CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Total Functions; Finite Data Types

March 6, 2025

A Simple Expression Language E

Syntax of E defined as Abstract Binding Trees:

Typ	τ	$::=$	$\texttt{num}$	$\texttt{num}$	numbers
			$\texttt{str}$	$\texttt{str}$	strings
Exp	e	$::=$	x	x	variable
			$\texttt{num}[n]$	n	numeral
			$\texttt{str}[s]$	$"s"$	literal
			$\texttt{plus}(e_1; e_2)$	$e_1 + e_2$	addition
			$\texttt{times}(e_1; e_2)$	$e_1 * e_2$	multiplication
			$\texttt{cat}(e_1; e_2)$	$e_1 \texttt{\^{}} e_2$	concatenation
			$\texttt{len}(e)$	$\lvert e \rvert$	length
			$\texttt{let}(e_1; x.e_2)$	$\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$	definition

Statics (Type System) for E

$\vec{x} \mid \Gamma \vdash e : \tau,$

An inductive definition of generic hypothetical judgments

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad (4.1a)$$

$$\frac{}{\Gamma \vdash \mathtt{str}[s] : \mathtt{str}} \quad (4.1b)$$

$$\frac{}{\Gamma \vdash \mathtt{num}[n] : \mathtt{num}} \quad (4.1c)$$

$$\frac{\Gamma \vdash e_1 : \mathtt{num} \quad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash \mathtt{plus}(e_1; e_2) : \mathtt{num}} \quad (4.1d)$$

$$\frac{\Gamma \vdash e_1 : \mathtt{num} \quad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash \mathtt{times}(e_1; e_2) : \mathtt{num}} \quad (4.1e)$$

$$\frac{\Gamma \vdash e_1 : \mathtt{str} \quad \Gamma \vdash e_2 : \mathtt{str}}{\Gamma \vdash \mathtt{cat}(e_1; e_2) : \mathtt{str}} \quad (4.1f)$$

$$\frac{\Gamma \vdash e : \mathtt{str}}{\Gamma \vdash \mathtt{len}(e) : \mathtt{num}} \quad (4.1g)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(e_1; x.e_2) : \tau_2} \quad (4.1h)$$

Structural Dynamics for E

A *structural dynamics* for the language **E** is given by a transition system whose states are closed expressions. All states are initial. The final states are the *(closed) values*, which represent the completed computations. The judgment e val, which states that e is a value, is inductively defined by the following rules:

$$\frac{}{\mathtt{num}[n]\ \mathsf{val}} \tag{5.3a}$$

$$\frac{}{\mathtt{str}[s]\ \mathsf{val}} \tag{5.3b}$$

The transition judgment $e \longmapsto e'$ between states is inductively defined by the following rules:

$$\frac{n_1 + n_2 = n}{\mathtt{plus}(\mathtt{num}[n_1];\mathtt{num}[n_2]) \longmapsto \mathtt{num}[n]} \tag{5.4a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{plus}(e_1;e_2) \longmapsto \mathtt{plus}(e_1';e_2)} \tag{5.4b}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{plus}(e_1;e_2) \longmapsto \mathtt{plus}(e_1;e_2')} \tag{5.4c}$$

Structural Dynamics for E

$$\frac{s_1 \hat{} s_2 = s \ \mathsf{str}}{\mathtt{cat}(\mathtt{str}[s_1]; \mathtt{str}[s_2]) \longmapsto \mathtt{str}[s]} \tag{5.4d}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{cat}(e_1; e_2) \longmapsto \mathtt{cat}(e_1'; e_2)} \tag{5.4e}$$

$$\frac{e_1 \ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{cat}(e_1; e_2) \longmapsto \mathtt{cat}(e_1; e_2')} \tag{5.4f}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\mathtt{let}(e_1; x.e_2) \longmapsto \mathtt{let}(e_1'; x.e_2)}\right] \tag{5.4g}$$

$$\frac{[e_1 \ \mathsf{val}]}{\mathtt{let}(e_1; x.e_2) \longmapsto [e_1/x]e_2} \tag{5.4h}$$

EF : E + Higher-Order Functions

The language **EF** enriches **E** with function types, as specified by the following grammar:

Typ	τ	::=	$\mathtt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
Exp	e	::=	$\mathtt{lam}\{\tau\}(x.e)$	$\lambda\,(x:\tau)\,e$	abstraction
			$\mathtt{ap}(e_1;e_2)$	$e_1(e_2)$	application

The statics of **EF** is given by extending rules (4.1) with the following rules:

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathtt{lam}\{\tau_1\}(x.e):\mathtt{arr}(\tau_1;\tau_2)} \tag{8.4a}$$

$$\frac{\Gamma \vdash e_1:\mathtt{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2):\tau} \tag{8.4b}$$

EF : E + Higher-Order Functions

Lemma 8.2 (Inversion). *Suppose that* $\Gamma \vdash e : \tau$.

1. *If* $e = \mathtt{lam}\{\tau_1\}(x.e_2)$, *then* $\tau = \mathtt{arr}(\tau_1; \tau_2)$ *and* $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.
2. *If* $e = \mathtt{ap}(e_1; e_2)$, *then there exists* τ_2 *such that* $\Gamma \vdash e_1 : \mathtt{arr}(\tau_2; \tau)$ *and* $\Gamma \vdash e_2 : \tau_2$.

Proof The proof proceeds by rule induction on the typing rules. Observe that for each rule, exactly one case applies and that the premises of the rule provide the required result. □

Lemma 8.3 (Substitution). *If* $\Gamma, x : \tau \vdash e' : \tau'$, *and* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash [e/x]e' : \tau'$.

Proof By rule induction on the derivation of the first judgment. □

EF : E + Higher-Order Functions

The dynamics of **EF** extends that of **E** with the following rules:

$$\frac{}{\texttt{lam}\{\tau\}(x.e)\ \mathsf{val}} \tag{8.5a}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)} \tag{8.5b}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1; e_2')}\right] \tag{8.5c}$$

$$\frac{[e_2\ \mathsf{val}]}{\mathtt{ap}(\texttt{lam}\{\tau_2\}(x.e_1); e_2) \longmapsto [e_2/x]e_1} \tag{8.5d}$$

EF Preservation

Theorem 8.4 (Preservation)**.** *If* $e : \tau$ *and* $e \longmapsto e'$*, then* $e' : \tau$*.*

Proof The proof is by induction on rules (8.5), which define the dynamics of the language.

Consider rule (8.5d),

$$\frac{}{\mathtt{ap}(\mathtt{lam}\{\tau_2\}(x.e_1); e_2) \longmapsto [e_2/x]e_1} \ .$$

Suppose that $\mathtt{ap}(\mathtt{lam}\{\tau_2\}(x.e_1); e_2) : \tau_1$. By Lemma 8.2, we have $e_2 : \tau_2$ and $x : \tau_2 \vdash e_1 : \tau_1$, so by Lemma 8.3, $[e_2/x]e_1 : \tau_1$.

The other rules governing application are handled similarly. □

EF Progress

Lemma 8.5 (Canonical Forms)**.** *If* $e : \mathtt{arr}(\tau_1; \tau_2)$ *and* e val*, then* $e = \lambda\,(x : \tau_1)\, e_2$ *for some variable* x *and expression* e_2 *such that* $x : \tau_1 \vdash e_2 : \tau_2$.

Proof By induction on the typing rules, using the assumption e val. □

Theorem 8.6 (Progress)**.** *If* $e : \tau$*, then either* e val*, or there exists* e' *such that* $e \longmapsto e'$.

Proof The proof is by induction on rules (8.4). Note that because we consider only closed terms, there are no hypotheses on typing derivations.

Consider rule (8.4b) (under the by-name interpretation). By induction either e_1 val or $e_1 \longmapsto e_1'$. In the latter case, we have $\mathtt{ap}(e_1; e_2) \longmapsto \mathtt{ap}(e_1'; e_2)$. In the former case, we have by Lemma 8.5 that $e_1 = \mathtt{lam}\{\tau_2\}(x.e)$ for some x and e. But then $\mathtt{ap}(e_1; e_2) \longmapsto [e_2/x]e$. □

EF Evaluation Dynamics

An inductive definition of the evaluation judgment $e \Downarrow v$ for **EF** is given by the following rules:

$$\frac{}{\mathtt{lam}\{\tau\}(x.e) \Downarrow \mathtt{lam}\{\tau\}(x.e)} \tag{8.6a}$$

$$\frac{e_1 \Downarrow \mathtt{lam}\{\tau\}(x.e) \quad [e_2/x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v} \tag{8.6b}$$

It is easy to check that if $e \Downarrow v$, then v val, and that if e val, then $e \Downarrow e$.

Theorem 8.7. *$e \Downarrow v$ iff $e \longmapsto^* v$ and v val.*

Proof In the forward direction, we proceed by rule induction on rules (8.6), following along similar lines as the proof of Theorem 7.2.

In the reverse direction, we proceed by rule induction on rules (5.1). The proof relies on an analog of Lemma 7.4, which states that evaluation is closed under converse execution, which is proved by induction on rules (8.5). □

System T

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

System T Statics

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* $\Gamma \vdash [e/x]e' : \tau'$.

System T Dynamics

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

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

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

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

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

System T Dynamics

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

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

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

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

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

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

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

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

System T Safety

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

1. *If $\tau = \mathtt{nat}$, then $e = \mathsf{s}(e')$ 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'.

System T 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 $f(n) \in \mathbb{N}$.

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

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

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

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

System T Definability

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:

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

System T Definability

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

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n)).$$

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 $A(m, n)$.

The key to showing that it is definable in **T** is to note that $A(m + 1, n)$ iterates n times the function $A(m, -)$, starting with $A(m, 1)$.

System T Definability

Let's define $\mathtt{it} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$

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:

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

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

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

System T 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]

System T Undefinability

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

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

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.)

System T Undefinability

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

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

System T Undefinability

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.

Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

Typ	τ	$::=$	$\mathtt{unit}$	$\mathtt{unit}$	nullary product
			$\mathtt{prod}(\tau_1; \tau_2)$	$\tau_1 \times \tau_2$	binary product
Exp	e	$::=$	$\mathtt{triv}$	$\langle\rangle$	null tuple
			$\mathtt{pair}(e_1; e_2)$	$\langle e_1, e_2\rangle$	ordered pair
			$\mathtt{pr[l]}(e)$	$e \cdot \mathtt{l}$	left projection
			$\mathtt{pr[r]}(e)$	$e \cdot \mathtt{r}$	right projection

The statics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}} \tag{10.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \tag{10.1b}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{l} : \tau_1} \tag{10.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2} \tag{10.1d}$$

Nullary and Binary Products

The dynamics of product types is defined by the following rules:

$$\frac{}{\langle\rangle\ \mathsf{val}} \tag{10.2a}$$

$$\frac{[e_1\ \mathsf{val}] \quad [e_2\ \mathsf{val}]}{\langle e_1, e_2\rangle\ \mathsf{val}} \tag{10.2b}$$

$$\left[\frac{e_1 \longmapsto e_1'}{\langle e_1, e_2\rangle \longmapsto \langle e_1', e_2\rangle}\right] \tag{10.2c}$$

$$\left[\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\langle e_1, e_2\rangle \longmapsto \langle e_1, e_2'\rangle}\right] \tag{10.2d}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{l} \longmapsto e' \cdot \mathtt{l}} \tag{10.2e}$$

$$\frac{e \longmapsto e'}{e \cdot \mathtt{r} \longmapsto e' \cdot \mathtt{r}} \tag{10.2f}$$

$$\frac{[e_1\ \mathsf{val}] \quad [e_2\ \mathsf{val}]}{\langle e_1, e_2\rangle \cdot \mathtt{l} \longmapsto e_1} \tag{10.2g}$$

$$\frac{[e_1\ \mathsf{val}] \quad [e_2\ \mathsf{val}]}{\langle e_1, e_2\rangle \cdot \mathtt{r} \longmapsto e_2} \tag{10.2h}$$

Finite Products

The syntax of finite product types is given by the following grammar:

Typ	τ	$::=$	$\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in I})$	$\langle \tau_i \rangle_{i \in I}$	product
Exp	e	$::=$	$\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i \in I})$	$\langle e_i \rangle_{i \in I}$	tuple
			$\mathtt{pr}[i](e)$	$e \cdot i$	projection

The variable I stands for a finite *index set* over which products are formed. The type $\mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in I})$, or $\prod_{i \in I} \tau_i$ for short, is the type of I*-tuples* of expressions e_i of type τ_i, one for each $i \in I$. An I-tuple has the form $\mathtt{tpl}(\{i \hookrightarrow e_i\}_{i \in I})$, or $\langle e_i \rangle_{i \in I}$ for short, and for each $i \in I$ the ith projection from an I-tuple e is written $\mathtt{pr}[i](e)$, or $e \cdot i$ for short.

When $I = \{ i_1, \ldots, i_n \}$, the I-tuple type may be written in the form

$$\langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle$$

where we make explicit the association of a type to each index $i \in I$. Similarly, we may write

$$\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle$$

for the I-tuple whose ith component is e_i.

Finite Products

The statics of finite products is given by the following rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle} \tag{10.3a}$$

$$\frac{\Gamma \vdash e : \langle i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n \rangle \quad (1 \leq k \leq n)}{\Gamma \vdash e \cdot i_k : \tau_k} \tag{10.3b}$$

In rule (10.3b), the index $i_k \in I$ is a *particular* element of the index set I, whereas in rule (10.3a), the indices $i_1, \ldots, i_n$ range over the entire index set I.

The dynamics of finite products is given by the following rules:

$$\frac{[e_1 \ \mathsf{val} \quad \ldots \quad e_n \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \ \mathsf{val}} \tag{10.4a}$$

$$\left[\frac{\left\{ \begin{array}{c} e_1 \ \mathsf{val} \quad \ldots \quad e_{j-1} \ \mathsf{val} \quad e'_1 = e_1 \quad \ldots \quad e'_{j-1} = e_{j-1} \\ e_j \longmapsto e'_j \quad e'_{j+1} = e_{j+1} \quad \ldots \quad e'_n = e_n \end{array} \right\}}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \longmapsto \langle i_1 \hookrightarrow e'_1, \ldots, i_n \hookrightarrow e'_n \rangle} \right] \tag{10.4b}$$

$$\frac{e \longmapsto e'}{e \cdot i \longmapsto e' \cdot i} \tag{10.4c}$$

$$\frac{[\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \ \mathsf{val}]}{\langle i_1 \hookrightarrow e_1, \ldots, i_n \hookrightarrow e_n \rangle \cdot i_k \longmapsto e_k} \tag{10.4d}$$

Primitive Mutual Recursion

Using products we may simplify the primitive recursion construct of **T** so that only the recursive result on the predecessor, and not the predecessor itself, is passed to the successor branch. Writing this as $\mathtt{iter}\{e_0; x.e_1\}(e)$, we may define $\mathtt{rec}\{e_0; x.y.e_1\}(e)$ to be $e' \cdot \mathtt{r}$, where e' is the expression

$$\mathtt{iter}\{\langle \mathtt{z}, e_0\rangle; x'.\langle \mathtt{s}(x' \cdot \mathtt{l}), [x' \cdot \mathtt{r}/x]e_1\rangle\}(e).$$

The idea is to compute inductively both the number n and the result of the recursive call on n, from which we can compute both $n+1$ and the result of another recursion using e_1. The base case is computed directly as the pair of zero and e_0. It is easy to check that the statics and dynamics of the recursor are preserved by this definition.

Primitive Mutual Recursion

We may also use product types to implement *mutual primitive recursion*, in which we define two functions simultaneously by primitive recursion. For example, consider the following recursion equations defining two mathematical functions on the natural numbers:

$$e(0) = 1$$
$$o(0) = 0$$
$$e(n+1) = o(n)$$
$$o(n+1) = e(n)$$

Intuitively, $e(n)$ is non-zero if and only if n is even, and $o(n)$ is non-zero if and only if n is odd.

To define these functions in **T** enriched with products, we first define an auxiliary function $e_{\mathtt{eo}}$ of type

$$\mathtt{nat} \to (\mathtt{nat} \times \mathtt{nat})$$

that computes both results simultaneously by swapping back and forth on recursive calls:

$$\lambda\,(n : \mathtt{nat}\quad)\,\mathtt{iter}\,n\,\{\mathtt{z} \hookrightarrow \langle 1, 0\rangle \mid \mathtt{s}(b) \hookrightarrow \langle b \cdot \mathtt{r}, b \cdot \mathtt{l}\rangle\}.$$

We may then define $e_{\mathtt{ev}}$ and $e_{\mathtt{od}}$ as follows:

$$e_{\mathtt{ev}} \triangleq \lambda\,(n : \mathtt{nat})\, e_{\mathtt{eo}}(n) \cdot \mathtt{l}$$
$$e_{\mathtt{od}} \triangleq \lambda\,(n : \mathtt{nat})\, e_{\mathtt{eo}}(n) \cdot \mathtt{r}.$$

Nullary and Binary Sums

The abstract syntax of sums is given by the following grammar:

Typ	τ	$::=$	$\mathtt{void}$	$\mathtt{void}$	nullary sum
			$\mathtt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	binary sum
Exp	e	$::=$	$\mathtt{abort}\{\tau\}(e)$	$\mathtt{abort}(e)$	abort
			$\mathtt{in}[\mathtt{l}]\{\tau_1;\tau_2\}(e)$	$\mathtt{l}\cdot e$	left injection
			$\mathtt{in}[\mathtt{r}]\{\tau_1;\tau_2\}(e)$	$\mathtt{r}\cdot e$	right injection
			$\mathtt{case}(e;x_1.e_1;x_2.e_2)$	$\mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\}$	case analysis

The nullary sum represents a choice of zero alternatives, and hence admits no introduction form. The elimination form, $\mathtt{abort}(e)$, aborts the computation in the event that e evaluates to a value, which it cannot do. The elements of the binary sum type are labeled to show whether they are drawn from the left or the right summand, either $\mathtt{l}\cdot e$ or $\mathtt{r}\cdot e$. A value of the sum type is eliminated by case analysis.

The statics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort}(e) : \tau} \tag{11.1a}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{l}\cdot e : \tau_1 + \tau_2} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r}\cdot e : \tau_1 + \tau_2} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\} : \tau} \tag{11.1d}$$

Nullary and Binary Sums

The dynamics of sums is given by the following rules:

$$\frac{e \longmapsto e'}{\mathtt{abort}(e) \longmapsto \mathtt{abort}(e')} \tag{11.2a}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{l} \cdot e\ \mathsf{val}} \tag{11.2b}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{r} \cdot e\ \mathsf{val}} \tag{11.2c}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{l} \cdot e \longmapsto \mathtt{l} \cdot e'}\right] \tag{11.2d}$$

$$\left[\frac{e \longmapsto e'}{\mathtt{r} \cdot e \longmapsto \mathtt{r} \cdot e'}\right] \tag{11.2e}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto \mathtt{case}\, e'\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\}} \tag{11.2f}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{l} \cdot e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_1]e_1} \tag{11.2g}$$

$$\frac{[e\ \mathsf{val}]}{\mathtt{case}\, \mathtt{r} \cdot e\, \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_2]e_2} \tag{11.2h}$$

Finite Sums

Typ	τ	$::=$	$\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$	$[\tau_i]_{i \in I}$	sum
Exp	e	$::=$	$\mathtt{in}[i]\{\vec{\tau}\}(e)$	$i \cdot e$	injection
			$\mathtt{case}(e; \{i \hookrightarrow x_i.e_i\}_{i \in I})$	$\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I}$	case analysis

The variable I stands for a finite index set over which sums are formed. The notation $\vec{\tau}$ stands for a finite function $\{i \hookrightarrow \tau_i\}_{i \in I}$ for some index set I. The type $\mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in I})$, or $\sum_{i \in I} \tau_i$ for short, is the type of I-classified values of the form $\mathtt{in}[i]\{I\}(e_i)$, or $i \cdot e_i$ for short, where $i \in I$ and e_i is an expression of type τ_i. An I-classified value is analyzed by an I-way case analysis of the form $\mathtt{case}(e; \{i \hookrightarrow x_i.e_i\}_{i \in I})$.

When $I = \{i_1, \ldots, i_n\}$, the type of I-classified values may be written

$$[i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]$$

specifying the type associated with each class $l_i \in I$. Correspondingly, the I-way case analysis has the form

$$\mathtt{case}\, e\, \{i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n\}.$$

Finite Sums

The statics of finite sums is defined by the following rules:

$$\frac{\Gamma \vdash e : \tau_k \quad (1 \leq k \leq n)}{\Gamma \vdash i_k \cdot e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n]} \tag{11.3a}$$

$$\frac{\Gamma \vdash e : [i_1 \hookrightarrow \tau_1, \ldots, i_n \hookrightarrow \tau_n] \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \mathtt{case}\, e\, \{i_1 \cdot x_1 \hookrightarrow e_1 \mid \ldots \mid i_n \cdot x_n \hookrightarrow e_n\} : \tau} \tag{11.3b}$$

These rules generalize the statics for nullary and binary sums given in Section 11.1.

The dynamics of finite sums is defined by the following rules:

$$\frac{[e\ \mathsf{val}]}{i \cdot e\ \mathsf{val}} \tag{11.4a}$$

$$\left[\frac{e \longmapsto e'}{i \cdot e \longmapsto i \cdot e'}\right] \tag{11.4b}$$

$$\frac{e \longmapsto e'}{\mathtt{case}\, e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I} \longmapsto \mathtt{case}\, e'\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I}} \tag{11.4c}$$

$$\frac{i \cdot e\ \mathsf{val}}{\mathtt{case}\, i \cdot e\, \{i \cdot x_i \hookrightarrow e_i\}_{i \in I} \longmapsto [e/x_i]e_i} \tag{11.4d}$$

Application of Sum Types: Boolean

Typ	τ	$::=$	`bool`	`bool`	booleans
Exp	e	$::=$	`true`	`true`	truth
			`false`	`false`	falsity
			`if`$(e; e_1; e_2)$	`if` e `then` e_1 `else` e_2	conditional

The statics of Booleans is given by the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \tag{11.5a}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \tag{11.5b}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \tau} \tag{11.5c}$$

Application of Sum Types: Boolean

The dynamics is given by the following value and transition rules:

$$\frac{}{\texttt{true}\ \mathsf{val}} \tag{11.6a}$$

$$\frac{}{\texttt{false}\ \mathsf{val}} \tag{11.6b}$$

$$\frac{}{\texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longmapsto e_1} \tag{11.6c}$$

$$\frac{}{\texttt{if}\ \texttt{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longmapsto e_2} \tag{11.6d}$$

$$\frac{e \longmapsto e'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longmapsto \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \tag{11.6e}$$

The type $\texttt{bool}$ is definable in terms of binary sums and nullary products:

$$\texttt{bool} = \texttt{unit} + \texttt{unit} \tag{11.7a}$$

$$\texttt{true} = \texttt{l} \cdot \langle\rangle \tag{11.7b}$$

$$\texttt{false} = \texttt{r} \cdot \langle\rangle \tag{11.7c}$$

$$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 = \texttt{case}\ e\ \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.7d}$$

Application of Sum Types: Options

Another use of sums is to define the *option* types, which have the following syntax:

Typ	τ	$::=$	$\texttt{opt}(\tau)$	$\tau\,\texttt{opt}$	option
Exp	e	$::=$	$\texttt{null}$	$\texttt{null}$	nothing
			$\texttt{just}(e)$	$\texttt{just}(e)$	something
			$\texttt{ifnull}\{\tau\}\{e_1; x.e_2\}(e)$	$\texttt{which}\,e\,\{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(x) \hookrightarrow e_2\}$	
				null test	

The type $\texttt{opt}(\tau)$ represents the type of "optional" values of type τ. The introduction forms are $\texttt{null}$, corresponding to "no value," and $\texttt{just}(e)$, corresponding to a specified value of type τ. The elimination form discriminates between the two possibilities.

The option type is definable from sums and nullary products according to the following equations:[1]

$$\tau\,\texttt{opt} = \texttt{unit} + \tau \tag{11.8a}$$

$$\texttt{null} = \texttt{l} \cdot \langle\rangle \tag{11.8b}$$

$$\texttt{just}(e) = \texttt{r} \cdot e \tag{11.8c}$$

$$\texttt{which}\,e\,\{\texttt{null} \hookrightarrow e_1 \mid \texttt{just}(x_2) \hookrightarrow e_2\} = \texttt{case}\,e\,\{\texttt{l} \cdot \_ \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \tag{11.8d}$$