Semantics of Programming Languages
Exercise Sheet

Andrej Ivašković (`ai294`)
Compiled on: 31st October 2019

Before attempting the problems

The goal of this course is to introduce you to a formal way of reasoning about programs. You consider the operational approach to formal semantics of a programming language (you might see other approaches in Part II). It is vital that you do not forget why we are looking at these issues – everything here is based on real programming languages, and understanding what programs do and translating them into a formal language is important if we wish to mechanise program verification. Formal verification is gaining more traction, and programming languages are seeing more versatile type systems nowadays.

The lecture notes contain more 'drill' exercises on typing and deriving transitions. You are more than welcome to attempt some of them, though the main lesson you will learn is that you need to leave a sufficient amount of space for the proof tree. Other exercises encourage you to play with the interpreter – please do, it will give you some valuable background for *Compiler Construction*.

You should know off the top of your head the main concepts and definitions, especially properties of typing. Consider compiling a glossary.

1 L1 and types

Exercise 1.1. What are the first eight reduction steps of the expression

$$\textbf{while } !l_1 \geq 0 \textbf{ do } (l_2 := !l_2 + !l_3;\ l_1 := !l_1 - 1)$$

when $\{l_1 \mapsto 3, l_2 \mapsto 1, l_3 \mapsto 5\}$ is the initial store? Show the proof trees for the first two of these reductions.

Exercise 1.2. What is the type of the expression

$$\textbf{if } !l_2 \geq !l_1 \textbf{ then } l_1 := !l_2 \textbf{ else skip};\ !l_1,$$

given the typing environment $l_1 : \mathrm{intref}, l_2 : \mathrm{intref}$? Show the typing derivation.

Exercise 1.3. Explain why the following is not an L1 rule:

$$\frac{\langle e_1,\ s\rangle \longrightarrow \langle e_1',\ s'\rangle}{\langle \textbf{while}\ e_1\ \textbf{do}\ e_2,\ s\rangle \longrightarrow \langle \textbf{while}\ e_1'\ \textbf{do}\ e_2,\ s'\rangle}$$

Exercise 1.4. State the *Progress*, *Safety* and *Type Preservation* properties for a particular programming language (both informally and formally). Which ones are consequences of the other two?

Exercise 1.5. L1 stores contain only integer references. How can you handle references of any type? State the additional (or revised) operational semantics and typing rules.

Exercise 1.6. In this exercise we look at a simple imperative language that extends L1 that adds handling fixed-length arrays whose elements are all integers. Define the syntax, operational semantics and type system for this language. There should be dedicated expressions for accessing the array item at a particular index, as well as creating a new array of a given length whose items are all the same value. Is your language type safe?

(This problem is underspecified – you are free to interpret it in any way you want, as long as you state your assumptions and justify them.)

2 Proofs by induction

Exercise 2.1. Complete the proof of Progress for L1.

Exercise 2.2. State and prove Uniqueness of Typing for L1. What proof principle are you using?

Exercise 2.3. The lecture notes present operational semantics in *small-step* style: only 'one step' is observed in a reduction sequence, and the reflexive transitive closure $\longrightarrow^*$ of the transition relation $\longrightarrow$ says something about eventually reaching a state. Contrary to that, *big-step* semantics consider transitions of the form $\langle e, s_1\rangle \Downarrow \langle v, s_2\rangle$, where e is an expression, v is a value, and s_1 and s_2 are stores – meaning that the expression e will, when the store state is s_1, eventually reduce to a value v, with the state of the store s_2. For example:

$$\langle \textbf{if}\ !l_1 = 5\ \textbf{then}\ l_2 := 0\ \textbf{else}\ l_1 := 5,\ \{l_1 \mapsto 3, l_2 \mapsto 1\}\rangle \Downarrow \langle \textbf{skip},\ \{l_1 \mapsto 5, l_2 \mapsto 1\}\rangle$$

(*a*) State the big-step operational semantics of L1.

(*b*) Prove that, if $\langle e,\ s_1\rangle \Downarrow \langle v,\ s_2\rangle$ (according to your big step operational semantics), then $\langle e,\ s_1\rangle \longrightarrow^* \langle v,\ s_2\rangle$. Does the converse hold?

3 L2 and functions

Exercise 3.1. Let e be the following closed L2 expression:

$$e = \mathbf{fn}\ x : \alpha \Rightarrow (\mathbf{fn}\ y : \beta \Rightarrow ((\mathbf{fn}\ x : (\beta \to \alpha) \Rightarrow x\ y)\ (\mathbf{fn}\ z : \beta \Rightarrow x)))$$

(a) What is the type of e? Show the typing derivation.

(b) Does e eventually reduce to a value? If so, what value? If not, why?

(c) What is the De Bruijn representation of e?

Exercise 3.2.

(a) Give an example of a well-typed L2 configuration that eventually reduces to a value with both call-by-value and call-by-name calling semantics, but the values differ.

(b) Does there exist a well-typed L2 configuration that eventually reduces to a value in call-by-value semantics, but loops forever in call-by-name semantics?

(c) Does there exist a well-typed L2 configuration that eventually reduces to a value in call-by-name semantics, but loops forever in call-by-value semantics?

Exercise 3.3. Define the operational semantics for *call-by-need* calling semantics in a pure functional programming language, so that the actual parameters in function calls are evaluated at most once.

Exercise 3.4. Prove Type Preservation for L2.

Hint. You might need to state and prove a substitution lemma.

Exercise 3.5. It this exercise we look at the following purely functional programming language (that is, there is no mutable store) with support for handling ML-style lists. Its expressions are given by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid \mathbf{fn}\ x : T \Rightarrow e_1 \mid e_1\ e_2 \mid [\,]_T \mid e_1 :: e_2 \\ & \mid & \mathbf{case}\ e_1\ \mathbf{of}\ ([\,]_T \Rightarrow e_2 \mid x_1 :: x_2 \Rightarrow e_3) \\ & \mid & \mathbf{let\ val}\ x : T = e_1\ \mathbf{in}\ e_2 \mid \mathbf{let\ rec}\ x : T = e_1\ \mathbf{in}\ e_2 \end{array}$$

The **case** syntax distinguishes the cases of an empty and a non-empty list: if e_1 is empty, the expression evaluates to e_2; otherwise, it evaluates to e_3, which might make use of the head (fresh variable x_1) and the tail (fresh x_2) of the list.

(a) Define the operational semantics and the type system for this language. The language should have call-by-value calling semantics and the semantics should satisfy all 'desirable' typing properties (Progress, Safety and others).

(b) Write a curried function *append* in this language that takes two lists, ℓ_1 and ℓ_2, and returns the concatenated list (like ML $\ell_1 @ \ell_2$). Infer the type of *append* using your type system.

(*c*) State Progress for this language. What proof principle is used to prove it? Show it on the case of the three expressions dealing with lists.

4 L3 and data

Exercise 4.1. Show how the reduction sequences for $\langle e_1, \{\}\rangle$ and $\langle e_2, \{\}\rangle$ differ, where $e_1 = (\mathbf{ref}\ 0,\ \mathbf{ref}\ 0)$ and $e_2 = \mathbf{let\ val}\ x : \mathsf{int\ ref} = \mathbf{ref}\ 0\ \mathbf{in}\ (x,\ x)$.

Exercise 4.2. The lecture notes introduce the *Curry-Howard correspondence*, where types correspond to theorems and programs correspond to proofs of said theorems. For example, the theorem $\{\} \vdash P \land (P \to Q) \to Q$ corresponds to the type $T_1 * (T_1 \to T_2) \to T_2$ in an empty context. A closed term of this type, corresponding to a proof of the theorem, is:

$$\mathbf{fn}\ x : (T_1 * (T_1 \to T_2)) \Rightarrow ((\#2\ x)\ (\#1\ x))$$

Give terms that correspond to proofs of the following theorems:

(*a*) $\{\} \vdash (P \land Q) \lor (\neg P \land R) \to Q \lor R$

(*b*) $(P \land Q) \lor R \vdash (P \lor R) \land (Q \lor R)$

(*c*) $(P \lor R), (Q \lor R) \vdash (P \land Q) \lor R$

Exercise 4.3. Design type rules and evaluation rules for ML-style exceptions. Start with exceptions that do not carry any values.

Hint 1. Take care with nested handlers within recursive functions.

Hint 2. You might want to express your semantics using evaluation contexts.

5 Subtyping

Exercise 5.1.

(*a*) Explain the reasoning behind the subtyping rule for function types.

(*b*) For each of the two bogus T ref subtype rules on slide 202, give an example program that is typable with that rule but gets stuck at runtime.

Exercise 5.2. For each of the following, either give a type derivation or explain why it is untypable:

(*a*) $\{\} \vdash \{\mathrm{p} = \{\mathrm{p} = \{\mathrm{p} = \{\mathrm{p} = 3\}\}\}\} : \{\mathrm{p} : \{\}\}$

(*b*) $\{\} \vdash \mathbf{fn}\ x : \{\mathrm{p} : \mathsf{bool}, \mathrm{q} : \{\mathrm{p} : \mathsf{int}, \mathrm{q} : \mathsf{bool}\}\} \Rightarrow \#q\ \#p\ x : ?$

(*c*) $\{\} \vdash \mathbf{fn}\ x : \{\mathrm{p} : \mathsf{int}\} \to \mathsf{int} \Rightarrow (f\ \{\mathrm{q} = 3\}) + (f\ \{\mathrm{p} = 4\}) : ?$

(*d*) $\{\} \vdash \mathbf{fn}\ x : \{\mathrm{p} : \mathsf{int}\} \to \mathsf{int} \Rightarrow (f\ \{\mathrm{q} = 3, \mathrm{p} = 2\}) + (f\ \{\mathrm{p} = 4\}) : ?$

Exercise 5.3. State the subtyping rules for sums, $\textbf{let val}\ x : e_1 = T\ \textbf{in}\ e_2$ and $\textbf{let rec}\ x : e_1 = T\ \textbf{in}\ e_2$.

6 Concurrency

Exercise 6.1. Show all possible reduction sequences for $e_1 \parallel e_2$ from the initial store $\{l_1 \mapsto 10, l_2 \mapsto 40\}$ and no locks acquired, where:

$$\begin{aligned} e_1 &= \textbf{lock}\ m;\ l_1 := !l_1 - 2;\ l_2 := !l_1 + 1;\ \textbf{unlock}\ m \\ e_2 &= \textbf{lock}\ m;\ l_2 := !l_2 + 3;\ l_1 := !l_1 - 3;\ \textbf{unlock}\ m \end{aligned}$$

Show the derivations of the possible candidates for the first reduction.

Exercise 6.2. We sometimes extend type systems with additional information, giving rise to *type-and-effect systems*. The judgements are now $\Gamma \vdash e : T \& F$, where F is an effect (meaning and structure depends on usage). In this exercise, we will be looking at the concurrent language from the notes (if the language has functions, we add latent effect annotations to function types: $T_1 \xrightarrow{F} T_2$).

Let F represent the sequence of locks and unlocks of mutexes. It will be represented by a list over the set $\{L_m, U_m \mid m \in \mathbb{M}\}$. For example, $[L_{m_1}, L_{m_2}, U_{m_2}, U_{m_1}]$ is the effect of an expression that first acquires the lock m_1, then m_2, and then unlocks m_2 and m_1.

Devise a type-and-effect system that only accepts those concurrent programs obeying the O2PL discipline (concurrent expressions with inadequate locking disciplines do not type check).

Exercise 6.3. Attempt *2014 Paper 6 Question 9* (very challenging!).

7 Semantic equivalence

Exercise 7.1. Let e_1 and e_2 be expressions and Γ_1 and Γ_2 be contexts such that $\Gamma_1 \vdash e_1 : \mathsf{unit}$ and $\Gamma_2 \vdash e_2 : \mathsf{unit}$. Show that, if Γ_1 and Γ_2 are disjoint, then $e_1;\ e_2 \simeq_{\Gamma}^{\mathsf{unit}} e_2;\ e_1$, where $\Gamma = \Gamma_1 \cup \Gamma_2$.

Exercise 7.2. The following L3 judgements hold:

$$\begin{aligned} l : \mathsf{int\ ref} &\vdash l := 0 : \mathsf{unit} \\ l : \mathsf{int\ ref} &\vdash l := 1 : \mathsf{unit} \end{aligned}$$

Show that these two assignments are not contextually equivalent.

Exercise 7.3. Prove or disprove Conjectures 30, 31, 32 from the notes.