CIS 7000-1 Homework 2

NAME: FILL IN HERE

September 22, 2025

1 Big-steps for let expressions and natural number recursion

We talked in class about the small-step and typing rules for let expressions and the successor operation and for natural number recursion. You can find the small-step rules in Chapter 3 of the lecture notes, and for convenience, the typing rules are below:

$\boxed{\Gamma \vdash e \in \tau}$ *(typing)*

T-LET
$$\frac{\Gamma \vdash e_1 \in \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 \in \tau_2}{\Gamma \vdash \mathbf{let}\, x = e_1\, i_n\, e_2 \in \tau_2}$$

T-SUCC
$$\frac{\Gamma \vdash e \in \mathbf{Nat}}{\Gamma \vdash \mathbf{succ}\, e \in \mathbf{Nat}}$$

T-NREC
$$\frac{\Gamma \vdash e \in \mathbf{Nat} \qquad \Gamma \vdash e_0 \in \tau \qquad \Gamma, x{:}\mathbf{Nat} \vdash e_1 \in \tau \to \tau}{\Gamma \vdash \mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_0; S\, x \Rightarrow e_1\} \in \tau}$$

Now consider adding rules to the *big-step semantics* for these operations (this is in addition to the existing val and app rules).

$\boxed{e \Rightarrow v}$ *(term e big-steps to v)*

BS-LET
$$\frac{e_1 \Rightarrow v_1 \qquad e_2[v_1/x] \Rightarrow v_2}{\mathbf{let}\, x = e_1\, i_n\, e_2 \Rightarrow v_2}$$

BS-SUCC
$$\frac{e \Rightarrow k}{\mathbf{succ}\, e \Rightarrow \mathbf{succ}\, k}$$

(bs-nrec-zero?)(bs-nrec-succ?)

1. Prove that the small-step and big-step languages produce the same values, extending the proof from Chapter 3.

   Theorem 1.1 (Equivalence of semantics)**.** For closed expressions e, we have $e \rightsquigarrow^* v$ if and only if $e \Rightarrow v$.

2. Extend the semantic soundness proof from Chapter 4. This means that you need to prove semantic typing rules equivalent to rules T-LET, T-SUCC, and T-NREC.

   Lemma 1.1 (Semantic let)**.** If $\Gamma \vDash e_1 : \tau_1$ and $\Gamma, x{:}\tau_2 \vDash e_2 : \tau_2$ then $\Gamma \vDash \mathbf{let}\, x = e_1\, i_n\, e_2 : \tau_2$.

   Lemma 1.2 (Semantic succ)**.** If $\Gamma \vDash e : \mathbf{Nat}$ then $\Gamma \vDash \mathbf{succ}\, e : \mathbf{Nat}$

   Lemma 1.3 (Semantic nrec)**.** If $\Gamma \vDash e : \mathbf{Nat}$ and $\Gamma \vDash e_0 : \tau$ and $\Gamma, x : \mathbf{Nat} \vDash e_1 : \tau \to \tau$, then $\Gamma \vDash \mathbf{nrec}\, e\, \mathbf{of}\, \{0 \Rightarrow e_0; S\, x \Rightarrow e_1\} : \tau$.

2 Big-steps with errors

In class, we observed that we cannot use a direct induction on the typing judgment to show type safety for the big-step semantics. Instead, we switched to a logical relations based argument. However, this proof is rather strong as it shows that all expressions terminate.

The issue is that the big-step semantics cannot distinguish between terms that fail to produce a value due to some type error and those that fail to produce a value because they diverge.

In this problem, let's revise the big step semantics so that it can talk about type errors. In this version, the result of evaluation is either some value v, or a special error result, written **Stuck**.

$$\textit{result } r \ ::= \ \textbf{Stuck} \mid v$$

Our rules for the big step semantics this result when we try to apply a non-function (rule TS-APP-STUCK), or when one of the subterms of an application produces an error (rules TS-APP1 and TS-APP2). (You do not need to consider **let**, **succ** or **nrec** terms in this problem.)

Definition 2.1 (Big-step)**.**

$\boxed{e \Rightarrow r}$ *(term e bigsteps to v or gets stuck)*

$$\text{TS-VAL}\ \frac{}{v \Rightarrow v} \qquad \text{TS-APP}\ \frac{e_1 \Rightarrow \lambda x.e_1' \qquad e_2 \Rightarrow v_1 \qquad e_1'[v_1/x] \Rightarrow r}{e_1\ e_2 \Rightarrow r} \qquad \text{TS-APP-STUCK}\ \frac{e_1 \Rightarrow k}{e_1\ e_2 \Rightarrow \textbf{Stuck}}$$

$$\text{TS-APP1}\ \frac{e_1 \Rightarrow \textbf{Stuck}}{e_1\ e_2 \Rightarrow \textbf{Stuck}} \qquad \text{TS-APP2}\ \frac{e_1 \Rightarrow v \qquad e_2 \Rightarrow \textbf{Stuck}}{e_1\ e_2 \Rightarrow \textbf{Stuck}}$$

1. Prove the following lemma:

   Lemma 2.1 (Type Safety (Not stuck))**.** If $\Gamma \vdash e \in \tau$ and $e \Rightarrow r$ then r is not **Stuck**.

2. How does this statement of the lemma compare to the preservation and progress lemmas described in Chapter 1?

   For example, suppose that we somehow made a design error in the big-step semantics or the typing rules. For example, say we forgot to add a rule, or that we added an extra, nonsensical rule. (You might consider variants of the unsafe type systems from homework 1.)

   What sort of errors would this safety lemma catch? What would still be considered type safe?

   Furthermore, we discussed that there were some languages that we wanted to call type safe, but did not satisfy the small-step properties of preservation and progress. Would these languages satisfy this lemma?

3 Big-steps with timeouts

Recall our step-counting definition of type-safety for the small-step semantics:

Definition 3.1 ((Small-step) Safe for k)**.** An expression evaluates safely for k steps if it either there is some e', such that $e \rightsquigarrow^k e'$, or there is some number of steps j strictly less than k where the term terminates with a value (i.e. there is some v and $j < k$ such that $e \rightsquigarrow^j v$).

We can use this idea to prove a form of type safety theorm for a language with a big-step semantics. The first step is to revise our semantics to incorporate a count, using the notation $e \Rightarrow_i r$. Here, i is the maximum *height* of the derivation; every rule must use a smaller i for its premises. Furthermore, it may or may not be possible to fully evaluate an expression within a bounded height derivation. So we modify the second argument of this judgement to be a result: either a value or **Timeout**, indicating an incomplete evaluation.

$$\textit{result } r \ ::= \ \textbf{Timeout} \mid v$$

Definition 3.2 ((Big-step) Safe for k)**.**

$\boxed{e \Rightarrow_i r}$ *(term e times out or steps to v in derivation of height less than i)*

$$\frac{}{e \Rightarrow_0 \textbf{Timeout}}\ \text{TK-TIMEOUT} \qquad \frac{}{v \Rightarrow_{S\,i} v}\ \text{TK-VAL}$$

$$\frac{e_1 \Rightarrow_i \lambda x.e_1' \qquad e_2 \Rightarrow_i v_1 \qquad e_1'[v_1/x] \Rightarrow_i r}{e_1\ e_2 \Rightarrow_{S\,i} r}\ \text{TK-APP}$$

$$\frac{e_1 \Rightarrow_i \textbf{Timeout}}{e_1\ e_2 \Rightarrow_{S\,i} \textbf{Timeout}}\ \text{TK-APP1}$$

$$\frac{e_1 \Rightarrow_i \lambda x.e_1' \qquad e_2 \Rightarrow_i \textbf{Timeout}}{e_1\ e_2 \Rightarrow_{S\,i} r}\ \text{TK-APP2}$$

With this definition, we can use this definition of type safety.

Theorem 3.1 (Type Safety (step-counting))**.** If $\vdash e \in \tau$ then for all natural numbers k, $e \Rightarrow_k r$, i.e. e is Safe for k.

1. Prove this theorem by induction on k.
2. Analyze this statement of type safety just as you did for the previous problem. Is it possible for a language to satisfy the step-counting theorem, but not the not-stuck theorem? Or vice versa?