9 System T of Higher-Order Recursion

System **T**, well-known as *Gödel's T*, is the combination of function types with the type of natural numbers. In contrast to **E**, which equips the naturals with some arbitrarily chosen arithmetic operations, the language **T** provides a general mechanism, called *primitive recursion*, from which these primitives may be defined. Primitive recursion captures the essential inductive character of the natural numbers, and hence may be seen as an intrinsic termination proof for each program in the language. Consequently, we may only define *total* functions in the language, those that always return a value for each argument. In essence, every program in **T** "comes equipped" with a proof of its termination. Although this may seem like a shield against infinite loops, it is also a weapon that can be used to show that some programs cannot be written in **T**. To do so would demand a master termination proof for every possible program in the language, something that we shall prove does not exist.

9.1 Statics

The syntax of **T** is given by the following grammar:

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

We write $\overline{n}$ for the expression $\mathtt{s}(\ldots\mathtt{s}(\mathtt{z}))$, in which the successor is applied $n \geq 0$ times to zero. The expression $\mathtt{rec}\{e_0; x.y.e_1\}(e)$ is called the *recursor*. It represents the e-fold iteration of the transformation $x.y.e_1$ starting from e_0. The bound variable x represents the predecessor and the bound variable y represents the result of the x-fold iteration. The "`with`" clause in the concrete syntax for the recursor binds the variable y to the result of the recursive call, as will become clear shortly.

Sometimes the *iterator*, $\mathtt{iter}\{e_0; y.e_1\}(e)$, is considered as an alternative to the recursor. It has essentially the same meaning as the recursor, except that only the result of the recursive

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

call is bound to y in e_1, and no binding is made for the predecessor. Clearly, the iterator is a special case of the recursor, because we can always ignore the predecessor binding. Conversely, the recursor is definable from the iterator, provided that we have product types (Chapter 10) at our disposal. To define the recursor from the iterator, we simultaneously compute the predecessor while iterating the specified computation.

The statics of **T** is given by the following typing rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{9.1a}$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}} \tag{9.1b}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}} \tag{9.1c}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}\{e_0; x.y.e_1\}(e) : \tau} \tag{9.1d}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{lam}\{\tau_1\}(x.e) : \mathtt{arr}(\tau_1; \tau_2)} \tag{9.1e}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau} \tag{9.1f}$$

As usual, admissibility of the structural rule of substitution is crucially important.

Lemma 9.1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

9.2 Dynamics

The closed values of **T** are defined by the following rules:

$$\frac{}{\mathtt{z}\ \mathsf{val}} \tag{9.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{s}(e)\ \mathsf{val}} \tag{9.2b}$$

$$\frac{}{\mathtt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{9.2c}$$

The premise of rule (9.2b) is included for an *eager* interpretation of successor, and excluded for a *lazy* interpretation.

The transition rules for the dynamics of **T** are as follows:

$$\left[\frac{e \longmapsto e'}{\mathtt{s}(e) \longmapsto \mathtt{s}(e')}\right] \tag{9.3a}$$

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{9.3b}$$

$$\left[\frac{e_1 \text{ val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{9.3c}$$

$$\frac{[e_2 \text{ val}]}{\mathtt{ap}(\mathtt{lam}\{\tau\}(x.e); e_2) \longmapsto [e_2/x]e} \tag{9.3d}$$

$$\frac{e \longmapsto e'}{\mathtt{rec}\{e_0; x.y.e_1\}(e) \longmapsto \mathtt{rec}\{e_0; x.y.e_1\}(e')} \tag{9.3e}$$

$$\frac{}{\mathtt{rec}\{e_0; x.y.e_1\}(\mathtt{z}) \longmapsto e_0} \tag{9.3f}$$

$$\frac{\mathtt{s}(e) \text{ val}}{\mathtt{rec}\{e_0; x.y.e_1\}(\mathtt{s}(e)) \longmapsto [e, \mathtt{rec}\{e_0; x.y.e_1\}(e)/x, y]e_1} \tag{9.3g}$$

The bracketed rules and premises are included for an eager successor and call-by-value application, and omitted for a lazy successor and call-by-name application. Rules (9.3f) and (9.3g) specify the behavior of the recursor on $\mathtt{z}$ and $\mathtt{s}(e)$. In the former case, the recursor reduces to e_0, and in the latter case, the variable x is bound to the predecessor e and y is bound to the (unevaluated) recursion on e. If the value of y is not required in the rest of the computation, the recursive call is not evaluated.

Lemma 9.2 (Canonical Forms)**.** *If $e : \tau$ and e val, then*

1. *If $\tau = \mathtt{nat}$, then $e = \mathtt{s}(e')$ for some e'.*
2. *If $\tau = \tau_1 \to \tau_2$, then $e = \lambda\,(x : \tau_1)\, e_2$ for some e_2.*

Theorem 9.3 (Safety)**.** *1. If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*
2. If $e : \tau$, then either e val or $e \longmapsto e'$ for some e'.

9.3 Definability

A mathematical function $f : \mathbb{N} \to \mathbb{N}$ on the natural numbers is *definable* in **T** iff there exists an expression e_f of type $\mathtt{nat} \to \mathtt{nat}$ such that for every $n \in \mathbb{N}$,

$$e_f(\overline{n}) \equiv \overline{f(n)} : \mathtt{nat}. \tag{9.4}$$

That is, the numeric function $f : \mathbb{N} \to \mathbb{N}$ is definable iff there is an expression e_f of type $\mathtt{nat} \to \mathtt{nat}$ such that, when applied to the numeral representing the argument $n \in \mathbb{N}$, the application is definitionally equal to the numeral corresponding to $f(n) \in \mathbb{N}$.

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

Definitional equality for **T**, written $\Gamma \vdash e \equiv e' : \tau$, is the strongest congruence containing these axioms:

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{ap}(\mathtt{lam}\{\tau_1\}(x.e_2); e_1) \equiv [e_1/x]e_2 : \tau_2} \tag{9.5a}$$

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}\{e_0; x.y.e_1\}(\mathtt{z}) \equiv e_0 : \tau} \tag{9.5b}$$

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathtt{rec}\{e_0; x.y.e_1\}(\mathtt{s}(e)) \equiv [e, \mathtt{rec}\{e_0; x.y.e_1\}(e)/x, y]e_1 : \tau} \tag{9.5c}$$

For example, the doubling function, $d(n) = 2 \times n$, is definable in **T** by the expression $e_d : \mathtt{nat} \to \mathtt{nat}$ given by

$$\lambda\,(x : \mathtt{nat})\,\mathtt{rec}\,x\,\{\mathtt{z} \hookrightarrow \mathtt{z} \mid \mathtt{s}(u)\,\mathtt{with}\,v \hookrightarrow \mathtt{s}(\mathtt{s}(v))\}.$$

To check that this defines the doubling function, we proceed by induction on $n \in \mathbb{N}$. For the basis, it is easy to check that

$$e_d(\overline{0}) \equiv \overline{0} : \mathtt{nat}.$$

For the induction, assume that

$$e_d(\overline{n}) \equiv \overline{d(n)} : \mathtt{nat}.$$

Then calculate using the rules of definitional equality:

$$\begin{aligned} e_d(\overline{n+1}) &\equiv \mathtt{s}(\mathtt{s}(e_d(\overline{n}))) \\ &\equiv \mathtt{s}(\mathtt{s}(\overline{2 \times n})) \\ &= \overline{2 \times (n+1)} \\ &= \overline{d(n+1)}. \end{aligned}$$

As another example, consider the following function, called *Ackermann's function*, defined by the following equations:

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

The Ackermann function grows very quickly. For example, $A(4, 2) \approx 2^{65,536}$, which is often cited as being larger than the number of atoms in the universe! Yet we can show that the Ackermann function is total by a lexicographic induction on the pair of arguments (m, n). On each recursive call, either m decreases, or else m remains the same, and n decreases, so inductively the recursive calls are well-defined, and hence so is $A(m, n)$.

A *first-order primitive recursive function* is a function of type $\mathtt{nat} \to \mathtt{nat}$ that is defined using the recursor, but without using any higher-order functions. Ackermann's function is defined so that it is not first-order primitive recursive but is higher-order primitive recursive. The key to showing that it is definable in **T** is to note that $A(m+1, n)$ iterates n times

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

the function $A(m, -)$, starting with $A(m, 1)$. As an auxiliary, let us define the higher-order function

$$\mathtt{it} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the λ-abstraction

$$\lambda\,(f : \mathtt{nat} \to \mathtt{nat})\,\lambda\,(n : \mathtt{nat})\,\mathtt{rec}\,n\,\{\mathtt{z} \hookrightarrow \mathtt{id} \mid \mathtt{s}(\_)\,\mathtt{with}\,g \hookrightarrow f \circ g\},$$

where $\mathtt{id} = \lambda\,(x : \mathtt{nat})\,x$ is the identity, and $f \circ g = \lambda\,(x : \mathtt{nat})\,f(g(x))$ is the composition of f and g. It is easy to check that

$$\mathtt{it}(f)(\overline{n})(\overline{m}) \equiv f^{(n)}(\overline{m}) : \mathtt{nat},$$

where the latter expression is the n-fold composition of f starting with $\overline{m}$. We may then define the Ackermann function

$$e_a : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

to be the expression

$$\lambda\,(m : \mathtt{nat})\,\mathtt{rec}\,m\,\{\mathtt{z} \hookrightarrow \mathtt{s} \mid \mathtt{s}(\_)\,\mathtt{with}\,f \hookrightarrow \lambda\,(n : \mathtt{nat})\,\mathtt{it}(f)(n)(f(\overline{1}))\}.$$

It is instructive to check that the following equivalences are valid:

$$e_a(\overline{0})(\overline{n}) \equiv \mathtt{s}(\overline{n}) \tag{9.6}$$

$$e_a(\overline{m+1})(\overline{0}) \equiv e_a(\overline{m})(\overline{1}) \tag{9.7}$$

$$e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\overline{m})(e_a(\mathtt{s}(\overline{m}))(\overline{n})). \tag{9.8}$$

That is, the Ackermann function is definable in **T**.

9.4 Undefinability

It is impossible to define an infinite loop in **T**.

Theorem 9.4. *If $e : \tau$, then there exists v val such that $e \equiv v : \tau$.*

Proof See Corollary 46.15. □

Consequently, values of function type in **T** behave like mathematical functions: if $e : \tau_1 \to \tau_2$ and $e_1 : \tau_1$, then $e(e_1)$ evaluates to a value of type τ_2. Moreover, if $e : \mathtt{nat}$, then there exists a natural number n such that $e \equiv \overline{n} : \mathtt{nat}$.

Using this, we can show, using a technique called *diagonalization*, that there are functions on the natural numbers that are not definable in **T**. We make use of a technique, called *Gödel-numbering*, that assigns a unique natural number to each closed expression of **T**. By assigning a unique number to each expression, we may manipulate expressions as data values in **T** so that **T** is able to compute with its own programs.[1]

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

The essence of Gödel-numbering is captured by the following simple construction on abstract syntax trees. (The generalization to abstract binding trees is slightly more difficult, the main complication being to ensure that all α-equivalent expressions are assigned the same Gödel number.) Recall that a general ast a has the form $o(a_1, \ldots, a_k)$, where o is an operator of arity k. Enumerate the operators so that every operator has an index $i \in \mathbb{N}$, and let m be the index of o in this enumeration. Define the *Gödel number* $\ulcorner a \urcorner$ of a to be the number

$$2^m \, 3^{n_1} \, 5^{n_2} \, \cdots \, p_k^{n_k},$$

where p_k is the kth prime number (so that $p_0 = 2$, $p_1 = 3$, and so on), and $n_1, \ldots, n_k$ are the Gödel numbers of $a_1, \ldots, a_k$, respectively. This procedure assigns a natural number to each ast. Conversely, given a natural number, n, we may apply the prime factorization theorem to "parse" n as a unique abstract syntax tree. (If the factorization is not of the right form, which can only be because the arity of the operator does not match the number of factors, then n does not code any ast.)

Now, using this representation, we may define a (mathematical) function $f_{univ} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that, for any $e : \mathtt{nat} \to \mathtt{nat}$, $f_{univ}(\ulcorner e \urcorner)(m) = n$ iff $e(\overline{m}) \equiv \overline{n} : \mathtt{nat}$.[2] The determinacy of the dynamics, together with Theorem 9.4, ensure that f_{univ} is a well-defined function. It is called the *universal function* for **T** because it specifies the behavior of any expression e of type $\mathtt{nat} \to \mathtt{nat}$. Using the universal function, let us define an auxiliary mathematical function, called the *diagonal function* $\delta : \mathbb{N} \to \mathbb{N}$, by the equation $\delta(m) = f_{univ}(m)(m)$. The δ function is chosen so that $\delta(\ulcorner e \urcorner) = n$ iff $e(\overline{\ulcorner e \urcorner}) \equiv \overline{n} : \mathtt{nat}$. (The motivation for its definition will become clear in a moment.)

The function f_{univ} is not definable in **T**. Suppose that it were definable by the expression e_{univ}, then the diagonal function δ would be definable by the expression

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

But in that case we would have the equations

$$\begin{aligned} e_\delta(\overline{\ulcorner e \urcorner}) &\equiv e_{univ}(\overline{\ulcorner e \urcorner})(\overline{\ulcorner e \urcorner}) \\ &\equiv e(\overline{\ulcorner e \urcorner}). \end{aligned}$$

Now let e_Δ be the function expression

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

so that we may deduce

$$\begin{aligned} e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) &\equiv \mathtt{s}(e_\delta(\overline{\ulcorner e_\Delta \urcorner})) \\ &\equiv \mathtt{s}(e_\Delta(\overline{\ulcorner e_\Delta \urcorner})). \end{aligned}$$

But the termination theorem implies that there exists n such that $e_\Delta(\overline{\ulcorner e_\Delta \urcorner}) \equiv \overline{n}$, and hence we have $\overline{n} \equiv \mathtt{s}(\overline{n})$, which is impossible.

We say that a language $\mathcal{L}$ is *universal* if it is possible to write an interpreter for $\mathcal{L}$ in $\mathcal{L}$ itself. It is intuitively clear that f_{univ} is computable in the sense that we can define it in

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

some sufficiently powerful programming language. But the preceding argument shows that **T** is not up to the task; it is not a universal programming language. Examination of the foregoing proof reveals an inescapable trade-off: by insisting that all expressions terminate, we necessarily lose universality—there are computable functions that are not definable in the language.

9.5 Notes

System **T** was introduced by Gödel (1958) in his study of the consistency of arithmetic (Gödel, 1980). Gödel showed how to "compile" proofs in arithmetic into well-typed terms of system **T**, and to reduce the consistency problem for arithmetic to the termination of programs in **T**. It was perhaps the first programming language whose design was directly influenced by the verification (of termination) of its programs.

Exercises

9.1. Prove Lemma 9.2.

9.2. Prove Theorem 9.3.

9.3. Attempt to prove that if $e : \mathtt{nat}$ is closed, then there exists n such that $e \longmapsto^* \overline{n}$ under the eager dynamics. Where does the proof break down?

9.4. Attempt to prove termination for all well-typed closed terms: if $e : \tau$, then there exists e' val such that $e \longmapsto^* e'$. You are free to appeal to Lemma 9.2 and Theorem 9.3 as necessary. Where does the attempt break down? Can you think of a stronger inductive hypothesis that might evade the difficulty?

9.5. Define a closed term e of type τ in **T** to be *hereditarily terminating at type* τ by induction on the structure of τ as follows:

(a) If $\tau = \mathtt{nat}$, then e is hereditarily terminating at type τ iff e is terminating (that is, iff $e \longmapsto^* \overline{n}$ for some n.)

(b) If $\tau = \tau_1 \rightarrow \tau_2$, then e is hereditarily terminating iff when e_1 is hereditarily terminating at type τ_1, then $e(e_1)$ is hereditarily terminating at type τ_2.

Attempt to prove hereditary termination for well-typed terms: if $e : \tau$, then e is hereditarily terminating at type τ. The stronger inductive hypothesis bypasses the difficulty that arose in Exercise **9.4** but introduces another obstacle. What is the complication? Can you think of an even stronger inductive hypothesis that would suffice for the proof?

9.6. Show that if e is hereditarily terminating at type τ, $e' : \tau$, and $e' \longmapsto e$, then e is also hereditarily terminating at type τ. (The need for this result will become clear in the solution to Exercise **9.5**.)

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press

9.7. Define an open, well-typed term

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$$

to be *open hereditarily terminating* iff every substitution instance

$$[e_1, \ldots, e_n/x_1, \ldots, x_n]e$$

is closed hereditarily terminating at type τ when each e_i is closed hereditarily terminating at type τ_i for each $1 \leq i \leq n$. Derive Exercise **9.3** from this result.

Notes

1 The same technique lies at the heart of the proof of Gödel's celebrated incompleteness theorem. The undefinability of certain functions on the natural numbers within **T** may be seen as a form of incompleteness like that considered by Gödel.

2 The value of $f_{univ}(k)(m)$ may be chosen arbitrarily to be zero when k is not the code of any expression e.

https://doi.org/10.1017/CBO9781316576892.011 Published online by Cambridge University Press