CIS 7000-1 Homework 3

NAME: FILL IN HERE

October 5, 2025

1 Recursive Nats

We saw in OCaml that we can define an "infinite" natural number using a recursive value definition.

```
type nat = zero | succ of nat

let rec omega : nat = succ omega
```

However, recall that the premise of our introduction rule for recursive values limits the types of values that may be used in recursive definitions.

$$\frac{\tau\ \mathsf{ok} \qquad \Gamma, x\!:\!\tau \vdash v \in \tau}{\Gamma \vdash \mathbf{rec}\, x.v \in \tau}\ \textsc{tv-rec}$$

In the type system in the lecture notes (and that we discussed in class), there were two types of values that could be used.

$$\frac{}{(\tau_1 \to \tau_2)\ \mathsf{ok}}\ \textsc{fun-ok} \qquad \frac{}{(\tau_1 * \tau_2)\ \mathsf{ok}}\ \textsc{prod-ok}$$

Suppose we add the ability to define recursive nats to REC.

$$\frac{}{\mathbf{Nat}\ \mathsf{ok}}\ \textsc{nat-ok}$$

With this rule, we can define $\omega = \mathbf{rec}\, x.\mathbf{S}\, x$ in REC. (Note: I've updated the REC language slightly compared to the version we discussed in class. Please take a look at the lecture notes to see how this expression type checks and evaluates.)

1. What small step rule(s) do we need to add to REC so that the progress lemma holds?
2. How do the following expressions evaluate using the small step semantics? Write the sequence of steps that they take, stopping when you get to a value or loop back to a prior term. Use your new step rule(s) from the previous part.
   - $(\lambda x.x)\ \omega \leadsto$
   - $\mathbf{case}\, \omega\, \mathbf{of}\, \{0 \Rightarrow 0;\ \mathbf{S}\, y \Rightarrow y\} \leadsto$
   - $(\mathbf{rec}\, x.\lambda y.\mathbf{case}\, y\, \mathbf{of}\, \{0 \Rightarrow 0;\ \mathbf{S}\, y \Rightarrow \mathbf{S}\, y\})\ \omega \leadsto$
   - $(\mathbf{rec}\, x.\lambda y.\mathbf{case}\, y\, \mathbf{of}\, \{0 \Rightarrow 0;\ \mathbf{S}\, y \Rightarrow x\ y\})\ \omega \leadsto$

2 Small-step CBV: derived forms

Fine-grained CBV requires that subterms be values in many cases. We showed in class that we could derive the usual forms using let terms.

2.1 Successor

In this language, the syntax of values include 0 and the successor of some value, written $\mathbf{S}\,v$.

However, in STLC, $\mathbf{succ}\,e$ was an expression and could be applied to any term, not just values. Even though this term does not appear in this language, we can define it using let expressions.

Definition 2.1 (Extended Successor)**.** Define $\mathbf{succ}\,e$ as $\mathbf{let}\ x\ = e\ \mathbf{in}\ \mathbf{ret}\,(\mathbf{S}\,x)$.

Now, prove that this definition acts like a successor term, by showing these properties of the encoding.

1. If $\Gamma \vdash e \in \mathbf{Nat}$ then $\Gamma \vdash \mathbf{succ}\,e \in \mathbf{Nat}$.
2. $\mathbf{succ}\,(\mathbf{ret}\,v) \rightsquigarrow \mathbf{ret}\,(\mathbf{S}\,v)$.
3. If $e \rightsquigarrow e'$ then $\mathbf{succ}\,e \rightsquigarrow^* \mathbf{succ}\,e'$.

2.2 Derived products

Now recall the definition of the "eager let" form:

Definition 2.2 (Eager let)**.** Define $\mathbf{let}\ x\ \Leftarrow\ e_1\ \mathbf{in}\ e_2$ as $e_2[v/x]$ when e_1 is $\mathbf{ret}\,v$ and $\mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2$ otherwise.

In fine-grained CBV, products are values and must have values as their component. Define an expression form for products using eager let.

Definition 2.3 (Extended prod)**.** Define (e_1, e_2) as $\mathbf{let}\ x_1\ \Leftarrow\ e_1\ \mathbf{in}\ \mathbf{let}\ x_2\ \Leftarrow\ e_2\ \mathbf{in}\ \mathbf{ret}\,(x_1, x_2)$.

Now, prove that this definition acts like a product term, by showing these properties of the encoding.

1. If $\Gamma \vdash e_1 \in \tau_1$ and $\Gamma \vdash e_2 \in \tau_2$ then $\Gamma \vdash (e_1, e_2) \in \tau_1 * \tau_2$.
2. If $e_1 \rightsquigarrow e_1'$ then $(e_1, e_2) \rightsquigarrow^* (e_1', e_2)$.
3. If $e_2 \rightsquigarrow e_2'$ then $(\mathbf{ret}\,v_1, e_2) \rightsquigarrow^* (\mathbf{ret}\,v_1, e_2')$.

What if we used regular let in the definition of (e_1, e_2). Are the two properties still true for this encoding? If any fail, provide a counterexample.

3 Small-step semantic soundness proof

If we consider the fine-grained call-by-value language without recursive values or recursive types, then we can prove that all expressions in this language terminate with a value. Recall the definition of our logical relation from class.

Definition 3.1 (Logical Relation)**.**

$$
\begin{array}{lcl}
\mathcal{C}[\![\tau]\!] & = & \{\ e \mid e \rightsquigarrow^* \mathbf{ret}\, v \textit{ and } \quad v \in \mathcal{V}[\![\tau]\!]\,\} \\
\\
\mathcal{V}[\![\mathbf{Nat}]\!] & = & \mathbb{N} \\
\mathcal{V}[\![\mathbf{Void}]\!] & = & \{\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] & = & \{\ v \mid \forall v_2,\ v_2 \in \mathcal{V}[\![\tau_1]\!] \textit{ implies } v\ v_2 \in \mathcal{C}[\![\tau_2]\!]\ \} \\
\mathcal{V}[\![\tau_1 * \tau_2]\!] & = & \{\ v \mid \mathbf{prj}_1 v \in \mathcal{C}[\![\tau_1]\!] \textit{ and } \mathbf{prj}_2 v \in \mathcal{C}[\![\tau_2]\!]\ \} \\
\mathcal{V}[\![\tau_1 + \tau_2]\!] & = & \{\ \mathbf{inj}_1 v \mid v_1 \in \mathcal{V}[\![\tau_1]\!]\ \} \cup \{\ \mathbf{inj}_2 v \mid v_2 \in \mathcal{V}[\![\tau_2]\!]\ \}
\end{array}
$$

As well as the definitions for semantic typing for values and expressions:

1. Define $\sigma \in \mathcal{G}[\![\Gamma]\!]$ when $\forall x \in \mathsf{dom}\,\Gamma, \sigma\, x \in \mathcal{V}[\![\Gamma\, x]\!]$.
2. Define $\Gamma \vDash e : \tau$ when forall $\sigma \in \mathcal{G}[\![\Gamma]\!]$, $e[\sigma] \in \mathcal{C}[\![\tau]\!]$.
3. Define $\Gamma \vDash v : \tau$ when forall $\sigma \in \mathcal{G}[\![\Gamma]\!]$, $v[\sigma] \in \mathcal{V}[\![\tau]\!]$.

Complete the small-step semantic soundness proof for fine-grained CBV by proving semantic soundness lemmas for products and (first) projections.

Lemma 3.1 (Semantic prod rule)**.** If $\Gamma \vDash v_1 : \tau_1$ and $\Gamma \vDash v_2 : \tau_2$ then $\Gamma \vDash (v_1, v_1) : \tau_1 * \tau_2$.

Lemma 3.2 (semantic projection)**.** If $\Gamma \vDash v_1 : \tau_1 * \tau_2$ then $\Gamma \vDash \mathbf{prj}_1 v_1 : \tau_1$.

4 Step-indexed logical relations

Now remember the step-indexed logical relation.

Definition 4.1 (Step-indexed logical relation)**.**

$$
\begin{array}{lcl}
\mathcal{C}[\![\, e \in \tau \,]\!]_k & = & e\,\mathbf{irreducible}\ \textit{implies that there exists } v \textit{ such that} \\
 & & \qquad\qquad e = \mathbf{ret}\ v \textit{ and } \mathcal{V}[\![\, v \in \tau \,]\!]_k \\
 & & \textit{and } e \rightsquigarrow e' \textit{ implies } \triangleright_k\ \mathcal{C}[\![\, e' \in \tau \,]\!] \\
 & & \\
\mathcal{V}[\![\, v \in \mathbf{Void} \,]\!]_k & = & \textit{never} \\
\mathcal{V}[\![\, v \in \mathbf{Nat} \,]\!]_k & = & v \in \mathbb{N} \\
\mathcal{V}[\![\, \lambda x.e \in \tau_1 \to \tau_2 \,]\!]_k & = & \forall v_1, \mathcal{V}[\![\, v_1 \in \tau_1 \,]\!] \Longrightarrow_k \mathcal{C}[\![\, e[v_2/x] \in \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{rec}\, x.v \in \tau_1 \to \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v[\mathbf{rec}\, x.v/x] \in \tau_1 \to \tau_2 \,]\!] \\
\mathcal{V}[\![\, (v_1, v_2) \in \tau_1 * \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v_1 \in \tau_1 \,]\!] \textit{ and } \triangleright_k\ \mathcal{V}[\![\, v_3 \in \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{rec}\, x.v \in \tau_1 * \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v[\mathbf{rec}\, x.v/x] \in \tau_1 * \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{inj}_1 v_1 \in \tau_1 + \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v_1 \in \tau_1 \,]\!] \\
\mathcal{V}[\![\, \mathbf{inj}_2 v_2 \in \tau_1 + \tau_2 \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v_2 \in \tau_2 \,]\!] \\
\mathcal{V}[\![\, \mathbf{fold}\, v \in \mu\alpha.\tau \,]\!]_k & = & \triangleright_k\ \mathcal{V}[\![\, v \in \tau[\mu\alpha.\tau/\alpha] \,]\!]
\end{array}
$$

and its notion of semantic typing:

1. Define $[\![\sigma \in \Gamma]\!]_k$ when for all $x \in \mathsf{dom}\,\Gamma$, we have $\mathcal{V}[\![\, \sigma\, x \in \Gamma\, x \,]\!]_k$.
2. Define $\Gamma \vDash_k e \in \tau$ when $[\![\sigma \in \Gamma]\!] \Longrightarrow_k \mathcal{C}[\![\, e[\sigma] \in \tau \,]\!]$.
3. Define $\Gamma \vDash_k v \in \tau$ when $[\![\sigma \in \Gamma]\!] \Longrightarrow_k \mathcal{V}[\![\, v[\sigma] \in \tau \,]\!]$.

Finish the step-indexed logical relations proof for products (including recursive products).

Lemma 4.1 (ST_prod)**.** If $\Gamma \vDash_k v_1 \in \tau_1$ and $\Gamma \vDash_k v_2 \in \tau_2$ then $\Gamma \vDash_k (v_1, v_1) \in \tau_1 * \tau_2$.

Lemma 4.2 (ST_rec_prod)**.** If $\Gamma, x\!:\!\tau_1 * \tau_2 \vDash_k v \in \tau_1 * \tau_2$ then $\Gamma \vDash_k \mathbf{rec}\, x.v \in \tau_1 * \tau_2$.

Lemma 4.3 (ST_prj1)**.** If $\Gamma \vDash_k v_1 \in \tau_1 * \tau_2$ then $\Gamma \vDash_k \mathbf{prj}_1 v_1 \in \tau_1$.

If you would like an extra challenge, you can also prove the semantic soundness lemma for let expressions.

Lemma 4.4 (ST_let)**.** If $\Gamma \vDash e_1 : \tau_1$ and $\Gamma, x\!:\!\tau_1 \vDash e_2 : \tau_2$ then $\Gamma \vDash \mathbf{let}\ x\ = e_1\ \mathbf{in}\ e_2 : \tau_2$.