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:

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

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

\[
\begin{align*}
\Gamma, x : \tau_1 \vdash e_2 : \tau_2 & \quad \Gamma \vdash \lambda(x : \tau_1) e_2 : \tau_1 \to \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1 & \quad \Gamma \vdash e_1(e_2) : \tau_2
\end{align*}
\]
2.2 Dynamics

These dynamics rules are for the *eager* form of System \( \mathbf{T} \). All arguments are evaluated before being substituted into the body of a function. For a lazy dynamics, the \( e_2 \rightarrow e'_2 \) 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.\(^1\)

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

\[
(\lambda(x: \tau)e)(e_2) \rightarrow [e_2/x]e
\]

3 Natural Numbers

In System \( \mathbf{T} \), 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

\[
\begin{align*}
\Gamma \vdash z : \text{nat} \\
\Gamma \vdash e : \text{nat} \\
\Gamma \vdash s(e) : \text{nat}
\end{align*}
\]

3.2 Dynamics

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

\[
\begin{align*}
z \text{ val} \\
e \text{ val} \\
s(e) \text{ val}
\end{align*}
\]

4 Recursion

Now let’s consider the recursion operation for System \( \mathbf{T} \):

\[
\text{rec}\{e_0; y.e_1\}(e)
\]

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

\(^1\)As they say in 15-150.
4.1 Statics

\[ \Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, y : \tau \vdash e_1 : \tau \]

\[ \Gamma \vdash \text{rec}\{e_0; y.e_1\}(e) : \tau \]

4.2 Dynamics

\[ e \mapsto e' \quad \text{rec}\{e_0; y.e_1\}(e) \mapsto \text{rec}\{e_0; y.e_1\}(e') \]

\[ \text{rec}\{e_0; y.e_1\}(z) \mapsto e_0 \quad \text{s}(e) \text{ val} \]

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

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:

\[ \lambda (e : \text{nat}) \text{rec}\{z; y.s(s(y))\}(e) \]

Let’s see how our dynamics rule captures the computation exactly the way we want:

\[
\begin{align*}
\text{double } s(z) & \mapsto \text{rec}\{z; y.s(s(y))\}(s(z)) \\
& \mapsto [\text{rec}\{z; y.s(s(y))\}(z)/y]s(s(y)) \\
& = s(s[\text{rec}\{z; y.s(s(y))\}(z)]) \\
& \mapsto s(s(z))
\end{align*}
\]

Application  Since \( s(z) \) val  Substitution  By evaluation rule of \( s() \)

As an exercise to make sure you understand the recursor, try to implement addition in the same manner.
5 Expressiveness of System T

5.1 Termination

Every well-typed expression e in System T terminates, i.e. if \( \vdash e : \tau \), then e \( \rightarrow^* v \) where v val. However, this seemingly harmless property makes our language not Turing-complete. The full proof is in PFPL; here we present the general idea.

We first define some relevant concepts:

In eager dynamics\(^2\), we say a function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is definable in T, iff there exists an expression \( e : \text{nat} \rightarrow \text{nat} \) in T such that for all \( n \in \mathbb{N} \), \( e(n) \rightarrow^* f(n) \).\(^3\)

Two assumptions in this proof:

1. We can encode every expression in T as natural numbers. (One example: Gödel’s numbering) Fix some encoding, let \( \lceil e \rceil \) be the encoded numeral of expression e in T.
2. We curry products in the domain of mathematical functions. So a function \( f : \mathbb{N}^2 \rightarrow \mathbb{N} \) has its correspondence in T as \( e_f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \).

Consider a function \( \text{eval} : \mathbb{N}^2 \rightarrow \mathbb{N} \) defined as \( \text{eval}(\lceil e \rceil, m) = n \) iff \( e(\lceil m \rceil) \rightarrow^* \pi \).

Claim: \( \text{eval} \) is not definable in System T.

The proof is by the classic diagonalization argument. Assume for the sake of contradiction that it is definable, i.e. there exists \( e_{\text{eval}} : \text{nat} \rightarrow \text{nat} \) such that \( e_{\text{eval}}(\lceil e \rceil)(m) \rightarrow^* \text{eval}(\lceil e \rceil, m) \).

Then consider an expression \( e_\delta = \lambda(x : \text{nat}) s(e_{\text{eval}}(x)(x)) \).

By termination property of T, we know \( e_\delta(\lceil e_\delta \rceil) \rightarrow^* v \) for some natural number v.

Then by definition \( e_\delta(\lceil e_\delta \rceil) \rightarrow^* s(e_{\text{eval}}(\lceil e_\delta \rceil)(\lceil e_\delta \rceil)) \rightarrow^* s(e_\delta(\lceil e_\delta \rceil)) \rightarrow^* v + 1 \).

Contradiction! \( \text{eval} \) is not definable in System T. However, \( \text{eval} \) is clearly computable.\(^4\)

Notice the argument above applies to System T, as well as any other language that is total.

5.2 Extra Reading: 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 deconstructed at every step. However, consider currying \( A(m, n) \):

\(^2\)this definition is not so great under lazy dynamics, why and how do we fix that?
\(^3\)\( \pi \) is a shorthand for numeral values in System T. So \( \overline{2} = s(s(z)) \)
\(^4\)If you are not convinced, you will get to implement an interpreter for System T in Assignment 2.
\[ A(0)(n) = s(n) \]
\[ A(s(m))(0) = A(m)(1) \]
\[ A(s(m))(s(n)) = A(m)(A(s(m))(n)) \]

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 T allows us to write this. Consider the definitions:

\[
\begin{align*}
\text{id} &: \text{nat} \to \text{nat} \\
\text{id} &\triangleq \lambda (x : \text{nat}) x \\
\text{comp} &: (\text{nat} \to \text{nat}) \to (\text{nat} \to \text{nat}) \to \text{nat} \to \text{nat} \\
\text{comp} &\triangleq \lambda (f : \text{nat} \to \text{nat}) \lambda (g : \text{nat} \to \text{nat}) \lambda (x : \text{nat}) f(g(x)) \\
\text{iter} &: (\text{nat} \to \text{nat}) \to \text{nat} \to \text{nat} \to \text{nat} \\
\text{iter} &\triangleq \lambda (f : \text{nat} \to \text{nat}) \lambda (n : \text{nat}) \text{rec}\{\text{id}; y.\text{comp}(f)(y)\}(n)
\end{align*}
\]

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 T (using a \( \text{succ} \) function that just takes the successor of a \( \text{nat} \)):

\[
\begin{align*}
\text{succ} &: \text{nat} \to \text{nat} \\
\text{succ} &\triangleq \lambda (n : \text{nat}) s(n) \\
\text{ack} &: \text{nat} \to \text{nat} \to \text{nat} \\
\text{ack} &\triangleq \lambda (m : \text{nat}) \text{rec}\{\text{succ}; y.\lambda (n : \text{nat}) \text{iter}(y)(n)(y(s(z)))\}(m)
\end{align*}
\]

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). We showed in the previous section that this means System T is not Turing-complete.