Homework 2: Induction, Coinduction, and Polymorphism

15-814: Types and Programming Languages
Fall 2015
TA: Evan Cavallo (ecavallo@cs.cmu.edu)

Out: 9/24/15
Due: 10/8/15 at 10:30am

1 Termination in System $\mathbf{T}$

Gödel's System $\mathbf{T}$, 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 (Normalization): If $\cdot \vdash e : \tau$, then there exists v val such that $e \mapsto^* v$.

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

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

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

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

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: 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 $1 \leq i \leq n$.

Now we state the theorem we will actually prove:

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

Theorem A follows as 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 $\mathsf{Red}_{\tau_2}(ee')$.
- $\mathsf{Red}_{\mathtt{nat}}(e)$ 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

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

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

Task 1 *Prove closure under head expansion.*

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

Task 2 *Prove the remaining cases of Theorem B.*

2 Programming with nats and streams

Consider an extension of Gödel's **T** with sums, products, and a coinductive type of streams. The type `stream` τ consists of streams which produce values of type τ. We will call this language **LNS**.

$$\begin{array}{rcl} \tau & ::= & \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathtt{nat} \mid \mathtt{stream}\,\tau \\ e & ::= & \cdots \mid \mathtt{z} \mid \mathtt{s}(e) \mid \mathtt{rec}(e; e; x.e) \mid \mathtt{hd}(e) \mid \mathtt{tl}(e) \mid \mathtt{strgen}\, e\, \{\mathtt{hd}(x) \hookrightarrow e \mid \mathtt{tl}(x) \hookrightarrow e\} \end{array}$$

(The statics and dynamics for `stream` τ are given in Appendix B. The rules for products and sums are as in PFPL 10 and 11; we'll assume lazy evaluation to match our presentation of **T**, but it won't make a difference here.) In class, we briefly mentioned *iteration* and *primitive recursion* in relation to the operator `rec`. In this context, `rec` in **LNS** corresponds to iteration and may be called the *iterator*. We can define an alternate form $\mathtt{rec}'(e; e_0; x.y.e_1)$ which corresponds to primitive recursion. In this form, e_1 has access to two bound variables. As in the iterator, x is bound to the result of recursing on the predecessor of e (assuming e is of the form $\mathtt{s}(e')$). In the new recursor, y is bound to the predecessor itself.

$$\mathtt{rec}'(\mathtt{z}; e_0; x.y.e_1) \mapsto e_0$$
$$\mathtt{rec}'(\mathtt{s}(e); e_0; x.y.e_1) \mapsto [\mathtt{rec}'(e; e_0; x.y.e_1), e/x, y]e_1$$

It is simple to define $\mathtt{rec}(e; e_0; x.e_1)$ in terms of $\mathtt{rec}'(e; e_0; x.y.e_1)$; simply ignore y. Using the other constructs of **LNS**, it is also possible to define $\mathtt{rec}'$ in terms of `rec`. (It is not possible, however, in plain System **T**!) We can write

$$\mathtt{rec}'(e; e_0; x.y.e_1) \triangleq (\mathtt{rec}(e; \langle e_0, \mathtt{z}\rangle; p.\langle [p \cdot \mathtt{l}, p \cdot \mathtt{r}/x, y]e_1, \mathtt{s}(p \cdot \mathtt{r})\rangle)) \cdot \mathtt{l}$$

Now that we have a more useful recursor, we'll get some practice working with nats and streams. In all of the below definitions, you may use any of the constructs of **LNS**, as well as $\mathtt{rec}'$. You may also find it helpful to break large definitions into intermediate ones, give them names, and refer to them in the larger definition.

Task 3

1. *Define* `plus`*, where* $\mathtt{plus}\,\overline{m}\,\overline{n} \mapsto^* \overline{m+n}$.[1]
2. *Define* `minus`*, where* $\mathtt{minus}\,\overline{m}\,\overline{n} \mapsto^* \overline{m-n}$ *if* $m > n$*. It should produce* 0 *otherwise.*
3. *Define* `leq`*, where* $\mathtt{leq}\,\overline{m}\,\overline{n} \mapsto^* \mathtt{l} \cdot \langle\rangle$ *if* $m \leq n$ *and* $\mathtt{leq}\,\overline{m}\,\overline{n} \mapsto^* \mathtt{r} \cdot \langle\rangle$ *otherwise.*
4. *Define* `mod`*, where* $\mathtt{mod}\,\overline{m}\,\overline{n} = \overline{m \mod n}$.

Next, we define some functions on nat streams.

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

Task 4

1. *Define a function* `delay`*, which takes a nat and a nat stream and produces a stream whose head is the given nat and whose tail is the given stream; if we think of the input stream as a signal, this function delays the input signal by one clock tick and maintains the delayed value in a buffer. For example,* `delay` 0 `nats` *should return the stream* $0, 0, 1, 2, 3, \ldots$.
2. *Define a function* `csum`*, which takes a nat stream and produces a nat stream that is the cumulative sum of the input stream;* `csum nats` *should be* $0, 1, 3, 6, \ldots$.

The type `stream` τ is isomorphic (in a sense we will make precise later in the class) to the type `nat` $\to \tau$. If f is the function representation of a stream s, $f\ \overline{n}$ returns the n^{th} element of s (i.e. $\texttt{hd}(\texttt{tl}^{(n)}(s))$, where $\texttt{tl}^{(n)}$ indicates n applications of `tl`.)

Task 5 *Rewrite the definitions of Task 4, using* `nat` $\to$ `nat` *functions anywhere nat streams appear (i.e. the stream arguments should be functions as described above, as should the result.)*

3 Inductive and Coinductive Types

3.1 Generic Programming

In this section, we will generalize `nat` and `stream` to arbitrary inductive and coinductive types. Recall that the generalized versions of the recursor and generator are defined using the *generic extension* primitive $\texttt{map}\{t.\tau\}(x.e)$. Before diving into the full generality of inductive and coinductive types, let's do a little generic programming.

A database schema may be viewed as giving a type to the records of the database. With sums, we can cleanly express optional fields or alternatives. For example (using the generalized notation for products), we might define a type

$$\texttt{string} \times (\texttt{string} + \texttt{unit}) \times \texttt{string} \times (\texttt{int} + \texttt{string})$$

for a database that has fields for a first name, an optional middle name, a last name, and either a phone number or email address.

Task 6

1. *Write a function* $\texttt{crecord} : \texttt{record} \to \texttt{record}$ *which uses a given capitalization function* $c : \texttt{string} \to \texttt{string}$ *to capitalize the first, middle, and last name entries,* without *using the* `map` *operator. (Assume that* $\times$ *associates to the right, so* $\tau_1 \times \tau_2 \times \tau_3$ *is shorthand for* $\tau_1 \times (\tau_2 \times \tau_3)$*.)*
2. *Now define* `crecord` *as an application of* `map`.

Recall the statics and dynamics rules for `map`. There is a different dynamics rule for each type constructor that `map` can operate over, so we'll just show the representative case of products (see PFPL 14.2 for others).

$$\frac{t.\tau \ \mathsf{pos} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{map}\{t.\tau\}(x.e')(e) : [\rho'/t]\tau}$$

$$\frac{}{\mathtt{map}\{t.\tau_1 \times \tau_2\}(x.e')(e) \mapsto \langle \mathtt{map}\{t.\tau_1\}(x.e')(e \cdot \mathtt{l}), \mathtt{map}\{t.\tau_2\}(x.e')(e \cdot \mathtt{r}) \rangle}$$

We have to restrict `map` to a certain class of type operators, because not all type operators are covariant. (For example, it would not be possible to define `map` for the operator $t.t \to t$.) Here, we've restricted to positive operators, which are defined by the following judgment.

$$\frac{}{t.t \ \mathsf{pos}} \qquad \frac{}{t.\mathtt{unit} \ \mathsf{pos}} \qquad \frac{t.\tau_1 \ \mathsf{pos} \quad t.\tau_2 \ \mathsf{pos}}{t.\tau_1 \times \tau_2 \ \mathsf{pos}} \qquad \frac{}{t.\mathtt{void} \ \mathsf{pos}} \qquad \frac{t.\tau_1 \ \mathsf{pos} \quad t.\tau_2 \ \mathsf{pos}}{t.\tau_1 + \tau_2 \ \mathsf{pos}} \qquad \frac{t \notin \tau_1 \quad \tau_2 \ \mathsf{pos}}{t.\tau_1 \to \tau_2 \ \mathsf{pos}}$$

Task 7 *Suppose we add a new rule*

$$\frac{t.\tau \ \mathsf{pos}}{t.\mathtt{stream}\ \tau \ \mathsf{pos}}$$

Give a new dynamics rule for `map` *to cover this case. Prove the case of preservation corresponding to your dynamics rule.*

3.2 Trees

For the rest of this section, we will work in a language **M** with general inductive and coinductive types, in addition to sums, products and functions. The formal definition of this language is given in PFPL 15.2 and 15.3. For the sake of convenience, we'll also include the type `nat` from System **T**.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mu t.\tau \mid \nu t.\tau \\ e & ::= & \ldots \mid \mathtt{fold}\{t.\tau\}(e) \mid \mathtt{rec}\{t.\tau\}(x.e_1; e_2) \mid \mathtt{unfold}\{t.\tau\}(e) \mid \mathtt{gen}\{t.\tau\}(x.e_1; e_2) \end{array}$$

Task 8

1. *Define a type* `btree` *of binary trees with values of type* `nat` *at each node (excluding leaves).*
2. *Define the empty tree* `bemp` : `btree`*.*
3. *Define the function* `val` : `btree` $\to$ (`nat` + `unit`)*, which returns the value at the root node or returns* $\mathtt{r} \cdot \langle\rangle$ *on the empty tree.*
4. *Define the function* `sum` : `btree` $\to$ `nat` *which returns the sum of all of the numbers in a tree. It should return* `z` *on the empty tree.*

Task 9

1. *Define a type* `itree` *of* infinite *binary trees with values of type* `nat` *at every node. Every node has two children; there are no leaves and there is no empty tree.*
2. *Define the infinite tree containing zero at every node.*

3. *Define the function* $\texttt{val} : \texttt{btree} \to \texttt{nat}$*, which returns the value at the root node.*
4. *Define the function* $\texttt{embed} : \texttt{btree} \to \texttt{itree}$*, which embeds an finite tree in an infinite one by extending with zeroes.*

Task 10 *Assume we generalize all of our definitions to handle nested type operators (e.g.* $\mu t.\mu s.\tau$ *where* τ *may contain both* s *and* t*.) This allows us to combine inductive and coinductive types.*

1. *Define a type of nat-valued tree with finite depth but infinite branching factor. A tree of this type is either empty or is a node with a value of type* `nat` *and a stream of children.*
2. *Define the dual of the above type: a nat-valued tree with (potentially) infinite depth and finite but variable branching factor. A tree of this type is a node with a value of type* `nat` *and a* finite list *of children.*

In both cases, you need only define the type. You do not need to define any operations over it.

4 System F

4.1 Church Encoding

Recall the syntax of System **F**.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha.\tau \\ e & ::= & \lambda x{:}\tau.e \mid ee \mid \Lambda \alpha.e \mid e[\tau] \end{array}$$

Using the few type constructors available in System **F**, it is possible to define types with the same behavior as almost any of the types we have studied. For example, we can define products and sums with

$$\begin{array}{rcl} \tau_1 \times \tau_2 & \triangleq & \forall t.(\tau_1 \to \tau_2 \to t) \to t \\ \langle e_1, e_2 \rangle & \triangleq & \Lambda t.\lambda f{:}\tau_1 \to \tau_2 \to t.f e_1 e_2 \\ e \cdot \texttt{l} & \triangleq & e[\tau_1](\lambda x{:}\tau_1.\lambda y{:}\tau_2.x) \\ e \cdot \texttt{r} & \triangleq & e[\tau_2](\lambda x{:}\tau_1.\lambda y{:}\tau_2.y) \\ \\ \tau_1 + \tau_2 & \triangleq & \forall t.(\tau_1 \to t) \to (\tau_2 \to t) \to t \\ \texttt{l} \cdot e_1 & \triangleq & \Lambda t.\lambda f_1{:}\tau_1 \to t.\lambda f_1{:}\tau_2 \to t.f_1 e_1 \\ \texttt{r} \cdot e_2 & \triangleq & \Lambda t.\lambda f_1{:}\tau_1 \to t.\lambda f_1{:}\tau_2 \to t.f_2 e_2 \\ \texttt{case}\ e\ \{x_1.e_1; x_2.e_2\} & \triangleq & e[\tau](\lambda x_1{:}\tau_1.e_1)(\lambda x_2{:}\tau_2.e_2) \\ \texttt{unit} & \triangleq & \forall t.t \to t \\ \langle\rangle & \triangleq & \Lambda t.\lambda x{:}t.x \\ \\ \texttt{void} & \triangleq & \forall t.t \\ \texttt{abort}[\tau](e) & \triangleq & e[\tau] \end{array}$$

These are called *Church encodings* and were originally defined by Alonzo Church in the untyped λ-calculus.

Task 11 *Using the definitions above, we are able to define a System* F *equivalent for any type operator* $t.\tau$ pos *as defined in Section 3.1. Since we have also translated the constructors and recursors for these types, we can define* `map` *as well. Use these to define, for any* $t.\tau$ pos*, the System* F *equivalent of* $\mu t.\tau$ *along with* `fold` *and* `rec`*. (You may want to try encoding some specific cases first to develop some intuition.)*

Task 12 *Define* $\nu t.\tau$ *for any* $t.\tau$ pos*, along with* `gen` *and* `unfold`*.* **(Hint)** *Recall that coinductive types use hidden internal state in order to respond to* `unfold`*. As such, you may find it useful to use existential types, which are definable in System* F *per PFPL 17.3.*

A System T

A.1 Statics

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

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}(e; e_0; x.e_1) : \tau}\ (\textsc{Rec}) \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau}\ (\textsc{Lam})$$

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

A.2 Dynamics

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

$$\frac{}{\mathtt{rec}(\mathtt{z}; e_0; x.e_1) \mapsto e_0}\ (\textsc{Rec-IZ}) \qquad \frac{}{\mathtt{rec}(\mathtt{s}(e); e_0; x.e_1) \mapsto [\mathtt{rec}(e; e_0; x.e_1)/x]e_1}\ (\textsc{Rec-IS})$$

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

B Streams

B.1 Statics

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma, x : \tau' \vdash e_1 : \tau \quad \Gamma, x : \tau' \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{strgen}\ e\ \{\mathtt{hd}(x) \hookrightarrow e_1 \mid \mathtt{tl}(x) \hookrightarrow e_2\} : \mathtt{stream}\ \tau}\ (\textsc{Str}) \qquad \frac{\Gamma \vdash e : \mathtt{stream}\ \tau}{\Gamma \vdash \mathtt{hd}(e) : \tau}\ (\textsc{Hd})$$

$$\frac{\Gamma \vdash e : \texttt{stream}\ \tau}{\Gamma \vdash \texttt{tl}(e) : \texttt{stream}\ \tau}\ (\textsc{Tl})$$

B.2 Dynamics

$$\frac{}{\texttt{strgen}\ e\ \{\texttt{hd}(x) \hookrightarrow e_1 \mid \texttt{tl}(x) \hookrightarrow e_2\}\ \mathsf{val}}\ (\textsc{Str-V}) \qquad \frac{e \mapsto e'}{\texttt{hd}(e) \mapsto \texttt{hd}(e')}\ (\textsc{Hd-S})$$

$$\frac{}{\texttt{hd}(\texttt{strgen}\ e\ \{\texttt{hd}(x) \hookrightarrow e_1 \mid \texttt{tl}(x) \hookrightarrow e_2\}) \mapsto [e/x]e_1}\ (\textsc{Hd-I}) \qquad \frac{e \mapsto e'}{\texttt{tl}(e) \mapsto \texttt{tl}(e')}\ (\textsc{Tl-S})$$

$$\frac{}{\texttt{tl}(\texttt{strgen}\ e\ \{\texttt{hd}(x) \hookrightarrow e_1 \mid \texttt{tl}(x) \hookrightarrow e_2\}) \mapsto \texttt{strgen}\ ([e/x]e_2)\ \{\texttt{hd}(x) \hookrightarrow e_1 \mid \texttt{tl}(x) \hookrightarrow e_2\}}\ (\textsc{Tl-I})$$