lecture 4 · TIME 55 min

This section aims to introduce you to how type systems can be used to judge if an expression is given meaning by a semantics. Further, it aims to help clarify subtleties regarding what type safety actually means.

QUESTION 10

PRACTICAL · 10 MIN

● ● ● ● ● BASIC

Give a type derivation for $(\ell_0 := 7); (\ell_1 := (!\ell_0 + 2))$ under $\Sigma = \{\ell_0 : \text{int ref}, \ell_1 : \text{int ref}\}$.

The notes use Γ where I use Σ . I explain the change in notation in the **Language Definitions** sheet.

QUESTION 11

CONCEPTUAL · 25 MIN

●●●●● CORE

In the previous section, you were asked to extend Arith with an expression former e_1/e_2 . I will now extend Arith with an **err** keyword, that represents division by zero, with the rule

$$\frac{}{n/0 \rightsquigarrow \mathbf{err}}$$

- (a) If your original answer to question 8 did not have a rule for division by zero, what can a programmer expect when they divide by zero?
- (b) Assume **err** has type Bad. Explain, using an example, why this language no longer has the type preservation property.
- (c) Assume **err** has type int.
 - (i) Explain, using an example, why this language is no longer type safe.
 - (ii) Without modifying the semantics, modify the statement of type safety to restore type safety. Use the notation $E[\mathbf{err}]$ to refer to “any expression that contains **err**, like $\mathbf{err} + 0$, $1 + \mathbf{err}$, etc”.
 - (iii) Without modifying the definition of type safety, modify the semantics to restore type safety.

QUESTION 12

CONCEPTUAL · 20 MIN

●●●●● CORE

Type safety is a guarantee given to the programmer, by the language designer. Type safety guarantees that well-typed programs do not get stuck. Give a precise definition of what it means for a program to “get stuck”. To validate your definition, give simple, but illustrative examples of terms that are stuck, and terms that are not. Do this for:

- (a) L1
- (b) The extension of Arith with e_1/e_2 and **err**, with type safety defined as in part (c)(ii) of question 11. Is $\mathbf{err} + 1$ a stuck term by your definition?
- (c) **For discussion:** Is your definition of “getting stuck” a syntactic, or semantic notion? Briefly explain why.

4 Inductive definitions

TIME 30 min (+ 5 min)

A big focus of the course is on **inductive proofs**. Inductive proofs are a proof strategy that works over inductively defined data. The aim of this section is to introduce inductive definitions. Hopefully, you will begin to see why inductive proofs make sense as a way of proving properties of inductively defined data, and additionally spot when data is defined inductively (so you know instinctively when you can apply induction).

A set of elements is *defined inductively* if it is defined using a set of rules. Each rule must take one of two forms:

- i. $\frac{}{x \in X}$, which states that x must be in X
- ii. $\frac{x_1 \in X \quad x_2 \in X \quad \dots \quad x_n \in X}{y \in X}$, which gives a “formula” for forming new elements (y) in the set, from existing elements in the set $(x_1, \dots, x_n)$

(To be pedantic, the first form is subsumed by the second form, where $n = 0$, so there is only really one form of rule.) As an example, the set of natural numbers can be defined inductively, as follows:

- i. $\frac{}{0 \in \mathbb{N}}$, which states that 0 is a natural number
- ii. $\frac{n \in \mathbb{N}}{s(n) \in \mathbb{N}}$, which states that, if n is a natural number, then its successor ($s(n)$, or $n + 1$) is a natural number.

Thus, inductively defined sets are defined as a set of base elements (in the case of natural numbers, 0), and additionally a set of rules that form new elements from existing elements.

QUESTION 13

PRACTICAL · 10 MIN

● ● ● ● ● BASIC

- (a) Write down an OCaml datatype for lists of natural numbers. *You can assume a datatype `nat` of natural numbers.*
- (b) Define lists of natural numbers (`ListNat`) inductively. *You can assume that natural numbers have already been defined.*

QUESTION 14

PRACTICAL · 15 MIN

● ● ● ● ● CORE

Recall that a binary relation over a set X is defined as a subset of $X \times X$. For example, the $<$ relation over naturals is the subset $\{(0, 1), (1, 2), \dots\}$. Assume a set X , and a binary relation $R \subseteq X \times X$.

- (a) Define, inductively, the transitive closure $R^+ \subseteq X \times X$.
- (b) Define, inductively, the reflexive-transitive closure $R^* \subseteq X \times X$.

QUESTION 15

CONCEPTUAL · 5 MIN (+ 5 MIN)

● ● ● ● ● CORE

There is something missing from the definition of “inductively defined sets”. Consider, for example, the rules for the natural numbers, which state that the natural numbers are a set that (a) contains 0, and (b) is closed under s : so for any element n in the set, $s(n)$ is in the set. This does not uniquely identify a set.

- (a) Give an example of a set that fits these restrictions, but is not the natural numbers. *Give a set that is not even in bijection with the natural numbers.*
- (b) How can the definition be extended to uniquely identify one set?

HINT What we *really* mean is that the natural numbers contain 0, and is closed under the successor function s , and anything else is not a natural number.

5 Inductive proof

TIME 25 min (+ 30 min)

The aim of this section is to understand why induction is a natural way to prove properties about inductively defined sets, and additionally familiarise you with the technique of induction.

There is a very natural way to prove properties of a set that is defined inductively. This is known as the **induction principle**. For example, to prove a property Φ about the naturals, we can:

- i. Prove that the property holds for 0
- ii. Prove that if the property holds for n , then it holds for $n + 1$.

Since these are the **only** ways to build natural numbers (see question 15), we have proved it for all natural numbers. This proof strategy is the “induction principle for the natural numbers” (or “mathematical induction”). Thus, induction over the natural numbers is simply *a special case* of rule induction.

RULE VS STRUCTURAL INDUCTION

The course carefully distinguishes between rule and structural induction, but structural induction is an instantiation of rule induction at a particular rule. Since it is easier to learn one thing rather than two, I only teach rule induction.

QUESTION 16

PRACTICAL · 5 MIN

● ● ● ● ● BASIC

Define addition on the natural numbers as follows:

$$\begin{aligned} 0 + n &= n && (\text{ADD-Z}) \\ s(m) + n &= s(m + n) && (\text{ADD-S}) \end{aligned}$$

- (a) Define an OCaml datatype for the natural numbers, `nat`, with a constructor `Zero` for 0, and a constructor `Succ` for s .
- (b) Define an `add: nat -> nat -> nat` function that implements the specification above.
- (c) $0 + n = n$ is true by definition. Using induction on the natural numbers, prove that $\forall n \in \mathbb{N}. n + 0 = n$

QUESTION 17

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

Using your inductive definition of lists of naturals, derive the induction principle over lists of naturals.

QUESTION 18

PROOF · 10 MIN

●●●●● CORE

This question was taken from the 12–13 *DiscMath II exercises*. The set of well-bracketed strings (Brackets) is inductively defined as follows

$$\frac{}{\varepsilon \in \text{Brackets}} \text{ (EMPTY)} \quad \frac{x \in \text{Brackets}}{(x) \in \text{Brackets}} \text{ (BRACK)} \\ \frac{x \in \text{Brackets} \quad y \in \text{Brackets}}{xy \in \text{Brackets}} \text{ (CONCAT)}$$

- Derive the induction principle for well-bracketed strings.
- Use your induction principle to prove that any well-bracketed string has an equal number of left and right brackets.

★ **QUESTION 19**

CONCEPTUAL · 30 MIN

●●●●● ADVANCED

This question has three aims, in descending order of priority. First, to get you to think about the relationship between recursion and induction. Second, to introduce you to dependent types: proofs as programs. Third, to introduce you to OCaml's Generalised Algebraic Datatypes.

In OCaml, we can inductively define the natural numbers using a generalised algebraic datatype, or GADT. GADTs let us declare more information at the type level, and, as we will see, this expressiveness let us write proofs as programs.

OCaml

```

1  type zero      (* Declares a type for 0 *)
2  type !'a succ  (* Declares a type for s.
3                  The ! annotation indicates that the type is injective.
4                  It is not important, it can be ignored. *)
5  type _ nat' =  (* a GADT for natural numbers *)
6    | Z : zero nat'
7    | S : 'n nat' -> 'n succ nat'
```

`type _ nat'` is a generalised algebraic datatype, or GADT. The 2014-15 Advanced Functional Programming Master's course, by Dr. Jeremy Yallop, has some [good notes on GADTs](#), though most of the knowledge will not be needed for this exercise.

- Give a type representing the natural 3.

HINT This question might be easier after part (b). I would also recommend writing the code into a compiler with `ocaml-lsp` enabled.

- Using `type _ nat'`, write an expression `three'` representing the natural number 3.
- What is the type of `three'`? Consider your alternative definition of the naturals in Question 16, and the following definition of `three`. What is the type of `three`? How do the types of `three` and `three'` differ?

OCaml

```
1 let three : nat = Succ(Succ(Succ(Zero)))
```

- (d) The power of GADTs is that they let us *index* types. Identify the index in the type of `three`. Using your answer to the previous part, explain how the index of a GADT gives us more expressive power than for regular ADTs.

HINT Try specifying a type such that any program of that type **must** supply the natural 3, not just any natural. Can you do it with `nat`? Can you do it with `nat'`?

- (e) Complete the following function

OCaml

```
1 let rec forget: type m. m nat' -> nat = (* TODO *)
```

`type m. m nat' -> nat` is a type annotation that says `forget` is *polymorphically recursive*: that is, it can be instantiated with a new index `m` on each recursive call.

- (f) Now consider a new GADT, that defines addition on the natural numbers

OCaml

```
1 type (_,_,_) add =
2   | ZAdd : (zero, 'n, 'n) add
3     (* ZAdd is a proof that 0 + n = n *)
4   | SAdd : ('m, 'n, 'p) add -> ('m succ, 'n, 'p succ) add
5     (* SAdd is a proof that, if m + n = p, then s(m) + n = s(p) *)
```

Contrast the `add` type to the `add` function in Question 16. How are they similar, and how are they different?

- (g) Here's a new way of thinking about types. A type can be thought of as a *proposition*. The type `int` can be seen as the proposition “the type `int` has at least one member”.

Programs with type `int`, like `3`, can be seen as proofs of that proposition: `3` has type `int`, so it is a witness that the type `int` has a member. GADTs give us a more expressive language that lets us declare more specific propositions. For example, we can write the proposition that $1 + 3 = 4$, as

```
val one_plus_three_eq_four
  : (zero succ, zero succ succ succ, zero succ succ succ succ) add
```

A program with this type is a proof that $1 + 3 = 4$, just like `3` is a proof that the type `int` has a member.¹ Using the `add` type, complete the proof that $1 + 3 = 4$

OCaml

```
1 let one_plus_three_eq_four
2   : (zero succ, zero succ succ succ, zero succ succ succ succ) add
3     (*      1 +          3 =          4      *)
4   = (* proof goes here *)
```

- (h) Write a proof that $\forall n \in \mathbb{N}. n + 0 = n$

¹This is known as the *Curry–Howard correspondence*. To be pedantic, for the proof to be non-vacuous, the type system should be logically consistent. Programmatically, all programs should terminate: otherwise, we can prove *false*, and conclude anything we'd like.

```

1  let rec n_plus_zero_eq_n
2    : type m. m nat' -> (m, zero, m) add
3    = (* proof goes here *)

```

Compare your answer to your proof in Question 16.

- (i) Using your answer to the previous part, explain the relationship between induction and recursion.
- (j) Can you write `n_plus_zero_eq_n` using the ADT `nat` rather than the GADT `nat'`?

6 L1 Metatheory

PREREQUISITES Lecture 6 · TIME 1 hour 50 min

*Induction is the main proof technique used in this course. We have seen inductive definitions and induction in the general case. This section aims to highlight the inductive definitions **in the course**, and illustrate how induction is used **in the course** (and also, more practically). Once it becomes familiar, induction is **not hard**, but it is **tedious**.*

QUESTION 20

CONCEPTUAL · 15 MIN

● ● ● ● ● CORE

Review the definitions of L1, and the corresponding sets:

- (a) Syntax (the set of L1 expressions)
- (b) Operational Semantics (the set of L1 reductions $\{(e, s, e', s') \mid \langle e, s \rangle \rightsquigarrow \langle e', s' \rangle\}$)
- (c) Typing Rules (the set of well-typed L1 expressions $\{(\Sigma, e, T) \mid \Sigma \vdash e : T\}$)

For each, determine if the corresponding set is inductively defined. For those that are inductively defined, derive one case of the induction principle, and list (by name), the other cases.

QUESTION 21

PROOF · 30 MIN

● ● ● ● ● CORE

Without looking at the notes:

- (a) State the determinacy property of L1, first informally, then formally.
- (b) State the induction principle that you will use to prove Determinacy.
- (c) Complete the cases of the proof of Determinacy for

$$(i) \frac{e_1 \quad e_2}{e_1 \text{ op } e_2} \text{ (E-Op)}$$

$$(ii) \frac{e_1 \quad e_2 \quad e_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} \text{ (E-If)}$$

- (d) Which set are you performing induction over?

QUESTION 22

PROOF · 30 MIN

●●●●● CORE

- (a) State the progress property of L1, first informally, then formally. You may use the following definitions. A store s is well-typed, written $\Sigma \vdash s$, if $\text{dom}(\Sigma) \subseteq \text{dom}(s)$. A configuration $\langle e, s \rangle$ is well-typed, written $\Sigma \vdash \langle e, s \rangle : T$, if $\Sigma \vdash e : T$ and $\Sigma \vdash s$.
- (b) State the induction principle that you will use to prove Progress.
- (c) Prove the following cases of the Progress theorem:
- (i) $\frac{}{\Sigma \vdash \text{skip} : \text{unit}}$ (T-SKIP)
 - (ii) $\frac{\Sigma \vdash e_1 : \text{unit} \quad \Sigma \vdash e_2 : T}{\Sigma \vdash e_1; e_2 : T}$ (T-SEQ)
 - (iii) $\frac{\Sigma(\ell) = \text{int ref} \quad \Sigma \vdash e : \text{int}}{\Sigma \vdash \ell := e : \text{unit}}$ (T-ASSIGN)
- (d) Which set are you performing induction over?

QUESTION 23

PROOF · 35 MIN

●●●●● CORE

Without looking at the notes:

- (a) State the type preservation property of L1, first informally, and then formally. Once again, use $\Sigma \vdash s$ for “well-typed store”, and $\Sigma \vdash \langle e, s \rangle : T$ for “well-typed configuration”.
- (b) State the induction principle that you will use to prove Type Preservation.
- (c) Prove the following cases of the Type Preservation theorem:
- (i) $\frac{\langle e_2, s \rangle \rightsquigarrow \langle e'_2, s' \rangle}{\langle v_1 \text{ op } e_2, s \rangle \rightsquigarrow \langle v_1 \text{ op } e'_2, s' \rangle}$ (R-OP2)
 - (ii) $\frac{}{\langle \text{skip}; e_2, s \rangle \rightsquigarrow \langle e_2, s \rangle}$ (R-SEQ1)
 - (iii) $\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \rightsquigarrow \langle e'_1; e_2, s' \rangle}$ (R-SEQ2)
 - (iv) $\frac{s(\ell) = n}{\langle !\ell, s \rangle \rightsquigarrow \langle n, s \rangle}$ (R-DEREF)
 - (v) $\frac{}{\langle \text{while } e_1 \text{ do } e_2 \text{ done}, s \rangle \rightsquigarrow \langle \text{if } e_1 \text{ then } e_2; \text{while } e_1 \text{ do } e_2 \text{ done else skip}, s \rangle}$ (R-WHILE)
- (d) Which set are you performing induction over?