System T of Higher-Order Recursion

Yu Zhang

http://staff.ustc.edu.cn/~yuzhang/tpl/
yuzhang@ustc.edu.cn

Recap

- $E\ (L^{\text{num,str}})$
  - Terms: introduction and elimination forms of a type
  - Statics: introduction and elimination rules
  - Dynamics: values, structural small-step semantics
  - Type safety: progress and preservation
- Algebraic Data Types $L^{\times,+,\rightarrow}$
  - void (0), unit (1), bool (1+1), Maybe a (a+1), Either a or b (a+b), (a,b) (axb), a->b (b^a)
- Useful sum types: enumerate, option$_\tau$
  - Why using option$_\tau$?

Recap

- Reduction strategies $(\lambda u.\lambda v.v)((\lambda x.x\ x)(\lambda x.x\ x))$
  - Normal-order reduction: left-most, outer-most redex first
  - Applicative-order reduction: left-most, **inner-most** redex first

Recap

- Reduction strategies $(\lambda u.\lambda v.v)((\lambda x.x\ x)(\lambda x.x\ x))$
  - Normal-order reduction: left-most, outer-most redex first

    $(\lambda u.\lambda v.v)((\lambda x.x\ x)(\lambda x.x\ x))$
    $\rightarrow \lambda v.v$
  - Applicative-order reduction: left-most, **inner-most** redex first

    $(\lambda u.\lambda v.v)((\lambda x.x\ x)(\lambda x.x\ x))$
    $\rightarrow (\lambda u.\lambda v.v)((\lambda x.x\ x)(\lambda x.x\ x))$
    $\rightarrow \ldots$

Recap

- Reduction strategies similar to (but subtly different from) ***evaluation strategies*** in language theories
  - **Call-by-name** (like normal-order)
    - ALGOL 60
    - arguments are not evaluated, but directly substituted into function body
  - **Call-by-need** ("memorized version" of call-by-name)
    - Haskell, R, …
    - *called "lazy evaluation", avoids repeated evaluations*
  - **Call-by-value** (like applicative-order)
    - C, …
    - *called "eager evaluation"*
  - …

References

- PFPL
  - Chapters:
    - 8 Function Definitions and Values
    - 9 System T of Higher-order Recursion
- TAPL

System T: *Gödel's T*

- E: $L^{\text{num,str}}$; T: $L^{\text{nat},\to}$ (primitive recursion)
- Syntax

Typ	τ ::=	nat	nat	naturals 自然数类型
		$\text{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function 函数类型
Exp	e ::=	x	x	variable 变量
introduction		z	z	zero 零
		$\text{s}(e)$	$\text{s}(e)$	successor 后继
elimination		$\text{rec}\{e_0; x.y.e_1\}(e)$	$\text{rec}\, e\{\text{z} \hookrightarrow e_0 \mid \text{s}(x) \text{ with } y \hookrightarrow e_1\}$	recursion 递归式
		$\text{lam}\{\tau\}(x.e)$	$\lambda(x:\tau)\, e$	abstraction 抽象
		$\text{ap}(e_1;e_2)$	$e_1(e_2)$	application 应用

  - Recursor: $\text{rec}\{e_0; x.y.e_1\}(e)$
    - e-fold iteration of the transformation $x.y.e_1$ starting from e_0

      x : predecessor; y : result of the x-fold iteration; e_1: result of recursive call

System T

- Recursor vs. Iterator 迭代式 $\text{iter}\{e_0; y.e_1\}(e)$

  y: result of the recursive call, no binding for the predecessor
- Derivability vs. Admissibility
  - Derivability (可导性) $\Gamma \vdash_{\text{R}} \text{K}$
    - K is derivable from rules $\text{R} \cup \Gamma$
  - Admissibility (可纳性、可容许性) $\Gamma \models_{\text{R}} \text{J}$
    - $\vdash_{\text{R}} \Gamma$ implies $\vdash_{\text{R}} \text{J}$

      if any of the hypotheses Γ are *not* derivable relative to R, then J is *vacuously* true
  - If $\Gamma \vdash_{\text{R}} \text{J}$, then $\Gamma \models_{\text{R}} \text{J}$; but the converse fails

$\text{succ}(\text{zero})\ \text{even} \not\vdash_{(2.8)} \text{zero odd}$,

$\text{succ}(\text{zero})\ \text{even} \models_{(2.8)} \text{zero odd}$ valid, because the hypothesis is false

$\text{succ}(\text{succ}(\text{zero}))\ \text{odd}$

$$\frac{}{\text{zero even}} \qquad \frac{b\ \text{odd}}{\text{succ}(b)\ \text{even}} \qquad \frac{a\ \text{even}}{\text{succ}(a)\ \text{odd}} \qquad (2.8)$$

System T

- Statics of T

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

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

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \text{lam}\{\tau_1\}(x.e):\text{arr}(\tau_1;\tau_2)} \qquad \frac{\Gamma \vdash e_1:\text{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \text{ap}(e_1;e_2):\tau}$$

  - Substitution (Admissibility)

    If $\Gamma \vdash e:\tau$ and $\Gamma, x:\tau \vdash e':\tau'$, then $\Gamma \vdash [e/x]e':\tau'$
- Dynamics ([…] omitted for lazy dynamics)
  - Value $\quad \dfrac{}{z\ \text{val}} \qquad \dfrac{[e\ \text{val}]}{s(e)\ \text{val}} \qquad \dfrac{}{\text{lam}\{\tau\}(x.e)\ \text{val}}$

    Search transitions 决定指令执行次序 Instruction transitions 基础的求值步
  - Transition $\quad \left[\dfrac{e \mapsto e'}{s(e) \mapsto s(e')}\right]$

$$\frac{[e_2\ \text{val}]}{\text{ap}(\text{lam}\{\tau\}(x.e_2);e_2) \mapsto [e_2/x]e} \qquad \frac{e_1 \mapsto e_1'}{\text{ap}(e_1;e_2) \mapsto \text{ap}(e_1';e_2)} \qquad \left[\frac{e_1\ val \quad e_2 \mapsto e_2'}{\text{ap}(e_1;e_2) \mapsto \text{ap}(e_1;e_2')}\right]$$

System T

- Dynamics ([…] omitted for lazy dynamics)
  - Transition

$$\frac{e \mapsto e'}{\text{rec}\{e_0; x.y.e_1\}(e) \mapsto \text{rec}\{e_0; x.y.e_1\}(e')}$$

$$\frac{}{\text{rec}\{e_0; x.y.e_1\}(z) \mapsto e_0} \qquad \frac{s(e)\ val}{\text{rec}\{e_0; x.y.e_1\}(s(e)) \mapsto [e, \text{rec}\{e_0; x.y.e_1\}(e)/x, y]e_1}$$

- Canonical Forms(范式)
  - If $e:\tau$ and e val, then
    - If $\tau = nat$, then e=z or $e = s(e')$ for some e'
    - If $\tau = \tau_1 \to \tau_2$, then $e = \lambda(x:\tau_1)e_2$ for some e_2
- Safety
  - If $e:\tau$ and $e \mapsto e'$, then $e':\tau$
  - If $e:\tau$, then either e val or $e \mapsto e'$ for some e'

Examples for Recursion

- Doubling

```
datatype nat = z | s of nat

fun double z = z
  | double (s x) = s (s (double x))

fun double e =
  case e of
    z => z
  | s x => let val y = double x in s (s y) end

λ(e : nat) rec{ z; x.y.s (s (y)) } (e)
```

double s(z) ↦ rec{z; x.y.s(s(y))}{s(z)} Application

↦ [rec{z; x.y.s(s(y))}{z}/y] s(s(y)) Since s(z) val

= s(s(rec{z; x.y.s(s(y))}{z})) Substitution

↦ s(s(z)) By evaluation rule of s()

Expressiveness of System T

- Function $f:\mathbb{N} \to \mathbb{N}$ is **definable** in T, iff

  there exists $e_f:\text{nat} \to \text{nat}$ such that for all $n \in \mathbb{N}$, $e_f(\bar{n}) \equiv \overline{f(n)}:\text{nat}$
- Definitional equality for **T**, $\Gamma \vdash e \equiv e':\tau$

$$\frac{\Gamma, x:\tau_1 \vdash e_2:\tau_2 \quad \Gamma \vdash e_1:\tau_1}{\Gamma \vdash \text{ap}(\text{lam}\{\tau_1\}(x.e_2);e_1) \equiv [e_1/x]e_2:\tau_2} \qquad \frac{\Gamma \vdash e_0:\tau \quad \Gamma, x:\text{nat}, y:\tau \vdash e_1:\tau}{\Gamma \vdash \text{rec}\{e_0; x.y.e_1\}(z) \equiv e_0:\tau}$$

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

- Doubling function $d(n) = 2 \times n$ is definable in T by

$e_d:\text{nat} \to \text{nat} \qquad e_d = \lambda(x:\text{nat})\ \text{rec}\, x\{z \hookrightarrow z | s(u) \text{ with } v \hookrightarrow s(s(v))\}$

Proof: $e_d(\bar{0}) \equiv \bar{0}:\text{nat}$ (basis)

Assume $e_d(\bar{n}) \equiv \overline{d(n)}:\text{nat} \qquad e_d(\overline{n+1}) \equiv s(s(e_d(\bar{n}))) \equiv s(s(\overline{2 \times n})) = \overline{2 \times (n+1)} = \overline{d(n+1)}$

Ackermann

- Well-known Ackermann function

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

  - not first-order primitive recursive
  - but is higher-order primitive recursive

  is incompatible with the recursor construct $\mathrm{rec}\{e_0; x.y.e_1\}(e)$
    - its argument be *deconstructed* at every step
- Ackermann is definable in T
  - $A(m+1,n)$ iterates n times $A(m,-)$, starting with $A(m,1)$
  - Define it: $(\mathrm{nat} \longrightarrow \mathrm{nat}) \longrightarrow \mathrm{nat} \longrightarrow \mathrm{nat} \longrightarrow \mathrm{nat}$ as
    $\lambda(f\colon \mathrm{nat} \longrightarrow \mathrm{nat})\ \lambda\ (n\colon \mathrm{nat})\ \mathrm{rec}\ n\ \{\mathrm{z} \hookrightarrow \mathrm{id} | \mathrm{s}(x)\ \mathrm{with}\ g \hookrightarrow f \circ g\}$
    - $\mathrm{id} = \lambda(x\colon \mathrm{nat})x$, $f \circ g = \lambda(x\colon \mathrm{nat}) f(g(x))$
  - $\mathrm{it}(f)(\bar{n})(\bar{m}) \equiv f^{(n)}(\bar{m})\colon \mathrm{nat}$

Ackermann

- Ackermann is definable in T
  - Define it: $(\mathrm{nat} \longrightarrow \mathrm{nat}) \longrightarrow \mathrm{nat} \longrightarrow \mathrm{nat} \longrightarrow \mathrm{nat}$ as
    $\lambda(f\colon \mathrm{nat} \longrightarrow \mathrm{nat})\ \lambda\ (n\colon \mathrm{nat})\ \mathrm{rec}\ n\ \{\mathrm{z} \hookrightarrow \mathrm{id} | \mathrm{s}(x)\ \mathrm{with}\ g \hookrightarrow f \circ g\}$
    - $\mathrm{id} = \lambda(x\colon \mathrm{nat})x$, $f \circ g = \lambda(x\colon \mathrm{nat}) f(g(x))$
  - $\mathrm{it}(f)(\bar{n})(\bar{m}) \equiv f^{(n)}(\bar{m})\colon \mathrm{nat}$
  - Ackermann $e_a\colon \mathrm{nat} \longrightarrow \mathrm{nat} \longrightarrow \mathrm{nat}$ is
    $\lambda(m\colon \mathrm{nat})\ \mathrm{rec}\ m\ \{z \hookrightarrow s | s(x)\ \mathrm{with}\ f \hookrightarrow \lambda(n\colon \mathrm{nat})\ \mathrm{it}(f)(n)(f(\bar{1}))\}$
    - $e_a(\bar{0})(\bar{n}) \equiv s(\bar{0})$
    - $e_a(\overline{m+1})(\bar{0}) \equiv e_a(\bar{m})(\bar{1})$
    - $e_a(\overline{m+1})(\overline{n+1}) \equiv e_a(\bar{m})\big(e_a\big(s(\bar{m})\big)(\bar{n})\big)$

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

Primitive Mutual Recursion原始互递归

- Using products to simplify primitive recursors in T
  - System T: $\mathrm{rec}\{e_0; x.y.e_1\}(e)$
  - Using products: $\mathrm{iter}\{e_0; x.e_1\}(e)$,
    - x : the recursive result on the predecessor
    - define $\mathrm{rec}\{e_0; x.y.e_1\}(e)$ to be $e' \cdot \mathrm{r}$
      $e' \triangleq \mathrm{iter}\{\langle \mathrm{z}, e_0\rangle; x'.\langle \mathrm{s}(x'.\mathrm{l}), [x'.\mathrm{l}, x'.\mathrm{r}/x, y]e_1\rangle\}(e)$
- Mutual primitive recursion
  - $\lambda(n\colon \mathrm{nat})\mathrm{iter}\ n\{\mathrm{z} \hookrightarrow \langle 1,0\rangle | \mathrm{s}(b) \hookrightarrow \langle b.\mathrm{r}, b.\mathrm{l}\rangle\}$

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

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