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

December, 2007

第八章 多态(Polymorphism)

8.1 多态[PFPL, 23]

对参数化多态函数(**generic function**) 形式化，函数带类型参数.

8.2 数据抽象[PFPL, 24]

用存在类型形式化数据抽象(接口)，如元素类型为 τ 的队列可以有多种实现(队列的表示及实现).

8.3 构造子和种类[PFPL, 25]

对**generic class**(即带类型参数的类型 / 类)进行形式化，如支持任意元素类型t的列表等.

8.1 多态

在L{nat →}和PCF及其扩展语言中

- 每一表达式最多有一个类型
- 函数有唯一确定的定义域类型和值域类型

每一类型都有一个不同的恒等函数 $id_\tau = \lambda(x:\tau.x)$

针对每一个类型三元组，就有一个不同的复合函数

$$\circ_{\tau_1,\tau_2,\tau_3} = \lambda(f:\tau_2 \to \tau_3.\lambda(g:\tau_1 \to \tau_2.\lambda(x:\tau_1.f(g(x)))))$$

这些函数功能一样，只是类型不太一样!

引入多态来避免为不同的类型重复定义功能相同的函数：以类型为参数，即函数是类属的(泛型的generic)、参数化的(parametric)——参数化多态 (parametric polymorphism)(本章讨论的主题)

其他形式的多态

Ad-hoc多态：允许一个多态值在遇到不同的类型时能产生不同的操作，如函数重载(overloading)

8.1 多态

8.1 多态λ演算L{→ ∀ } (又称系统F)

8.2 多态的可定义性

多态λ演算可以定义原始递归函数

8.3 多态的受限形式

多态λ演算是不可预言的，这里给出可预言的子集,称为前束片段

8.1.1 多态λ演算-语法

多态λ演算L{→ ∀ }是具有多态定型的最小的函数式语言！

❖ 语法

Type $\tau ::= t \mid \mathrm{arr}(\tau_1, \tau_2) \mid \mathrm{all}(t.\tau)$

Expr $e ::= x \mid \mathrm{lam}[\tau](x.e) \mid \mathrm{ap}(e_1, e_2) \mid \mathrm{Lam}(t.e) \mid \mathrm{App}[\tau](e)$

- t：元变元，取值范围为一组类型名(类型变元)
- x：元变元，取值范围为一组表达式名(表达式变元)
- $\mathrm{all}(t.\tau)$：全称类型(universal type)，是以t为参数、以τ为结果的函数
- $\mathrm{Lam}(t.e)$：类型抽象，定义一个带有类型参数t的参数化函数，是$\mathrm{all}(t.\tau)$的引入形式
- $\mathrm{App}[\tau](e)$：类型应用或实例化(instantiation)，将一个多态函数应用到具体的类型上，是$\mathrm{all}(t.\tau)$的消去形式

8.1.1 多态λ演算-语法

- 语法

抽象语法	具体语法
all($t.\tau$ **)**	$\forall(t.\tau)$
Lam(*t.e***)**	**Λ(***t.e***)**
App[τ**](***e***)**	***e*[**τ**]**

- 静态语义
  - 直言断言
    - τ **type**： τ是一个良类型**(**类型形成断言**)**
    - ***e*** **:** τ ： ***e***是类型为τ的良形表达式**(**表达式形成断言**)**
  - 上述直言断言采用 $\mathcal{T} \mid \Delta \vdash \tau \text{ type}$和 $\mathcal{T}\ \mathcal{X} \mid \Delta\ \Gamma \vdash e : \tau$ 断言来定义
    - $\mathcal{T}$ **:**一组形如***t*** **cons**的类型构造子变元声明的有限集
    - $\mathcal{X}$ **:**一组形如***x*** **exp**的表达式变元声明的有限集
    - Δ **:** 一组形如***t*** **type**的假设，使得 $\mathcal{T} \vdash t \text{ cons}$
    - Γ **:** 一组形如***x*** **:** τ 的假设，其中 $\mathcal{X} \vdash x \text{ exp}$, $\mathcal{T} \mid \Delta \vdash \tau \text{ type}$

    一般地，省去$\mathcal{T}$和$\mathcal{X}$，因为它们可以从假设Δ和Γ重新获得

8.1.1 多态λ演算-静态语义

- 静态语义
  - 类型的形成规则

$$\frac{}{\Delta, t\ \mathtt{type} \vdash t\ \mathtt{type}}$$

$$\frac{\Delta \vdash \tau_1\ \mathtt{type} \quad \Delta \vdash \tau_2\ \mathtt{type}}{\Delta \vdash \mathtt{arr}(\tau_1, \tau_2)\ \mathtt{type}}$$

$$\frac{\Delta, t\ \mathtt{type} \vdash \tau\ \mathtt{type}}{\Delta \vdash \mathtt{all}(t.\tau)\ \mathtt{type}}$$

(PFPL 23.1)

  - 表达式的定型规则

$$\frac{\Delta, t\ \mathtt{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\tau)}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau') \quad \Delta \vdash \tau\ \mathtt{type}}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\tau'}$$

(PFPL 23.2)

8.1.1 多态λ演算-举例

❖ 举例

- 多态复合函数

具体语法表示 $\Lambda(t_1.\Lambda(t_2.\Lambda(t_3.\lambda(f:t_2\to t_3.\lambda(g:t_1\to t_2.\lambda(x:t_1.f(g(x)))))))))$

抽象语法表示

$\mathrm{Lam}(t_1.\mathrm{Lam}(t_2.\mathrm{Lam}(t_3.\mathrm{lam}[\mathrm{arr}(t_2,t_3)](f.\mathrm{lam}[\mathrm{arr}(t_1,t_2)](g.\mathrm{lam}[t_1](x.f(g(x)))))))))$

- 多态复合函数的类型

具体语法表示 $\forall(t_1.\forall(t_2.\forall(t_3.(t_2\to t_3)\to(t_1\to t_2)\to t_1\to t_3)))$

等同于 $\forall(t_1.\forall(t_2.\forall(t_3.(t_2\to t_3)\to(t_1\to t_2)\to(t_1\to t_3))))$

抽象语法表示

$\mathrm{all}(t_1.\mathrm{all}(t_2.\mathrm{all}(t_3.\mathrm{arr}(\mathrm{arr}(t_2,t_3),\ \mathrm{arr}(\mathrm{arr}(t_1,t_2),\mathrm{arr}(t_1,t_3))))))$

8.1.1 多态λ演算-置换引理

❖ 置换引理(PFPL Lemma 23.1)

1.如果$\Delta, t\ \text{type} \vdash \tau'\ \text{type}$且$\Delta \vdash \tau\ \text{type}$，则$\Delta \vdash [\tau/t]\tau'\ \text{type}$.

2.如果$\Delta, t\ \text{type}\ \Gamma \vdash e' : \tau'$且$\Delta \vdash \tau\ \text{type}$，则$\Delta\ [\tau/t]\Gamma \vdash [\tau/t]e' : [\tau/t]\tau'$.

3.如果$\Delta\ \Gamma, x : \tau \vdash e' : \tau'$且$\Delta \vdash e : \tau$，则$\Delta\ \Gamma \vdash [e/x]e' : \tau'$.

注意：第1条是关于类型形成断言的置换。

第2,3条是关于表达式形成断言的置换。其中第2条中的Γ、e'和τ'可以包含自由出现的类型变元，故需要对Γ、e'和τ'置换。

证明：1.对$\Delta, t\ \text{type} \vdash \tau'\ \text{type}$归纳证明。需要考虑$\tau'$的每一种可能情况(针对(PFPL 23.1)的每一条规则)，这里只考虑(23.1b):

设τ'为arr(τ_1, τ_2)，由归纳假设有$\Delta \vdash [\tau/t]\tau_1\ \text{type}$和$\Delta \vdash [\tau/t]\tau_2\ \text{type}$

由置换定义有[τ/t]arr(τ_1, τ_2)=arr([τ/t]τ_1, [τ/t]τ_2).

再由(23.1b)，有 $\Delta \vdash [\tau/t]\text{arr}(\tau_1, \tau_2)\ \text{type}$

8.1.1 多态λ演算-动态语义

- 动态语义
  - 增加如下值规则

$$\frac{}{\mathtt{Lam}(t.e)\ \mathtt{val}}$$

类型抽象是值

  - 增加如下转换规则

$$\frac{}{\mathtt{App}[\tau](\mathtt{Lam}(t.e)) \mapsto [\tau/t]e}$$

对类型抽象进行类型应用，则对e中的类型变元进行置换

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

e未完成求值时，则允许在类型应用下归约

(PFPL 23.3)

- 范式引理(PFPL Lemma 23.2)

假设$e : \tau$且e val，则：

1.如果τ=arr(τ_1, τ_2)，则e=lam[τ_1]($x.e_2$)，其中$x : \tau_1 \vdash e_2 : \tau_2$.

2.如果τ=all($t.\tau'$)，则e=Lam($t.e'$)，其中$t\ \mathrm{type} \vdash e' : \tau'$.

证明：由定型规则和值规则可证明。例如，由(23.2a)和(23.3a)可证明上述的2。

8.1.1 多态λ演算-安全性

- 保持性定理(PFPL Theorem 23.3)

  如果$e:\tau$且$e \mapsto e'$，则$e':\tau$.

  证明：对转换规则归纳证明。

- 进展性定理(PFPL Theorem 23.4)

  如果$e:\tau$，则或者e val，或者存在e'使得$e \mapsto e'$.

  证明：对定型规则归纳证明。

8.1.2 多态的可定义性-1

多态λ演算的每一个良类型表达式都可以求得为值。

- 用L{→ ∀}可以定义原始递归函数
  - 原始递归函数的结构：nat、z、s(*e*)、iter[τ](e_0, e_1, *x*.e_2)
    (rec[τ] (e_0, e_1, *x*.*y*.e_2))

回顾:在第4章(PFPL 20)讨论了用未类型化λ演算表示PCF

Church数值　　z = λ*b*.λ*s*.*b*　　s = λ*x*.λ*b*.λ*s*.*s* (*x* (*b*) (*s*))

ifz(*u*, u_0, *x*.u_1) = *u* (u_0) (λ*x*.u_1)

如何求*u*的前驱?

tt = λ*x*.λ*y*.*x*　　ff = λ*x*.λ*y*.*y*

pair = λ*f*.λ*s*.λ*p*. *p*(*f*)(s)　　<u_1, u_2> = pair(u_1)(u_2) = λ*p*.*p*(u_1)(u_2)

fst = λ*p*.*p* (tt)　　fst(*u*) = *u* (tt) = *u* (λ*x*.λ*y*.*x*)

snd = λ*p*.*p* (ff)　　snd(*u*) = *u* (ff) = *u* (λ*x*.λ*y*.*y*)

prdpair = λ*x*.*x* (pair($\bar{0}$, $\bar{0}$)) (λ*y*. <snd(*y*), s(snd(*y*))>)

pred = λ*x*.fst(prdpair(*x*))

8.1.2 多态的可定义性-2

- 用L{→∀}可以定义原始递归函数
  - L{→∀}是类型化语言，它包含函数类型和全称类型
    用→和∀表示nat!
  - 用L{→∀}表示原始递归函数的关键：考虑iter[τ](e_0, e_1, $x.e_2$)的定型

$$\frac{e_0 : \mathtt{nat} \quad e_1 : \tau \quad x : \tau \vdash e_2 : \tau}{\mathtt{iter}[\tau](e_0, e_1, x.e_2) : \tau}$$

由于τ是任意的，故迭代式的类型为

$\mathbf{nat} \rightarrow \forall t.(t \rightarrow (t \rightarrow t) \rightarrow t)$

$t \rightarrow (t \rightarrow t) \rightarrow t$ 中，第1个t表示e_1的类型，最后1个t表示迭代式的结果类型， $t \rightarrow t$表示根据x在迭代式的结果来求$s(x)$的结果的函数类型。

自然数上的唯一操作是上述的迭代，故可用多态函数来标识自然数n

- $\mathbf{nat} = \forall t.(t \rightarrow (t \rightarrow t) \rightarrow t)$
- $z = \Lambda(t.\lambda(z{:}t.\lambda(s{:}t \rightarrow t.z)))$ $\quad s(e) = \Lambda(t.\lambda(z{:}t.\lambda(s{:}t \rightarrow t.s(e[t](z)(s)))))$
- $\mathbf{iter}[\tau](e_0, e_1, x.e_2) = e_0[\tau](e_1)(\lambda(x{:}t.e_2))$

8.1.2 多态的可定义性-3

- 用L{→∀}可以定义原始递归函数的正确性
  - 静态语义：可由定义直接检查得证
  - 动态语义
    - 自然数 **z**：从**z**开始按**s** 迭代，迭代**0**次
    - **s(*e*)**：***e***是从**z**开始迭代**s**共***e*** 次，之后再迭代**s** **1**次
    - $\overline{n}$：表示复合**s(…s(z)….)**的***n***次折叠形式

    假设对函数应用按**call-by-value**语义，可以对***n*** 归纳证明：

    $$\overline{n}[\tau](e_1)(e_2) \mapsto^* e_2(\ldots e_2(e_1)\ldots)$$

    - $z[\tau](e_1)(e_2) \mapsto^* e_1$
    - 如果 $\overline{n}[\tau](e_1)(e_2) \mapsto^* e$，则 $s(\overline{n})[\tau](e_1)(e_2) \mapsto^* e_2(e)$

    由此可证明动态语义可正确地由这些定义来模拟！
  - 举例 加法：**λ(*x*:nat.λ(*y*:nat.*y*[nat](*x*)(λ(*z*:nat.s(*z*)))))**

8.1.2 多态的可定义性-4

- 用L{→∀}定义其他类型

$$\begin{aligned}
\texttt{unit} &= \forall(t.t \to t) \\
\tau_1 \times \tau_2 &= \forall(t.(\tau_1 \to \tau_2 \to t) \to t) \\
\texttt{void} &= \forall(t.t) \\
\tau_1 + \tau_2 &= \forall(t.(\tau_1 \to t) \to (\tau_2 \to t) \to t) \\
\tau\ \texttt{list} &= \forall(t.t \to (\tau \to t \to t) \to t)
\end{aligned}$$

8.1.3 多态的受限形式-1

- 不可预言性(impredicativity)
  - L{→ ∀}是不可预言的

    例如，若$\tau = \forall(t.t \to t)$，类型变量$t$的范围是所有类型，包括$\tau$本身
  - 可预言的量化：限制量词的取值范围是单一类型，该类型不含量词
- 前束片段(prenex fragment)
  - 前束量化：限制多态只出现在最外层：不仅是量词可预言的，而且量词不允许出现在任何其他类型构造子的参数中。

    ML中的多态就是前束量化，以便进行类型推断。
  - L{→ ∀}的前束片段可通过将类型分成单型和多型两层来得到

    Monotype $\tau ::= t \mid \mathrm{arr}(\tau_1, \tau_2)$

    Polytype $\sigma ::= \tau \mid \mathrm{all}(t.\tau)$

    多型总是形如：$\forall(t_1. \ldots \forall(t_n.\tau)\ldots)$

8.1.3 多态的受限形式-2

- L{→∀}的前束片段
  - 静态语义

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{arr}(\tau_1, \tau_2)}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \sigma}{\Delta\ \Gamma \vdash \mathtt{Lam}(t.e) : \mathtt{all}(t.\sigma)}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type} \quad \Delta\ \Gamma \vdash e : \mathtt{all}(t.\sigma)}{\Delta\ \Gamma \vdash \mathtt{App}[\tau](e) : [\tau/t]\sigma}$$

(PFPL 23.4)

可以用任何自由类型变元来一般化，也可以实例化任何带量词的多型.

每个单型都是一个多型. 应用上述这些规则，初始时将表达式定型为某个单型，然后再对其中的自由类型变元归纳

  - 加入Let结构的类型规则，可得ML类型系统的核心

$$\frac{\Delta\ \Gamma \vdash e_1 : \sigma_1 \quad \Delta\ \Gamma, x : \sigma_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{let}[\sigma_1](e_1, x.e_2) : \tau_2}$$

(PFPL 23.5)

8.2 数据抽象

数据抽象的基本思想：将客户机程序和抽象的实现通过接口分隔开. 当实现发生变化但接口不变时，就不会改变客户机程序的行为。

数据抽象可以通过在L{→ ∀}中扩展存在类型(existence types)来形式化.

- 接口用存在类型来建模，提供在抽象类型上的一组操作。
- 实现用存在类型的引入形式——包(packages)来建模
- 客户机程序用存在类型的消去形式来建模

存在类型与全称类型密切联系，经常被一起处理:

- 二者都是类型量化形式，都需要类型变元机制
- 存在类型可从全称类型定义，数据抽象实际是多态的一种形式

8.2.1 存在类型

8.2.2 通过存在类型进行数据抽象

8.2.3 存在类型的可定义性

8.2.1 存在类型-1

- L{→ ∀ ∃}的语法：在L{→ ∀}上扩展以下结构

Type τ **::=** **some(**$t.\tau$**)**

Expr e **::=** **pack[**$t.\tau;\rho$**](**e**) | open[**$t.\tau$**](**$e_1;t, x.e_2$**)**

- **some(**$t.\tau$**)**：存在类型，即接口
- **pack[**$t.\tau;\rho$**](**e**)**：存在类型的引入形式，即包。ρ是包的表示类型；e是类型为$[\rho/t]\tau$的表达式，称为包的实现.
- **open[**$t.\tau$**](**$e_1;t, x.e_2$**)**：存在类型的消去形式，即客户机程序。将包e_1解开并在客户机程序e_2中使用，e_1的表示类型绑定到t，e_1的实现绑定到x.

抽象语法	具体语法
some($t.\tau$**)**	$\exists(t.\tau)$
pack[$t.\tau;\rho$**](**e**)**	**pack** ρ **with** e **as** $\exists(t.\tau)$
open[$t.\tau$**](**$e_1;t, x.e_2$**)**	**open** e_1 **as** t **with** $x:\tau$ **in** e_2

8.2.1 存在类型-静态语义

❖ 静态语义

$$\frac{\Delta, t\ \text{type} \vdash \tau\ \text{type}}{\Delta \vdash \text{some}(t.\tau)\ \text{type}}$$

$$\frac{\Delta \vdash \rho\ \text{type} \quad \Delta, t\ \text{type} \vdash \tau\ \text{type} \quad \Delta\ \Gamma \vdash e : [\rho/t]\tau}{\Delta\ \Gamma \vdash \text{pack}[t.\tau;\rho]\,(e) : \text{some}(t.\tau)}$$

(PFPL 24.1)

$$\frac{\Delta\ \Gamma \vdash e_1 : \text{some}(t.\tau) \quad \Delta, t\ \text{type}\ \Gamma, x : \tau \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \text{type}}{\Delta\ \Gamma \vdash \text{open}[t.\tau]\,(e_1; t, x.e_2) : \tau_2}$$

- 对于第3条，
  - 客户机程序的类型τ_2一定不含抽象类型t。它防止客户机程序试图将抽象类型的值输出到其定义的作用域之外。
  - 客户机程序的体e_2在不知道表示类型t的情况下被类型检查。结果，客户机程序是多态的，它以类型变元t为参数。

8.2.1 存在类型-动态语义

❖ 动态语义

$$\frac{\{e\ \mathsf{val}\}}{\mathsf{pack}[t.\tau;\rho](e)\ \mathsf{val}}$$

$$\left\{\frac{e \mapsto e'}{\mathsf{pack}[t.\tau;\rho](e) \mapsto \mathsf{pack}[t.\tau;\rho](e')}\right\}$$

(PFPL 24.2)

$$\frac{e_1 \mapsto e_1'}{\mathsf{open}[t.\tau](e_1;t,x.e_2) \mapsto \mathsf{open}[t.\tau](e_1';t,x.e_2)}$$

$$\frac{e\ \mathsf{val}}{\mathsf{open}[t.\tau](\mathsf{pack}[t.\tau;\rho](e);t,x.e_2) \mapsto [\rho,e/t,x]e_2}$$

在运行时没有抽象类型！

- 表示类型在求值期间已完全暴露给客户机程序。
- 数据抽象是编译时的规定，没有运行时开销。

8.2.1 存在类型-安全性

- 保持性 (PFPL Theorem 24.1)

  如果$e:\tau$且$e \mapsto e'$，则$e':\tau$.

  证明：对转换规则归纳证明。

- 范式引理(PFPL Lemma 24.2)

  假设$e:\text{some}(t.\tau)$且e val，则$\text{pack}[t.\tau;\rho](e')$，其中存在某一类型$\rho$和某$e'$ val，使得$e':[\rho/t]\tau$.

  证明：由定型规则和值规则可证明。

- 进展性定理(PFPL Theorem 24.3)

  如果$e:\tau$，则或者e val，或者存在e'使得 $e \mapsto e'$.

  证明：对定型规则归纳证明。

8.2.2 通过存在类型进行数据抽象-1

例：抽象队列，支持三种操作

- 空队列的构造
- 在队尾插入一个元素
- 删去队头

队列元素可以是任一类型τ。

抽象队列的定义

- 抽象队列的接口 $\exists(t.\langle \text{emp: } t, \text{ ins: } \tau \times t \to t, \text{ rem: } t \to \tau \times t \rangle)$
  $\text{some}(t.\text{rcd}[\text{emp,ins,rem}](t, \text{arr}(\text{prod}(\tau, t), t), \text{arr}(t, \text{prod}(\tau, t))))$
- 队列的表示类型t是抽象的，它支持emp、ins、rem操作
- 队列的实现由指定表示类型的包、以及该表示所关联的操作的实现组成。
  在实现内部，队列的表示是已知的，操作依赖于表示。

8.2.2 通过存在类型进行数据抽象-2

例：抽象队列，支持三种操作

- 抽象队列的接口 $\exists(t.\langle emp: t, ins: \tau \times t \to t, rem: t \to \tau \times t \rangle)$
  $some(t.rcd[emp,ins,rem](t, arr(prod(\tau, t), t), arr(t, prod(\tau, t))))$

队列的一个简单实现e_l：将队列表示成一个list

- $pack\ \tau\ list\ with \langle emp=nil, ins=e_i, rem=e_r \rangle\ as\ \exists(t.\sigma)$
  $pack[t.\sigma; list(\tau)](rcd[emp, ins, rem](nil[\tau], e_i, e_r))$

  - $e_i: \tau \times \tau\,list \to \tau\,list = \lambda(x:\tau \times \tau\,list.e'_i)$,
    $e_i: arr(prod(\tau, list(\tau)), list(\tau)) = lam[prod(\tau, list(\tau))](x.e'_i)$,
    e'_i将x的第1个成员(即元素)联接到x的第2个成员(即队列)上

注意：这里利用list的cons操作完成插入，故所得的表是队列的逆置

8.2.2 通过存在类型进行数据抽象-3

例：抽象队列，支持三种操作

- 抽象队列的接口 $\exists(t.\langle emp: t, ins: \tau \times t \to t, rem: t \to \tau \times t \rangle)$
  $some(t.rcd[emp,ins,rem](t, arr(prod(\tau, t), t), arr(t, prod(\tau, t))))$

队列的一个简单实现e_l：将队列表示成一个list

- $pack\ \tau\ list\ with \langle emp=nil, ins=e_i, rem=e_r \rangle\ as\ \exists(t.\sigma)$
  $pack[t.\sigma; list(\tau)](rcd[emp, ins, rem](nil[\tau], e_i, e_r))$

  - $e_r : \tau\,list \to \tau \times \tau\,list = \lambda(x:\tau\,list.e'_r)$.
    $e_r : arr(list(\tau), prod(\tau, list(\tau))) = lam[list(\tau)](x.e'_r)$.
    e'_r将x逆置，然后返回队头元素和队尾的逆置

在实现这些操作时，知道队列表示成类型为$\tau\,list$的值。

8.2.2 通过存在类型进行数据抽象-4

队列的另一个实现e_p：将队列表示为list的二元组，即由实际队列的后一部分的逆置(如$a_n,\ldots,a_{i+1}$)以及前一部分(如$a_1,\ldots,a_i$)组成。

这种表示避免了在每次调用都需要执行逆置，使得通过分摊可达到常量时间的复杂度。

- pack τ list × τ list with <emp=<nil,nil>, ins=e_i, rem=e_r> as $\exists(t.\sigma)$
  pack[$t.\sigma$; prod(list(τ), list(τ))]
  (rcd[emp, ins, rem](pair(nil[τ], nil[τ]), e_i, e_r))

  - e_i: τ × (τ list × τ list) → (τ list × τ list)
    e_i: arr(prod(τ, prod(list(τ), list(τ))), prod(list(τ), list(τ)))
    将参数的第1个成员联接到第2个成员中的第1成员。

8.2.2 通过存在类型进行数据抽象-5

队列的另一个实现e_p：将队列表示为list的二元组，即由队列的后一部分以及前一部分的逆置组成。

这种表示避免了在每次调用都需要执行逆置，使得通过分摊可达到常量时间的复杂度。

- pack τ list × τ list with <emp=<nil,nil>, ins=e_i, rem=e_r> as $\exists(t.\sigma)$
  pack[$t.\sigma$; prod(list(τ), list(τ))]
  (rcd[emp, ins, rem](pair(nil[τ], nil[τ]), e_i, e_r))

  - e_r : (τ list × τ list) → τ × (τ list × τ list)
    e_r : arr(prod(list(τ), list(τ)), prod(τ, prod(list(τ), list(τ)))).

当参数的第2个成员不为空时，则直接取得表中的第1个元素(即队头)并将参数的第1个成员和第2个成员的剩余部分组织成二元组返回。

若参数的第2个成员为空，则可取出第1个成员并将其逆置放到第2个成员，再进行删除。——大幅度降低了逆置操作的频度！

8.2.2 通过存在类型进行数据抽象-6

例：抽象队列

抽象队列的客户机程序通过open结构与实现细节相隔离

- 抽象队列的客户机程序

  open e **as** t **with** $x{:}\sigma$ **in** $e'{:}\tau'$

  open$[t.\sigma](e;t, x.e')$

  e'的类型τ'不含抽象类型t.

  $\sigma = \langle \text{emp: } t, \text{ ins: } \tau \times t \to t, \text{ rem: } t \to \tau \times t \rangle$,其中$t$仍是抽象的

注意：只将存在类型中的类型信息传播到客户机程序，供类型检查；对于客户机程序在类型检查时不关心表达式e的具体实现。这个性质称为表示无关(representation independence)。

8.2.3 存在类型的可定义性-1

存在类型的引入原因：对数据抽象建模

存在类型可由全称类型定义！

- 抽象类型的客户机程序是以表示类型t为参数的多态

  open e as t with $x:\tau$ in $e':\tau'$　　**open[$t.\tau$](e;t, $x.e'$)**

  – 其中$e:\exists(t.\tau)$，该open表达式的定型规则表示$e':\tau'$是以t type和$x:\tau$为假设的

  – 客户机程序本质上是类型为$\forall(t.\tau\rightarrow\tau')$的多态函数，$t$可能出现在$\tau$中，但不会出现在$\tau'$中。

- 存在类型及其操作可以定义如下

  – $\exists(t.\tau)=\forall(t'.\forall(t.\tau\rightarrow t')\rightarrow t')$

  some($t.\tau$) = all(t', arr(all(t.arr(τ, t')), t'))

  存在类型是一个多态函数类型，该类函数以结果类型t'为参数，后跟表示客户机程序的、结果类型为t'的多态函数，产生类型为t'的值作为整个结果。

8.2.3 存在类型的可定义性-2

- $\exists(t.\tau) = \forall(t'.\forall(t.\tau \to t') \to t')$
  some($t.\tau$) = all(t', arr(all(t.arr(τ, t')), t'))
  存在类型是一个多态函数类型，该类函数以结果类型t'为参数，后跟表示客户机程序的、结果类型为t'的多态函数，产生类型为t'的值作为整个结果。
- **pack ρ with e as $\exists(t.\tau) = \Lambda(t'.\lambda(x{:}\forall(t.\tau \to t').x[\rho](e)))$**
  pack[$t.\tau$;ρ](e) = Lam(t'.lam[all(t.arr(τ, t'))](x.ap(App[ρ](x), e)))
  由表示类型ρ和实现e组成的包是一个多态函数，该函数在给定结果类型t'和客户机程序x时，将用ρ实例化x，再应用到实现e上。
- **open e as t with $x{:}\tau$ in e' = $e[\tau'](\Lambda(t.\lambda(x{:}\tau.e')))$**
  open[$t.\tau$](e;t, $x.e'$) = ap(App[τ'](e), Lam(t.lam[τ]($x.e'$)))
  将客户机程序打包成一个多态函数Lam(t.lam[τ]($x.e'$))，将存在类型的结果类型(即$\forall(t'.\forall(t.\tau \to t') \to t')$中的$t'$) 实例化为$\tau'$ (注意e最终为一个pack值，即多态函数Lam(t'.lam…))，再将App[τ'](e)应用到多态客户机程序上。

8.3 构造子(Constructors)和种类(kinds)

在类型上的量化：全称类型(对一般性建模)、存在类型(对抽象建模).

只有类型量化不足以对许多实际的编程情况建模。例如，

- 类型的抽象族(abstract families of types),如τ list
  同时引入一个无限的类型集合，这些类型共享一组公共的操作集.
  全称类型：参数多态　存在类型：抽象出操作的接口
- 如何同时引入两个相互关联的抽象类型,如树类型和森林类型
  树中的结点有多个孩子(可看成是森林)，森林中的元素是树.

这就要求允许量化在种类(kinds)上建模，而不只是在类型上建模。例如，

- 类型构造子(type constructors)：是将类型映射到类型的函数
- 类型结构(type structures)：本质上是类型元组

在语言中，用更高级的kinds来对构造子分类，这就好比用类型来对表达式分类。

类型本身以某种构造子形式出现，这种构造子的种类(kind)称为Type.

8.3 构造子和种类

引入更高级的kinds后，就可以在任意kind上进行全称或存在量化，量化的初始形式是在Type上进行量化。

"在kind上量化和在type上量化"这种两层结构可以对2.7.3[PFPL, 12]中的阶段上的区别(phase distinction)进行建模:

- 构造子和kind级别形成静态层：静态层提供定义语言的静态语义所需要的设施，如类型的类别(以某种构造子形式出现)
- 表达式和type级别形成动态层：动态层定义语言的设施，如函数、数据结构，它们在运行时计算

构造子是语言的静态数据，表达式是语言的动态数据。

8.3.1 类型族语言L{→ $\forall_K$ $\exists_K$} 的静态语义

8.3.2 表达式的形成

8.3.3 区分构造子与类型

8.3.4 动态语义

8.3.1 类型族语言的静态语义-1

- 抽象语法

Kind K ::= Type | Prod(K_1, K_2) | Arr(K_1, K_2)

Cons C ::= t | arr | all[K] | some[K] | pair(C_1, C_2) | fst(C) | snd(C) | lambda[K]($t.C$) | app(C_1, C_2)

Type τ ::= C

Expr e ::= x | lam[τ]($x.e$) | ap(e_1, e_2) | Lam[K]($t.e$) | App[C](e) | pack[K, C, C'](e) | open[K, C](e_1, t, $x.e_2$)

  - 前两个语法范畴K和C组成了语言的静态层；后两个语法范畴τ和e组成了语言的动态层
  - 这里的类型只是某种构造子，它们的kind都是Type.

8.3.1 类型族语言的静态语义-2

抽象语法	具体语法
Prod(K_1, K_2)	$K_1 \times K_2$
Arr(K_1, K_2)	$K_1 \rightarrow K_2$
app(app(arr, C_1), C_2)	$C_1 \rightarrow C_2$
app(all[K], lambda[K](t.C))	$\forall_K[\lambda(t::K.C)]$或$\forall_K[t::K.C]$
app(some[K], lambda[K](t.C))	$\exists_K[\lambda(t::K.C)]$或$\exists_K[t::K.C]$
pair(C_1, C_2)	$<C_1, C_2>$
fst(C)	fst(C)
snd(C)	snd(C)
lambda[K](t.C)	$\lambda(t::K.C)$
app(C_1, C_2)	$C_1[C_2]$
Lam[K](t.e)	$\Lambda(t::K.e)$
App[C](e)	$e[C]$
pack[K, C, C'](e)	pack C' with e as $\exists_K[C]$
open[K, C](e_1, t, x.e_2)	open e_1 as $t::K$ with x:app(C, t) in e_2

8.3.1 类型族语言的静态语义-3

❖静态语义

➢ 断言

- $\mathcal{T}|\Delta \vdash \mathcal{C} :: \mathcal{K}$ 构造子的形成(可类比到表达式的形成)
- $\mathcal{T}|\Delta \vdash \mathcal{C}_1 \equiv \mathcal{C}_2 :: \mathcal{K}$ 两个构造子的定义是等价的

  如 **fst(<C_1, C_2>)**与**C_1**相等
- $\mathcal{T}|\Delta \vdash \tau$ type 类型的形成
- $\mathcal{T}\ \mathcal{X}|\Delta\ \Gamma \vdash e : \tau$ 表达式的形成

在断言中，参数集$\mathcal{T}$指定类型构造子变元，$\mathcal{X}$指定表达式变元

Δ中的假设形如$t::K$，其中$\mathcal{T} \vdash t$ cons；

Γ中的假设形如$x:\tau$，其中$\mathcal{X} \vdash x$ exp且$\mathcal{T}|\Delta \vdash \tau$ type

为避免符号混乱，这里省去$\mathcal{T}$和$\mathcal{X}$。

8.3.1 类型族语言的静态语义-4

- 构造子的形成

$$\frac{}{\Delta, t :: \kappa \vdash t :: \kappa}$$

$$\frac{}{\Delta \vdash \mathtt{arr} :: \mathtt{Arr}(\mathtt{Type}, \mathtt{Arr}(\mathtt{Type}, \mathtt{Type}))}$$

$$\frac{}{\Delta \vdash \mathtt{all}[\kappa] :: \mathtt{Arr}(\mathtt{Arr}(\kappa, \mathtt{Type}), \mathtt{Type})}$$

$$\frac{}{\Delta \vdash \mathtt{some}[\kappa] :: \mathtt{Arr}(\mathtt{Arr}(\kappa, \mathtt{Type}), \mathtt{Type})}$$

$$\frac{\Delta \vdash c_1 :: \kappa_1 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{pair}(c_1, c_2) :: \mathtt{Prod}(\kappa_1, \kappa_2)}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \mathtt{fst}(c) :: \kappa_1}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \mathtt{snd}(c) :: \kappa_2}$$

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{lambda}[\kappa_1](t.c_2) :: \mathtt{Arr}(\kappa_1, \kappa_2)}$$

$$\frac{\Delta \vdash c_1 :: \mathtt{Arr}(\kappa_2, \kappa) \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{app}(c_1, c_2) :: \kappa}$$

(PFPL 25.1)

8.3.1 类型族语言的静态语义-5

- 定义的等价性

$$\frac{\Delta \vdash \mathtt{pair}(c_1, c_2) :: \mathtt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \mathtt{fst}(\mathtt{pair}(c_1, c_2)) \equiv c_1 :: \kappa_1}$$

$$\frac{\Delta \vdash \mathtt{pair}(c_1, c_2) :: \mathtt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \mathtt{snd}(\mathtt{pair}(c_1, c_2)) \equiv c_2 :: \kappa_2}$$

$$\frac{\Delta \vdash c :: \mathtt{Prod}(\kappa_1, \kappa_2)}{\Delta \vdash \mathtt{pair}(\mathtt{fst}(c), \mathtt{snd}(c)) \equiv c :: \mathtt{Prod}(\kappa_1, \kappa_2)}$$

(PFPL 25.2)

$$\frac{\Delta, t :: \kappa_1 \vdash c_2 :: \kappa_2 \quad \Delta \vdash c_2 :: \kappa_2}{\Delta \vdash \mathtt{app}(\mathtt{lambda}[\kappa](t.c_1), c_2) \equiv [c_2/t]c_1 :: \kappa_2}$$

$$\frac{\Delta \vdash c_2 :: \mathtt{Arr}(\kappa_1, \kappa_2) \quad t \# \Delta}{\Delta \vdash \mathtt{lambda}[\kappa_1](t.\mathtt{app}(c_2, t)) \equiv c_2 :: \mathtt{Arr}(\kappa_1, \kappa_2)}$$

$$\frac{\Delta\, \Gamma \vdash e : \tau' \quad \Delta \vdash \tau \equiv \tau' :: \mathtt{Type}}{\Delta\, \Gamma \vdash e : \tau}$$

(PFPL 25.3)

8.3.2 表达式的形成

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{25.4a}$$

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{lam}[\tau_1](x.e_2) : \mathtt{app}(\mathtt{app}(\mathtt{arr}, \tau_1), \tau_2)} \tag{25.4b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{app}(\mathtt{app}(\mathtt{arr}, \tau_2), \tau) \quad \Delta\ \Gamma \vdash e_2 : \tau_2}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1, e_2) : \tau} \tag{25.4c}$$

$$\frac{\Delta, t :: \kappa\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{Lam}[\kappa](t.e) : \mathtt{app}(\mathtt{all}[\kappa], \mathtt{lambda}[\kappa](t.\tau))} \tag{25.4d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{app}(\mathtt{all}[\kappa], c') \quad \Delta \vdash c :: \kappa}{\Delta\ \Gamma \vdash \mathtt{App}[c](e) : \mathtt{app}(c', c)} \tag{25.4e}$$

$$\frac{\Delta \vdash c' :: \mathtt{Arr}(\kappa, \mathtt{Type}) \quad \Delta \vdash c :: \kappa \quad \Delta\ \Gamma \vdash e : \mathtt{app}(c', c)}{\Delta\ \Gamma \vdash \mathtt{pack}[\kappa, c', c](e) : \mathtt{app}(\mathtt{some}[\kappa], c')} \tag{25.4f}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{app}(\mathtt{some}[\kappa], c) \quad \Delta, t :: \kappa\ \Gamma, x : \mathtt{app}(c, t) \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \mathtt{type}}{\Delta\ \Gamma \vdash \mathtt{open}[\kappa, c](e_1, t, x.e_2) : \tau_2} \tag{25.4g}$$

8.3.3 区分构造子与类型-1

- 构造子的Kind Type有双重角色
  - 作为静态的数据值，可以
    - 充当参数传递到多态函数中，即实例化多态函数中的类型参数
    - 绑定到存在类型的包中，即绑定到包的表示类型上
  - 作为动态数据值的分类器。
    - 通过表达式的定型规则来对表达式进行分类，即说明表达式具有什么类型。

8.3.3 区分构造子与类型-2

- 上述两个角色可在语法上分离开来

  将Kind为Type的构造子显式作为类型的一个类别，得到语言变种 $L\{typ, \rightarrow \forall_K \exists_K\}$

  Cons $C ::= t \mid arr \mid all[K] \mid some[K] \mid pair(C_1, C_2) \mid fst(C) \mid snd(C) \mid lambda[K](t.C) \mid app(C_1, C_2)$

  Type $\tau ::= typ(C) \mid arr(\tau_1, \tau_2) \mid all[K](t.\tau) \mid some[K](t.\tau)$

  - 这种将构造子和类型区分开来，会存在一定的冗余，但是能使人清晰地理解$L\{\rightarrow \forall_K \exists_K\}$中类型的两个角色
    - 作为表达式的分类器
      ——静态角色(静态语义:表达式的定型规则)
    - 作为多态函数的参数，或作为包的组成成分(表示类型)
      ——动态角色

8.3.3 区分构造子与类型-3

- 类型的两个角色之间的对应关系

用以下定义的等价性公理来说明

$$\frac{}{\mathtt{typ}(\mathtt{app}(\mathtt{app}(\mathtt{arr}, c_1), c_2)) \equiv \mathtt{arr}(\mathtt{typ}(c_1), \mathtt{typ}(c_2)) :: \mathtt{Type}} \quad (25.5a)$$

$$\frac{}{\mathtt{typ}(\mathtt{app}(\mathtt{all}[\kappa], c)) \equiv \mathtt{all}[\kappa](t.\mathtt{typ}(\mathtt{app}(c, t))) :: \mathtt{Type}} \quad (25.5b)$$

$$\frac{}{\mathtt{typ}(\mathtt{app}(\mathtt{some}[\kappa], c)) \equiv \mathtt{some}[\kappa](t.\mathtt{typ}(\mathtt{app}(c, t))) :: \mathtt{Type}} \quad (25.5c)$$

- 等价性说明类型可由构造子来标明
  - 规则(25.5a)指出：常量构造子**arr**应用到**Kind**为**Type**的构造子C_1和C_2时，标明一个由构造子C_1和C_2分别标明的类型之间的函数类型
  - 规则(25.5b)指出：常量构造子**all[*K*]**应用到**Kind**为**Type**的构造子C时，标明一个全称类型，该类型将C所标明的类型应用到任何**kind**为K的类型变元t上

8.3.3 区分构造子与类型-4

- L{typ, $\rightarrow \forall_K \exists_K$}的优点
  - 清楚地显示出可预言类型量化与不可预言类型量化之间的区别。
    在这个语言中，全称量词和存在量词只会取特定kind的构造子。

  在L{$\rightarrow \forall_K \exists_K$}中，Type包含所有的作为分类器的类型($\tau ::= C$)，从而量化是不可预言的.

  在L{typ $\rightarrow \forall_K \exists_K$}中，可以进行选择:

  - 选择包含带量词的类型作为构造子➔不可预言的片段
  - 选择不包含带量词的类型作为构造子(即不使用all[K]和some[K]作为构造子)➔可预言的片段.

8.3.4 动态语义-1

许多语言(针对那些动态语义的转换关系不受类型信息影响的语言)采用类型抹除(type erasure)语义，在执行完类型检查之后，抹除所有的类型信息，然后将所得的未类型化项解释或编译成机器代码。

Why? 类型注解的唯一作用是：保证源表达式以及根据转换关系由此推导出的表达式在源语言中是良类型的

从实现的角度看，这些类型注解可以被抹除，因为在表达式推导中(即程序运行时) 不会进行类型检查。

当引入类型量化时，类型抹除就会有问题:

类型会作为多态函数的参数，或者作为包的成分，在求值时，类型会置换表达式中出现的类型变元。这时，类型在动态语义中就起作用了，因此不能在执行前被抹除!

8.3.4 动态语义-2

区分类型的两类角色：

- 作为数据：任意kind的构造子是在运行时必须维护和操纵的数据形式——这部分类型不可以在执行前被抹除
- 作为分类器：不会影响求值——这部分类型可以在执行前被抹除

作业

8.1 证明置换引理(PFPL,Lemma 23.1)中的第2条

Thanks!