Theory of Programming Languages
程序设计语言理论

张昱

Department of Computer Science and Technology
University of Science and Technology of China

October, 2008

第四章 简单类型

4.1 函数(Function)

λ抽象：函数定义

λx:nat.x+x，x是函数的参数，x+x 是函数体

λ应用：函数应用，左结合

(λx:nat.x+x) 2

4.1.1 函数类型-1

- 函数类型：$\sigma \to \tau$，σ是定义域(论域)，τ是值域
  - 例：d : nat→nat，如果e : nat,则$d\ e$: nat
  - →是右结合的，λ抽象体尽可能向右扩展

  在函数式语言中,函数是first-class对象(能参加计算,传递)
- α等价公理(约束变元改名公理)

  $\lambda x:\sigma.M = \lambda y:\sigma.[y/x]M$， M中无自由出现的y
- β等价公理(等式公理)

  $(\lambda x:\sigma.M)N = [N/x]M$

  对函数应用求值就是在函数体中用实在变元代替形式变元
- β归约

  $(\lambda x:\sigma.M)N \mapsto [N/x]M$

  归约是建立在α等价上的，因为在代换时约束变元可能需要改名

4.1.1 函数类型-2

- 同余(Congruence)规则

$$\frac{M_1 = M_2 \quad N_1 = N_2}{M_1 N_1 = M_2 N_2}$$

  相等的函数作用于相等的变元产生相等的结果
- 高阶函数类型：参数或函数值类型为函数类型
  - nat→(nat→nat) 即为 nat→nat→nat

    例：加法 plus = λx : nat. λy : nat. x+y

    plus 2 : nat→nat　　　plus 2 3 : nat (结果为5)
  - (nat→nat)→nat→nat

    例：对函数 f 执行两次 twicef = λf : nat→nat. λn : nat.f (f n)

    ~~twicef plus~~　~~twicef plus 2~~　plus的类型为nat→nat→nat

    twicef (plus 2): nat →nat

    twicef (plus 2) 3 : nat (结果为7)

4.1.1 函数类型-3

- 高阶函数类型：参数或函数值类型为函数类型
  - (nat→nat)→nat→nat 即为 (nat→nat)→(nat→nat)

    例：对函数 f 执行两次 twicef = λf : nat→nat. λn : nat.f (f n)

    例：恒等函数 identf = λf : nat→nat. f

    ~~identf plus~~　~~identf plus 2~~　plus的类型为nat→nat→nat

    identf (plus 2): nat →nat

    identf (plus 2) 3 : nat (结果为5)
  - nat→(nat →nat) →nat

    例：对函数 f 执行两次 twicef1 = λn : nat. λf : nat→nat. f (f n)

    twicef1 3 : (nat →nat) →nat

    twicef1 3 (plus 2): nat (结果为7)

4.1.2 语言L{→}的语法

- 抽象语法
  - Types　$\tau ::= \mathrm{arr}(\tau_1;\tau_2)$
  - Expr's　$e ::= x \mid \mathrm{lam}[\tau](x.e) \mid \mathrm{ap}(e_1;e_2)$

x : λ抽象的形式变元(形参)
e : λ抽象的体
e_1 : 函数，e_2 : 函数实参

- 抽象语法　vs. 具体语法 (PFPL中的表示法)
  - $\mathrm{arr}(\tau_1;\tau_2)$　$\tau_1 \to \tau_2$
  - $\mathrm{lam}[\tau](x.e)$　$\lambda(x:\tau.e)$　从$e_0:\tau$映射到$[e_0/x]e_1$的函数
  - $\mathrm{ap}(e_1;e_2)$　$e_1(e_2)$　$[e_2/x]e_1$
- 定义 $\mathrm{let}[\tau](e_2;x.e_1)$ 代表$\mathrm{ap}(\mathrm{lam}[\tau](x.e_1);e_2)$

例：$(\lambda x{:}\mathrm{nat}.x+x)\ 2$

具体语法 $\lambda(x{:}\mathrm{nat}.x+x)(2)$

抽象语法 $\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{num}[2])$

$\mathrm{let}[\mathrm{nat}](\mathrm{num}[2];x.\mathrm{plus}(x;x))$

4.1.3 L{→}的静态语义-1

❖ 定型规则 (Γ: 定型上下文)

- 变元　$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\quad (14.2a)$$
- λ抽象　$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathrm{lam}[\tau_1](x.e):\mathrm{arr}(\tau_1;\tau_2)}\quad (14.2b)$$
- λ应用　$$\frac{\Gamma \vdash e_1:\mathrm{arr}(\tau_2;\tau) \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \mathrm{ap}(e_1;e_2):\tau}\quad (14.2c)$$

❖ 引理4.1(定型的逆转)　假设 $\Gamma \vdash e:\tau$

1. 如果$e = x$，则 $\Gamma = \Gamma', x:\tau$
2. 如果$e = \mathrm{lam}[\tau_1](x.e)$, 则$\tau = \mathrm{arr}(\tau_1;\tau_2)$且 $\Gamma, x:\tau_1 \vdash e_2:\tau_2$
3. 如果$e = \mathrm{ap}(e_1;e_2)$, 则存在τ_2使得$\Gamma \vdash e_1:\mathrm{arr}(\tau_2,\tau)$
   且 $\Gamma \vdash e_2:\tau_2$

对逆转引理的证明可按定型规则进行归纳证明。

4.1.3 L{→}的静态语义-2

❖ 定型断言满足置换性质

- 引理14.2(置换) 如果$\Gamma, x:\tau \vdash e':\tau'$并且 $\Gamma \vdash e:\tau$，那么 $\Gamma \vdash [e/x]e':\tau'$

证明：对$\Gamma, x:\tau \vdash e':\tau'$的推导进行归纳，考虑(14.2)中的每一条规则，即e'的每一种可能，分别证明。

例：对$\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{num}[2])$进行类型检查

自底向上的类型检查

$$\frac{\Gamma, x:\mathrm{nat} \vdash \mathrm{plus}(x;x):\mathrm{nat}}{\Gamma \vdash \mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x)):\mathrm{arr}(\mathrm{nat};\mathrm{nat})}$$

$$\frac{\Gamma \vdash \mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x)):\mathrm{arr}(\mathrm{nat};\mathrm{nat}) \quad \Gamma \vdash \mathrm{num}[2]:\mathrm{nat}}{\Gamma \vdash \mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{num}[2]):\mathrm{nat}}$$

4.1.4 L{→}的动态语义-1

❖ 语言L{→}的动态语义

由在闭式上的结构操作语义来给出

- λ抽象是值　$$\frac{}{\mathrm{lam}[\tau](x.e)\ \mathsf{val}}$$

这里对函数体 e 的形式没有限制

函数的两种动态语义

- call-by-value(按值调用)语义：实参在通过置换传递到函数之前被求值。
- call-by-name(按名调用)语义：实参未经求值即传递到函数，对参数的求值将推迟到实际被需要的时候进行。

4.1.4 L{→}的动态语义-2

❖ call-by-value(按值调用)语义

$$\frac{e_1 \mapsto e_1'}{\mathrm{ap}(e_1;e_2) \mapsto \mathrm{ap}(e_1';e_2)} \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathrm{ap}(e_1;e_2) \mapsto \mathrm{ap}(e_1;e_2')}$$

$$\frac{e_2\ \mathsf{val}}{\mathrm{ap}(\mathrm{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1}\quad (14.4)$$

例 $\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x,x));\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]))$

$$\begin{aligned}
&\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]))\\
\mapsto\ &\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{num}[4])\\
\mapsto\ &\mathrm{plus}(\mathrm{num}[4];\mathrm{num}[4])\\
\mapsto\ &\mathrm{num}[8]
\end{aligned}$$

4.1.4 L{→}的动态语义-3

❖ call-by-name(按名调用)语义

$$\frac{e_1 \mapsto e_1'}{\mathrm{ap}(e_1;e_2) \mapsto \mathrm{ap}(e_1';e_2)} \qquad \frac{}{\mathrm{ap}(\mathrm{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1}\quad (14.5)$$

注意：未要求 e_2一定是值

例 $\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]))$

$$\begin{aligned}
&\mathrm{ap}(\mathrm{lam}[\mathrm{nat}](x.\mathrm{plus}(x;x));\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]))\\
\mapsto\ &\mathrm{plus}(\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]);\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]))\\
\mapsto\ &\mathrm{plus}(\mathrm{num}[4];\mathrm{plus}(\mathrm{num}[2];\mathrm{num}[2]))\\
\mapsto\ &\mathrm{plus}(\mathrm{num}[4];\mathrm{num}[4])\\
\mapsto\ &\mathrm{num}[8]
\end{aligned}$$

4.1.5 L{→}的类型安全-1

❖ **定理14.3(保持性)** *If* $e : \tau$ *and* $e \mapsto e'$, *then* $e' : \tau$.

证明：假设采用**call-by-value**语义，证明对转换规则**(14.4)**归纳。

考虑规则 $\dfrac{e_2\ \mathsf{val}}{\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) \mapsto [e_2/x]e_1}$

假设 $\mathtt{ap}(\mathtt{lam}[\tau_2](x.e_1);e_2) : \tau_1$

由定型逆转引理**14.1**的**2**，有 $e_2 : \tau_2$ 和 $x : \tau_2 \vdash e_1 : \tau_1$

再由置换引理**14.2**,有 $[e_2/x]e_1 : \tau_1$

……

4.1.5 L{→}的类型安全-2

❖ **引理14.4(范式)** *If* e *val and* $e : \mathtt{arr}(\tau_1;\tau_2)$, *then* $e = \mathtt{lam}[\tau_1](x.e_2)$ *for some* x *and* e_2 *such that* $x : \tau_1 \vdash e_2 : \tau_2$.

❖ **定理14.5(进展性)** 如果**e:τ**,则**e**或者是一个值，或者存在**e'**使得 $e \mapsto e'$

证明：证明对定型规则**(14.2)**归纳。注意这里只考虑闭项，在定型推导上没有假设，即Γ为空。

考虑规则**(14.2c)**

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

对e_1归纳(e_1是值或者是$e_1 \to e'_1$)，再结合转换规则证明……

4.1.6 L{→}的大步语义-1

❖ 计算(求值)语义

➢ λ抽象 $\mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)$ **(14.6)**

➢ λ应用 $\dfrac{e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v}$

❖ 环境语义：引入环境(记录自由变元的绑定)

➢ λ抽象 $\mathcal{E} \vdash \mathtt{lam}[\tau](x.e) \Downarrow \mathtt{lam}[\tau](x.e)$ **(14.7)**

➢ λ应用 $\dfrac{\mathcal{E} \vdash e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad \mathcal{E} \vdash e_2 \Downarrow v_2 \quad \mathcal{E}, x \Downarrow v_2 \vdash e \Downarrow v}{\mathcal{E} \vdash \mathtt{ap}(e_1;e_2) \Downarrow v}$

当将函数应用到实参时，在整个函数体求值过程中，函数的形参被绑定到实参值。

➢ 这个环境语义是不正确的，因为它与求值语义中的置换语义不一致。

4.1.6 L{→}的大步语义-2

❖ 为什么环境语义不正确？它使用<u>环境(记录自由变元绑定)</u>假设，该假设对那些<u>返回值包含自由变元的函数</u>的求值会不正确.

例： $e = \mathtt{ap}(\mathtt{lam}[\mathtt{nat}](x.\mathtt{lam}[\mathtt{nat}](y.x));\mathtt{num}[3])$

由求值语义规则， $\dfrac{e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\mathtt{ap}(e_1;e_2) \Downarrow v}$

由于 $[\mathtt{num}[3]/x]\mathtt{lam}[\mathtt{nat}](y.x) \Downarrow \mathtt{lam}[\mathtt{nat}](y.\mathtt{num}[3])$

故*e* 求值为 $\mathtt{lam}[\mathtt{nat}](y.\mathtt{num}[3])$

对于包含 *e* 的表达式 $e' = \mathtt{let}(e; f.\mathtt{ap}(f;\mathtt{num}[4]))$

由置换语义可得 $e' \mapsto^* \mathtt{ap}(\mathtt{lam}[\mathtt{nat}](y.\mathtt{num}[3]);\mathtt{num}[4])$ (**[e/f]**)

$e' \Downarrow \mathtt{num}[3]$ (**[num[4]/y]**)

4.1.6 L{→}的大步语义-3

例： $e = \mathtt{ap}(\mathtt{lam}[\mathtt{nat}](x.\mathtt{lam}[\mathtt{nat}](y.x));\mathtt{num}[3])$

但是由环境语义规则， $e' = \mathtt{let}(e; f.\mathtt{ap}(f;\mathtt{num}[4]))$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \mathtt{lam}[\tau](x.e) \quad \mathcal{E} \vdash e_2 \Downarrow v_2 \quad \mathcal{E}, x \Downarrow v_2 \vdash e \Downarrow v}{\mathcal{E} \vdash \mathtt{ap}(e_1;e_2) \Downarrow v} \quad \textbf{(14.8)}$$

e 的值由外层λ抽象确定,它以 $x \Downarrow \mathtt{num}[3]$ 为假设

由**(14.7a)** $\mathtt{lam}[\mathtt{nat}](y.x) \Downarrow \mathtt{lam}[\mathtt{nat}](y.x)$

e 求值到开式 $\mathtt{lam}[\mathtt{nat}](y.x)$,其中*x* 是自由的

但是对*e'* 的求值来说,其假设为$f \Downarrow \mathtt{lam}[\mathtt{nat}](y.x)$,而没有对 *x* 的假设,这样对 *f* **num[4]**求值就有麻烦,因为*f* 的函数体有内部的λ抽象,即*x*. ($\mathtt{lam}[\mathtt{nat}](y.x) \Downarrow \mathtt{lam}[\mathtt{nat}](y.x)$)

这就导致求值受阻,因为没有对*x* 的绑定.

4.1.6 L{→}的大步语义-4

例： $e = \mathtt{ap}(\mathtt{lam}[\mathtt{nat}](x.\mathtt{lam}[\mathtt{nat}](y.x));\mathtt{num}[3])$

$e' = \mathtt{let}(e; f.\mathtt{ap}(f;\mathtt{num}[4]))$

导致麻烦的原因是：出现在*e*内层λ抽象中的变元 *x* 作为外层λ抽象的值返回时，将逃逸出其作用域。

结果，在环境假设中没有对 *x* 的绑定，从而导致求值受阻.

求值断言中的环境假设：类似于栈的行为

高阶语言中的变元：类似于堆的行为

前者不符合后者！

4.1.7 闭包(Closures)-1

如何解决在环境语义中遇到的问题?

必须保证λ抽象中的自由变元没有从环境中脱离绑定!

❖ 将环境当作显式置换

- 显式置换是一个数据结构:记录变元将被置换成什么
- 仅当遇到变元时,才将变元代换为在环境中对应的绑定 ——推迟置换 $[v_1,\ldots,v_k/x_1,\ldots,x_k]\mathtt{lam}[\tau](x.e)$
- 在对λ抽象求值的地方，将环境附加在该λ抽象上，从而有：clo[τ](*E*,*x.e*) ——闭包

  环境*E*通过为λ抽象中的自由变量提供绑定，来"封闭"自由变量。

4.1.7 闭包(Closures)-2

❖ L{→}的环境语义

- 值 *Values* $V ::= \mathrm{clo}[\tau](E; x.e)$

  值不再是表达式形式，而是一种特有的语法范畴
- 环境 *Env's* $E ::= \bullet \mid E,\ x \mapsto v$

  环境不再是假言求值断言中的假设，而是可以出现在闭包中的一个数据结构

$$\frac{}{\mathcal{E}, x \Downarrow v \vdash x \Downarrow v} \quad (14.9a)$$

$$\frac{E = \{\, x_1 \mapsto v_1 \ldots x_k \mapsto v_k \,\}}{x_1 \Downarrow v_1, \ldots, x_k \Downarrow v_k \vdash \mathtt{lam}[\tau]\,(x.e) \Downarrow \mathtt{clo}[\tau]\,(E; x.e)} \quad (14.9b)$$

将k个变量的求值假设保存到环境E中

4.1.7 闭包(Closures)-3

❖ L{→}的环境语义

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \mathtt{clo}[\tau]\,(E; x.e) \quad \mathcal{E} \vdash e_2 \Downarrow v \quad E = \{\, x_1 \mapsto v_1 \ldots x_k \mapsto v_k \,\} \quad x_1 \Downarrow v_1, \ldots, x_k \Downarrow v_k, x \Downarrow v \vdash e \Downarrow w}{\mathcal{E} \vdash \mathrm{ap}\,(e_1; e_2) \Downarrow w} \quad (14.9c)$$

从环境E中取得的假设

例：$e = \mathrm{ap}(\mathtt{lam}[\mathtt{nat}](x.\mathtt{lam}[\mathtt{nat}](y.x)), \mathtt{num}[3])$

$e' = \mathtt{let}(e, f.\mathrm{ap}(f, \mathtt{num}[4]))$

对*e*求值，应用(14.9c)，对于前提(红圈部分)，由(14.9b)有

$$\vdash \mathtt{lam}[\mathtt{nat}](x.\mathtt{lam}[\mathtt{nat}](y.x)) \Downarrow \mathtt{clo}[\mathtt{nat}](E; x.\mathtt{lam}[\mathtt{nat}](y.x))$$

$$E = \{\ \}$$

4.1.7 闭包(Closures)-4

例：需要计算

$$x \Downarrow \mathtt{num}[3] \vdash \mathtt{lam}[\mathtt{nat}](y.x) \Downarrow ?$$

由(14.9b)有

$$x \Downarrow \mathtt{num}[3] \vdash \mathtt{lam}[\mathtt{nat}](y.x) \Downarrow \mathrm{clo}[\mathtt{nat}](E'; y.x)$$

$$E' = \{x \mapsto \mathtt{num}[3]\}$$

则有 $\vdash e \Downarrow \mathrm{clo}[\mathtt{nat}](E'; y.x) \qquad E' = \{x \mapsto \mathtt{num}[3]\}$

接下来计算 $e' = \mathtt{let}(e, f.\mathrm{ap}(f, \mathtt{num}[4]))$

由 $\dfrac{\mathcal{E} \vdash e_1 \Downarrow v_1 \quad \mathcal{E}, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\mathcal{E} \vdash \mathtt{let}(e_1; x.e_2) \Downarrow v_2}$

需要计算 $f \Downarrow \mathtt{clo}[\mathtt{nat}](E'; y.x) \vdash \mathrm{ap}(f, \mathtt{num}[4]) \Downarrow ?$

由(14.9c)有 $x \Downarrow \mathtt{num}[3], y \Downarrow \mathtt{num}[4] \vdash x \Downarrow \mathtt{num}[3]$

$$\vdash e' \Downarrow \mathtt{num}[3]$$

4.2 积类型(元组、记录)

二元积(binary product)：一组序对(pair);

消去形式(运算)：投影－选择序对中的第一项或第二项

空积(nullary product)：唯一的没有值的空元组，没有消去形式

有限积(finited product)：n元组(记录)；消去形式：投影

4.2.1 空积和二元积 [PFPL, 17.1]

4.2.2 有限积 [PFPL, 17.2]

4.2 积类型(元组、记录)

- 序对(二元组)示例
  - 复数 nat × nat <3, 4> 3-实部, 4-虚部
  - 二元函数 如 plus1 : nat × nat → nat plus1 <*x*, *y*>
- 元组示例
  - 如 <str, str, nat> <"*Zhang*", "*DS*", *88* >
  - 多元函数 如 max3 : <nat,nat,nat> → nat max3 <*3, 2, 5*>
- 记录示例
  - 类型 <sname:str, cname:str, score:nat>

    记录 <sname = "*Zhang*", cname = "*DS*", score = *88* >

4.2.1 空积和二元积-1

❖ 抽象语法

Types τ ::= unit | prod (τ_1; τ_2)

Expr's e ::= triv | pair(e_1; e_2) | fst(e) | snd(e)

抽象语法	具体语法	
unit	unit	空积(nullary)类型
triv	< >	空元组
prod (τ_1; τ_2)	$\tau_1 \times \tau_2$	二元积类型，prod:二元积类型构造子
pair(e_1; e_2)	$<e_1, e_2>$	二元组序对，pair:二元组项构造子
fst(e)	fst(e)	消去形式：第一投影
snd(e)	snd(e)	消去形式：第二投影

pair : $\tau_1 \to \tau_2 \to$ prod (τ_1, τ_2)   fst : prod (τ_1, τ_2) $\to \tau_1$

4.2.1 空积和二元积-2

❖ 静态语义

空积的引入规则

$$\frac{}{\Gamma \vdash \texttt{triv} : \texttt{unit}}$$

二元积的引入规则

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1; e_2) : \texttt{prod}(\tau_1; \tau_2)}$$

二元积的消去规则

$$\frac{\Gamma \vdash e : \texttt{prod}(\tau_1; \tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \texttt{prod}(\tau_1; \tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2}$$

(17.1)

❖ 动态语义

空值和序对都是值

$$\frac{}{\texttt{triv val}} \qquad \frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\texttt{pair}(e_1; e_2) \text{ val}}$$

$$\left\{\frac{e_1 \mapsto e_1'}{\texttt{pair}(e_1; e_2) \mapsto \texttt{pair}(e_1'; e_2)}\right\}$$

$$\left\{\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\texttt{pair}(e_1; e_2) \mapsto \texttt{pair}(e_1; e_2')}\right\}$$

尚未完成到序对的求值时，允许在投影下归约

$$\frac{e \mapsto e'}{\texttt{fst}(e) \mapsto \texttt{fst}(e')} \qquad \frac{e \mapsto e'}{\texttt{snd}(e) \mapsto \texttt{snd}(e')}$$

(17.2)

求值到序对时，可以进行投影，结果是相应的分量

$$\frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\texttt{fst}(\texttt{pair}(e_1; e_2)) \mapsto e_1} \qquad \frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\texttt{snd}(\texttt{pair}(e_1; e_2)) \mapsto e_2}$$

•惰性(lazy)语义
省去{ }中的规则或前提
•急切(eager)语义
包含{ }中的规则或前提

4.2.1 空积和二元积-3

例 $\texttt{ap}(\texttt{lam}[\texttt{prod}(\texttt{nat}; \texttt{nat})](x.\texttt{fst}(x)); \texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$

➢ Call-by-value, eager急切求值语义

$\texttt{ap}(\texttt{lam}[\texttt{prod}(\texttt{nat}; \texttt{nat})](x.\texttt{fst}(x)); \texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$
$\mapsto \texttt{ap}(\texttt{lam}[\texttt{prod}(\texttt{nat}; \texttt{nat})](x.\texttt{fst}(x)); \texttt{pair}(\texttt{num}[2]; \texttt{num}[7]))$
$\mapsto \texttt{fst}(\texttt{pair}(\texttt{num}[2]; \texttt{num}[7]))$
$\mapsto \texttt{num}[2]$

➢ Call-by-name, eager急切求值语义

$\texttt{ap}(\texttt{lam}[\texttt{prod}(\texttt{nat}; \texttt{nat})](x.\texttt{fst}(x)); \texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$
$\mapsto \texttt{fst}(\texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$
$\mapsto \texttt{fst}(\texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[7])))$
$\mapsto \texttt{num}[2]$

4.2.1 空积和二元积-4

例 $\texttt{ap}(\texttt{lam}[\texttt{prod}(\texttt{nat}; \texttt{nat})](x.\texttt{snd}(x)); \texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$

➢ Call-by-value, lazy惰性求值语义

$\texttt{ap}(\texttt{lam}[\texttt{prod}(\texttt{nat}; \texttt{nat})](x.\texttt{snd}(x)); \texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$
$\mapsto \texttt{snd}(\texttt{pair}(\texttt{num}[2]; \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])))$
$\mapsto \texttt{plus}(\texttt{num}[3]; \texttt{num}[4])$
$\mapsto \texttt{num}[7]$

➢ Call-by-name, lazy惰性求值语义

同上

4.2.2 有限积-1

❖ 抽象语法

Types τ ::= prod[I]($i \mapsto \tau_i$) I:索引集合

Expr's e ::= tuple[I]($i \mapsto e_i$) | proj[I][i](e)

抽象语法	具体语法	
prod[I]($i \mapsto \tau_i$)	$\Pi_{i \in I} \tau_i$	n元积类型, prod: 类型构造子
tuple[I]($i \mapsto e_i$)	$< e_i >_{i \in I}$	n元组，tuple: 项构造子
proj[I][i](e)	$e.i$	第i投影 ($0 \leqslant i \leqslant n$-1)

❖ 静态语义

$$\frac{(\forall i \in I)\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{tuple}[I]\,(i \mapsto e_i) : \texttt{prod}[I]\,(i \mapsto \tau_i)}$$

$$\frac{\Gamma \vdash e : \texttt{prod}[I]\,(i \mapsto e_i) \quad j \in I}{\Gamma \vdash \texttt{proj}[I]\,[j]\,(e) : \tau_j}$$

(17.3)

4.2.2 有限积-2

❖ 动态语义

$$\frac{\{(\forall i \in I)\ e_i \text{ val}\}}{\texttt{tuple}[I]\,(i \mapsto e_i) \text{ val}}$$

$$\frac{e_j \mapsto e_j' \quad (\forall i \neq j)\ e_i' = e_j}{\texttt{tuple}[I]\,(i \mapsto e_i) \mapsto \texttt{tuple}[I]\,(i \mapsto e_i')}$$

改为: $e_i' = e_i$

$$\frac{\texttt{tuple}[I]\,(i \mapsto e_i) \text{ val}}{\texttt{proj}[I]\,[j]\,(\texttt{tuple}[I]\,(i \mapsto e_i)) \mapsto e_j}$$

(17.4)

❖ 安全性

4.2.2 有限积-3

- 多元函数与高阶函数
  - 二元函数　例：plus1 ： nat × nat → nat
    plus1 = lam[prod(nat; nat)] (*x*.plus(fst(*x*); snd(*x*)))
    对ap(plus1;pair(num[2]; num[3]))求值得num[5]
  - 高阶函数　例：plus2 ： nat → nat → nat
    plus2 = lam[nat](*x*.lam[nat](*y*.plus(*x*, y)))
    对ap(ap(plus2;num[2]); num[3])求值得num[5]
- 多元函数与高阶函数的相互转换
  - Currying：多元函数到高阶函数的转换
    curry = λ *f* : nat × nat → nat. λ *x*:nat. λ *y*:nat. *f* <*x*, *y*>
  - Uncurrying：高阶函数到多元函数的转换
    uncurry = λ*f* : nat → nat → nat. λ *x* : nat × nat. *f* (fst *x*) (snd *x*)

4.3 和类型

二元和(binary sum)：从两者选择其一
空和(nullary sum)：从空中选择
n元和(n-ary sum)：从n个中选择其一

4.3.1 二元和与空和 [PFPL, 18.1]
4.3.2 有限和 [PFPL, 18.2]
4.3.3 一些有用的和类型 [PFPL, 18.3]

4.3.1 二元和与空和-1

- 二元和示例
  - 表　list = unit + nat × list
    表或者是空表unit，
    　或者是由表头和表尾组成的表 nat × list
  - 二叉树　bitree = unit + < label, bitree, bitree >
    二叉树或者是空树unit，
    　或者是由label和两棵子树组成的树
    　　< label, bitree, bitree >　// 三元积类型
    假设lab[str]表示值为str的标签,则下面是一棵二叉树
    　null表示bitree的左标记值(类型为unit)
    　<lab[“a”], null, <lab[“b”], null, null> > // 三元组

4.3.1 二元和与空和-2

- 地址
  Addr = PhysicalAddr + VirtualAddr　// 二元和类型
  PhysicalAddr = <firstlast : str, addr : str > // 记录类型,左标记类型
  VirtualAddr = <name : str, email : str >　// 记录类型,右标记类型

通过左/右标记类型的分量产生Addr类型的元素
- in[l] : PhysicalAddr → Addr　左标记
- in[r] : VirtualAddr → Addr　右标记

引入case 构造子，以区分一个值是来自和类型左边的分支还是右边的分支

getName = λ *a* : Addr. case *a* {
　　in[l](*x*) => *x*.firstlast
　| in[r](*y*) => *y*.name }

4.3.1 二元和与空和-3

- 抽象语法
  Types　τ ::= void | sum (τ_1; τ_2)
  Expr’s　*e* ::= abort[τ](*e*) | in[l][τ](*e*) | in[r][τ](*e*) |
  　case(*e*; $x_1.e_1$; $x_2.e_2$)
- 空和类型(nullary sum)

抽象语法	具体语法	
void	void	空和类型
abort[τ](*e*)	abort$_\tau$(*e*)	消去形式：中止对*e*的求值

空和类型的值表示从 0 个可选项中选择一个,故空和类型没有值,因此也就没有引入形式.

其消去形式abort[τ](*e*)表示在*e*不能再求值时,中止对*e* 的求值. (如Java中的异常机制)

4.3.1 二元和与空和-4

- 二元和类型(binary sum)

抽象语法	具体语法	
sum(τ_1; τ_2)	$\tau_1 + \tau_2$	二元和类型，sum:二元和类型构造子
in[l][τ](*e*)	in[l](*e*)	引入形式：左标记；$e : \tau_1$
in[r][τ](*e*)	in[r](*e*)	引入形式：右标记；$e : \tau_2$
case(*e*; $x_1.e_1$; $x_2.e_2$)	case *e*{ in[l](x_1) =>e_1 | in[r](x_2) =>e_2}	

消去形式：分情况分析值标记，得到相应的体e_1或e_2

引入形式定义如何由类型为τ_1 或 τ_2的表达式*e* 构造类型为sum(τ_1; τ_2)的值in[l][τ](*e*)(左标记值)或in[r][τ](*e*) (右标记值)

消去形式定义对类型为sum(τ_1; τ_2)的值的运算，它需要区分值是左标记值in[l][τ](*e*)还是右标记值in[r][τ](*e*)来分情况处理。

4.3.1 二元和与空和-4

❖ 静态语义

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort}[\tau](e) : \tau}$$

空和的消去规则

二元和的引入规则

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \mathtt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{in[l]}[\tau](e) : \tau}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \mathtt{sum}(\tau_1;\tau_2)}{\Gamma \vdash \mathtt{in[r]}[\tau](e) : \tau}$$

(18.1)

二元和的消去规则

$$\frac{\Gamma \vdash e : \mathtt{sum}(\tau_1;\tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}(e; x_1.e_1; x_2.e_2) : \tau}$$

不是取记录中的成员；而是e_2中带约束变元 x_2

$$\frac{e \mapsto e'}{\mathtt{abort}[\tau](e) \mapsto \mathtt{abort}[\tau](e')}$$

❖ 动态语义

左标记值和右标记值是值

$$\frac{\{e \ \mathsf{val}\}}{\mathtt{in[l]}[\tau](e) \ \mathsf{val}}$$

$$\frac{\{e \ \mathsf{val}\}}{\mathtt{in[r]}[\tau](e) \ \mathsf{val}}$$

(18.2)

急切语义下，e未完成求值时，允许在引入形式(标记)下归约

$$\left\{\frac{e \mapsto e'}{\mathtt{in[l]}[\tau](e) \mapsto \mathtt{in[l]}[\tau](e')}\right\}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{in[r]}[\tau](e) \mapsto \mathtt{in[r]}[\tau](e')}\right\}$$

•惰性(lazy)语义
省去{ }中的规则或前提
•急切(eager)语义
包含{ }中的规则或前提

4.3.1 二元和与空和-5

❖ 动态语义 (18.2)

尚未求值到左/右标记值时，该规则允许在消去形式下归约

$$\frac{e \mapsto e'}{\mathtt{case}(e; x_1.e_1; x_2.e_2) \mapsto \mathtt{case}(e'; x_1.e_1; x_2.e_2)}$$

$$\frac{\{e \ \mathsf{val}\}}{\mathtt{case}(\mathtt{in[l]}[\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1}$$

$$\frac{\{e \ \mathsf{val}\}}{\mathtt{case}(\mathtt{in[r]}[\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2}$$

•惰性(lazy)语义
省去{ }中的规则或前提
•急切(eager)语义
包含{ }中的规则或前提

对标记值根据其标记，决定按哪一种情况进行置换

4.3.1 二元和与空和-6

❖ 安全性(略)

❖ unit vs. void

- 空积类型unit只有一个值triv
  空和类型void没有值
- 如果e : unit，假使e 求值到v，则v : unit
  如果e : void，则由e 一定不会求得值，因为如果e 能求值到v，则v : void，而void类型没有值
- 在许多程序语言(如C语言)中的void类型实际上是unit类型。

4.3.2 有限和-1

❖ n元和（略）

❖ 带标签的和/变式(labelled variants)

Types τ ::= sum[I]($i \mapsto \tau_i$)

Expr's e ::= inj[I][j](e) | case[I] (e; $i \mapsto x_i.e_i$)

抽象语法	具体语法	
sum[I]($i \mapsto \tau_i$)	$\Sigma_{i \in I} \tau_i$	有限和类型
inj[I][j](e)	in[j](e)	引入形式
case[I] (e; $i \mapsto x_i.e_i$)	case e { in[i](x_i)=>e_i} $_{i \in I}$	消去形式

- C语言中的union类型就是变式类型

4.3.2 有限和-2

❖ 静态语义

引入规则

$$\frac{\Gamma \vdash e : \tau_j \quad j \in I}{\Gamma \vdash \mathtt{inj}[I][j](e) : \mathtt{sum}[I](i \mapsto \tau_i)} \quad (18.3)$$

$$\frac{\Gamma \vdash e : \mathtt{sum}[I](i \mapsto \tau_i) \quad (\forall i \in I)\ \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \mathtt{case}[I](e; i \mapsto x_i.e_i) : \tau}$$

消去规则

❖ 动态语义

$$\frac{\{e \ \mathsf{val}\}}{\mathtt{inj}[I][j](e) \ \mathsf{val}} \quad (18.4)$$

$$\left\{\frac{e \mapsto e'}{\mathtt{inj}[I][j](e) \mapsto \mathtt{inj}[I][j](e')}\right\}$$

4.3.2 有限和-3

❖ 动态语义

$$\frac{e \mapsto e'}{\mathtt{case}[I](e; i \mapsto x_i.e_i) \mapsto \mathtt{case}[I](e'; i \mapsto x_i.e_i)}$$

$$\frac{\mathtt{inj}[I][j](e) \ \mathsf{val}}{\mathtt{case}[I](\mathtt{inj}[I][j](e); i \mapsto x_i.e_i) \mapsto [e/x_j]e_j} \quad (18.4)$$

❖ 安全性（略）

4.3.3 一些有用的和类型-Boolean

❖ Boolean类型

- 语法

  Types τ ::= bool

  Expr's e ::= tt | ff | if(e; e_1; e_2)

  tt 和 ff是引入形式，分别表示真和假， if(e; e_1; e_2)是消去形式

- 具体的定义（由空积与二元和定义）

  bool = sum(unit, unit)

  tt = in[l][bool](triv)　　(18.5)

  ff = in[r][bool](triv)

  if(e; e_1; e_2) = case(e; $x_1.e_1$; $x_2.e_2$)

  x_1 和 x_2 是任意的变元，使得 $x_1\#e_1$ (e_1中无自由出现的x_1)且 $x_2\#e_2$

4.3.3 一些有用的和类型-枚举类型

❖ 枚举类型

例如：扑克牌的花色

- 类型 card = unit + (unit + (unit + unit))
- 引入形式： hearts | spades | diamonds | clubs

  hearts = in[l](triv)　spades = in[r](in[l](triv))

  diamonds = in[r](in[r](in[l](triv)))

  clubs = in[r](in[r](in[r](triv)))

- 消去形式

  case e { hearts => e_0, spades => e_1, diamonds => e_2,
  clubs => e_3 }

4.3.3 一些有用的和类型-option-1

❖ 选项类型option

Types τ ::= opt(τ)

Expr's e ::= null | just(e) | ifnull[τ](e; e_1; $x.e_2$)

- 类型 opt(τ) = sum(unit; τ)表示类型τ 的可选值类型
- 引入形式
  - null = in[l][opt(τ)](triv) 左标记，表示由空值形成的左标记值
  - just(e) = in[r][opt(τ)](e) 右标记，表示类型为τ 的表达式 e 形成的右标记值
- 消去形式

  ifnull[τ](e; e_1; $x.e_2$) = case(e; _.e_1; $x_2.e_2$)

  下划线表示任意不出现在e_1中的变元

4.3.3 一些有用的和类型-option-2

❖ 理解空指针错误(null pointer fallacy)

——option类型的意义之一

- 起因：在OO语言中，所有对象都是引用(指针)，对象的引用可能为空，不能通过空引用来访问对象的域.
- 如何避免空指针错误?

  一些语言提供空指针的检测函数 null : $\tau\to$bool

  if null(e) then …*error* … else …*ok* …

- 但是空指针异常仍然普遍, 原因: 1)缺少空指针检测; 2)极少在程序的异常处进行空指针检测
- 解决：用opt(τ)描述类型为τ的可选值类型，其值或者为τ类型的值，或者为空.

  消去形式 ifnull[τ](e;…*error*…; x. …*ok*…)

  在静态语义和动态语义中，针对这两种情况分别处理，并进行传播。

4.4 一般递归-1

递归论:能行性论(讨论可计算/可判定)，数理逻辑的一个分支，研究递归函数及其推广的学科。递归函数是数论函数的一种，其定义域与值域都是自然数集。

函数$f(x)$的可计算性：当x 的值给出后，如果$f(x)$有定义，则可以在有限步内得出该函数的值；当$f(x)$无定义时，如果能在有限步内判知，则说 f是可完全计算的，如果不能在有限步内判知，则说 f 是可半计算的。

17世纪,Pascal(法):正式使用与递归式密切相关的数学归纳法

19世纪,Dedekind(德),Peano(意):用原始递归式定义加和乘

1923,Skolem:提出并证明“一切初等数论中的函数都可以由原始递归式定义，即都是原始递归函数”。

4.4 一般递归-2

1931，Gödel(奥地利):在证明其著名的不完全性定理时，以原始递归式为主要工具把所有元数学的概念都算术化了。=>出现了原始递归函数论

不完全性定理：在形式数论(算术逻辑)演绎系统中，总可以找出一个合理的命题使得在该系统中既无法证明它为真，也无法证明它为假。

什么是原始递归式？什么是原始递归函数？

本原函数：1)零函数$O(x)$=0；2)后继函数$S(x)=x+1$；3)广义幺函数或射影函数$Imn(x_1,\ldots, x_m)= x_n$ $(1\leqslant n\leqslant m)$

4.4 一般递归-3

原始递归式(primitive recursion):

$$\begin{cases} f(u_1,\cdots,u_n,0) = A(u_1,\cdots,u_n) \\ f(u_1,\cdots,u_n,S(x)) = B(u_1,\cdots,u_n,x,f(u_1,\cdots,u_n,x)) \end{cases}$$ 多参数(u)

$$\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,x)) \end{cases}$$ 单参数(u)

由A、B两函数,依次计算$f(u,0)$, $f(u,1)$, $f(u,2)$,…。

只要A、B为全函数且可计算,则新函数f也是全函数且可计算

原始递归函数: 由本原函数出发，经过原始递归式与有限次的复合而作出的函数。

叠置(复合):由一个 m元函数 f与 m个 n元函数 $g_1, g_2, \ldots, g_m$ 而造成新函数 $f(g_1(x_1, \ldots, x_n), g_2(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$, 后者亦可记为 $f(g_1, \ldots, g_m)(x_1, \ldots, x_n)$。

本原函数是全函数且可计算，故原始递归函数也是全函数且可计算。

4.4 一般递归-4

Ackermann(德): 提出非原始递归的可计算函数，否定了"原始递归函数可能穷尽一切可计算的函数"的猜测.

$$\begin{cases} g(0,n) = n+1 \\ g(S(m),0) = g(m,1) \\ g(S(m),S(n)) = g(m,g(S(m),n)) \end{cases}$$

1934, Gödel(奥地利):提出一般递归函数的定义

有序递归式(半递归式): $\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,g(u,S(x)))) \end{cases}$

与原始递归式的不同在于: 它不是把$f(u, S(x))$的计算化归于$f(u, x)$的计算，而是先化归于$f(u,g(u, S(x)))$的计算，然后化归于$f(u, g(u, g(u, S(x))))$的计算(记做 $f(u, g_u^2(S(x)))$)，再化归于 $f(u, g_u^3(S(x)))$ 的计算, ……。

如果有一个m，使得 $g_u^m(S(x))=0$，即函数 g_u 在 $S(x)$ 处归宿于0，则 $f(u, g_u^m(S(x))) = f(u, 0) = A(u)$

4.4 一般递归-5

有序递归式(半递归式): $\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,g(u,S(x)))) \end{cases}$

如果不存在一个m，使得 $g_u^m(S(x))=0$，即函数 g_u 在$S(x)$处不归宿于0，将导致永远化归下去而得不到结果，从而$f(u, S(x))$不仅不能被计算，而且没有定义。

即使A、B与g是全函数且可计算，而由半递归式所定义的函数未必是全函数，也可能是部分函数。但只要有定义的地方，即g_u归宿于0的地方，就一定能计算。

递归半函数(递归部分函数): 由本原函数出发，经过半递归式与有限次的复合而作出的函数。

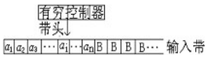
如果作出的函数是全函数，则称做递归全函数(一般递归函数)

4.4 一般递归-6

1936, Church(美) :提出"可计算函数恰巧是一般递归函数"（判定性问题 Entscheidungs problem ）
和 Kleene在 20 世纪三十年代引入λ 演算(无类型的)

1936, Turing(英) :提出Turing机，指出可计算函数恰巧是可用Turing机所计算的函数。

1936，Kleene(美) : 证明一般递归函数就是Turing机所计算的函数。
——Church-Turing论点



4.4.1 Gödel的T [PFPL, 15]

4.4.2 Ackermann函数 [PFPL, 15.4]

4.4.3 Plotkin的PCF [PFPL, 16]

4.4.4 递归类型 [PFPL, 21]

4.4.1 Gödel的T-1

❖ L{nat,→}: Gödel的T

- 原始递归函数: 由本原函数出发，经过原始递归式与有限次的复合而作出的函数。

$$\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,x)) \end{cases}$$

❖ 语法

Types τ ::= nat | arr(τ_1; τ_2)

Expr's e ::= x | z | s(e) | rec [τ](e; e_0; x. y.e_1) | lam[τ](x.e) | ap(e_1; e_2)

- nat的引入形式: z －零; s(e)－e的后继，记 $\overline{n}$ 表示对z 应用n次s
- rec[τ](e; e_0; x. y.e_1) : 原始递归式。
  表示从初值e_0开始，按 $x.y.e_1$ 进行复迭变换，折叠 e 次所得的结果。其中，x 表示e的前驱，y 表示复迭 x 次的结果。

注: 在 rec[τ](e; e_0; x. y.e_1) 中，e_0与$A(u)$对应，e_1与$B(u, x, f(x))$对应，x与x对应, y与$f(x)$对应。

4.4.1 Gödel的T-2

复迭式(iteration) iter[τ](e; e_0; y.e_1)有时作为 rec[τ](e; e_0; x. y.e_1)的替换物。但复迭式的约束变元只有y，而没有 x 。

- 复迭式是原始递归式的特例,因为总能忽略对前驱的绑定
- 原始递归式可以由复迭式定义(在复迭计算的同时计算前驱)

抽象语法	具体语法	
arr (τ_1; τ_2)	$\tau_1 \to \tau_2$	全函数类型
lam[τ](x.e)	λ(x:τ.e)	函数定义（引入形式）
ap(e_1; e_2)	e_1(e_2)	函数应用（消去形式）
rec[τ](e; e_0; x. y.e_1)	rec e { z=>e_0 | s(x) with y =>e_1}	
	具有引入和消去的双重含义	

- with子句的目的是重复将 y 绑定到递归调用的结果上。

4.4.1 Gödel的T-3

例：阶乘函数 $\begin{cases} f(0) = S(0) \\ f(S(x)) = S(x) * f(x) \end{cases}$

具体语法表示

fct = λ(*n*:nat.rec *n* { z => s(z) | s(x) with *y* => s(*x*)*y})

抽象语法表示

fct = lam[nat](*n*.rec[arr(nat;nat)](*n*; s(z); *x*.*y*.times(s(*x*); y)))

4.4.1 Gödel的T-4

❖ 静态语义

变元的定型规则

$$\frac{}{\Gamma, x : \mathtt{nat} \vdash x : \mathtt{nat}}$$

nat 的引入规则(零)

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}$$

nat 的引入规则(后继)

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

原始递归式的定型规则

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

函数的引入规则

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \# \Gamma}{\Gamma \vdash \mathtt{lam}[\sigma](x.e) : \mathtt{arr}(\sigma; \tau)} \qquad (15.1)$$

函数的消去规则

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

➢ 引理15.1(置换) 如果$\Gamma, x : \tau \vdash e' : \tau'$并且$\Gamma \vdash e : \tau$，那么$\Gamma \vdash [e/x]e' : \tau'$

4.4.1 Gödel的T-5

❖ 动态语义 (15.2)

假设对s(*e*)采用惰性语义，对函数应用采用按名调用语义

闭值

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

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1; e_2) \mapsto \mathtt{ap}(e_1'; e_2)} \qquad \frac{}{\mathtt{ap}(\mathtt{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e}$$

函数应用的动态语义(基于置换语义)

$$\frac{e \mapsto e'}{\mathtt{rec}[\tau](e; e_0; x.y.e_1) \mapsto \mathtt{rec}[\tau](e'; e_0; x.y.e_1)}$$

e未完全求值时，允许在原始递归式下归约

$$\frac{}{\mathtt{rec}[\tau](\mathtt{z}; e_0; x.y.e_1) \mapsto e_0}$$

$$\frac{}{\mathtt{rec}[\tau](\mathtt{s}(e); e_0; x.y.e_1) \mapsto [e, \mathtt{rec}[\tau](e; e_0; x.y.e_1)/x, y]e_1}$$

原始递归式的两种求值：1) 初值；2)在计算e_1前对*e* 递归调用

由于函数应用采用惰性语义(按名调用)，如果e1无需y来定值，则该递归调用不会被执行！

4.4.1 Gödel的T-6

➢ 引理15.2(范式) (略)

➢ 定理15.3(安全性) (略)

❖ 观测等价(observational equivalence)

$e_1 \cong e_2 : \tau\ [\Gamma], \quad \Gamma \vdash e_1 : \tau, \quad \Gamma \vdash e_2 : \tau$

表示两个具有相同类型的开式e_1和e_2在L{nat,→}程序中是无区别的，从而可以自由地在任意上下文中互换。

观测等价满足的性质：

➢ 一致性：零和非零不等价

➢ 同余性(congruence):对于任何的子表达式，用一个等价的表达式代替该子表达式，则新表达式仍和源表达式等价。

➢ 符号执行(symbolic execution):应用动态语义规则求值时保持等价

4.4.1 Gödel的T-7

❖ 终止性

➢ 定理15.4 (终止性)如果$e : \tau$,则存在*v* val使得$e \mapsto^* v$.

➢ L{nat,→}不存在无限循环，用它能编写的函数是数学上的全函数。

❖ 可定义性

➢ 数学函数*f* :N→N在L{nat,→}中是可定义的，当且仅当存在类型为nat→nat的表达式e_f能正确地模仿*f*在所有可能输入上的行为。

$$e_f(\overline{n}) \cong \overline{f(n)} : nat, \quad n \in \mathbb{N}$$

例1:后继函数可定义为表达式 succ=λ(*x*:nat.s(*x*))

例2:加倍函数d(*n*)=2 × *n*可定义为表达式

e_d =λ(*x*:nat.rec *x*{z=>z | s(*u*) with *v*=>s(s(*v*))})

4.4.1 Gödel的T-8

❖ 可定义性

例2:加倍函数d(*n*)=2 × *n*可定义为表达式

e_d =λ(*x*:nat.rec *x*{z=>z | s(*u*) with *v*=>s(s(*v*))})

证明：

由观测可知$e_f(\overline{0}) \cong \overline{0} : nat$，

假设 $e_f(\overline{n}) \cong \overline{d(n)} : nat$

$$\begin{aligned} e_d(\overline{n+1}) &\cong s(s(e_d(\overline{n}))) \\ &\cong s(s(\overline{2 \times n})) \\ &\cong \overline{2 \times (n+1)} \\ &\cong \overline{d(n+1)} \end{aligned}$$

4.4.2 Ackermann函数-1

❖ **Ackermann函数**

$$\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,x)) \end{cases}$$

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

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

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

$$\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,g(u,S(x)))) \end{cases}$$

Ackermann函数不是原始递归函数。

❖ **Ackermann函数是可定义的**

观察$A(m+1, n)$，它是从$A(m, 1)$开始，对$A(m, _)$迭代n次

定义高阶函数 it : (nat→nat) →nat →nat →nat为

$\lambda(f : \text{nat} \to \text{nat}.\lambda(n : \text{nat.rec } n\{z \Rightarrow \text{id} \mid s(\_) \text{ with } g \Rightarrow f \circ g\}))$

其中id=$\lambda(x\text{:nat}.x)$，g是取n的前驱时it对应的函数值

$f \circ g = \lambda(x\text{:nat}.f(g(x)))$

容易得到 $\texttt{it}(f)(\overline{n})(\overline{m}) \cong f^{(n)}(\overline{m}) : \texttt{nat}$

右边的表达式是从$\overline{m}$开始的f的n次复合

4.4.2 Ackermann函数-2

❖ **Ackermann函数是可定义的**

it= $\lambda(f : \text{nat} \to \text{nat}.\lambda(n : \text{nat.rec } n\{z \Rightarrow \text{id} \mid s(\_) \text{ with } g \Rightarrow f \circ g\}))$

id=$\lambda(x\text{:nat}.x)$，$f \circ g = \lambda(x\text{:nat}.f(g(x)))$ $\texttt{it}(f)(\overline{n})(\overline{m}) \cong f^{(n)}(\overline{m}) : \texttt{nat}$

由此，可以定义Ackermann函数a : nat →nat →nat为

$\lambda(m : \text{nat.rec } m\{z \Rightarrow \text{succ} \mid s(\_) \text{ with } f \Rightarrow \lambda(n : \text{nat.it}(f)(n)(f(\overline{1})))\})$

可以得到以下等式，这些等式表明Ackermann函数是可定义的。

$$\begin{aligned} a\ (\overline{0})\ (\overline{n}) &\cong s\ (\overline{n}) \\ a\ (\overline{m+1})\ (\overline{0}) &\cong a\ (\overline{m})\ (\overline{1}) \\ a\ (\overline{m+1})\ (\overline{n+1}) &\cong a\ (\overline{m})\ (a\ (s\ (\overline{m}))\ (\overline{n})) \end{aligned}$$

f

$$\begin{aligned} a\ (\overline{m+1})\ (\overline{0}) &\cong \texttt{it}\ (a\ (\overline{m}))\ (\overline{0})\ (a\ (\overline{m})\ (\overline{1})) && [App.\ a] \\ &\cong a\ (\overline{m})\ (\overline{1}) && [App.\ \texttt{it}] \end{aligned}$$

4.4.2 Ackermann函数-3

❖ **Ackermann函数是可定义的**

it= $\lambda(f : \text{nat} \to \text{nat}.\lambda(n : \text{nat.rec } n\{z \Rightarrow \text{id} \mid s(\_) \text{ with } g \Rightarrow f \circ g\}))$

id=$\lambda(x\text{:nat}.x)$，$f \circ g = \lambda(x\text{:nat}.f(g(x)))$ $\texttt{it}(f)(\overline{n})(\overline{m}) \cong f^{(n)}(\overline{m}) : \texttt{nat}$

由此，可以定义Ackermann函数a : nat →nat →nat为

$\lambda(m : \text{nat.rec } m\{z \Rightarrow \text{succ} \mid s(\_) \text{ with } f \Rightarrow \lambda(n : \text{nat.it}(f)(n)(f(\overline{1})))\})$

可以得到以下等式，这些等式表明Ackermann函数是可定义的。

$$\begin{aligned} a\ (\overline{0})\ (\overline{n}) &\cong s\ (\overline{n}) \\ a\ (\overline{m+1})\ (\overline{0}) &\cong a\ (\overline{m})\ (\overline{1}) \\ a\ (\overline{m+1})\ (\overline{n+1}) &\cong a\ (\overline{m})\ (a\ (s(\overline{m}))\ (\overline{n})) \end{aligned}$$

$$\begin{aligned} a\ (\overline{m+1})\ (\overline{n+1}) &\cong \texttt{it}\ (a\ (\overline{m}))\ (\overline{n+1})\ (a\ (\overline{m})\ (\overline{1})) && [App.\ a] \\ &\cong a\ (\overline{m})\ (\texttt{it}\ (a\ (\overline{m}))\ \overline{n})\ (a\ (\overline{m})\ (\overline{1})) && [App.\ \texttt{it}] \end{aligned}$$

g

$$a\ (\overline{m})\ (a\ (s\ (\overline{m}))\ (\overline{n})) \cong a\ (\overline{m})\ (\texttt{it}\ (a\ (\overline{m}))\ (\overline{n}))\ (a\ (\overline{m})\ (\overline{1})) \quad [App.\ 2nd\ a]$$

4.4.3 Plotkin的PCF-1

❖ **L{nat ⇀}: Plotkin的PCF**

➢ 使用一般递归式集成函数和自然数

$$\begin{cases} f(u,0) = A(u) \\ f(u,S(x)) = B(u,x,f(u,g(u,S(x)))) \end{cases}$$

➢ 部分函数，类型系统不再保证终止性

➢ 递归定义的不动点(fixed point)

如果F:σ→σ是某类型σ到它自己的函数，那么F的不动点是使得$F(x)=x$的值x:σ.

- 自然数上的平方函数的不动点有0和1
- 恒等函数有无数个不动点
- 后继函数没有不动点

4.4.3 Plotkin的PCF-2

➢ 阶乘函数是方程

f : nat → nat = λy : nat.ifz y { z=>s(z)|s(x)=> y ∗ f (x) }

的解。

➢ 阶乘函数是

F =λ f :nat → nat. λy: nat.ifz y { z=>s(z)|s(x)=> y ∗ f (x) }

的不动点。

➢ 不动点算子fix_σ:(σ→σ)→σ：对每个类型σ，函数fix_σ为σ到σ的函数产生一个不动点。

e.g.针对上述阶乘函数，有$fix_\sigma(F)= f$.

$fix_\sigma = \lambda f$:σ→σ. f (fix_σ (F))

fix_σ (M)= M (fix_σ (M))

4.4.3 Plotkin的PCF-3

❖ **语法**

Types τ ::= nat | parr(τ_1; τ_2)

Expr's e ::= x | z | s(e) | ifz(e; e_0; $x.e_1$) |
lam[τ]($x.e$) | ap(e_1; e_2) | fix[τ]($x.e$)

➢ fix[τ]($x.e$) : 一般递归式。x:τ 且e:τ

➢ ifz(e; e_0; $x.e_1$) : e是z,则为e_0; 否则将e的前驱绑定到x计算e_1(递归计算)。

抽象语法	具体语法	
parr (τ_1; τ_2)	$\tau_1 \rightharpoonup \tau_2$	部分函数类型
ifz(e, e_0; $x.e_1$)	ifz e{z=>e_0 | s(x) =>e_1}	
fix[τ]($x.e$)	fix x:τ is e	一般递归式，不动点τ是函数类型

4.4.3 Plotkin的PCF-4

❖ 静态语义

(16.1)

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}$$ 变元的定型规则

$$\frac{}{\Gamma \vdash \mathtt{z}:\mathtt{nat}}$$ nat 的引入规则(零)

$$\frac{\Gamma \vdash e:\mathtt{nat}}{\Gamma \vdash \mathtt{s}(e):\mathtt{nat}}$$ nat的引入规则(后继)

$$\frac{\Gamma \vdash e:\mathtt{nat} \quad \Gamma \vdash e_0:\tau \quad \Gamma, x:\mathtt{nat} \vdash e_1:\tau}{\Gamma \vdash \mathtt{ifz}(e;e_0;x.e_1):\tau}$$ 条件分支的定型规则

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathtt{lam}[\tau_1](x.e):\mathtt{parr}(\tau_1;\tau_2)}$$ 函数的引入规则

4.4.3 Plotkin的PCF-5

❖ 静态语义 (16.1)

函数的消去规则

$$\frac{\Gamma \vdash e_1:\mathtt{parr}(\tau_2;\tau) \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \mathtt{ap}(e_1;e_2):\tau}$$

一般递归式的定型规则
递归自引用: 在类型检查期间，用递归式本身来代换e中出现的x

$$\frac{\Gamma, \mathtt{fix}[\tau](x.e):\tau \vdash [\mathtt{fix}[\tau](x.e)/x]e:\tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e):\tau}$$

与上一规则等价的规则
将递归自引用看成是变量

$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash \mathtt{fix}[\tau](x.e):\tau} \quad (16.2)$$

➢ 定型规则满足置换引理 (Lemma16.1)

如果$\Gamma, x:\tau \vdash e':\tau'$并且$\Gamma \vdash e:\tau$，那么 $\Gamma \vdash [e/x]e':\tau'$

4.4.3 Plotkin的PCF-6

❖ 动态语义

•惰性(lazy)语义 省去{ }中的规则或前提
•急切(eager)语义 包含{ }中的规则或前提

闭值

$$\frac{}{\mathtt{z}\ \mathsf{val}}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{s}(e)\ \mathsf{val}}$$

$$\frac{}{\mathtt{lam}[\tau](x.e)\ \mathsf{val}}$$

(16.3)

$$\left\{\frac{e \mapsto e'}{\mathtt{s}(e) \mapsto \mathtt{s}(e')}\right\}$$ 在eager语义下,e未完成求值,允许在后继下归约

(16.4)

$$\frac{e \mapsto e'}{\mathtt{ifz}(e;e_0;x.e_1) \mapsto \mathtt{ifz}(e';e_0;x.e_1)}$$

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \mapsto e_0}$$

$$\frac{}{\mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \mapsto [e/x]e_1}$$

条件分支的动态语义

$$\frac{e_1 \mapsto e_1'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1';e_2)}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathtt{ap}(e_1;e_2) \mapsto \mathtt{ap}(e_1;e_2')}\right\}$$

$$\frac{\{e_2\ \mathsf{val}\}}{\mathtt{ap}(\mathtt{lam}[\tau](x.e);e_2) \mapsto [e_2/x]e}$$

函数应用

用递归式本身代换递归式体中的变量来实现自引用——展开递归式(unfold)

$$\frac{}{\mathtt{fix}[\tau](x.e) \mapsto [\mathtt{fix}[\tau](x.e)/x]e}$$

4.4.3 Plotkin的PCF-7

❖ 上下文语义

➢ 翻译规则：将表达式分解成一个求值上下文和一个可归约式

$$\frac{e = \mathcal{E}\{e_0\} \quad e_0 \leadsto e_0' \quad e' = \mathcal{E}\{e_0'\}}{e \mapsto_{\mathsf{c}} e'} \quad (16.7)$$

➢ 指令步由如下规则定义

$$\frac{}{\mathtt{ifz}(\mathtt{z};e_0;x.e_1) \leadsto e_0}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{ifz}(\mathtt{s}(e);e_0;x.e_1) \leadsto [e/x]e_1} \quad (16.8)$$

$$\frac{\{e_2\ \mathsf{val}\}}{\mathtt{ap}(\mathtt{lam}[\tau_2](x.e);e_2) \leadsto [e_2/x]e}$$

$$\frac{}{\mathtt{fix}[\tau](x.e) \leadsto [\mathtt{fix}[\tau](x.e)/x]e}$$

•惰性(lazy)语义
省去{ }中的规则或前提
•急切(eager)语义
包含{ }中的规则或前提

4.4.3 Plotkin的PCF-8

❖ 上下文语义

➢ 求值上下文由如下规则定义

$$\frac{}{\circ\ \mathsf{ectxt}}$$

$$\left\{\frac{\mathcal{E}\ \mathsf{ectxt}}{\mathtt{s}(\mathcal{E})\ \mathsf{ectxt}}\right\}$$

$$\frac{\mathcal{E}\ \mathsf{ectxt}}{\mathtt{ifz}(\mathcal{E};e_0;x.e_1)\ \mathsf{ectxt}} \quad (16.9)$$

$$\frac{\mathcal{E}_1\ \mathsf{ectxt}}{\mathtt{ap}(\mathcal{E}_1;e_2)\ \mathsf{ectxt}}$$

$$\left\{\frac{e_1\ \mathsf{val} \quad \mathcal{E}_2\ \mathsf{ectxt}}{\mathtt{ap}(e_1;\mathcal{E}_2)\ \mathsf{ectxt}}\right\}$$

•惰性(lazy)语义
省去{ }中的规则或前提
•急切(eager)语义
包含{ }中的规则或前提

4.4.3 Plotkin的PCF-PCF的可定义性-1

❖ PCF的可定义性(definability)

➢ 一般递归式
- 缺点：所定义的递归函数的终止性不是程序固有的，而必须由程序员证明
- 优点：可以定义更多的函数，给程序员更多的自由

自然数上的可计算函数在PCF中都是可编程的
——Church-Turing论点

➢ 一般递归函数 fun[τ_1; τ_2](x.y.e)

x 是代表函数本身的变量，y 是函数的参数

静态语义

$$\frac{\Gamma, \mathtt{fun}[\tau_1;\tau_2](x.y.e):\mathtt{parr}(\tau_1;\tau_2), y:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \mathtt{fun}[\tau_1;\tau_2](x.y.e):\mathtt{parr}(\tau_1;\tau_2)} \quad (16.5)$$

4.4.3 Plotkin的PCF-PCF的可定义性-2

- 一般递归函数 fun[τ_1; τ_2]($x.y.e$)
  x 是代表函数本身的变量，y 是函数的参数

动态语义

$$\frac{\{e_1 \ \mathsf{val}\} \quad e = \mathtt{fun}[\tau_1;\tau_2](x.y.e')}{\mathtt{ap}(e;e_1) \mapsto [e, e_1/x, y]e'} \qquad (16.6)$$

在函数调用时，用函数本身代换函数体中的x

- 一般递归函数可由一般递归式和非递归函数定义
  fun[τ_1; τ_2]($x.y.e$) = fix[parr(τ_1; τ_2)](x.lam[τ_1]($y.e$))
- 原始递归式在PCF中是可定义的
  rec[τ](e; e_0; $x.y.e_1$) = ap(e' ; e)
  其中 e' = fun[nat; τ]($f.u$.ifz(u; e_0; x.[ap(f; x)/y]e_1))

4.4.3 Plotkin的PCF-PCF的可定义性-3

- 原始递归式在PCF中是可定义的
  rec[τ](e; e_0; $x.y.e_1$) = ap(e' ; e)
  其中 e' = fun[nat; τ]($f.u$.ifz(u; e_0; x.[ap(f; x)/y]e_1))

例：阶乘函数

Gödel的T－抽象语法表示

fct = lam [nat] (n.rec [arr(nat:nat)]
(n; s(z); $x.y$.times(s(x); y))

Plotkin的PCF－抽象语法表示

fct' = lam [nat] (n.ap(fun[nat; nat]
($f.u$.ifz(u; s(z); x.[ap(f; x)/ y] times (s(x); y))); n))
= lam [nat](n.ap(fix[parr(nat;nat)](f.lam[nat]
(u.ifz(u; s(z); x.[ap(f; x)/ y] times (s(x); y)))); n))

4.4.3 Plotkin的PCF-PCF的可定义性-4

- 观测同余(观测等价) $e \cong e' : \tau \ [\Gamma]$
  - 一致性：能终止的表达式和不能终止的表达式不等价
  - 同余性(congruence):对于任何的子表达式，用一个等价的表达式代替该子表达式，则新表达式仍和源表达式等价。
- 可定义性
  - 自然数的部分函数ϕ:N ⇀ N在PCF中是可定义的，当且仅当存在类型为nat ⇀ nat的表达式e_ϕ，使得
    $\phi(m)=n$当且仅当 $e_\phi(\overline{m}) \cong \overline{n} : \mathsf{nat}$
  - 如果ϕ完全没有定义，则e_ϕ必须是一个会导致死循环的函数

4.4.4 递归类型-1

- 递归类型示例：表
  - 用二元和表示表 natlist = nil:unit+ cons: nat × natlist
    上述等式蕴涵着递归定义。
  - 对类型引入一个明确的递归操作符μ
    natlist = μt. nil:unit+cons: nat × t
    读作"将natlist定义为满足t = nil:unit+cons: nat × t 的无穷的类型"
- 递归类型形式化的方法
  $\mu t.\tau$和其展开[$\mu t.\tau/t$]τ 之间的关系是什么?
  - 相等递归(equi-recursive)：将这两个类型表达式作为相同的定义。这种方法使类型表达式可以为无穷。
  - 同构递归(iso-recursive)：将一个递归类型和其展开式视为是不同的，但是二者是同构的。
    递归类型$\mu t.\tau$的展开式是用该递归类型代换τ中出现的t，即[$\mu t.\tau/t$]τ

4.4.4 递归类型-2

- 同构递归类型
  - 举例(具体语法表示)
    - 递归类型：natlist = μt. nil:unit+cons: nat × t
    - 展开式：nil:unit+cons: nat × μt.(nil:unit+ cons: nat × t)
  - 语法
    Types τ ::= t | rec($t.\tau$) t是关于类型名的元变量
    Expr's e ::= fold[$t.\tau$](e) | unfold(e)

抽象语法	具体语法	
rec($t.\tau$)	$\mu t.\tau$	递归类型，其展开式为[rec($t.\tau$) /t]τ
fold[$t.\tau$](e)	fold(e)	引入形式：折叠， e : [rec($t.\tau$) /t]τ
unfold(e)	unfold(e)	消去形式：展开， e : rec($t.\tau$)

4.4.4 递归类型-3

- 同构递归类型
  - 静态语义
    - 一般断言：$\Delta \mid \tau$ type
      Δ是一组有限的形如t_i type的假设集合
      t_i为类型变量

$$\frac{}{\Delta, t \ \mathsf{type} \mid t \ \mathsf{type}}$$

$$\frac{\Delta \mid \tau_1 \ \mathsf{type} \quad \Delta \mid \tau_2 \ \mathsf{type}}{\Delta \mid \mathtt{arr}(\tau_1;\tau_2) \ \mathsf{type}} \qquad (21.1)$$

$$\frac{\Delta, t \ \mathsf{type} \mid \tau \ \mathsf{type}}{\Delta \mid \mathtt{rec}(t.\tau) \ \mathsf{type}}$$

    - 定型断言：$\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Gamma \vdash \mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)}$$

$$\frac{\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \qquad (21.2)$$

4.4.4 递归类型-4

❖ 同构递归类型

➢ 动态语义

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{fold}[t.\tau](e)\ \mathsf{val}}$$

$$\left\{\frac{e \mapsto e'}{\mathtt{fold}[t.\tau](e) \mapsto \mathtt{fold}[t.\tau](e')}\right\}$$

$$\frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')}$$

$$\frac{\{e\ \mathsf{val}\}}{\mathtt{unfold}(\mathtt{fold}[t.\tau](e)) \mapsto e}$$

(21.3)

eager语义下, e是值，则其fold形式是值

eager语义下,e未完成求值,则允许在fold下归约

e未完成求值，则允许在unfold下归约

eager语义下,对值e执行fold再unfold，所得为e

•惰性(lazy)语义 省去{ }中的规则或前提

•急切(eager)语义 包含{ }中的规则或前提

➢ 安全性（略） (Theorem 21.1)

4.4.4 递归类型-5

❖ 同构递归类型

➢ 举例(抽象语法表示)

natlist = sum(nil:unit; cons:prod(nat;natlist))

– 展开式: natlist ≅ sum(nil:unit; cons:prod(nat;natlist))

– 则 nil = fold[natlist] (in[nil](triv))

cons = lam[nat](*n*.
lam[natlist](*l*.
fold[natlist](in[cons]pair(*n*;*l*)))

判空isnil = lam[natlist](*l*. case (unfold(*l*); *u*.tt; *p*.ff))

取表头head = lam[natlist](*l*.case(unfold(*l*); *u*.*z*; *p*.fst(*p*))

作业

❖ 4.1 试用4.1.6和4.1.7中介绍的求值语义和两种环境语义，对twicef 3 (plus 4)求值，其中

twicef = $\lambda\, n$: nat. $\lambda\, f$: nat→nat. f (f n)

plus = λx : nat. λy : nat. x+y

要求：写出求值所依赖的规则和相应的推导结果

作业

❖ 4.2 为语言T{list}(在Gödel的T上增加list类型)定义静态语义和动态语义,并证明安全性。

➢ 语法

τ ::= nat	nat	
| arr(τ_1, τ_2)	$\tau_1 \rightarrow \tau_2$	
| list(τ)	τ list	//元素类型为τ的list
e ::= x | z | s(e)	x | zero | s(e)	
| natrec [τ](e; e_0; x. y.e_1)	natrec e { z=>e_0 | s(x) with y =>e_1}	
| lam[τ](x.e)	λ(x.e)	
| ap(e_1; e_2)	e_1(e_2)	
| nil[τ]	nil[τ]	
| cons(e_1; e_2)	e_1::e_2	
| listrec[τ](e; e_{nil}; x.xs.y.e_{cons})	listrec e { nil=>enil | x::xs with y =>econs}	

作业

❖ 4.2 为语言T{list}(在Gödel的T上增加list类型)定义静态语义和动态语义,并证明安全性。

➢ 值

$$\frac{}{\mathrm{z\ val}} \qquad \frac{e\ \mathrm{val}}{\mathrm{s}(e)\ \mathrm{val}} \qquad \frac{}{\mathrm{lam}[\tau](x.e)\ \mathrm{val}}$$

$$\frac{}{\mathrm{nil}[\tau]\ \mathrm{val}} \qquad \frac{e_1\ \mathrm{val} \quad e_2\ \mathrm{val}}{\mathrm{cons}(e_1;e_2)\ \mathrm{val}}$$

Thanks!