Lecture Notes on Properties of System T

15-814: Types and Programming Languages
Jan Hoffmann

Lecture 6
Thursday, September 12, 2024

1 Introduction

In the previous lecture, we defined the static and dynamic semantics of System T. In this lecture, we study the properties of System T.

First, we show the dynamic semantics has the properties that we were aiming for when defining it:

- Values are final states that do not appear on the left side of transitions.
- For programs that are not values, there is exactly one transition to another program.

A key to proving these properties is *type soundness*, which links the static and dynamic semantics of a language through *progress* and *preservation*.

Next, we consider *normalization*: all System T programs evaluate to a value. We also say all programs terminate and call System T a total language. We discuss the difficulties of proving *normalization*. On Homework Assignment 3, you will prove *canonicity*, which corresponds to normalization at the base type `nat` for System T.

Finally, we discuss the expressivity of System T. We characterize the functions $\mathbb{N}^k \to \mathbb{N}$ which can be defined and construct a (total) computable function that cannot be defined in T (namely, a self-interpreter). We also discuss Blum's Size Theorem, which shows a shortcoming of total languages that hampers their practicality.

2 Type Soundness

Static and Dynamic Semantics Recall the definition of the static semantics from the previous lecture. We defined the judgment

$$\Gamma \vdash e : \tau$$

which states that expression e has type τ in context Γ. *Programs* are well-typed closed expressions e, that is, we have $\cdot \vdash e : \tau$ and usually just write $e : \tau$.

We defined one inference rule for each syntactic form. For example, for function application we defined the rule T_{ap}.

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{app}(e_1, e_2) : \tau'}\ T_{\text{ap}}$$

For the dynamic semantics we defined two judgments $e\ \mathsf{val}$, which states that the expression e is a value, and $e \longmapsto e'$, which states that e steps to e'. If we consider *call-by-name evaluation order* then we have the following rules for function application.

$$\frac{e_1 \longmapsto e_1'}{\mathtt{app}(e_1, e_2) \longmapsto \mathtt{app}(e_1', e_2)}\ E_{\text{ap1}} \qquad \frac{}{\mathtt{app}((\lambda\,(x : \tau)\,e), e_2) \longmapsto [e_2/x]e}\ E_{\text{ap4}}$$

We now want to show that (1) values are final states that do not appear on the left side of transitions and (2) there is exactly one transition for programs that are not values. The first part and the determinacy of the transition relation can be directly proved by rule induction.

Lemma 1 (Finality of Values) *There is no expression e such that $e\ \mathsf{val}$ and $e \longmapsto e'$ for some expression e'.*

The previous lemma can be proved by rule induction on $e\ \mathsf{val}$ and $e \longmapsto e'$.

Lemma 2 (Determinacy) *If $e \longmapsto e'$ and $e \longmapsto e''$ then $e' \equiv_\alpha e''$.*

In the previous lemma, we write $e' \equiv_\alpha e''$ to emphasize the fact that bound variables may be renamed during evaluation. However, we identify α-equivalent expressions, so $e = e'$ means $e \equiv_\alpha e'$ anyway. The proof of the lemma is by rule induction on $e \longmapsto e'$.

Progress and Preservation What remains to be shown is that the evaluation does not *get stuck* before reaching a value. We first show that programs are not stuck states: either they are values or there is a transition. This property is called *progress* and we prove it by rule induction on the type judgment $e : \tau$.

Theorem 3 (Progress) *If $e : \tau$ then either e val or there exists an expression e' such that $e \longmapsto e'$.*

During an evaluation, we might still encounter a bad expression like $\mathtt{app}(\mathtt{s}(\mathtt{z}), \mathtt{z})$, which is not a value but also does not have a transition. However, this does not happen: Programs always transition to programs of the same type and the expression $\mathtt{app}(\mathtt{s}(\mathtt{z}), \mathtt{z})$ is not a program. This property is called preservation.

Theorem 4 (Preservation) *If $e : \tau$ and $e \longmapsto e'$ then $e' : \tau$.*

The proof proceeds by rule induction on the judgment $e \longmapsto e'$. We will apply inversion to $e : \tau$. To get a flavor of the proof, we go through the case E_{ap1}.

Proof: Case E_{ap1}
Then $e = \mathtt{app}(e_1, e_2)$, $e_1 \longmapsto e_1'$, and $e' = \mathtt{app}(e_1', e_2')$.
Our goal is to show $\mathtt{app}(e_1', e_2') : \tau$.
We first apply inversion to $\mathtt{app}(e_1, e_2) : \tau$ and derive $e_1 : \tau' \to \tau$, and $e_2 : \tau'$.
Now we can use the induction hypothesis with $e_1 \longmapsto e_1'$ and obtain $e_1' : \tau' \to \tau$. But then it follows that $e' = \mathtt{app}(e_1', e_2') : \tau$ because we can derive this judgement by applying rule E_{ap1} with the premises $e_1' : \tau' \to \tau$ and $e_2 : \tau'$. □

Together, progress and preservation ensure that programs do not get stuck during their evaluation. This is sometimes expressed with the slogan *well typed programs don't go wrong*. While this is the case for System T, programs in more expressive languages can diverge or terminate with an error even if they are well typed.

3 Normalization

In System T, all programs terminate. One the one hand, this seems intuitively clear since recursive (or looping) computations are implemented with the recursor, which terminates after a bounded number of iterations. One the other hand, the λ-Calculus does not even have numbers or recursion in the

definition of its syntax but we can still define λ-expressions, such as the Y combinator, that give rise to general recursion (and divergence).

We can for instance attempt to implement the Church numerals in System T. However, we now have to annotate the function abstractions in the numerals with types. We can define $\overline{n}$ using an arbitrary but fixed type τ as follows.

$$\overline{n} \triangleq \lambda\,(s : \tau \to \tau)\,\lambda\,(z : \tau)\,s^n(z)$$

However, since we have to fix the type τ, we can only express iterations whose result is of type τ. In this way, we can express the successor, addition, and multiplication but not much more. Helmut Schwichtenberg showed that these functions are the extended polynomials [Sch75].

The reason programs in System T terminate is that the type system rules out expressions like the Y combinator. For example, recall the lambda expression

$$\Omega \triangleq (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$$

An expression like Ω cannot by typed in System T because for all types τ and τ'

$$\cdot \not\vdash (\lambda\,(x : \tau)\,x(x)) : \tau'\,.$$

Because of the rule T_{ap}, we know that $\tau = \tau \to \tau_2$ but for all τ and τ_2, we have $\tau \neq \tau \to \tau_2$.

Normalization and Logical Relations We define

$$e \Downarrow v \quad \text{if} \quad e \longmapsto^* v \text{ and } v\ \mathsf{val}$$

We can then formulate the normalization theorem as follows.

Theorem 5 (Normalization) *If $e : \tau$ then $e \Downarrow v$ for some v.*

The proof of Theorem 5 is challenging and requires a technique called *Tait's method* or *logical relations*. To motivate the need for this technique, let us observe why induction on the typing judgment alone does not suffice. First, note that the theorem is immediate for *values*. For instance, consider the case of successor:

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}}\ T_{\mathrm{s}}$$

Observe that, by assumption in the theorem, we have $\Gamma = \cdot$. By induction hypothesis, we know $e \Downarrow v$. We can apply the canonical-forms lemma (using

$\tau = \texttt{nat}$) to obtain $v = \overline{n}$. Therefore, we know that $\texttt{s}(e) \mapsto^* \texttt{s}(\overline{n}) \triangleq \overline{n+1}$, and so the theorem holds.

However we are not as fortunate in case of an elimination rule. Consider application:

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \texttt{app}(e_1, e_2) : \tau}\ T_{\text{ap}}$$

By assumption, we have again that $\Gamma = \cdot$. By induction, we get $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$ for values v_1 and v_2. So we have that $\texttt{app}(e_1, e_2) \mapsto^* \texttt{app}(v_1, v_2)$. We can now use the canonical-forms lemma to show that v_1 has the form $\lambda(x : \tau)e'$. Then we can step $\texttt{app}(v_1, v_2) \mapsto [v_2/x]e'$. But now we are stuck because we cannot apply the induction hypothesis to $[v_1/x]e' : \tau$ since it is not a premise of the rule T_{ap}.

To make the proof go through we have to strengthen the induction to a property of expressions called hereditary termination, which is defined inductively on the type structure:

- $\text{HT}_{\texttt{nat}}(e)$ if $e \Downarrow v$ for some v.
- $\text{HT}_{\tau_1 \to \tau_2}(e)$ if $e \Downarrow \texttt{lam}\{\tau_1\}(x.e')$ and for all e_1 such that $\text{HT}_{\tau_1}(e_1)$, we have $\text{HT}_{\tau_2}([e_1/x]e')$.

You will use such a construction in the homework, where hereditary termination is replaced by a more general methodology based on candidates to prove canonicity, which is normalization at the base type for System T.

Theorem 6 (Canonicity) *Given a closed program $e : \texttt{nat}$ of base type, we have that $e \Downarrow \overline{n}$ for some n.*

4 Definability and Undefinability

System T is very expressive. It can express all primitive recursive functions but also very fast growing functions that are not primitive recursive. An example of such a function is Ackermann's function $A : \mathbb{N}^2 \to \mathbb{N}$ which is defined as follows.

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

Ackermann's function grows fast. For instance $A(3, n) = 2^{n+3} - 3$ and $A(4, n) = 2^{2^{\cdots^2}} - 3$ where there are $n + 3$ powers in the stack function.

We say that a function $f : \mathbb{N}^k \to \mathbb{N}$ is *definable* in System T if there exists an expression $e_f : \sigma_k$ such that $e_f(\overline{n_1})\cdots(\overline{n_k}) \Downarrow \overline{f(n_1, \ldots, n_k)}$ for all natural numbers $n_1, \ldots, n_k$. Here we define $\sigma_0 = \texttt{nat}$ and $\sigma_{n+1} = \texttt{nat} \to \sigma_n$.

We can characterize the definable functions as follows.

> The functions definable in System T are all commutable functions that can be proved to be terminating in Peano arithmetic; intuitively, by a nested induction on the natural numbers.

Undefinability System T is incomplete in the following sense. There are total and computable functions that cannot be defined in T. The classic example of such a function is a self-interpreter. An interpreter is a function that accepts two arguments, a program $e : \texttt{nat} \to \texttt{nat}$ and an argument $n \in \mathbb{N}$, and returns the result of evaluating $e(\overline{n})$, that is, $m \in \mathbb{N}$ such that $e(\overline{n}) \Downarrow \overline{m}$. Intuitively, an interpreter is a computable function. We can for instance implement the dynamic semantics of T and perform steps until we reach a value. The normalization theorem guarantees that this interpreter terminates for each program.

A self-interpreter is a definition of an interpreter for System T in System T. To show that we cannot define such a function in T, we use diagonalization to show that if we have a self-interpreter then we can construct an expression that *evaluates to the successor of the result of its evaluation*, which is a contradiction.

To define a self-interpreter, we need to talk about functions on the natural numbers. So instead of a program e, the self-interpreter should accept an argument $n \in \mathbb{N}$. To this end, we observe that we can effectively encode an expression e as a number $\ulcorner e \urcorner \in \mathbb{N}$ so that e can be computed from $\ulcorner e \urcorner$. Practically, the encoding can be done by writing a System T program to a file and saving it on a hard drive. The decoding can be to read the file from the hard drive. We can express such an encoding mathematically by using *Gödel numbering*. However, we are not defining it here.

Fix an effective encoding $\ulcorner e \urcorner$ of expressions.

We call the function $f_{\text{univ}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ a *universal function* if for every expression $e : \texttt{nat} \to \texttt{nat}$

$$f_{\text{univ}}(\ulcorner e \urcorner, m) = n \quad \text{iff} \quad e(\overline{m}) \Downarrow \overline{n}$$

We do not need to specify what happens if f_{univ} is applied to an argument whose first component does not correspond to an encoded expression to make the proof work. So the result can be an arbitrary number in this case.

Theorem 7 *We can implement a universal function for System T in the λ-calculus.*

The previous theorem follows from the fact that we can compute the decoding of an expression and then feed it to an interpreter for T.

Now we prove that universal functions cannot be implemented in T. The proof idea is the following: If we can implement a self-interpreter e_{self} in T then we can call it with it's own encoding as in the following expression $e_{\text{self}}(\overline{\ulcorner e_{\text{self}} \urcorner})$. This form of self-reference enables us to create a looping computation. However, this contradicts the normalization theorem.

Theorem 8 (Incompleteness) *Universal functions are not definable in System T.*

Proof: Assume that $e_{\text{univ}} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ implements a universal function. Then

$$e_\delta = \lambda\,(m : \mathtt{nat})\, e_{\text{univ}}(m)(m)$$

implements the diagonal function $\delta : \mathbb{N} \to \mathbb{N}$. We have

$$e_\delta(\overline{\ulcorner e \urcorner}) \Downarrow \overline{n} \quad \text{if} \quad e(\overline{\ulcorner e \urcorner}) \Downarrow \overline{n} \tag{1}$$

Now we can construct the offending expression $e_\Delta(\overline{\ulcorner e_\Delta \urcorner})$ where

$$e_\Delta \triangleq \lambda\,(x : \mathtt{nat})\, \mathtt{s}(e_\delta(x))$$

Assume $e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \Downarrow \overline{n}_0$.
The we have the following contradiction:

$$e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \longmapsto \mathtt{s}(e_\delta(\overline{\ulcorner e_\Delta \urcorner})) \longmapsto^* \mathtt{s}(\overline{n_0})$$

The first step follows from the definition of e_Δ and the evaluation rules for function application. The following steps follow from the evaluation rule for the successor and Equation 1. □

Note that, in the proof, we did not use any properties of System T other than being able to define functions using the successor function. Therefore, the theorem effectively applies to every total language.

However, the proof only applies to total languages. We used the normalization theorem when we assumed $e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \Downarrow \overline{n}_0$. We can construct such an expression from a self interpreter in a partial language like the λ-Calculus. However, we would not get a contradiction but a proof that shows that $e_\Delta(\overline{\ulcorner e_\Delta \urcorner})$ does not terminate.

5 Blum's Size Theorem

We have already experienced that programming in a total language can be difficult when we implemented the predecessor function using the schema of iteration in the λ-Calculus. Another example is the implementation of Ackermann's function in System T.

It is in fact a general limitation of (expressive) total languages that there are functions that are difficult to implement while they are easy to implement in the λ-Calculus. The reason is that a program in a total language already includes its termination proof. However, there are (succinct) programs that require elaborate termination proofs. In general, these programs cannot be directly expressed in a total language and sometimes each equivalent program in the total language has to be very large. This fact is formalized by Blum's Size Theorem [MY78] (page 166).

Theorem 9 (Blum's Size Theorem) *Let $\mathcal{L}$ be a total programming language in which we can define an infinite number of functions $\mathbb{N} \to \mathbb{N}$. Then there exists a family of functions $(f_n)_{n\in\mathbb{N}}$ such that $f_n : \mathbb{N} \to \mathbb{N}$ and*

- *f_n is definable in the λ-Calculus with an implementation of size n*
- *f_n is definable in $\mathcal{L}$ but the shortest implementation has size $m \geq 2^{2^n}$.*

I picked the function 2^{2^n} as an example to make the theorem concrete but it could be another fast growing computable function.

To prove the theorem, we have to precisely define the concepts *programming language* and *size of a program*. Machtey and Young [MY78] simply say a programming language is an enumeration of computable functions so that for two functions in the enumeration their composition appears in the enumeration as well. They also leave the notion of size abstract and only require that there are finitely many programs of a fixed size.

Exercises

Exercise 1 *Prove Lemma 2.*

Exercise 2 *Define Ackermann's function in System T.*

References

[MY78] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Elsevier Science Inc., USA, 2nd edition, 1978.

[Sch75] Helmut Schwichtenberg. Definierbare funktionen imλ-kalkül mit typen. *Arch. Math. Log.*, 17(3-4):113–114, 1975.