Recitation 3:
Gödel's System T

15-312: Foundations of Programming Languages

Jeanne Luning Prak, Charles Yuan

January 31, 2018

1 Syntax

We now define and explore a language called **System T**. System **T** extends **E** with function types and replaces **E**'s primitive arithmetic operations with a more general operation on the natural numbers: **primitive recursion**. The syntax of System **T** is given by the following grammar:

Typ	τ	$::=$	$\texttt{nat}$	number
			$\tau_1 \to \tau_2$	function
Exp	e	$::=$	x	variable
			$\texttt{z}$	zero
			$\texttt{s}(e)$	successor
			$\texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(e)$	recursion
			$\lambda\,(x : \tau)\,e$	abstraction
			$e_1(e_2)$	application

Surprisingly, despite the loss of the arithmetic operations, **T** is capable of expressing every numeric computation in **E** and much more.

2 Abstraction and Application

Abstraction and application behave much as we would intuitively expect. An abstraction (function) binds a variable of type τ in e_1, and an application substitutes an expression $e_2 : \tau$ for that bound variable. Abstractions are first-class expressions: they have a type and can be passed to and returned from other abstractions. Because of this, System **T** is a language with *higher-order functions.*

The statics and dynamics for abstraction and application are given below.

2.1 Statics

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda\,(x : \tau_1)\,e_2 : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}$$

2.2 Dynamics

These dynamics rules are for the *eager* form of System **T**. All arguments are evaluated before being substituted into the body of a function. For a lazy dynamics, the $e_2 \longmapsto e_2'$ rule would be left out, along with the requirement on the last rule that e_2 be a value. Note the first rule, which states that functions are values.[1]

$$\frac{}{\lambda\,(x:\tau)\,e\ \mathsf{val}}$$

$$\frac{e_1 \longmapsto e_1'}{e_1(e_2) \longmapsto e_1'(e_2)}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{e_1(e_2) \longmapsto e_1(e_2')}$$

$$\frac{e_2\ \mathsf{val}}{(\lambda\,(x:\tau)\,e)(e_2) \longmapsto [e_2/x]e}$$

3 Natural Numbers

In System **T**, the natural numbers are defined as either zero, or the successor of a natural number. In addition to this definition, we also now have a single operation that works on naturals: recursion. The statics and dynamics of `nats` is given below, while recursion is discussed in the next section.

3.1 Statics

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}}$$

3.2 Dynamics

For a lazy form of System **T**, the requirement $e\ \mathsf{val}$ would be removed.

$$\frac{}{\mathtt{z}\ \mathsf{val}} \qquad \frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}}$$

4 Recursion

Now let's consider the recursion operation for System **T**:

$$\mathtt{rec}\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x)\ \mathtt{with}\ y \hookrightarrow e_1\}(e)$$

This operation cases on the value of e (either $\mathtt{z}$ or $\mathtt{s}(e')$). If e is $\mathtt{z}$ then the expression evaluates to e_0, the base case. If e is $\mathtt{s}(e')$ for some natural number e', then it recurs on e', binding the result of the recursion to y and e' to x for use in e_1.

[1] As they say in 15-150.

4.1 Statics

$$\frac{\Gamma \vdash e : \texttt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \texttt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(e) : \tau}$$

4.2 Dynamics

$$\frac{e \longmapsto e'}{\texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(e) \longmapsto \texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(e')}$$

$$\frac{}{\texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(\texttt{z}) \longmapsto e_0}$$

$$\frac{\texttt{s}(e) \ \mathsf{val}}{\texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(\texttt{s}(e)) \longmapsto [e, \texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow e_1\}(e)/x, y]e_1}$$

4.3 Examples for Recursion

4.3.1 Doubling

Understanding the recursor can be tricky, so let's go through an example. We'll write a function that doubles a number using the recursor. To do this, let's consider how we would implement doubling in Standard ML given the following datatype for natural numbers:

```
datatype nat = z | s of nat
```

We can double a number by doubling its predecessor and then taking the successor of that number twice:

```
fun double z = z
  | double (s x) = s (s (double x))
```

Let's rewrite this so that it matches the format of the recursor, with the predecessor of e bound to x and the result of the recursion bound to y:

```
fun double e =
  case e of
    z => z
  | s x => let val y = double x in s (s y) end
```

This makes it easier to now implement this using the recursor:

$$\lambda\,(e : \texttt{nat})\,\texttt{rec}\{\texttt{z} \hookrightarrow \texttt{z} \mid \texttt{s}(x) \texttt{ with } y \hookrightarrow \texttt{s}(\texttt{s}(y))\}(e)$$

As an exercise to make sure you understand the recursor, try to implement addition in the same manner.

4.3.2 Ackermann

System **T** is notable for its only explicit recursion operator being primitive recursion. However, its higher-order functions means that it is capable of computing non-primitive-recursive functions, like the well-known Ackermann function $A(m, n)$, defined as follows:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$

Ackermann is not primitive recursive since with a given recursive call, it is possible for n to increase. This is incompatible with the recursor construct, which requires its argument be *deconstructed* at every step. However, consider currying $A(m, n)$:

$$\begin{aligned} A(0)(n) &= \mathtt{s}(n) \\ A(\mathtt{s}(m))(0) &= A(m)(1) \\ A(\mathtt{s}(m))(\mathtt{s}(n)) &= A(m)(A(\mathtt{s}(m))(n)) \end{aligned}$$

If we treat $A(\mathtt{s}(m))$ as the function in question, we observe that whenever it is called recursively, its argument n decreases in value. We arrive at an insight: $A(\mathtt{s}(m))$ is a primitive recursive function in as of itself, and we should try writing it as a recursor.

However, there is one hiccup in computing $A(\mathtt{s}(m))$: the intermediate value we are collecting is not a number, but a function which applies $A(m)$ every step. Fortunately, System **T** allows us to write this. Consider the definitions:

$$\begin{aligned} \mathtt{id} &: \mathtt{nat} \to \mathtt{nat} \\ \mathtt{id} &\triangleq \lambda\,(x : \mathtt{nat})\,x \\ \mathtt{comp} &: (\mathtt{nat} \to \mathtt{nat}) \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \\ \mathtt{comp} &\triangleq \lambda\,(f : \mathtt{nat} \to \mathtt{nat})\,\lambda\,(g : \mathtt{nat} \to \mathtt{nat})\,\lambda\,(x : \mathtt{nat})\,f(g(x)) \\ \mathtt{iter} &: (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \\ \mathtt{iter} &\triangleq \lambda\,(f : \mathtt{nat} \to \mathtt{nat})\,\lambda\,(n : \mathtt{nat})\,\mathtt{rec}\{\mathtt{z} \hookrightarrow \mathtt{id} \mid \mathtt{s}(x)\ \mathtt{with}\ y \hookrightarrow \mathtt{comp}(f)(y)\}(n) \end{aligned}$$

What does `iter` do? Given a function f and a number n, it computes the n-th iterate of f, f^n. That's exactly what we need!

Rearranging, we have:

$$\begin{aligned} A(0)(n) &= \mathtt{s}(n) \\ A(\mathtt{s}(m))(n) &= \mathtt{iter}(A(m))(n)(A(m)(1)) \end{aligned}$$

Now we can move up one level to express A as a recursor, and write the Ackermann function in **T** (using a `succ` function that just takes the successor of a `nat`):

$$\begin{aligned} \mathtt{succ} &: \mathtt{nat} \to \mathtt{nat} \\ \mathtt{succ} &\triangleq \lambda\,(n : \mathtt{nat})\,\mathtt{s}(n) \\ \mathtt{ack} &: \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \\ \mathtt{ack} &\triangleq \lambda\,(m : \mathtt{nat})\,\mathtt{rec}\{\mathtt{z} \hookrightarrow \mathtt{succ} \mid \mathtt{s}(x)\ \mathtt{with}\ y \hookrightarrow \lambda\,(n : \mathtt{nat})\,\mathtt{iter}(y)(n)(y(\mathtt{s}(\mathtt{z})))\}(m) \end{aligned}$$

This is a constructive proof that despite not being primitive recursive, Ackermann is higher-order primitive recursive. System **T** allows us to compute a large set of functions like Ackermann, though all expressions in **T** provably terminate (cannot diverge). What does that mean from a computability theory perspective?