Lecture Notes on Gödel's System T

15-814: Types and Programming Languages
Jan Hoffmann

Lecture 5
Tuesday, September 12, 2023

1 Introduction

In this lecture we discuss *System T*, which has been developed by Kurt Gödel as a logical system in 1930s [Göd80]. We present System T as a programming language that features natural numbers, higher-order functions, and the schema of primitive recursion. In contrast to the λ-Calculus, System T has a type system that is part of the *static semantics*, which defines the set of programs. This makes System T a *normalizing* language, that is, every System T program terminates. While termination is a desirable property, we will see that normalizing languages cannot express all total functions we can implement in the λ-Calculus and have other limitations that make them difficult to use as general purpose programming languages.

2 Historical Context

This lecture is the first time we encounter the *Curry–Howard Correspondence* that links mathematical logic and (normalizing) programming languages. We will revisit and explore this connection later in the course. Here, we just note that System T was introduced as a logical reasoning system that has been part of Gödel's response to *Hilbert's program*, which aimed at grounding mathematics in a set of axioms and reasoning rules that implies all mathematical theorems, including the consistency of the axioms. The work on Hibert's program was triggered by the foundational crises of mathematics that emerged after the discovery of *Russel's paradox* in 1901.

Gödel approached Hilbert's program by investigating if a solution is at all possible. In 1929, he proved his *completeness theorem* which states that there is a set of *decent* inference rules so that

> If a theorem follows from a set of assumptions then it can be proved by a derivation tree using the inference rules.

This was followed in 1931 by Gödel's famous *incompleteness theorems*, which imply that the goals of Hilbert's program cannot be achieved. The incompleteness theorems [Raa20] state that

> for every set of *decent* axioms and inference rules there are theorems that we cannot derive.

The incompleteness theorem only applies to sets of axioms that are powerful enough to reason about integer arithmetic. The proof of the first incompleteness theorem constructs a theorem that intuitively states *I'm not provable* and cannot be proved or disproved. The second incompleteness theorem constructs a theorem that intuitively states *the set of axioms is consistent* and also cannot be proved or disproved (consistent means that the axioms are not contradictory).

Gödel's work was extremely innovative and introduced ideas such as encoding of data in numbers and the necessity a mathematical notion of compatibility. The latter arose from the need to precisely define what a *decent* axiom or rule is. Gödel's idea was that a rule or axiom is decent if it can be mechanically determined if it was used correctly. Today, we would say that rules and axioms should be *decidable*. To this end, Gödel proposed the notion of *general recursive functions* as a formalization of compatibility in 1934. This definition was later shown to be equivalent to Church's λ-Calculus (proposed in 1935) and the Turing's machines (proposed in 1936).

3 System T

System T has been introduced by Gödel in 1941 as a mitigation of the second incompleteness theorem. He showed that it is possible to prove the consistency of the theory of arithmetic (Peano arithmetic) in a *higher-order* version of the same theory in which theorems can quantify over theorems (not only numbers). Here, we define System T as the corresponding programming language with natural numbers, higher-order functions, and a type system.

Syntax The syntax of System T is given by types and expressions. The types consists of natural numbers `nat` and function types $\tau_1 \to \tau_2$.

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

Like in the λ-Calculus, we have syntactic forms for function abstraction and function application. A difference is that we function abstractions $\lambda\,(x:\tau)\, e$ are annotated with types τ that indicated the type of the function argument. This is not a requirement but ensures the desirable property that types of closed expressions are unique.

The introduction for natural numbers are zeroand the successor, which yields a unary encoding. The elimination form for natural numbers is the recursor $\texttt{rec}\{e_0; x.y.e_1\}(e)$, which implements the schema of primitive recursion.

For $n \in \mathbb{N}$, we define the numeral $\overline{n}$ inductively as follows.

$$\begin{array}{lcl} \overline{0} & \triangleq & \texttt{z} \\ \overline{n+1} & \triangleq & \texttt{s}(\overline{n}) \end{array}$$

Recursor The recursor defines a terminating recursive computation using *primitive recursion*. To understand how it works, recall the schema of primitive recursion from Lecture 3:

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & h\ n\ (f\ n) \end{array}$$

Translating f to our recursor lead the following expression

$$f = \lambda\,(n : \texttt{nat})\, \texttt{rec}\, n\, \{\texttt{z} \hookrightarrow e_c \mid \texttt{s}(x)\, \texttt{with}\, y \hookrightarrow e_h(x)(y)\}$$

where the expression e_c is the implementation of the constant c and the expression e_h is the implementation of the function h. So e_c is the base case of the recursion and e_h is the step function that will be applied n times.

For example, we can define the addition function in System T as follows.

$$\mathit{add} = \lambda\,(n : \mathtt{nat})\,\lambda\,(m : \mathtt{nat})\,\mathtt{rec}\,n\,\{\mathtt{z} \hookrightarrow m \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow \mathtt{s}(y)\}$$

We will see how such an expression behaves in an evaluation when we discuss the dynamic semantics.

4 Static Semantics

We use a type system to define the programs of System T. We want to create a normalizing language, so every program should evaluate to a number or a function, which are the values of System T. But what are the programs of System T? In the λ-Calculus, we could define programs to be closed expressions, that is, expression that do not contain free variables. However, this approach seems to not work for System T. What should for instance be the result of evaluating the expression $\mathtt{app}(\mathtt{s}(\mathtt{z}), \mathtt{z})$? It does not make sense to apply $\overline{1}$ to $\overline{0}$ because $\overline{1}$ is not a function. Using a type system, we exclude such nonsensical programs and define programs to be well-typed closed expressions.

We inductively define the judgment

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

where e is an expression, τ is a type, and Γ is variable context that maps variables to types. We define

$$\begin{array}{rcll} \Gamma & ::= & \cdot & \text{empty context} \\ & & \Gamma, x : \tau & \text{mapping} \end{array}$$

We require that every variable appears at most once in a context Γ. So if we write Γ_1, Γ_2 we refer to the joined map of Γ_1 and Γ_2 with the implicit side condition $\mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_1) = \varnothing$. The order in which variable bindings appear in Γ does not matter.

The *programs* of System T are well-typed and closed expressions e or, more precisely, expressions for which we can derive the judgment

$$\cdot \vdash e : \tau\,.$$

We usually just write $e : \tau$ for a program of type τ.

The type rules of System T are *syntax directed*. That means that there is exactly one type rule for each syntactic form. The rule T_{var} states that the

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ T_{\text{var}} \qquad \frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}\ T_{\mathtt{z}} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}}\ T_{\mathtt{s}}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}\{e_0; x.y.e_1\}(e) : \tau}\ T_{\text{rec}}$$

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

Figure 1: Type Rules of System T

expression x is type τ if the hypothesis $x : \tau$ is part of the context Γ. The rule $T_{\mathtt{z}}$ states that the constant $\mathtt{z}$ has type $\mathtt{nat}$ in every context. The rule $T_{\mathtt{s}}$ states that $\mathtt{s}(e)$ has type $\mathtt{nat}$ in context Γ if e has type $\mathtt{nat}$ in the same context.

To understand the rule T_{rec} we can consult our previous discussion of the intended meaning of the recursor. The expression e is supposed to evaluate to a natural number that indicates the number of times we should iterate the step function. The result of the computation of the recursor is of type τ. Consequently, the expression e_0 for the base is also of type τ. Finally, the step functions e_1 also has type τ and consumes the predecessor of the current iteration ($x : \mathtt{nat}$) and the result of the previous iteration ($y : \tau$). It is important to note that the result type τ of the recursor is arbitrary and includes function types. This makes System T very expressive and lets us, for instance, implement functions that are not primitive recursive such as Ackermann's function.

The rule T_{lam} states that a function abstraction has type $\tau \to \tau'$ if the function body e has type τ' under the assumption that the argument x has type τ. An interesting detail is that we use the context $\Gamma, x : \tau$ in the typing of the expression e without verifying that $x \notin \mathit{dom}(\Gamma)$. We can get away with this because of α-equivalence: $\mathtt{lam}\{\tau\}(x.e)$ binds the variable x and we pick a representative from the equivalence class so that the variable x does not appear in Γ.

The rule T_{ap} for function applications states that e_1 has to have a function type $\tau \to \tau'$ in context Γ and that e_2 has to have a matching argument type τ in the same context. Then the application $\mathtt{app}(e_1, e_2)$ has type τ' in context Γ.

The type judgment $\Gamma \vdash e : \tau$ is a *hypothetical judgment*. The variable typings $x : \tau'$ in the context Γ are the hypotheses. The meaning of a hypothesis

is that the variable x does not stand for arbitrary expressions like in the λ-Calculus but for expressions of type τ'. We can replace the occurrences of x in the expression e with an expression of the same type τ' and can then derive that the resulting expression has type τ. This is made precise by the following lemma, which can be proved by rule induction on $\Gamma, x : \tau' \vdash e : \tau$.

Lemma 1 (Substiution) *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash [e'/x]e : \tau$.

We can prove inversion lemmas. For example, we show the following one for the successor. The proof is by induction on the judgment $\Gamma \vdash \mathtt{s}(e) : \tau$.

Lemma 2 (Inversion Successor) *If* $\Gamma \vdash \mathtt{s}(e) : \tau$ *then* $\tau = \mathtt{nat}$ *and* $\Gamma \vdash e : \mathtt{nat}$.

Another property we can prove is that types are unique in a given context. This property is not a requirement for a programming language but desirable.

Lemma 3 *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e : \tau'$ *then* $\tau = \tau'$.

We can prove the lemma by induction on $\Gamma \vdash e : \tau$ and by applying inversion to $\Gamma \vdash e : \tau$

The type system also enjoys structural properties like *weakening* and *contraction*. Weakening states that we can add variables to our context that are not used without hampering a type derivation. Contraction states that we can use variables in the context as often as we want in an expression. Later in this curse, we will study *substructural* type systems that do not enjoy some (or all) of these structural properties.

Lemma 4 (Weakening) *If* $\Gamma \vdash e : \tau$ *then* $\Gamma, x : \tau' \vdash e : \tau$.

Lemma 5 (Contraction) *If we have* $\Gamma, x_1 : \tau', x_2 : \tau' \vdash e : \tau$ *then* $\Gamma, x : \tau' \vdash [x, x/x_1, x_2]e : \tau$.

The proof of both lemmas proceeds by induction on the type judgment on the left-hand side of the implication.

We can prove inversion lemmas. For example, we show the one for the successor.

Lemma 6 (Inversion Successor) *If* $\Gamma \vdash \mathtt{s}(e) : \tau$ *then* $\tau = \mathtt{nat}$ *and* $\Gamma \vdash e : \mathtt{nat}$.

5 Dynamic Semantics

We now define the *dynamic semantics* (or just dynamics), which defines the result of evaluating a program. There are different ways in which we can define the dynamics. In this course, we focus on an operational approach. It is called operational because it is close to the implementation of an *interpreter*.

For System T we define a *structural dynamic semantics*. It is sometimes also called small-step operational semantics or just small-step semantics. The idea is to define a transition system, so that states are programs and transitions represent computational steps. Our goal is that programs either transition to another state or are final states (which we call *values*) that do not transition further. We want transitions to be deterministic as well.

When defining the structural dynamics, we have some degree of freedom. For instance, we can decide to evaluate function *by-name* or *by-value*. In the case of System T, this choice is inconsequential in sense that programs $e : \mathtt{nat}$ at base type evaluate to the same value in both versions.

Values The values, the final states in the transition system, are inductively defined by the judgment $v\ \mathsf{val}$. There are two kinds of values function abstractions and numerals $\overline{n}$.

$$\frac{}{\mathtt{lam}\{\tau\}(x.e)\ \mathsf{val}}\ V_{\mathrm{lam}} \qquad \frac{}{\mathtt{z}\ \mathsf{val}}\ V_{\mathrm{z}} \qquad \frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}}\ V_{\mathrm{s}}$$

Call-By-Value Transitions We inductively define the judgement $e \longmapsto e'$, which states that e steps to e' in one step. Multi-step evaluation $e \longmapsto^* e'$ is defined inductive by the following rules.

$$\frac{}{e \longmapsto^* e'}\ M_1 \qquad \frac{e \longmapsto^* e' \quad e' \longmapsto e''}{e \longmapsto^* e''}\ M_2$$

Figure 2 contains the rules for the step relation. Rule E_s states that to make a step in the evaluation $\mathtt{s}(e)$, we have to make a step in e. The rules E_{rec1}, E_{rec2}, and E_{rec3} specify how to evaluate the recursor $\mathtt{rec}\{e_0; x.y.e_1\}(e)$. We first evaluate the "argument" e that specifies the number of recursive steps. If e is already a value, we consider two cases. If $e = \mathtt{z}$ then we step to e_0. If $e = \mathtt{s}(e')$ then we evaluate e_1 with e' and $\mathtt{rec}\{e_0; x.y.e_1\}(e')$ substituted for the variables x and y. This is the self-reference that powers the recursive computation.

$$\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\ E_{\mathrm{s}} \qquad \frac{e \longmapsto e'}{\mathtt{rec}\{e_0; x.y.e_1\}(e) \longmapsto \mathtt{rec}\{e_0; x.y.e_1\}(e')}\ E_{\mathrm{rec1}}$$

$$\frac{}{\mathtt{rec}\{e_0; x.y.e_1\}(\mathtt{z}) \longmapsto e_0}\ E_{\mathrm{rec2}}$$

$$\frac{\mathtt{s}(e)\ \mathsf{val}}{\mathtt{rec}\{e_0; x.y.e_1\}(\mathtt{s}(e)) \longmapsto [e, \mathtt{rec}\{e_0; x.y.e_1\}(e)/x, y]e_1}\ E_{\mathrm{rec3}}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{app}(e_1, e_2) \longmapsto \mathtt{app}(e_1', e_2)}\ E_{\mathrm{ap1}} \qquad \frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{app}(e_1, e_2) \longmapsto \mathtt{app}(e_1, e_2')}\ E_{\mathrm{ap2}}$$

$$\frac{e_2\ \mathsf{val}}{\mathtt{app}((\lambda\,(x:\tau)\,e), e_2) \longmapsto [e_2/x]e}\ E_{\mathrm{ap3}}$$

Figure 2: Call-By-Value Step Relation

The function application $\mathtt{app}(e_1, e_2)$ is evaluated in call-by-value or eager evaluation order. Rule E_{ap1} specifies that we first evaluate the expression e_1. Rules E_{ap2} and Rule E_{ap3} ensure that we first evaluate e_2 to a avlue bevore we perform the substitution of the argument.

In an eager language, variables stand for values. This should be reflected by the structural dynamic semantics that only substitutes values. The rules we present are therefore not truly eager since we substitute the recursor for y in the rule E_{rec3}.

Call-By-Name Transitions If we use the call-by-name (or lazy) evaluation order then we replace the rules E_{ap2} and E_{ap3} with the following rule E_{ap4}.

$$\frac{}{\mathtt{app}((\lambda\,(x:\tau)\,e), e_2) \longmapsto [e_2/x]e}\ E_{\mathrm{ap4}}$$

This rule is similar to E_{ap3} but without the premise e_2 val.

Call-by-name function evaluation is only suitable for a lazy language in which variables stand for general expression.

Exercises

Exercise 1 *Provide a type τ and a derivation tree for the judgment*

$$\lambda\,(n:\mathtt{nat})\,\lambda\,(m:\mathtt{nat})\,\mathtt{rec}\,n\,\{\mathtt{z} \hookrightarrow m \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow \mathtt{s}(y)\}\tau\ .$$

Exercise 2 *Prove Lemma 1.*

Exercise 3 *Prove Lemma 3.*

Exercise 4 *Let*

$$\mathit{add} \triangleq \lambda\,(n : \mathtt{nat})\,\lambda\,(m : \mathtt{nat})\,\mathtt{rec}\,n\,\{\mathtt{z} \hookrightarrow m \mid \mathtt{s}(x)\,\mathtt{with}\,y \hookrightarrow \mathtt{s}(y)\}\tau\,.$$

Provide a derivation of the judgment

$$\mathit{add}(\overline{2})(\overline{2}) \longmapsto^* \overline{4}\,.$$

References

[Göd80] Kurt Gödel. On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosophical Logic*, 9:133–142, 1980.

[Raa20] Panu Raatikainen. Gödel's incompleteness theorems. *Stanford Encyclopedia of Philosophy*, 2020.