
PART IB

Semantics of Programming Languages

Supervision Exercises

Michaelmas Term 2026 · Supervisions 1–3, with language definitions

Michael Lee

Supervision 1	1
1 Syntax and Semantics	1
2 Small-step operational semantics	2
3 Types	3
4 Inductive definitions	5
5 Inductive proof	6
6 L1 Metatheory	9
Supervision 2	11
1 Function basics	11
2 Functions, in L2	13
3 L2 Metatheory	14
4 Products, sums, & Curry–Howard	16
5 Mutable references and store typing	17
Supervision 3	20
1 Evaluation Contexts	20
2 Exceptions	22
3 Subtyping	23
4 Concurrency	25
5 Contextual equivalence	26
Language Definitions	29
A Notation	29
B L1	30
C L2	31
D L3	33

● ● ● ● ● Difficulty runs from one to five, and is independent of how long a question takes: beside each time is a word for the type of work. *Practical* is meant to build up your programming skills; *conceptual* is hard but short, and relies on insight; *proof* trains formal rigour: often tedious, but not difficult; *design* asks you to invent or evaluate novel semantics, or extend existing ones in novel settings.

★ A ★ in the margin means the question, or that part of the question, is optional.

OCaml A code panel is a link: click anywhere on it to open that code, and whatever it builds on, in an editor in your browser.

HINT Set in grey after each question.

SOLUTION

Only in the solutions build.

PICKED UP IN

PART 1B Another Course · *what it calls the idea*

A block like this says where an idea is revisited: the part of the Tripos, the course, and the name that course gives it. In passing, the same pointer is written inline as (**1B** ANOTHER COURSE).

Supervision 1

Michaelmas Term 2026 · Michael Lee

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).

PICKED UP IN

PART IB Compiler Construction · abstract syntax trees and interpreters

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

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

- (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 $E[\text{err}]$ to refer to “any expression that contains **err**, like **err** + 0, 1 + **err**, etc”.
 - (iii) Without modifying the definition of type safety, modify the semantics to restore type safety.

PICKED UP IN

PART IB Programming in C and C++ · *undefined behaviour***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 **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.

- 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.
- 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:

- Prove that the property holds for 0
- 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:

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

- Define an OCaml datatype for the natural numbers, `nat`, with a constructor `Zero` for 0, and a constructor `Succ` for s .
- 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.

```

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'
```

OCaml

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.

- (a) 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.

- (b) Using type `_ nat'`, write an expression `three'` representing the natural number 3.
- (c) 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?

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

OCaml

- (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

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

OCaml

`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

```
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) *)
```

OCaml

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 $1 + 3 = 4$

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.

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 $\forall n \in \mathbb{N}. n + 0 = n$

OCaml

```

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) $\frac{}{\Sigma \vdash \text{skip} : \text{unit}} \text{ (T-SKIP)}$
- (ii) $\frac{\Sigma \vdash e_1 : \text{unit} \quad \Sigma \vdash e_2 : T}{\Sigma \vdash e_1; e_2 : T} \text{ (T-SEQ)}$
- (iii) $\frac{\Sigma(\ell) = \text{int ref} \quad \Sigma \vdash e : \text{int}}{\Sigma \vdash \ell := e : \text{unit}} \text{ (T-ASSIGN)}$

(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) $\frac{\langle e_2, s \rangle \rightsquigarrow \langle e'_2, s' \rangle}{\langle v_1 \text{ op } e_2, s \rangle \rightsquigarrow \langle v_1 \text{ op } e'_2, s' \rangle} \text{ (R-OP2)}$
- (ii) $\frac{}{\langle \text{skip}; e_2, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-SEQ1)}$
- (iii) $\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \rightsquigarrow \langle e'_1; e_2, s' \rangle} \text{ (R-SEQ2)}$
- (iv) $\frac{s(\ell) = n}{\langle !\ell, s \rangle \rightsquigarrow \langle n, s \rangle} \text{ (R-DEREF)}$
- (v) $\frac{}{\langle \text{while } e_1 \text{ do } e_2 \text{ done}, s \rangle \rightsquigarrow \langle \text{if } e_1 \text{ then } e_2; \text{while } e_1 \text{ do } e_2 \text{ done else skip}, s \rangle} \text{ (R-WHILE)}$

(d) Which set are you performing induction over?

Supervision 2

Michaelmas Term 2026 · Michael Lee

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	6 hours 10 min (+ 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 ,

$$\Sigma \vdash s \iff \text{dom}(\Sigma) \subseteq \text{dom}(s)$$

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 1 hour 5 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 Appendix A.

QUESTION 19

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Consider the following definition for a well-typed L3 store:

- i. A store s is well-typed under assumptions Σ , written

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

Consider simplifying this definition to the following:

- i. A store s is well-typed under assumptions Σ , written

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

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

QUESTION 20

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Consider the following definition for a well-typed L3 store:

- i. A store s is well-typed under assumptions Σ , written

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

Consider simplifying this definition to the following:

- i. A store s is well-typed under assumptions Σ , written

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

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

QUESTION 21

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 22

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

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

QUESTION 23

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

OCaml

```
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
```

- (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

Supervision 3

Michaelmas Term 2026 · Michael Lee

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	6 hours 30 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
```

OCaml

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

PICKED UP IN

PART IB Compiler Construction · *continuation-passing style*

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

$$\begin{array}{c} \frac{\langle \rho; e \rangle \rightsquigarrow \langle \rho'; e' \rangle}{\langle \rho; E[e] \rangle \rightsquigarrow \langle \rho'; E[e'] \rangle} \text{ (R-CTX)} \quad \frac{x \notin \text{dom}(\rho_1)}{\langle \rho_0, x = v, \rho_1; x \rangle \rightsquigarrow \langle \rho_0, x = v, \rho_1; v \rangle} \text{ (R-VAR)} \\[10pt] \frac{n = n_1 \pm n_2}{\langle \rho; n_1 + n_2 \rangle \rightsquigarrow \langle \rho; n \rangle} \text{ (R-OP+)} \quad \frac{}{\langle \rho; \text{let } x = v \text{ in } e \rangle \rightsquigarrow \langle \rho, x = v; \text{pop}(e) \rangle} \text{ (R-LET)} \\[10pt] \frac{}{\langle \rho; \text{if true then } e_2 \text{ else } e_3 \rangle \rightsquigarrow \langle \rho; e_2 \rangle} \text{ (R-IF1)} \\[10pt] \frac{}{\langle \rho; \text{if false then } e_2 \text{ else } e_3 \rangle \rightsquigarrow \langle \rho; e_3 \rangle} \text{ (R-IF2)} \\[10pt] \frac{}{\langle \rho; (\text{fun } x : T \rightarrow e) v \rangle \rightsquigarrow \langle \rho, x = v; \text{pop}(e) \rangle} \text{ (R-APP)} \\[10pt] \frac{}{\langle \rho, x = v'; \text{pop}(v) \rangle \rightsquigarrow \langle \rho; v \rangle} \text{ (R-POP)} \end{array}$$

(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:

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

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** carries 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.

You can assume evaluation contexts E are as they are in the notes, except extended with **raise** and **try**

$$E ::= \dots \mid \text{raise } X([-]) \mid \text{try } [-] \text{ with } h$$

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 2 hours

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,

- 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 $\Gamma \vdash e : T$)
- Give a type derivation that shows the record has type $\{q : \text{int}\}$
- 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) $\text{fun } x : \{p : \text{bool}, q : \{p : \text{int}, q : \text{bool}\}\} \rightarrow x.p.q$
- (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 (T-APP) case of Type Preservation with subtyping, justifying (but not proving) the lemmas that you rely upon. Explain why the proof of the (T-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

PRACTICAL · 10 MIN

● ● ● ● ● CORE

What does the following Java program do at compile-time and run-time?

Java

```
1 Object[] a = new String[1];
2 a[0] = 1;
```

What trade-off does Java accept in exchange for covariant arrays?

PICKED UP IN

PART IA Object-Oriented Programming · *covariant arrays*

QUESTION 15

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 $\{p : \text{int}\} \rightarrow \text{int}$ and g has type $\{q : \text{bool}\} \rightarrow \text{int}$. 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 16

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 17

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 18

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.

PICKED UP INPART IB Concurrent and Distributed Systems · *two-phase locking, serialisability*PART II Multicore Semantics and Programming · *sequential consistency***5 Contextual equivalence**

TIME 1 hour 55 min

QUESTION 19

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 20

PROOF · 25 MIN

● ● ● ● ● CORE

Prove the **if** $[-]$ **then** e_1 **else** e_2 case of the congruence theorem for L1

QUESTION 21

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Recall the definition of a monoid. A monoid is a triple $(X, \iota \in X, \cdot : X \times X \rightarrow X)$, where ι is the unit of the monoid, and $\cdot$ is the multiplication. It is subject to the following laws:

$$\begin{array}{lll} \forall x & \iota \cdot x = x & \text{left-unit} \\ \forall x & x \cdot \iota = x & \text{right-unit} \\ \forall x, y, z & (x \cdot y) \cdot z = x \cdot (y \cdot z) & \text{associativity} \end{array}$$

Examples of monoids include integers with 0 and addition, $(\mathbb{Z}, 0, +)$, naturals with 1 and multiplication, $(\mathbb{N}, 1, \times)$, functions on a set T with identity and composition, $(T \rightarrow T, \text{id}, \circ)$, and lists with the empty list and append $(\text{List } T, [], @)$.

Consider the following equivalence relation $\sim$ on the monoid $(\mathbb{Z}, 0, +)$: $\forall x, y \in \mathbb{Z}. x \sim y \iff x^2 = y^2$

- Check that $\sim$ is an equivalence relation
- Explain, with reference to an example, why $\sim$ is a poor equivalence for the monoid
- Explain what additional property an equivalence should respect
- Loosely explain how this definition relates to the notion of a congruence for contextual equivalence.

PICKED UP IN

PART 1A Discrete Mathematics · monoids

QUESTION 22

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”. This question explains why a naive translation of that idea to L2/L3 does not work.

Consider the following definition for L3.

L3 CONTEXTUAL EQUIVALENCE

Changes from the notes have been highlighted in orange. The critical change is that the notes only compare contexts with type unit, whereas here we allow for any type T' .

Consider typed L3 programs $;\Gamma \vdash e_1 : T$ and $;\Gamma \vdash e_2 : T$. We say they are contextually equivalent if, for every context C such that $;\cdot \vdash C[e_1] : T'$ and $;\cdot \vdash C[e_2] : T'$, we have either

- $\langle C[e_1], \emptyset \rangle \rightsquigarrow^\omega$ and $\langle C[e_2], \emptyset \rangle \rightsquigarrow^\omega$
- $\langle C[e_1], \emptyset \rangle \rightsquigarrow^* \langle v, s_1 \rangle$ and $\langle C[e_2], \emptyset \rangle \rightsquigarrow^* \langle v, s_2 \rangle$

- Explain, with reference to **fun** $x : \text{int} \rightarrow x + 0$ and **fun** $x : \text{int} \rightarrow x$, why x and $x + 0$ cannot be contextually equivalent.
- Explain, with reference to **ref** 0, why this definition is not a congruence.

HINT Recall that a congruence is an equivalence relation that respects the operations: in this case, context composition. The definition guarantees the “respects the operations” part by default: show that the relation is not an equivalence relation.

- (c) A *ground type* is, informally speaking, a “simple” type, like `int` or `bool`. Importantly, function types, like $T_1 \rightarrow T_2$, and reference types $T \text{ ref}$, are not ground types. How would you adapt the definition of semantic equivalence to work around the problem?

QUESTION 23

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

Consider the following definition for L3 contextual equivalence.

L3 CONTEXTUAL EQUIVALENCE, SAME STORE

Changes from the notes have been highlighted in orange. The critical change is that the notes allow the two stores at the end of evaluation to differ. In contrast, in this definition, they must be the same.

Consider typed L3 programs $;\Gamma \vdash e_1 : T$ and $;\Gamma \vdash e_2 : T$. We say they are contextually equivalent if, for every context C such that $;\cdot \vdash C[e_1] : \text{unit}$ and $;\cdot \vdash C[e_2] : \text{unit}$, we have either

1. $\langle C[e_1], \emptyset \rangle \rightsquigarrow^\omega$ and $\langle C[e_2], \emptyset \rangle \rightsquigarrow^\omega$
2. $\langle C[e_1], \emptyset \rangle \rightsquigarrow^* \langle \text{skip}, s_1 \rangle$ and $\langle C[e_2], \emptyset \rangle \rightsquigarrow^* \langle \text{skip}, s_1 \rangle$

With reference to a counter-example, explain why this definition does not produce a congruence.

HINT As in the previous question, the definition guarantees the “respects the operations” part by default: show that the relation is not an equivalence relation.

QUESTION 24

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 25

PROOF · 15 MIN

● ● ● ● ● CORE

Using the definition of contextual equivalence in the course notes, prove that the L3 programs `ref 0` and `ref 1` are not contextually equivalent.

Language Definitions

Michaelmas Term 2026 · Michael Lee

A Notation

I've deviated from the convention in the notes in three important ways.

1. Σ vs Γ . An important notion is that of the store-typing context. This context stores a mapping from locations to types, for example

$$\cdot, \ell_1 : \text{int ref}, \dots$$

The notes use Γ to represent this store typing context. Unfortunately, both later in the notes, and in the literature, we will require a mapping from *variables* to types, like

$$\cdot, x : \text{bool}, \dots$$

The name for this variable-typing context, both in the notes, and conventionally in the literature, is Γ . That is, in the course, Γ is *overloaded*: one syntax for two different semantics. I find that this is confusing and leads to avoidable mistakes. Hence, I use Σ to mean the store-typing context, and Γ to mean the variable-typing context.

2. **BNF vs Inductive Rules.** In your notes, and in other courses, you will probably see the syntax written out in Backus-Naur Form, or BNF:

$$e ::= e_1 \text{ op } e_2 \mid \dots$$

BNF is used as an alternative to inductive rules when the premises of the inductive rules are trivial. For example, the BNF rule above may be re-written

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

I use the inductive-rule form so that I can name the individual grammar rules, and reference them in the proof cases (since I do not need to name the type forming rules, they remain in BNF).

3. **Rule Naming.** In my convention, every rule name carries a prefix: E- for expression rules, R- for reduction (operational semantics) rules, and T- for typing rules. So, for example, the reduction rule for *deref* is named (R-DEREF), and the typing rule for *deref* is named (T-DEREF). In the notes, both are named (DEREF).

B L1

L1 SYNTAX

As in the notes, let n range over integers, b over booleans, ℓ over locations, and op over $\{+, \geq\}$.

EXPRESSIONS

$$\begin{array}{c}
\frac{}{n} \text{ (E-INT)} \quad \frac{}{b} \text{ (E-BOOL)} \quad \frac{}{\text{skip}} \text{ (E-SKIP)} \quad \frac{}{!\ell} \text{ (E-DEREF)} \quad \frac{e_1 \ e_2}{e_1 \text{ op } e_2} \text{ (E-OP)} \\
\frac{e}{\ell := e} \text{ (E-ASSIGN)} \quad \frac{e_1 \ e_2}{e_1; e_2} \text{ (E-SEQ)} \quad \frac{e_1 \ e_2 \ e_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} \text{ (E-IF)} \\
\frac{e_1 \ e_2}{\text{while } e_1 \text{ do } e_2 \text{ done}} \text{ (E-WHILE)}
\end{array}$$

VALUES

$$\text{value}(e) \iff e = n \vee e = b \vee e = \text{skip}$$

L1 OPERATIONAL SEMANTICS

v ranges over values. s is a store: a finite partial function from locations ℓ to integers n .

$$\begin{array}{c}
\frac{n = n_1 \pm n_2}{\langle n_1 + n_2, s \rangle \rightsquigarrow \langle n, s \rangle} \text{ (R-OP+)} \quad \frac{b = n_1 \geq n_2}{\langle n_1 \geq n_2, s \rangle \rightsquigarrow \langle b, s \rangle} \text{ (R-OP}\geq\text{)} \\
\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightsquigarrow \langle e'_1 \text{ op } e_2, s' \rangle} \text{ (R-OP1)} \quad \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} \text{ (R-OP2)} \\
\frac{s(\ell) = n}{\langle !\ell, s \rangle \rightsquigarrow \langle n, s \rangle} \text{ (R-DEREF)} \quad \frac{\ell \in \text{dom}(s)}{\langle \ell := n, s \rangle \rightsquigarrow \langle \text{skip}, s + \{\ell \mapsto n\} \rangle} \text{ (R-ASSIGN1)} \\
\frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \ell := e, s \rangle \rightsquigarrow \langle \ell := e', s' \rangle} \text{ (R-ASSIGN2)} \quad \frac{}{\langle \text{skip}; e_2, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-SEQ1)} \\
\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-SEQ2)} \quad \frac{}{\langle \text{if true then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-IF1)} \\
\frac{}{\langle \text{if false then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle e_3, s \rangle} \text{ (R-IF2)} \\
\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s' \rangle} \text{ (R-IF3)} \\
\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} \text{ (R-WHILE)}
\end{array}$$

TYPES

$$T ::= \text{int} \mid \text{bool} \mid \text{unit}$$

CONTEXTS

$$\Sigma ::= \cdot \mid \Sigma, \ell : \text{int ref}$$

RULES

$$\begin{array}{c}
\frac{}{\Sigma \vdash n : \text{int}} \text{ (T-INT)} \quad \frac{}{\Sigma \vdash b : \text{bool}} \text{ (T-BOOL)} \quad \frac{}{\Sigma \vdash \text{skip} : \text{unit}} \text{ (T-SKIP)} \\
\\
\frac{\Sigma \vdash e_1 : \text{int} \quad \Sigma \vdash e_2 : \text{int}}{\Sigma \vdash e_1 + e_2 : \text{int}} \text{ (T-OP+)} \quad \frac{\Sigma \vdash e_1 : \text{int} \quad \Sigma \vdash e_2 : \text{int}}{\Sigma \vdash e_1 \geq e_2 : \text{bool}} \text{ (T-OP}\geq\text{)} \\
\\
\frac{\Sigma(\ell) = \text{int ref}}{\Sigma \vdash !\ell : \text{int}} \text{ (T-DEREF)} \quad \frac{\Sigma(\ell) = \text{int ref} \quad \Sigma \vdash e : \text{int}}{\Sigma \vdash \ell := e : \text{unit}} \text{ (T-ASSIGN)} \\
\\
\frac{\Sigma \vdash e_1 : \text{unit} \quad \Sigma \vdash e_2 : T}{\Sigma \vdash e_1; e_2 : T} \text{ (T-SEQ)} \quad \frac{\Sigma \vdash e_1 : \text{bool} \quad \Sigma \vdash e_2 : \text{unit}}{\Sigma \vdash \text{while } e_1 \text{ do } e_2 \text{ done} : \text{unit}} \text{ (T-WHILE)} \\
\\
\frac{\Sigma \vdash e_1 : \text{bool} \quad \Sigma \vdash e_2 : T \quad \Sigma \vdash e_3 : T}{\Sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{ (T-IF)}
\end{array}$$

C L2

As in the notes, let n range over integers, b over booleans, ℓ over locations, x over variables, and op over $\{+, \geq\}$. • marks a rule that L2 adds, or states more generally than L1.

EXPRESSIONS

$$\begin{array}{c}
\frac{}{n} \text{ (E-INT)} \quad \frac{}{b} \text{ (E-BOOL)} \quad \frac{}{\text{skip}} \text{ (E-SKIP)} \quad \frac{}{!\ell} \text{ (E-DEREF)} \quad \frac{e_1 \ e_2}{e_1 \text{ op } e_2} \text{ (E-OP)} \\
\\
\frac{e}{\ell := e} \text{ (E-ASSIGN)} \quad \frac{e_1 \ e_2}{e_1; e_2} \text{ (E-SEQ)} \quad \frac{e_1 \ e_2 \ e_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} \text{ (E-IF)} \\
\\
\frac{e_1 \ e_2}{\text{while } e_1 \text{ do } e_2 \text{ done}} \text{ (E-WHILE)} \quad \frac{}{x} \text{ (E-VAR)} \bullet \quad \frac{e}{\text{fun } x : T \rightarrow e} \text{ (E-FUN)} \bullet \\
\\
\frac{e_1 \ e_2}{e_1 \ e_2} \text{ (E-APP)} \bullet \quad \frac{e_1 \ e_2}{\text{let } x : T = e_1 \text{ in } e_2} \text{ (E-LET)} \bullet \\
\\
\frac{e_1 \ e_2}{\text{let rec } f : T_1 \rightarrow T_2 = (\text{fun } x : T_1 \rightarrow e_1) \text{ in } e_2} \text{ (E-LETRECFUN)} \bullet
\end{array}$$

VALUES

$$\text{value}(e) \iff e = n \vee e = b \vee e = \text{skip} \vee e = \text{fun } x : T \rightarrow e'$$

v ranges over values. s is a store: a finite partial function from locations ℓ to integers n . $\bullet$ marks a rule that L2 adds, or states more generally than L1.

$$\begin{array}{c}
\frac{n = n_1 \pm n_2}{\langle n_1 + n_2, s \rangle \rightsquigarrow \langle n, s \rangle} \text{ (R-OP+)} \quad \frac{b = n_1 \geq n_2}{\langle n_1 \geq n_2, s \rangle \rightsquigarrow \langle b, s \rangle} \text{ (R-OP}\geq\text{)} \\
\\
\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightsquigarrow \langle e'_1 \text{ op } e_2, s' \rangle} \text{ (R-OP1)} \quad \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} \text{ (R-OP2)} \\
\\
\frac{s(\ell) = n}{\langle !\ell, s \rangle \rightsquigarrow \langle n, s \rangle} \text{ (R-DEREF)} \quad \frac{\ell \in \text{dom}(s)}{\langle \ell := n, s \rangle \rightsquigarrow \langle \text{skip}, s + \{\ell \mapsto n\} \rangle} \text{ (R-ASSIGN1)} \\
\\
\frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \ell := e, s \rangle \rightsquigarrow \langle \ell := e', s' \rangle} \text{ (R-ASSIGN2)} \quad \frac{}{\langle \text{skip}; e_2, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-SEQ1)} \\
\\
\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-SEQ2)} \quad \frac{}{\langle \text{if true then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-IF1)} \\
\\
\frac{}{\langle \text{if false then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle e_3, s \rangle} \text{ (R-IF2)} \\
\\
\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s' \rangle} \text{ (R-IF3)} \\
\\
\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} \text{ (R-WHILE)} \\
\\
\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)} \bullet \quad \frac{\langle e_2, s \rangle \rightsquigarrow \langle e'_2, s' \rangle}{\langle v e_2, s \rangle \rightsquigarrow \langle v e'_2, s' \rangle} \text{ (R-APP2)} \bullet \\
\\
\frac{}{\langle (\text{fun } x:T \rightarrow e) v, s \rangle \rightsquigarrow \langle e[v/x], s \rangle} \text{ (R-FUN)} \bullet \\
\\
\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle \text{let } x:T = e_1 \text{ in } e_2, s \rangle \rightsquigarrow \langle \text{let } x:T = e'_1 \text{ in } e_2, s' \rangle} \text{ (R-LET1)} \bullet \\
\\
\frac{}{\langle \text{let } x:T = v \text{ in } e_2, s \rangle \rightsquigarrow \langle e_2[v/x], s \rangle} \text{ (R-LET2)} \bullet \\
\\
\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)} \bullet
\end{array}$$

$\bullet$ marks a rule that L2 adds, or states more generally than L1.

TYPES

$$T ::= \text{int} \mid \text{bool} \mid \text{unit} \mid T \rightarrow T$$

CONTEXTS

$$\Sigma ::= \cdot \mid \Sigma, \ell : \text{int ref} \quad \Gamma ::= \cdot \mid \Gamma, x : T$$

RULES

$$\begin{array}{c}
\frac{}{\Sigma; \Gamma \vdash n : \text{int}} \text{(T-INT)} \quad \frac{}{\Sigma; \Gamma \vdash b : \text{bool}} \text{(T-BOOL)} \quad \frac{}{\Sigma; \Gamma \vdash \text{skip} : \text{unit}} \text{(T-SKIP)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : \text{int} \quad \Sigma; \Gamma \vdash e_2 : \text{int}}{\Sigma; \Gamma \vdash e_1 + e_2 : \text{int}} \text{(T-OP+)} \quad \frac{\Sigma; \Gamma \vdash e_1 : \text{int} \quad \Sigma; \Gamma \vdash e_2 : \text{int}}{\Sigma; \Gamma \vdash e_1 \geq e_2 : \text{bool}} \text{(T-OP}\geq\text{)} \\
\\
\frac{\Sigma(\ell) = \text{int ref}}{\Sigma; \Gamma \vdash !\ell : \text{int}} \text{(T-DEREF)} \quad \frac{\Sigma(\ell) = \text{int ref} \quad \Sigma; \Gamma \vdash e : \text{int}}{\Sigma; \Gamma \vdash \ell := e : \text{unit}} \text{(T-ASSIGN)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : \text{unit} \quad \Sigma; \Gamma \vdash e_2 : T}{\Sigma; \Gamma \vdash e_1; e_2 : T} \text{(T-SEQ)} \quad \frac{\Sigma; \Gamma \vdash e_1 : \text{bool} \quad \Sigma; \Gamma \vdash e_2 : \text{unit}}{\Sigma; \Gamma \vdash \text{while } e_1 \text{ do } e_2 \text{ done} : \text{unit}} \text{(T-WHILE)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : \text{bool} \quad \Sigma; \Gamma \vdash e_2 : T \quad \Sigma; \Gamma \vdash e_3 : T}{\Sigma; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-IF)} \quad \frac{\Gamma(x) = T}{\Sigma; \Gamma \vdash x : T} \text{(T-VAR)} \bullet \\
\\
\frac{\Sigma; \Gamma, x : T_1 \vdash e : T_2}{\Sigma; \Gamma \vdash \text{fun } x : T_1 \rightarrow e : T_1 \rightarrow T_2} \text{(T-FUN)} \bullet \quad \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)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : T_1 \quad \Sigma; \Gamma, x : T_1 \vdash e_2 : T_2}{\Sigma; \Gamma \vdash \text{let } x : T_1 = e_1 \text{ in } e_2 : T_2} \text{(T-LET)} \bullet \\
\\
\frac{\Sigma; \Gamma, f : T_1 \rightarrow T_2, x : T_1 \vdash e_1 : T_2 \quad \Sigma; \Gamma, f : T_1 \rightarrow T_2 \vdash e_2 : T_3}{\Sigma; \Gamma \vdash \text{let rec } f : T_1 \rightarrow T_2 = (\text{fun } x : T_1 \rightarrow e_1) \text{ in } e_2 : T_3} \text{(T-LETRECFUN)} \bullet
\end{array}$$

D L3

L3 SYNTAX

As in the notes, let n range over integers, b over booleans, ℓ over locations, x over variables, lab over labels, and op over $\{+, \geq\}$. No label occurs more than once in a record type or a record expression. $\bullet$ marks a rule that L3 adds, or states more generally than L2.

EXPRESSIONS

$$\begin{array}{c}
\frac{}{n} \text{(E-INT)} \quad \frac{}{b} \text{(E-BOOL)} \quad \frac{}{\text{skip}} \text{(E-SKIP)} \quad \frac{}{\ell} \text{(E-LOC)} \bullet \quad \frac{}{x} \text{(E-VAR)} \\
\\
\frac{e_1 \quad e_2}{e_1 \text{ op } e_2} \text{(E-OP)} \quad \frac{e}{\text{ref } e} \text{(E-REF)} \bullet \quad \frac{e}{!\ell} \text{(E-DEREF)} \bullet \quad \frac{e_1 \quad e_2}{e_1 := e_2} \text{(E-ASSIGN)} \bullet \\
\\
\frac{e_1 \quad e_2}{e_1; e_2} \text{(E-SEQ)} \quad \frac{e_1 \quad e_2 \quad e_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} \text{(E-IF)} \quad \frac{e_1 \quad e_2}{\text{while } e_1 \text{ do } e_2 \text{ done}} \text{(E-WHILE)} \\
\\
\frac{e}{\text{fun } x : T \rightarrow e} \text{(E-FUN)} \quad \frac{e_1 \quad e_2}{e_1 e_2} \text{(E-APP)} \quad \frac{e_1 \quad e_2}{\text{let } x : T = e_1 \text{ in } e_2} \text{(E-LET)} \\
\\
\frac{e_1 \quad e_2}{\text{let rec } f : T_1 \rightarrow T_2 = (\text{fun } x : T_1 \rightarrow e_1) \text{ in } e_2} \text{(E-LETRECFUN)} \quad \frac{e_1 \quad e_2}{(e_1, e_2)} \text{(E-PAIR)} \bullet \\
\\
\frac{e}{\text{fst } e} \text{(E-FST)} \bullet \quad \frac{e}{\text{snd } e} \text{(E-SND)} \bullet \quad \frac{e}{\text{inl } e : T} \text{(E-INL)} \bullet \quad \frac{e}{\text{inr } e : T} \text{(E-INR)} \bullet
\end{array}$$

$$\frac{e \quad e_1 \quad e_2}{\text{match } e \text{ with } \text{inl } (x_1 : T_1) \rightarrow e_1 \mid \text{inr } (x_2 : T_2) \rightarrow e_2} \text{ (E-MATCH)} \bullet$$

$$\frac{e_1 \quad \dots \quad e_k}{\{\text{lab}_1 = e_1, \dots, \text{lab}_k = e_k\}} \text{ (E-RECORD)} \bullet \quad \frac{e}{e.\text{lab}} \text{ (E-RECORDPROJ)} \bullet$$

VALUES

$$\begin{aligned} \text{value}(e) \iff & e = n \vee e = b \vee e = \text{skip} \vee e = \ell \vee e = \text{fun } x : T \rightarrow e' \\ & \vee e = (v_1, v_2) \vee e = \text{inl } v : T \vee e = \text{inr } v : T \\ & \vee e = \{\text{lab}_1 = v_1, \dots, \text{lab}_k = v_k\} \end{aligned}$$

L3 OPERATIONAL SEMANTICS

v ranges over values. s is a store: a finite partial function from locations ℓ to values v . $\bullet$ marks a rule that L3 adds, or states more generally than L2.

$$\begin{aligned} & \frac{n = n_1 \pm n_2}{\langle n_1 + n_2, s \rangle \rightsquigarrow \langle n, s \rangle} \text{ (R-OP+)} & \frac{b = n_1 \geq n_2}{\langle n_1 \geq n_2, s \rangle \rightsquigarrow \langle b, s \rangle} \text{ (R-OP}\geq\text{)} \\ & \frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightsquigarrow \langle e'_1 \text{ op } e_2, s' \rangle} \text{ (R-OP1)} & \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} \text{ (R-OP2)} \\ & \frac{\ell \notin \text{dom}(s)}{\langle \text{ref } v, s \rangle \rightsquigarrow \langle \ell, s + \{\ell \mapsto v\} \rangle} \text{ (R-REF1)} \bullet & \frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \text{ref } e, s \rangle \rightsquigarrow \langle \text{ref } e', s' \rangle} \text{ (R-REF2)} \bullet \\ & \frac{s(\ell) = v}{\langle !\ell, s \rangle \rightsquigarrow \langle v, s \rangle} \text{ (R-DEREF1)} \bullet & \frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle !e, s \rangle \rightsquigarrow \langle !e', s' \rangle} \text{ (R-DEREF2)} \bullet \\ & \frac{\ell \in \text{dom}(s)}{\langle \ell := v, s \rangle \rightsquigarrow \langle \text{skip}, s + \{\ell \mapsto v\} \rangle} \text{ (R-ASSIGN1)} \bullet \\ & \frac{\langle e_2, s \rangle \rightsquigarrow \langle e'_2, s' \rangle}{\langle \ell := e_2, s \rangle \rightsquigarrow \langle \ell := e'_2, s' \rangle} \text{ (R-ASSIGN2)} & \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-ASSIGN3)} \bullet \\ & \frac{}{\langle \text{skip}; e_2, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-SEQ1)} & \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-SEQ2)} \\ & \frac{}{\langle \text{if true then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle e_2, s \rangle} \text{ (R-IF1)} \\ & \frac{}{\langle \text{if false then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle e_3, s \rangle} \text{ (R-IF2)} \\ & \frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \rightsquigarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s' \rangle} \text{ (R-IF3)} \\ & \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} \text{ (R-WHILE)} \\ & \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)} & \frac{\langle e_2, s \rangle \rightsquigarrow \langle e'_2, s' \rangle}{\langle v e_2, s \rangle \rightsquigarrow \langle v e'_2, s' \rangle} \text{ (R-APP2)} \end{aligned}$$

$$\begin{array}{c}
\frac{}{\langle (\text{fun } x:T \rightarrow e) v, s \rangle \rightsquigarrow \langle e[v/x], s \rangle} \text{ (R-FUN)} \\
\\
\frac{\langle e_1, s \rangle \rightsquigarrow \langle e'_1, s' \rangle}{\langle \text{let } x:T = e_1 \text{ in } e_2, s \rangle \rightsquigarrow \langle \text{let } x:T = e'_1 \text{ in } e_2, s' \rangle} \text{ (R-LET1)} \\
\\
\frac{}{\langle \text{let } x:T = v \text{ in } e_2, s \rangle \rightsquigarrow \langle e_2[v/x], s \rangle} \text{ (R-LET2)} \\
\\
\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)} \\
\\
\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-PAIR1)} \bullet \quad \frac{\langle e_2, s \rangle \rightsquigarrow \langle e'_2, s' \rangle}{\langle (v_1, e_2), s \rangle \rightsquigarrow \langle (v_1, e'_2), s' \rangle} \text{ (R-PAIR2)} \bullet \\
\\
\frac{}{\langle \text{fst } (v_1, v_2), s \rangle \rightsquigarrow \langle v_1, s \rangle} \text{ (R-PROJ1)} \bullet \quad \frac{}{\langle \text{snd } (v_1, v_2), s \rangle \rightsquigarrow \langle v_2, s \rangle} \text{ (R-PROJ2)} \bullet \\
\\
\frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \text{fst } e, s \rangle \rightsquigarrow \langle \text{fst } e', s' \rangle} \text{ (R-PROJ3)} \bullet \quad \frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \text{snd } e, s \rangle \rightsquigarrow \langle \text{snd } e', s' \rangle} \text{ (R-PROJ4)} \bullet \\
\\
\frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \text{inl } e:T, s \rangle \rightsquigarrow \langle \text{inl } e':T, s' \rangle} \text{ (R-INL)} \bullet \quad \frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \text{inr } e:T, s \rangle \rightsquigarrow \langle \text{inr } e':T, s' \rangle} \text{ (R-INR)} \bullet \\
\\
\frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle \text{match } e \text{ with inl } (x_1:T_1) \rightarrow e_1 \mid \text{inr } (x_2:T_2) \rightarrow e_2, s \rangle \rightsquigarrow \langle \text{match } e' \text{ with inl } (x_1:T_1) \rightarrow e_1 \mid \text{inr } (x_2:T_2) \rightarrow e_2, s' \rangle} \text{ (R-MATCH1)} \bullet \\
\\
\frac{}{\langle \text{match inl } v:T \text{ with inl } (x_1:T_1) \rightarrow e_1 \mid \text{inr } (x_2:T_2) \rightarrow e_2, s \rangle \rightsquigarrow \langle e_1[v/x_1], s \rangle} \text{ (R-MATCH2)} \bullet \\
\\
\frac{}{\langle \text{match inr } v:T \text{ with inl } (x_1:T_1) \rightarrow e_1 \mid \text{inr } (x_2:T_2) \rightarrow e_2, s \rangle \rightsquigarrow \langle e_2[v/x_2], s \rangle} \text{ (R-MATCH3)} \bullet \\
\\
\frac{\langle e_i, s \rangle \rightsquigarrow \langle e'_i, s' \rangle}{\langle \{\text{lab}_1 = v_1, \dots, \text{lab}_i = e_i, \dots, \text{lab}_k = e_k\}, s \rangle \rightsquigarrow \langle \{\text{lab}_1 = v_1, \dots, \text{lab}_i = e'_i, \dots, \text{lab}_k = e_k\}, s' \rangle} \text{ (R-RECORD1)} \bullet \\
\\
\frac{}{\langle \{\text{lab}_1 = v_1, \dots, \text{lab}_k = v_k\}. \text{lab}_i, s \rangle \rightsquigarrow \langle v_i, s \rangle} \text{ (R-RECORD2)} \bullet \\
\\
\frac{\langle e, s \rangle \rightsquigarrow \langle e', s' \rangle}{\langle e. \text{lab}_i, s \rangle \rightsquigarrow \langle e'. \text{lab}_i, s' \rangle} \text{ (R-RECORD3)} \bullet
\end{array}$$

L3 TYPING

- marks a rule that L3 adds, or states more generally than L2.

TYPES

$$T ::= \text{int} \mid \text{bool} \mid \text{unit} \mid T_1 \rightarrow T_2 \mid T_1 * T_2 \mid T_1 + T_2 \mid \{\text{lab}_1:T_1, \dots, \text{lab}_k:T_k\} \mid T \text{ ref}$$

CONTEXTS

$$\Sigma ::= \cdot \mid \Sigma, \ell : T \text{ ref} \quad \Gamma ::= \cdot \mid \Gamma, x : T$$

RULES

$$\begin{array}{c}
\frac{}{\Sigma; \Gamma \vdash n : \text{int}} \text{(T-INT)} \quad \frac{}{\Sigma; \Gamma \vdash b : \text{bool}} \text{(T-BOOL)} \quad \frac{}{\Sigma; \Gamma \vdash \text{skip} : \text{unit}} \text{(T-SKIP)} \\
\\
\frac{\Sigma(\ell) = T \text{ ref}}{\Sigma; \Gamma \vdash \ell : T \text{ ref}} \text{(T-LOC)} \bullet \quad \frac{\Gamma(x) = T}{\Sigma; \Gamma \vdash x : T} \text{(T-VAR)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : \text{int} \quad \Sigma; \Gamma \vdash e_2 : \text{int}}{\Sigma; \Gamma \vdash e_1 + e_2 : \text{int}} \text{(T-OP+)} \quad \frac{\Sigma; \Gamma \vdash e_1 : \text{int} \quad \Sigma; \Gamma \vdash e_2 : \text{int}}{\Sigma; \Gamma \vdash e_1 \geq e_2 : \text{bool}} \text{(T-OP}\geq\text{)} \\
\\
\frac{\Sigma; \Gamma \vdash e : T}{\Sigma; \Gamma \vdash \text{ref } e : T \text{ ref}} \text{(T-REF)} \bullet \quad \frac{\Sigma; \Gamma \vdash e : T \text{ ref}}{\Sigma; \Gamma \vdash !e : T} \text{(T-DEREF)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : T \text{ ref} \quad \Sigma; \Gamma \vdash e_2 : T}{\Sigma; \Gamma \vdash e_1 := e_2 : \text{unit}} \text{(T-ASSIGN)} \bullet \quad \frac{\Sigma; \Gamma \vdash e_1 : \text{unit} \quad \Sigma; \Gamma \vdash e_2 : T}{\Sigma; \Gamma \vdash e_1; e_2 : T} \text{(T-SEQ)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : \text{bool} \quad \Sigma; \Gamma \vdash e_2 : \text{unit}}{\Sigma; \Gamma \vdash \text{while } e_1 \text{ do } e_2 \text{ done} : \text{unit}} \text{(T-WHILE)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : \text{bool} \quad \Sigma; \Gamma \vdash e_2 : T \quad \Sigma; \Gamma \vdash e_3 : T}{\Sigma; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-IF)} \quad \frac{\Sigma; \Gamma, x : T_1 \vdash e : T_2}{\Sigma; \Gamma \vdash \text{fun } x : T_1 \rightarrow e : T_1 \rightarrow T_2} \text{(T-FUN)} \\
\\
\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)} \quad \frac{\Sigma; \Gamma \vdash e_1 : T_1 \quad \Sigma; \Gamma, x : T_1 \vdash e_2 : T_2}{\Sigma; \Gamma \vdash \text{let } x : T_1 = e_1 \text{ in } e_2 : T_2} \text{(T-LET)} \\
\\
\frac{\Sigma; \Gamma, f : T_1 \rightarrow T_2, x : T_1 \vdash e_1 : T_2 \quad \Sigma; \Gamma, f : T_1 \rightarrow T_2 \vdash e_2 : T_3}{\Sigma; \Gamma \vdash \text{let rec } f : T_1 \rightarrow T_2 = (\text{fun } x : T_1 \rightarrow e_1) \text{ in } e_2 : T_3} \text{(T-LETRECFUN)} \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : T_1 \quad \Sigma; \Gamma \vdash e_2 : T_2}{\Sigma; \Gamma \vdash (e_1, e_2) : T_1 * T_2} \text{(T-PAIR)} \bullet \quad \frac{\Sigma; \Gamma \vdash e : T_1 * T_2}{\Sigma; \Gamma \vdash \text{fst } e : T_1} \text{(T-PROJ1)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e : T_1 * T_2}{\Sigma; \Gamma \vdash \text{snd } e : T_2} \text{(T-PROJ2)} \bullet \quad \frac{\Sigma; \Gamma \vdash e : T_1}{\Sigma; \Gamma \vdash (\text{inl } e : T_1 + T_2) : T_1 + T_2} \text{(T-INL)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e : T_2}{\Sigma; \Gamma \vdash (\text{inr } e : T_1 + T_2) : T_1 + T_2} \text{(T-INR)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e : T_1 + T_2 \quad \Sigma; \Gamma, x_1 : T_1 \vdash e_1 : T \quad \Sigma; \Gamma, x_2 : T_2 \vdash e_2 : T}{\Sigma; \Gamma \vdash \text{match } e \text{ with inl } (x_1 : T_1) \rightarrow e_1 \mid \text{inr } (x_2 : T_2) \rightarrow e_2 : T} \text{(T-MATCH)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e_1 : T_1 \quad \dots \quad \Sigma; \Gamma \vdash e_k : T_k}{\Sigma; \Gamma \vdash \{\text{lab}_1 = e_1, \dots, \text{lab}_k = e_k\} : \{\text{lab}_1 : T_1, \dots, \text{lab}_k : T_k\}} \text{(T-RECORD)} \bullet \\
\\
\frac{\Sigma; \Gamma \vdash e : \{\text{lab}_1 : T_1, \dots, \text{lab}_k : T_k\}}{\Sigma; \Gamma \vdash e.\text{lab}_i : T_i} \text{(T-RECORDPROJ)} \bullet
\end{array}$$