CIS 7000-1 Homework 5

NAME: FILL IN HERE

November 17, 2025

1 Stack-based operational semantics

In this problem you will design a stack based operational semantics for STLC with natural numbers and let expressions. For reference, the file `homework/rocq/hw5/stlc/stack.v` contains definitions to help you get started.

1. Define the syntax of stack frames and rules for a small-step stack-based semantics (i.e. the judgement $\langle s, e\rangle \mapsto \langle s', e'\rangle$). We have given you the frame and two rules associated with let expressions to get started.

   Note that because STLC is not a fine grained language, you will need to define stack frames for application, **succ**, and **nrec** terms. Each rule of your small-step stack-based semantics should be an axiom, i.e., should not include any premises in order for the rule to apply.

   Definition 1.1 (Frame Grammar)**.**

   $$\mathit{frame} ::= \mathbf{let}\ x\ =\ \_\ \mathbf{in}\ e \mid \mathit{ADD\ new\ frames\ here}\ldots$$

   Definition 1.2 (Stack-based small-step semantics)**.**

   $\boxed{\langle s, e\rangle \mapsto \langle s', e'\rangle}$ *(Add rules here)*

   SSM-LETV

   $$\frac{}{\langle \mathbf{let}\ x\ =\ \_\ \mathbf{in}\ e_2 : s, v\rangle \mapsto \langle s, e_2[v/x]\rangle}$$

   SSM-PUSH

   $$\frac{}{\langle s, \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2\rangle \mapsto \langle \mathbf{let}\ x\ =\ \_\ \mathbf{in}\ e_2 : s, e_1\rangle}$$

2. As part of making sure that your definitions produce the same semantics as our original semantics for STLC, you should prove several lemmas. One lemma, shown below, connects the big-step semantics to the stack based semantics.

   Lemma 1.1 (Bigstep completeness)**.** $e \Rightarrow v$ implies $\langle s, e\rangle \mapsto^* \langle s, v\rangle$ for any s.

   This proof proceeds by induction on the derivation of $e \Rightarrow v$. Typeset the case when the last rule used was rule BS-APP.

   BS-APP

   $$\frac{e_2 \Rightarrow v_1 \qquad \begin{array}{c} e_1 \Rightarrow \lambda x.e_1' \\ e_1'[v_1/x] \Rightarrow v_2 \end{array}}{e_1\ e_2 \Rightarrow v_2}$$

3. Complete the definition of the "unravel function", written $s\{e\}$ and prove the following lemma connecting the original small-step semantics with your new evaluation frames.

   Definition 1.3 (Unravel)**.** The unravel operation, written $s\{e\}$, takes a stack s and expression e and produces a combined expression.

$$\begin{array}{lcl} [\,]\{e\} & = & e \\ (\mathbf{let}\, x\, =\, \_\, \mathbf{in}\, e_2 : s)\{e\} & = & s\{\mathbf{let}\, x\, =\, e\, \mathbf{in}\, e_2\} \\ \textit{FILL IN HERE} & & \end{array}$$

   Lemma 1.2 (Stack congruence)**.** If $e \rightsquigarrow e'$ then $s\{e\} \rightsquigarrow s\{e'\}$.

2 Exceptions, Control Operators and Effect Handling

The next problems concern extension of the stack-based version of REC with various control operators. We call this base language CONTROL. For reference, the file `homework/rocq/hw5/control/control.v` contains definitions to help you get started.

4. In homework 1, we saw that the extension of STLC with lists and indexing was not type sound. In this problem, we will use exception throwing to safely extend the CONTROL language with list indexing.

$$\textsc{T-NTH}\ \frac{\Gamma \vdash e_1 \in \mathbf{List}\,\tau \quad \Gamma \vdash e_2 \in \mathbf{Nat}}{\Gamma \vdash e_1\ e_2 \in \tau} \qquad \textsc{S-APP-ZERO}\ \frac{}{(\mathbf{cons}\,v_1\,v_2)\ 0 \rightsquigarrow v_1} \qquad \textsc{S-APP-SUCC}\ \frac{}{(\mathbf{cons}\,v_1\,v_2)\ (\mathbf{S}\,k) \rightsquigarrow v_2\ k}$$

The problem is that indexing from an empty list is *stuck*. We can recover soundness by adding a new rule that raises an exception in this case. (This rule raises exception "0", which we can call "NoSuchElementException".)

$$\textsc{S-APP-NIL}\ \frac{}{\mathbf{nil}\ v \rightsquigarrow \mathbf{raise}\,0}$$

Prove the preservation preservation lemma for the CONTROL language with this extension. In your proof, you need only show the case for the rule T-NTH.

Lemma 2.1 (Primitive Preservation)**.** If $\vdash e \in \tau$ and $e \twoheadrightarrow e'$. The $\vdash e' \in \tau$.

Proof. By induction on $\vdash e \in \tau$.

- If rule T-NTH was the last rule used, then ....

□

5. The term **exit** v *immediately* terminates the execution of a program returning the value v, discarding any computation that is in progress.

   For example, we have:

   $$\langle [], (\lambda x.\mathbf{let}\ z\ =\ \mathbf{exit}\ x\ \mathbf{in}\ 1 + z)\ 3\rangle \mapsto^* \langle [], \mathbf{ret}\ 3\rangle$$

   What are the typing rules and operational semantics for **exit** v? You can add these rules to the Rocq development to make sure that they do not violate type safety.

6. Effect handlers in OCaml also include a **discontinue** term. See the OCaml Manual (https://ocaml.org/manual/5.3/effects.html) for more information.

   This term takes two values as arguments: a continuation and an exception value (i.e., a nat in our simple language). On execution, it jumps to that saved stack and raises the given exception.

   $$\textsc{ssm-discontinue}\ \frac{}{\langle s, \mathbf{discontinue}\ (\mathbf{cont}\ s')\ v\rangle \mapsto \langle s', \mathbf{raise}\ v\rangle}$$

   Design a typing rule for this term and prove the corresponding cases of the preservation and progress lemmas.

   Lemma 2.2 (Machine Preservation)**.** If $\vdash m\ \mathsf{ok}$ and $m \mapsto m'$. The $\vdash m'\ \mathsf{ok}$.

   Proof. By inversion on the typing and step judgements.

   - If the machine steps by rule SSM-DISCONTINUE, then ....

   □

   Lemma 2.3 (Machine Progress)**.** If $\vdash m\ \mathsf{ok}$ and there is no m' such that $m \mapsto m'$, m is a final machine state.

   Proof. By induction on the typing judgement.

   - If the last rule used was rule T-DISCONTINUE, then ....

   □

3 A type-and-effect system for exception handling

7. The file homework/rocq/hw5/exn/exn.v contains the definition of a type-and-effect system for exception handling. This language is like the CONTROL language, but for simplicity, only includes the exception handling extension and does not include recursive values or types.

   To make it easier to statically track exception, there are two small modifications to **raise** and **try**. First, raise terms must be called with concrete exception values—i.e. we need to know the identity of the exception that has been raised. Secondly, when installing an exception handler with **try**, the exception handler applies to only a single exception (indicated by the natural number k) and does not catch any other exception value. (In this language there is no way to catch all exceptions.) We've changed the syntax of the **try** term to make this explicit.

$$\begin{array}{llll} \textit{term} & e & ::= & \textbf{try}\ e_1\ \textbf{with exn}\ k \Rightarrow e_2 \mid \textbf{raise}\ (\textbf{exn}\ k) \\ \textit{frame} & f & ::= & \textbf{try}\ \_\ \textbf{with exn}\ k \Rightarrow e_2 \end{array}$$

   The stack-based small step semantics is similar to before. When an exception is raised, the **find_exn** operation searches the stack for a handler that exactly matches the exception that was raised. Frames for other exceptions are skipped.

$\boxed{m \mapsto m'}$ *(Stack-based small-step rules)*

SSM-TRY-EFF

$$\overline{\langle s, \textbf{try}\ e_1\ \textbf{with exn}\ k \Rightarrow e_2\rangle \mapsto \langle \textbf{try}\ \_\ \textbf{with exn}\ k \Rightarrow e_2 : s, e_1\rangle}$$

SSM-DISCARD-EFF

$$\overline{\langle \textbf{try}\ \_\ \textbf{with exn}\ k \Rightarrow e_2 : s, \textbf{ret}\ v\rangle \mapsto \langle s, \textbf{ret}\ v\rangle}$$

SSM-RAISE-EFF

$$\overline{\langle \mathit{frame} : s, \textbf{raise}\ (\textbf{exn}\ k)\rangle \mapsto \textbf{find\_exn}\ (\mathit{frame} : s)\ k}$$

$$\begin{array}{ll} \textbf{find\_exn}\ (\textbf{try}\ \_\ \textbf{with exn}\ k \Rightarrow e_2 : s)\ k & = \langle s, e_2\rangle \\ \textbf{find\_exn}\ (\mathit{frame} : s)\ k & = \textbf{find\_exn}\ s\ k \\ \textbf{find\_exn}\ []\ v & = \langle [], \textbf{raise}\ v\rangle \end{array}$$

   The type-and-effect rules for this extension track potentially raised exceptions. The core rules are similar to the ones that we used to track nontermination.

$\boxed{\Gamma \vdash v \in \tau}$ *(Value v has type τ)*

TV-VAR
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \in \tau}$$

TV-ABS-EFF
$$\frac{\Gamma, x : \tau_1 \vdash e \overset{\varepsilon}{\in} \tau_2}{\Gamma \vdash \lambda x.e \in \tau_1 \overset{\varepsilon}{\to} \tau_2}$$

TV-ZERO
$$\frac{}{\Gamma \vdash 0 \in \textbf{Nat}}$$

TV-SUCC
$$\frac{\Gamma \vdash v \in \textbf{Nat}}{\Gamma \vdash \textbf{S}\ v \in \textbf{Nat}}$$

TV-PAIR
$$\frac{\Gamma \vdash v_1 \in \tau_1 \qquad \Gamma \vdash v_2 \in \tau_2}{\Gamma \vdash (v_1, v_2) \in \tau_1 * \tau_2}$$

TV-INJ1
$$\frac{\Gamma \vdash v_1 \in \tau_1}{\Gamma \vdash \textbf{inj}_1 v_1 \in \tau_1 + \tau_2}$$

TV-INJ2
$$\frac{\Gamma \vdash v_2 \in \tau_2}{\Gamma \vdash \textbf{inj}_2 v_2 \in \tau_1 + \tau_2}$$

$\boxed{\Gamma \vdash e \overset{\varepsilon}{\in} \tau}$ *(Term e has type τ and effect ε)*

TEE-RET
$$\frac{\Gamma \vdash v \in \tau}{\Gamma \vdash \mathbf{ret}\, v \overset{\bot}{\in} \tau}$$

TEE-LET
$$\frac{\Gamma \vdash e_1 \overset{\varepsilon_1}{\in} \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 \overset{\varepsilon_2}{\in} \tau}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \overset{\varepsilon_1 \oplus \varepsilon_2}{\in} \tau}$$

TEE-APP
$$\frac{\Gamma \vdash v_1 \in \tau_1 \overset{\varepsilon}{\to} \tau_2 \qquad \Gamma \vdash v_2 \in \tau_1}{\Gamma \vdash v_1\ v_2 \overset{\varepsilon}{\in} \tau_2}$$

TEE-IFZ
$$\frac{\Gamma \vdash v \in \mathbf{Nat} \qquad \Gamma \vdash e_1 \overset{\varepsilon}{\in} \tau \qquad \Gamma, x_2{:}\mathbf{Nat} \vdash e_2 \overset{\varepsilon}{\in} \tau}{\Gamma \vdash \mathbf{case}\, v\, \mathbf{of}\, \{0 \Rightarrow e_1; \mathbf{S}\, x_2 \Rightarrow e_2\} \overset{\varepsilon}{\in} \tau}$$

TEE-PRJ1-BOT
$$\frac{\Gamma \vdash v \in \tau_1 * \tau_2}{\Gamma \vdash \mathbf{prj}_1 v \overset{\bot}{\in} \tau_1}$$

TEE-PRJ2-BOT
$$\frac{\Gamma \vdash v \in \tau_1 * \tau_2}{\Gamma \vdash \mathbf{prj}_2 v \overset{\bot}{\in} \tau_2}$$

TEE-CASE
$$\frac{\Gamma \vdash v \in \tau_1 + \tau_2 \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 \overset{\varepsilon}{\in} \tau \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 \overset{\varepsilon}{\in} \tau}{\Gamma \vdash \mathbf{case}\, v\, \mathbf{of}\, \{\mathbf{inj}_1 x_1 \Rightarrow e_1; \mathbf{inj}_2 x_2 \Rightarrow e_2\} \overset{\varepsilon}{\in} \tau}$$

TEE-SUB-EFF
$$\frac{\Gamma \vdash e \overset{\varepsilon_1}{\in} \tau \qquad \varepsilon_1 <: \varepsilon_2}{\Gamma \vdash e \overset{\varepsilon_2}{\in} \tau}$$

What is different this time is that effects are modeled by *sets of natural numbers*, where each natural number in the set indicates the identity of an exception that could be thrown.

That means that we interpret the $\bot$ effect as the empty set — pure expressions cannot throw any effects. We define effect subsumption $\varepsilon_1 <: \varepsilon_2$ using the subset relation. If we know that a program could raise exception 0, it is sound to say that it could raise either exception 0 or exception 1.

The typing rules for **raise** and **try** are specific for exception tracking. The effect of **raise** (**exn** k), is $\{k\}$, a singleton set that indicates that **exn** k could be raised by the program.

TEE-RAISE-EFF
$$\frac{}{\Gamma \vdash \mathbf{raise}\, (\mathbf{exn}\, k) \overset{\{k\}}{\in} \tau}$$

To type check exception handlers, we need a new operation on effects, written $\varepsilon_1 \ominus \varepsilon_2$. Exception handlers *mask* effects, so we need a way to remove specific exceptions from the effect of a **try** block. The effect of a try block includes both the effects of the body of the try (with the caught exception removed) plus the effects of the handler.

TEE-TRY-EFF
$$\frac{\Gamma \vdash e_1 \overset{\varepsilon \oplus \{k\}}{\in} \tau \qquad \Gamma \vdash e_2 \overset{\varepsilon}{\in} \tau}{\Gamma \vdash \mathbf{try}\, e_1\, \mathbf{with}\, \mathbf{exn}\, k \Rightarrow e_2 \overset{\varepsilon}{\in} \tau}$$

Your job for this problem is to:

(a) Define the typing judgements for frames, stacks, and machines. (You can start with the analogous definitions from the CONTROL language and effect tracking.)

(b) State and prove the preservation lemma for the **find_exn** $s\ k$ term.

(c) (Optional) Prove the preservation lemma.

(d) (Optional) Prove the progress lemma.

The type soundness lemma for this language gives us *effect soundness*. If a machine has effect $\perp$ is in a terminal state, then it cannot be an uncaught exception. In other words, we know that all raise exceptions will be caught by the program.