Homework 2: Principles of System T

15-814: Types and Programming Languages
Fall 2017
Instructor: Karl Crary
TA: Yong Kiam Tan

Out: Sep 25, 2017 (5 pm)
Due: Oct 9, 2017 (11 pm)

Notes:

- Welcome to 15-814's second homework assignment!
- Please email your work as a PDF file to `yongkiat@cs.cmu.edu` titled "15-814 Homework 2". Your PDF should be named "<your-name>-hw2-sol.pdf".

1 Termination in System T

Gödel's System T, presented in Appendix A as we defined it in class, has the valuable property that any program we can write will evaluate to a value in a finite number of steps. In this section, we will look at how to prove this fact using Tait's *reducibility method*, which is an instance of the ubiquitous technique of *logical relations*. The theorem we want to prove is the following:

Theorem 1 (Normalization) *If $\cdot \vdash e : \tau$, then there exists v val such that $e \mapsto^* v$.*

Here, $\mapsto^*$ is the reflexive transitive closure of the step judgment $\mapsto$. We might hope to prove this theorem directly by induction on the typing judgment. However, this approach is insufficient. The case for the application rule (App) is demonstrative.

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \text{ (App)}$$

In this case, our induction hypotheses tell us that $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$ for some values v_1 and v_2. By preservation and the appropriate canonical forms lemma, we know that $v_1 = \lambda x : \tau'.e'$ for some e'. It follows that $e_1 e_2 \mapsto^* v_1 e_2 \mapsto [e_2/x]e'$. Unfortunately, we are now stuck, as we have no information about the behavior of $[e_2/x]e'$.

We will solve this by generalizing and proving a stronger statement which gives us more information in the induction hypothesis. Specifically, we will define a *reducibility predicate* $\mathsf{Red}_\tau(e)$ and prove the following theorem.

Theorem 2 *If $\cdot \vdash e : \tau$, then $\mathsf{Red}_\tau(e)$.*

Since we'll define Red_τ such that $\mathsf{Red}_\tau(e)$ implies the existence of v val with $e \mapsto^* v$, this will give normalization as a special case. The definition will go by structural induction on the type τ, which makes Red_τ what is called a *logical relation.* (In particular, it is a *unary* logical relation; we will encounter *binary* logical relations, such as logical equivalence $e \sim_\tau e'$, later in the course.) Actually, we will prove an even more general theorem in order to account for open terms; to state it concisely, we first want to define some notation for substitutions.

Definition 1 *A substitution $\gamma = \{x_1 \hookrightarrow e_1, \ldots, x_n \hookrightarrow e_n\}$ is a finite mapping from variables to terms. Given an expression e, we write $\gamma(e)$ for the expression $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$, that is, the simultaneous substitution in e of each expression e_i for its corresponding variable x_i. For γ as above, we define $\gamma \Vdash \Gamma$ to mean that $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ for some $\tau_1, \ldots, \tau_n$ such that $\mathsf{Red}_{\tau_i}(e_i)$ holds for $1 \leq i \leq n$.*

This is the theorem we will actually prove:

Theorem 3 *If $\Gamma \vdash e : \tau$ and $\gamma \Vdash \Gamma$ then $\mathsf{Red}_\tau(\gamma(e))$.*

Theorem 2 (and hence Theorem 1) follows by the special case where $\Gamma = \cdot$ and $\gamma = \langle\rangle$. Finally, we define the predicate Red_τ by structural induction on τ:

- $\mathsf{Red}_{\tau_1 \to \tau_2}(e)$ holds if
  1. $\cdot \vdash e : \tau_1 \to \tau_2$,
  2. there exists v val such that $e \mapsto^* v$, and
  3. for any e' such that $\mathsf{Red}_{\tau_1}(e')$, we have $\mathsf{Red}_{\tau_2}(e\ e')$.
- $\mathsf{Red}_{\mathtt{nat}}(e)$ holds if
  1. $\cdot \vdash e : \mathtt{nat}$,
  2. there exists v val such that $e \mapsto^* v$, and
  3. $v \downarrow$, where $v \downarrow$ is a judgment defined by

$$\frac{}{\mathsf{z}\downarrow}\ (\downarrow\text{-Z}) \qquad \frac{e \mapsto^* v \quad v\ \mathsf{val} \quad v\downarrow}{\mathsf{s}(e)\downarrow}\ (\downarrow\text{-S})$$

Note that $\mathsf{Red}_{\tau_1 \to \tau_2}(e)$ is defined in terms of Red at the structurally smaller types τ_1 and τ_2, so the definition is well-founded. To get you started on the proof, and to see how this definition succeeds where the previous attempt failed, here is the (App) case:

- Case (App): We have $\Gamma \vdash e_1\ e_2 : \tau$ with $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$ for some τ'. Per the theorem statement, we assume we are given $\gamma \Vdash \Gamma$ and want to prove that $\mathsf{Red}_\tau(\gamma(e_1\ e_2))$. By definition of substitution, we have that $\gamma(e_1\ e_2) = \gamma(e_1)\ \gamma(e_2)$. Moreover, our induction hypotheses tell us that $\mathsf{Red}_{\tau' \to \tau}(\gamma(e_1))$ and $\mathsf{Red}_{\tau'}(\gamma(e_2))$. From condition 3 in the definition of $\mathsf{Red}_{\tau' \to \tau}$, we know that for any e' with $\mathsf{Red}_{\tau'}(e')$ we have $\mathsf{Red}_\tau(\gamma(e_1)\ e')$. Taking $e' = \gamma(e_2)$ thus gives our goal.

With the right definition Red in hand, the (App) case follows almost trivially. On the other hand, the (Lam) case becomes more difficult. In general, though, proving the theorem is the easy part of a logical relations argument – the hard part is choosing the right theorem to prove.

To complete the proof, you'll need the following lemma.

Lemma 1 (Closure under Head Expansion) *If $\mathsf{Red}_\tau(e')$ and $\cdot \vdash e : \tau$ and $e \mapsto e'$, then $\mathsf{Red}_\tau(e)$.*

Task 1 *Prove closure under head expansion.*

With the help of type preservation, closure under head expansion extends to apply when $e \mapsto^* e'$ in multiple steps (you may use this without proof).

Task 2 *Prove the remaining cases of Theorem 3. You may state (without proof) lemmas about substitution, but be sure to check that they are actually true.*

2 Programming in System **T**

Consider an extension of Gödel's **T** with products. We will call this language **LNS**.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \tau_1 \times \tau_2 \\ e & ::= & \cdots \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \end{array}$$

(We will use eager dynamics for products, following PFPL 10, and the definitions given in class.)

First we'll get some practice working with nats. You may find it helpful to break large definitions into intermediate ones, give them names, and refer to them in the larger definition.

Task 3 *For each definition below, briefly explain (in 1-2 lines) the intuition behind your answer.*

1. *Define* `mult`*, where* `mult` $\overline{m}\,\overline{n} \mapsto^* \overline{m * n}$.[1]
2. *Define* `minus`*, where* `minus` $\overline{m}\,\overline{n} \mapsto^* \overline{m - n}$ *if* $m > n$*. It should produce* 0 *otherwise.*
3. *Define* `leq`*, where* `leq` $\overline{m}\,\overline{n} \mapsto^*$ `s(z)` *if* $m \leq n$ *and* `leq` $\overline{m}\,\overline{n} \mapsto^*$ `z` *otherwise.*
4. *Define* `mod`*, where* `mod` $\overline{m}\,\overline{n} = \overline{m \mod n}$*. You may pick appropriate defaults when* $n = 0$.

In class, we briefly mentioned *iteration* and *primitive recursion* in relation to the operator `natrec`. In this context, `natrec` in **LNS** corresponds to primitive recursion and may be called the *recursor*. We also defined an alternate form $\texttt{rec}(e; e_0; y.e_1)$ which corresponds to iteration.

$$\frac{e \mapsto e'}{\texttt{rec}(e; e_0; y.e_1) \mapsto \texttt{rec}(e'; e_0; y.e_1)}$$

$$\texttt{rec}(\texttt{z}; e_0; y.e_1) \mapsto e_0$$

$$\texttt{rec}(\texttt{s}(e); e_0; y.e_1) \mapsto [\texttt{rec}(e; e_0; y.e_1)/y]e_1$$

It is simple to define $\texttt{rec}(e; e_0; x.e_1)$ in terms of $\texttt{natrec}(e; e_0; x.y.e_1)$; simply ignore x. Using the other constructs of **LNS**, it is also possible to define `natrec` in terms of `rec`.

Task 4 *Define* $\texttt{natrec}(e; e_0; x.y.e_1)$ *in terms of* `rec`*. Briefly explain the intuition behind your answer.*

[1] We will write $\overline{n}$ to indicate the representation of a natural number n as an element of type `nat`.

A System T

A.1 Statics

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\textsc{Hyp}) \qquad \frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}\ (\mathrm{Z}) \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}}\ (\mathrm{S})$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{natrec}(e; e_0; x.y.e_1) : \tau}\ (\textsc{Rec})$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau}\ (\textsc{Lam}) \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}\ (\textsc{App})$$

A.2 Dynamics

$$\frac{}{\mathsf{z}\ \mathsf{val}}\ (\mathrm{Z\text{-}V}) \qquad \frac{}{\mathtt{s}(e)\ \mathsf{val}}\ (\mathrm{S\text{-}V}) \qquad \frac{e \mapsto e'}{\mathtt{natrec}(e; e_0; x.y.e_1) \mapsto \mathtt{natrec}(e'; e_0; x.y.e_1)}\ (\textsc{Rec-S})$$

$$\frac{}{\mathtt{natrec}(\mathtt{z}; e_0; x.y.e_1) \mapsto e_0}\ (\textsc{Rec-IZ})$$

$$\frac{}{\mathtt{natrec}(\mathtt{s}(e); e_0; x.y.e_1) \mapsto [e, \mathtt{natrec}(e; e_0; x.y.e_1)/x, y]e_1}\ (\textsc{Rec-IS})$$

$$\frac{}{\lambda x{:}\tau.e\ \mathsf{val}}\ (\textsc{Lam-V}) \qquad \frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}\ (\textsc{App-S}) \qquad \frac{}{(\lambda x{:}\tau.e)e' \mapsto [e'/x]e}\ (\textsc{App-I})$$