Recitation 3:
Gödel’s System T
15-312: Foundations of Programming Languages
Jeanne Luning Prak, Charles Yuan
January 30, 2019

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:

\[
\begin{align*}
\text{Typ} & \quad \tau \ ::= \ \text{nat} \quad \text{number} \\
& \quad \tau_1 \to \tau_2 \quad \text{function} \\
\text{Exp} & \quad e \ ::= \ x \quad \text{variable} \\
& \quad z \quad \text{zero} \\
& \quad s(e) \quad \text{successor} \\
& \quad \text{rec}\{z \mapsto e_0 \mid s(x) \text{ with } y \mapsto e_1\}(e) \quad \text{recursion} \\
& \quad \lambda (x : \tau) \ e \quad \text{abstraction} \\
& \quad e_1(e_2) \quad \text{application}
\end{align*}
\]

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 \( \tau \) 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

\[
\begin{align*}
\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\
\Gamma \vdash \lambda (x : \tau_1) e_2 : \tau_1 \to \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \to \tau_2 \\
\Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash e_1(e_2) : \tau_2
\end{align*}
\]
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 \( e_2 \mapsto e'_2 \) 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.

\[
\begin{align*}
\lambda (x : \tau) e & \quad \text{val} \\
\frac{e_1 \mapsto e'_1}{e_1(e_2) \mapsto e'_1(e_2)} \\
\frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{e_1(e_2) \mapsto e_1(e'_2)} \\
\frac{e_2 \text{ val}}{\lambda (x : \tau) e(e_2) \mapsto [e_2/x]e}
\end{align*}
\]

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 \texttt{nats} is given below, while recursion is discussed in the next section.

3.1 Statics

\[
\frac{\Gamma \vdash z : \text{nat}}{\Gamma \vdash \text{z} : \text{nat}} \quad \frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{s}(e) : \text{nat}}
\]

3.2 Dynamics

For a lazy form of System T, the requirement \( e \text{ val} \) would be removed.

\[
\frac{z \text{ val}}{z \text{ val}} \quad \frac{e \text{ val}}{\text{s}(e) \text{ val}}
\]

4 Recursion

Now let’s consider the recursion operation for System T:

\[
\text{rec}\{z \leftarrow e_0 \mid \text{s}(x) \text{ with } y \leftarrow e_1\}(e)
\]

This operation cases on the value of \( e \) (either \( z \) or \( \text{s}(e') \)). If \( e \) is \( z \) then the expression evaluates to \( e_0 \), the base case. If \( e \) is \( \text{s}(e') \) 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

\[ \Gamma \vdash e : \text{nat} \quad \Gamma, x : \text{nat}, y : \tau \vdash e_1 : \tau \]
\[ \Gamma \vdash \text{rec}\{z \leftarrow e_0 \mid s(x) \text{ with } y \leftarrow e_1\}(e) : \tau \]

4.2 Dynamics

\[ e \mapsto e' \]
\[ \text{rec}\{z \leftarrow e_0 \mid s(x) \text{ with } y \leftarrow e_1\}(e) \mapsto \text{rec}\{z \leftarrow e_0 \mid s(x) \text{ with } y \leftarrow e_1\}(e') \]
\[ \text{rec}\{z \leftarrow e_0 \mid s(x) \text{ with } y \leftarrow e_1\}(e) \mapsto e_0 \]
\[ \text{s}(e) \text{ val} \]
\[ \text{rec}\{z \leftarrow e_0 \mid s(x) \text{ with } y \leftarrow e_1\}(e) \mapsto | e, \text{rec}\{z \leftarrow e_0 \mid s(x) \text{ with } y \leftarrow e_1\}(e)/x,y|e_1 \]

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:

\[ \lambda (e : \text{nat}) \text{rec}\{z \leftarrow z \mid s(x) \text{ with } y \leftarrow s(y)\}(e) \]

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:

\[ A(0, n) = n + 1 \]
\[ A(m + 1, 0) = A(m, 1) \]
\[ A(m + 1, n + 1) = A(m, A(m + 1, n)) \]
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 \textit{deconstructed} at every step. However, consider currying \( A(m, n) \):

\[
A(0)(n) = s(n) \\
A(s(m))(0) = A(m)(1) \\
A(s(m))(s(n)) = A(m)(A(s(m))(n))
\]

If we treat \( A(s(m)) \) as the function in question, we observe that whenever it is called recursively, its argument \( n \) decreases in value. We arrive at an insight: \( A(s(m)) \) 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 \( A(s(m)) \): the intermediate value we are collecting is not a number, but a function which applies \( A(m) \) every step. Fortunately, System \( \text{T} \) allows us to write this. Consider the definitions:

\[
\text{id} : \text{nat} \rightarrow \text{nat} \\
\text{id} ≜ \lambda (x : \text{nat}) x \\
\text{comp} : (\text{nat} \rightarrow \text{nat}) ightarrow (\text{nat} \rightarrow \text{nat}) ightarrow \text{nat} \rightarrow \text{nat} \\
\text{comp} ≜ \lambda (f : \text{nat} \rightarrow \text{nat}) \lambda (g : \text{nat} \rightarrow \text{nat}) \lambda (x : \text{nat}) f(g(x)) \\
\text{iter} : (\text{nat} \rightarrow \text{nat}) ightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \\
\text{iter} ≜ \lambda (f : \text{nat} \rightarrow \text{nat}) \lambda (n : \text{nat}) \text{rec} \{z ↪ \text{id} | s(x) \text{ with } y ↪ \text{comp}(f)(y)(n)\}
\]

What does \( \text{iter} \) 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:

\[
A(0)(n) = s(n) \\
A(s(m))(n) = \text{iter}(A(m))(n)(A(m)(1))
\]

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

\[
\text{succ} : \text{nat} \rightarrow \text{nat} \\
\text{succ} ≜ \lambda (n : \text{nat}) s(n) \\
\text{ack} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \\
\text{ack} ≜ \lambda (m : \text{nat}) \text{rec} \{z ↪ \text{succ} | s(x) \text{ with } y ↪ \lambda (n : \text{nat}) \text{iter}(y)(n)(y(s(z)))(m)\}
\]

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