# 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

# 第七章 状态

7.1 存储效应[PFPL, 35]

7.2 Monadic存储效应[PFPL, 36]

7.3 可扩展的和[PFPL, 37]

# 7.1 存储效应

纯(pure)语言：其执行模型只包括表达式求值，如L{→}

非纯(impure)语言：其执行模型不仅包括表达式求值，还包括控制效应和存储效应。如ML

- 控制效应(control effects)：控制的非局部转移，如异常[PFPL, 28]、延续[PFPL, 29]
- 存储效应(storage effects)：对易变存储的动态修改

# 7.1 存储效应-扩展引用类型-语法

## ❖ 为L{→}扩展引用类型

- 扩展引用类型：**ref($\tau$)**
  该类型的元素表示可变存储单元
- 扩展表达式

  **Expr** $e ::= l \mid \text{new}(e) \mid \text{get}(e) \mid \text{set}(e_1, e_2)$
  - **new($e$)**：分配一个"新的"引用单元，**ref($\tau$)**的引入形式
  - **get($e$)**：取回$e$单元的内容，**ref($\tau$)**的消去形式
  - **set($e_1$, $e_2$)**：设置$e_1$单元的内容为值$e_2$，**ref($\tau$)**的消去形式
  - $l$：取值范围为一组存储单元的变量，与表达式变量相区分。
    存储单元是动态语义所必需的，但程序员不直接使用。

# 7.1 存储效应-扩展引用类型-静态语义

## ❖ 带引用的L{→}的静态语义

- 定型断言$e:\tau$的上下文

  变量假设：形如$x:\tau$，表示引入类型为$\tau$的表达式变量$x$

  存储单元假设：形如$l:\tau$，表示引入存储单元$l$，其内容为$\tau$类型

- 假言定型断言：$\Lambda\Gamma\vdash e:\tau$

  $\Lambda$一组存储单元假设,$\Gamma$一组变量假设；相同变量只有一个假设

- 扩展如下定型规则

new(e)是引用类型

$$\frac{}{\Lambda, l:\tau\,\Gamma\vdash l:\mathtt{ref}(\tau)}$$

$$\frac{\Lambda\,\Gamma\vdash e:\tau}{\Lambda\,\Gamma\vdash \mathtt{new}(e):\mathtt{ref}(\tau)}$$

e是τ 的引用类型，
则get(e)是τ 类型

$$\frac{\Lambda\,\Gamma\vdash e:\mathtt{ref}(\tau)}{\Lambda\,\Gamma\vdash \mathtt{get}(e):\tau}$$

e1是τ 的引用类型
e2是τ 类型，则
set(e1,e2)是τ 类型

$$\frac{\Lambda\,\Gamma\vdash e_1:\mathtt{ref}(\tau_2)\quad \Lambda\,\Gamma\vdash e_2:\tau_2}{\Lambda\,\Gamma\vdash \mathtt{set}(e_1,e_2):\tau_2}$$

该规则在类型检查时不会用到!
存储单元只在运行时出现，不是程序的一部分,即程序中一定无任何自由的存储位置!
该规则在证明类型安全时有用!

(PFPL 35.1)

# 7.1 存储效应-扩展引用类型-动态语义

## ❖ 带引用的L{→}的动态语义

- 内存：是将存储单元映射到闭值的一个有限函数
- 抽象机的状态： **(M,*e*)**

  **M**表示内存，***e***是其存储单元都在**M**内的表达式

  – **(Ø,*e*)** ： 初始状态，其中***e***不包含任何存储单元

  – **(M,*e*)，*e* val** ： 终结状态
- 引用的动态语义(eager即call-by-value)

$$\frac{}{l \text{ val}}$$

存储单元是值

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{new}(e)) \mapsto (M', \texttt{new}(e'))}$$

e未完成求值，则允许在引入形式下归约

$$\frac{e \text{ val} \quad l \# M}{(M, \texttt{new}(e)) \mapsto (M[l = e], l)}$$

e是值时，执行new(e)会产生一个不在M中的存储单元*l*，将e存入该单元，并将*l* 到e的映射加入到M中。

(PFPL 35.2)

# 7.1 存储效应-扩展引用类型-动态语义

## ❖ 带引用的L{→}的动态语义

e未完成求值，则允许在get下归约

从M中取存储单元*l*的内容( *l* 在M中时)

$$\frac{(M,e) \mapsto (M',e')}{(M,\mathtt{get}(e)) \mapsto (M',\mathtt{get}(e'))}$$

$$\frac{e\ \mathsf{val}}{(M[l=e],\mathtt{get}(l)) \mapsto (M,e)}$$

e1未完成求值，则允许在set下归约

e1是值但e2未完成求值，则允许在set下归约

$$\frac{(M,e_1) \mapsto (M',e_1')}{(M,\mathtt{set}(e_1,e_2)) \mapsto (M',\mathtt{set}(e_1',e_2))}$$

$$\frac{e_1\ \mathsf{val} \quad (M,e_2) \mapsto (M',e_2')}{(M,\mathtt{set}(e_1,e_2)) \mapsto (M',\mathtt{set}(e_1,e_2'))}$$

$$\frac{e\ \mathsf{val}}{(M[l=e'],\mathtt{set}(l,e)) \mapsto (M[l=e],e)}$$

将M中的存储单元*l*的值设置为e

(PFPL 35.2)

# 7.1 存储效应-扩展引用类型-安全性

## ❖ 安全性

**辅助关系**：内存**M**与存储单元定型**Λ**之间的关系**M : Λ**

$$\frac{\forall l \in dom(\Lambda)\ \Lambda \vdash M(l) : \Lambda(l)}{M : \Lambda}$$

(PFPL 35.3)

对**Λ**中的每个存储单元***l***，存储在***l*** 中的值必须是**Λ**中规定的类型。

- 允许一个存储单元中的值直接或间接引用自身——循环的内存

用存储单元实现递归函数，如下面返回类型为**nat→nat**的求阶乘函数

```
let( new(lam[nat](n.n)), r.
   let(lam[nat](n.ifz(n, num[1], n'.times(n, ap(get(r), n')))), f.
      let(set(r, f ), _. f )))
```

- 分配一个引用单元***r***,它初始化为任意类型为**nat→nat**的函数
- 定义一个λ抽象***f***，其中的递归调用包含取***r*** 中的函数，再将该函数应用到***n′***
- 将***f*** 赋值到存储单元***r***
- 返回***f***

回填技术（**backpatching**）

# 7.1 存储效应-扩展引用类型-应用举例

例：求3的阶乘

**(Ø, ap( let( new(lam[nat](*n*.*n*)), *r*.**
**let(lam[nat](*n*.ifz(*n*, num[1], *n*′.times(*n*, ap(get(*r*), *n*′)))))), *f*.**
**let(set(*r*, *f* ), _. *f* ))), num[3] ) )**

$\mapsto$ ($[l = \text{lam[nat]}(n.n)]$, ap(let(*l*, *r*.
let(lam[nat](*n*.ifz(*n*, num[1], *n*′.times(*n*, ap(get(*r*), *n*′))))), *f* .
let(set(*r*, *f* ), _ . *f* ))), num[3]))

应用规则(PFPL 35.2c)

$\mapsto$ ($[l = \text{lam[nat]}(n.n)]$, ap(let(lam[nat](*n*.ifz(*n*, num[1],
*n*′.times(*n*, ap(get(*l*), *n*′)))), *f* .let(set(*l*, *f* ), _ . *f* )), num[3]))

应用规则(PFPL 9.2g)
用l 置换r（$\beta$归约）

$\mapsto$ ($[l = \text{lam[nat]}(n.n)]$, ap(let(set(*l*,lam[nat](*n*.ifz(*n*, num[1],
*n*′.times(*n*, ap(get(*l*), *n*′))))), _ . lam[nat](*n*.ifz(...))), num[3]))

应用规则(PFPL 9.2g)
置换f（$\beta$归约）

$\mapsto$ ($l = \text{lam[nat]}(n.\text{ifz}(...))$, ap(let(lam[nat](*n*.ifz(*n*, num[1],
*n*′.times(*n*, ap(get(*l*), *n*′)))), _ . lam[nat](*n*.ifz(...))), num[3]))

应用规则(PFPL 35.2h)

# 7.1 存储效应-扩展引用类型-应用举例

例：求3的阶乘

$\mapsto ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{ap}(\text{let}(\text{lam[nat]}(n.\text{ifz}(n,\ \text{num[1]},$
$n'.\text{times}(n,\ \text{ap}(\text{get}(l), n')))),\ \_\ .\ \text{lam[nat]}(n.\text{ifz}(...))),\ \text{num[3]}))$

$\mapsto ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{ap}(\text{lam[nat]}(n.\text{ifz}(n,\ \text{num[1]},$
$n'.\text{times}(n,\ \text{ap}(\text{get}(l), n')))),\ \text{num[3]}))$

应用规则(PFPL 9.2g)

$\mapsto ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{ifz}(\text{num[3]},\ \text{num[1]},$
$n'.\text{times}(\text{num[3]},\ \text{ap}(\text{get}(l), n'))))$

应用规则(PFPL 13.4c)

$\mapsto^* ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{times}(\text{num[3]},\ \text{ap}(\text{get}(l),\ \text{num[2]})))$

应用规则(PFPL 15.4d)

$\mapsto ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{times}(\text{num[3]},\ \text{ap}(\text{lam[nat]}(n.\text{ifz}(...)),\ \text{num[2]})))$

应用规则(PFPL 35.2e)

$\mapsto^* ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{times}(\text{num[3]},$
$\text{times}(\text{num[2]}, \text{times}(\text{num[1]}, \text{num[1]}))))$

$\mapsto^* ([l = \text{lam[nat]}(n.\text{ifz}(...))], \text{num[6]})$

# 7.1 存储效应-扩展引用类型-安全性

良形状态

$$\frac{M : \Lambda \quad \Lambda \vdash e : \tau}{(M, e)\ \mathsf{ok}}$$

(PFPL 35.4)

$(M, e)$是良形的，当且仅当$M$中与$e$有关的存储单元定型是良类型的

弱化引理(PFPL Lemma 35.1，Weakening)如果$\Lambda \vdash e : \tau$且$\Lambda' \supseteq \Lambda$，则$\Lambda' \vdash e : \tau$.

证明：对定型推导归纳.不在$\Lambda$中的存储单元假设不会影响推导的有效性.

保持性定理(PFPL Theorem 35.2)如果$(M, e)$ ok且$(M, e) \mapsto (M', e')$，则$(M', e')$ ok.

证明：对求值转换进行归纳.其中的证明技巧是证明：
如果$(M, e) \mapsto (M', e')$，$M: \Lambda$且$\Lambda \vdash e : \tau$，则存在$\Lambda' \supseteq \Lambda$使得$M' : \Lambda'$且$\Lambda' \vdash e' : \tau$.

进展性定理(PFPL Theorem 35.3)如果$(M, e)$ ok，则或者$(M, e)$是终结状态，或者存在$(M', e')$使得$(M, e) \mapsto (M', e')$.

证明：对定型规则进行归纳.其中的证明技巧是证明：
如果$M: \Lambda$且$\Lambda \vdash e : \tau$，则或者$e$是值，或者存在$M'$和$e'$使得$(M, e) \mapsto (M', e')$.

# 7.2 一元(Monadic)存储效应-1

组合函数式语言和命令式语言(有效应)的方法之一：增加引用类型

- 对于具有call-by-value语义的语言来说，是可行的。
  能预测表达式在哪被求值，从而能预测引用在哪被分配和赋值。
- 对于具有call-by-name语义的语言来说，会存在问题。
  经常难以预测表达式何时被求值以及如何求值。

在函数式语言中加入引用类型，还会导致：
不能单纯由类型确定表达式是否改变存储。

例如，对一个类型为nat→nat的函数，参数和返回值都必须是nat，但是它有可能会分配新的存储单元，或者修改现有的存储单元。
——类型系统的表达能力被削弱了，因为它不能区分纯表达式(无效应的)和非纯表达式(有效应的)。

# 7.2 一元(Monadic)存储效应-2

在纯语言中引入效应的另一方法是在类型系统中显式表达效应的可能性。

——引入模态(modality),称为单元(monad，unit的拉丁语),它将语言中无效应的部分与有效应的部分隔离开来。

这两个子语言之间的联系是:

- 每个无效应的表达式可看作是有(空)效应的表达式
- 可以挂起一个有(空)效应的表达式，并将它打包成无效应的表达式；相应地，解包和激活这种表达式就是有效应的操作

对有效应的表达式打包所得的就是一个一元(monadic)类型 $\tau$ comp 的值.类型 $\tau$ comp表示一类会产生类型为 $\tau$ 的值的非纯计算。

- nat→nat ： 纯函数
- nat→nat comp: 非纯函数，当应用到自然数时，会产生有效应的计算(可能会修改存储)

# 7.2.1 Monadic语言-1

## 用Monadic重新描述扩展有引用类型的L{nat⇀}

- 语法

**Type** $\tau$ **::= nat | parr(**$\tau_1$**,** $\tau_2$**) | ref(**$\tau$**) | comp(**$\tau$**)**

**Expr** $e$ **::=** $x$ **|** $l$ **| z | s(**$e$**) | ifz(**$e_0$**,** $e_1$**,** $e_2$**) | fix[**$\tau$**](**$x.e$**) |**
**lam[**$\tau$**](**$x.e$**) | ap(**$e_1$**,** $e_2$**) | comp(**$m$**)** ——纯表达式

**Comm** $m$ **::= return(**$e$**) | letcomp(**$e$**,** $x.m$**) | new(**$e$**) |**
**get(**$e$**) | set(**$e_1$**,** $e_2$**)** ——命令(非纯表达式)

- **comp(**$\tau$**)**：表示产生一个$\tau$类型的值的被挂起计算(有效应)的类型
  引入形式是**comp(**$m$**)**；消去形式是**letcomp(**$e$**,** $x.m$**)** ——monadic原语

| 抽象语法 | 具体语法 | |
|---|---|---|
| **comp(**$\tau$**)** | $\tau$ **comp** | 包装产生类型为$\tau$的非纯计算的类型 |
| **return(**$e$**)** | **return** $e$ | 将纯表达式$e$放入到命令中 |
| **letcomp(**$e$**,** $x.m$**)** | **let comp(**$x$**) be** $e$ **in** $m$ | 将纯表达式$e$放入到命令中 |
| **comp(**$m$**)** | **comp(**$m$**)** | 将命令(非纯表达式)$m$打包成纯表达式 |

# 7.2.1 Monadic语言-2

- do语法(为便于表示而引入，语法美化(syntax sugar))
  - 二元do结构：抽象语法 $\mathbf{do}(m_1, x.m_2)$　　具体语法 $\mathbf{do}\{x \leftarrow m_1; m_2\}$
    定义为 $\mathbf{letcomp}(\mathbf{comp}(m_1), x.m_2)$
  - 扩展的do结构
    - 具体语法 $\mathbf{do}\{ x_1 \leftarrow m_1; \ldots; x_k \leftarrow m_k; \mathbf{return}\ e \}$
    - 抽象语法 $\mathbf{do}(m_1, x_1.\mathbf{do}(\ldots\mathbf{do}(m_k, x_k.\mathbf{return}(e)\ldots)$
- 静态语义
  - 定型断言
    - $e : \tau$：纯表达式$e$的类型为$\tau$
    - $m \sim \tau$：非纯表达式$m$的类型为$\tau$
  - $e : \tau$和$m \sim \tau$均使用如下定型假设(因为变量永远只绑定到值上)
    - $x_i : \tau_i$：假设变量$x_i$的类型为$\tau_i$
    - $l_i : \tau_i$：假设存储单元$l_i$的类型为$\tau_i$

    $\Lambda$一组存储单元假设，$\Gamma$一组变量假设

# 7.2.1 Monadic语言-3

- 定型规则：在L{→}的定型规则上扩展以下规则

$$\frac{\Lambda\,\Gamma \vdash m \sim \tau}{\Lambda\,\Gamma \vdash \mathtt{comp}(m) : \mathtt{comp}(\tau)}$$

通过return将纯表达式e变为非纯表达式return(e)

comp将非纯表达式m包装为纯表达式comp(m)

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \mathtt{return}(e) \sim \tau}$$

$$\frac{\Lambda\,\Gamma \vdash e : \mathtt{comp}(\tau) \quad \Lambda\,\Gamma, x : \tau \vdash m \sim \tau'}{\Lambda\,\Gamma \vdash \mathtt{letcomp}(e, x.m) \sim \tau'}$$

comp($\tau$)类型的纯表达式e可以绑定到$\tau$类型的变量x

(PFPL 36.1)

$$\frac{}{\Lambda, l : \tau\,\Gamma \vdash l : \mathtt{ref}(\tau)}$$

该规则同样在类型检查时不会用到，但在证明类型安全时会用到！

$$\frac{\Lambda\,\Gamma \vdash e : \tau}{\Lambda\,\Gamma \vdash \mathtt{new}(e) \sim \mathtt{ref}(\tau)}$$

纯表达式e类型为$\tau$,则为e分配存储单元所得的new(e)是类型为ref($\tau$)的非纯表达式

$$\frac{\Lambda\,\Gamma \vdash e : \mathtt{ref}(\tau)}{\Lambda\,\Gamma \vdash \mathtt{get}(e) \sim \tau}$$

e为引用类型ref($\tau$),则get(e)是类型为$\tau$的非纯表达式

$$\frac{\Lambda\,\Gamma \vdash e_1 : \mathtt{ref}(\tau) \quad \Lambda\,\Gamma \vdash e_2 : \tau}{\Lambda\,\Gamma \vdash \mathtt{set}(e_1, e_2) \sim \mathtt{unit}}$$

e1是$\tau$的引用类型,e2是$\tau$类型,则set(e1,e2)是空积类型的非纯表达式

# 7.2.1 Monadic语言-4

- 动态语义
  - 转换断言
    - $e \mapsto e'$：针对非纯表达式
    - $(\mathbf{M}, e) \mapsto (\mathbf{M}', e')$：针对非纯表达式

    前者的定义与L{nat ⇀}一样，只是要补充对新的纯表达式的定义；后者类似于7.1，同样要考虑对monadic原语的定义
  - 在纯表达式中增加以下值

$$\frac{}{\mathtt{comp}(m)\ \mathsf{val}} \qquad \frac{}{l\ \mathsf{val}}$$

(PFPL 36.2)

  - monadic原语的转换规则

由纯表达式的状态转换，得到相应的非纯表达式的状态转换

$$\frac{e \mapsto e'}{(M, \mathtt{return}(e)) \mapsto (M, \mathtt{return}(e'))}$$

(PFPL 36.3ab)

$$\frac{e \mapsto e'}{(M, \mathtt{letcomp}(e, x.m)) \mapsto (M, \mathtt{letcomp}(e', x.m))}$$

# 7.2.1 Monadic语言-5

- monadic原语的转换规则 (PFPL 36.3cd)

$$\frac{(M, m_1) \mapsto (M', m_1')}{(M, \mathtt{letcomp}(\mathtt{comp}(m_1), x.m_2)) \mapsto (M', \mathtt{letcomp}(\mathtt{comp}(m_1'), x.m_2))}$$

m1未完成求值，则允许在letcomp下归约

$$\frac{e\ \mathsf{val}}{(M, \mathtt{letcomp}(\mathtt{comp}(\mathtt{return}(e)), x.m)) \mapsto (M, [e/x]m)}$$

e是值，则执行置换. 思考：为什么不直接写letcomp(e, ...)? 结合定型规则来理解

- 引用原语的转换规则

e未完成求值，则允许在new操作下归约

$$\frac{e \mapsto e'}{(M, \mathtt{new}(e)) \mapsto (M, \mathtt{new}(e'))}$$

(PFPL 36.3ef)

$$\frac{e\ \mathsf{val} \quad l \# M}{(M, \mathtt{new}(e)) \mapsto (M[l = e], \mathtt{return}(l))}$$

e是值，则分配新的存储单元*l*，将l到e的映射加入到M,表达式归约到return(*l*)

# 7.2.1 Monadic语言-6

- 引用原语的转换规则

$$\frac{e \mapsto e'}{(M, \mathtt{get}(e)) \mapsto (M, \mathtt{get}(e'))}$$

$$\frac{e\ \mathsf{val} \quad l \# M}{(M[l = e], \mathtt{get}(l)) \mapsto (M[l = e], \mathtt{return}(e))}$$

$$\frac{e_1 \mapsto e_1'}{(M, \mathtt{set}(e_1, e_2)) \mapsto (M, \mathtt{set}(e_1', e_2))}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{(M, \mathtt{set}(e_1, e_2)) \mapsto (M, \mathtt{set}(e_1, e_2'))}$$

$$\frac{e\ \mathsf{val} \quad l \# M}{(M[l = e'], \mathtt{set}(l, e)) \mapsto (M[l = e], \mathtt{return}(e))}$$

(PFPL 36.3g~k)

# 7.2.2 显式效应

## ❖ 显式效应

- 引入monad的主要动机
  - 使任何计算效应在类型中可显式表示——显式效应
- 针对存储效应，这种显式效应并不总是有优势

主要的问题：任何使用存储的计算都被强制在monad内

——一旦在monad中，就永远在monad中

始终被看作是非纯的！

良性效应(benign effects)：可用于某一计算内，该计算对外完全是纯的。

[自学]理解书上关于阶乘函数的回填实现!

# 7.3 可扩展的和类型

如何提供可动态扩展的异常类型 $\tau_{exn}$ ？

——可扩展的变式类型

**SML中的做法**

- **exn**是异常类型（变式类型，带标签的和类型）
- 用**exception**声明新的异常值构造器

```
exception Div
exception Fail of string
exception Recursive of exn
```

- 允许声明多个同名异常值构造器，它们代表不同的异常，在其中一个作用域下不能按名引用其他的。
- 异常声明可以出现在循环中，循环的每一次迭代引入一个新的异常，虽然它们有相同的名字

本节引入更一般的可扩展的和类型。

# 7.3 可扩展的和类型-语法

- 标记机制的语法

  **Type** $\tau$ **::= tagged | tag(**$\tau$**)**

  **Expr** $e$ **::=** $l$ **| tagwith[**$\tau$**](**$e_1$**;**$e_2$**) | lettag(**$e_1$**;**$t$ **,** $x$**,** $y$**.**$e_2$**) |**
  **newtag[**$\tau$**]( ) | istag[**$t.\tau$**](**$e_1$**;** $e_2$ **;** $e_3$ **;** $e_4$**)** -- $l$是标签

  - **tagged**：标记值所属的类型，与SML中的**exn**相对应
  - **tag(**$\tau$**)**：标签类型，其关联的值类型为$\tau$
    标签只是一个名字
  - **tagwith[**$\tau$**](**$e_1$**;**$e_2$**)**：将关联类型为$\tau$的标签$e_1$附在这种类型的值$e_2$上
    ——**tagged**的引入形式
  - **lettag(**$e_1$**;**$t$ **,** $x$**,** $y$**.**$e_2$**)**：将标记值$e_1$分解为标签和值分别绑定到$x$和$y$
    在$e_2$中使用 ——**tagged**的消去形式
  - **newtag[**$\tau$**]( )**：产生一个新标签
  - **istag[**$t.\tau$**](**$e_1$**;** $e_2$ **;** $e_3$ **;** $e_4$**)**：比较两个标签$e_1$和$e_2$是否相等，相等则执行$e_3$，否则执行$e_4$

# 7.3 可扩展的和类型-语法

- 标记机制的语法

| 抽象语法 | 具体语法 | |
|---|---|---|
| tagged | tagged | 标记值所属的类型 |
| tagwith[$\tau$]($e_1$;$e_2$) | tag$_\tau$ $e_2$ with $e_1$ | 将标签$e_1$附在值$e_2$上，值类型为$\tau$ |
| lettag($e_1$;$t$ , $x$, $y$.$e_2$) | let tag$_\tau$ $y$ with $x$ be $e_1$ in $e_2$ | 将$e_1$分解并在$e_2$中使用 |
| newtag[$\tau$]( ) | newtag[$\tau$] | 产生一个标签 |
| istag[$t.\tau$]($e_1$; $e_2$ ; $e_3$ ; $e_4$) | iftag$_{t.\tau}$ $e_1$ is $e_2$ then $e_3$ else $e_4$ | |

比较标签$e_1$和$e_2$是否相等，相等则执行$e_3$，否则执行$e_4$

- 静态语义
  - 断言形式：$\Theta\Delta\Gamma \vdash e : \tau$

$\Theta$:一组形如$l : \tau$的假设，用来假设标签及其关联的类型，一个标签只能有一个类型；$\Delta$:一组形如$t$ type的假设；$\Gamma$:一组形如$x : \tau$的假设

# 7.3 可扩展的和类型-静态语义

## ❖静态语义

$$\frac{}{\Theta, l : \tau\ \Delta\ \Gamma \vdash l : \mathtt{tag}(\tau)}$$

$$\frac{\Delta \vdash \tau\ \mathtt{type} \quad \Theta\ \Delta\ \Gamma \vdash e_1 : \mathtt{tag}(\tau) \quad \Theta\ \Delta\ \Gamma \vdash e_2 : \tau}{\Theta\ \Delta\ \Gamma \vdash \mathtt{tagwith}[\tau](e_1; e_2) : \mathtt{tagged}}$$ (PFPL 37.1)

$$\frac{\Theta\ \Delta\ \Gamma \vdash e_1 : \mathtt{tagged} \quad \Theta\ \Delta, t\ \mathtt{type}\ \Gamma, x : \mathtt{tag}(t), y : t \vdash e_2 : \tau_2}{\Theta\ \Delta\ \Gamma \vdash \mathtt{lettag}(e_1; t, x, y.e_2) : \tau_2}$$

$$\frac{\Delta \vdash \tau\ \mathtt{type}}{\Theta\ \Delta\ \Gamma \vdash \mathtt{newtag}[\tau]() : \mathtt{tag}(\tau)}$$

$$\frac{\Theta\ \Delta\ \Gamma \vdash e_1 : \mathtt{tag}(\tau_1) \quad \Theta\ \Delta\ \Gamma \vdash e_2 : \mathtt{tag}(\tau_2) \quad \Theta\ \Delta\ \Gamma \vdash e_3 : [\tau_2/t]\tau \quad \Theta\ \Delta\ \Gamma \vdash e_4 : [\tau_1/t]\tau \quad \Delta, t\ \mathtt{type} \vdash \tau\ \mathtt{type}}{\Theta\ \Delta\ \Gamma \vdash \mathtt{istag}[t.\tau](e_1; e_2; e_3; e_4) : [\tau_1/t]\tau}$$

**e1是比较的目标，故当e1和e2不同时，取$\tau_1$置换t**

# 7.3 可扩展的和类型-动态语义

## ❖动态语义

- 抽象机的状态：**(T, *e*)**
  **T** 是一组有限的标签；***e***是表达式，其标签都在**T**中
- 值：标签和标记值都是值

$$\frac{l \text{ name}}{l \text{ val}} \qquad \frac{e_1 \text{ val} \quad e_2 \text{ val}}{\texttt{tagwith}[\tau]\,(e_1; e_2) \text{ val}}$$

(PFPL 37.2ab)

- 初始状态和终结状态

$$\frac{}{(\emptyset, e) \text{ initial}} \qquad \frac{e \text{ val}}{(T, e) \text{ final}}$$

(PFPL 37.2cd)

- 转换规则

$$\frac{(T, e_1) \mapsto (T', e_1')}{(T, \texttt{tagwith}[\tau]\,(e_1; e_2)) \mapsto (T, \texttt{tagwith}[\tau]\,(e_1'; e_2))}$$

$$\frac{e_1 \text{ val} \quad (T, e_2) \mapsto (T', e_2')}{(T, \texttt{tagwith}[\tau]\,(e_1; e_2)) \mapsto (T', \texttt{tagwith}[\tau]\,(e_1; e_2'))}$$

# 7.3 可扩展的和类型-动态语义

- 转换规则

$$\frac{(T, e_1) \mapsto (T', e_1')}{(T, \texttt{lettag}(e_1; t, x, y.e_2)) \mapsto (T', \texttt{lettag}(e_1'; t, x, y.e_2))}$$

$$\frac{\texttt{tagwith}[\tau](e_1; e_2)\ \text{val}}{(T, \texttt{lettag}(\texttt{tagwith}[\tau](e_1; e_2); t, x, y.e)) \mapsto (T, [\tau, e_1, e_2/t, x, y]e)}$$

$$\frac{l \# T}{(T, \texttt{newtag}[\tau]()) \mapsto (T \cup \{l\}, l)}$$

$$\frac{(T, e_1) \mapsto (T', e_1')}{(T, \texttt{istag}[t.\tau](e_1; e_2; e_3; e_4)) \mapsto (T', \texttt{istag}[t.\tau](e_1'; e_2; e_3; e_4))}$$

$$\frac{e_1\ \text{val} \quad (T, e_2) \mapsto (T', e_2')}{(T, \texttt{istag}[t.\tau](e_1; e_2; e_3; e_4)) \mapsto (T', \texttt{istag}[t.\tau](e_1; e_2'; e_3; e_4))}$$

$$\frac{(l_1 = l_2)}{(T, \texttt{istag}[t.\tau](l_1; l_2; e_3; e_4)) \mapsto (T, e_3)}$$

$$\frac{(l_1 \neq l_2)}{(T, \texttt{istag}[t.\tau](l_1; l_2; e_3; e_4)) \mapsto (T, e_4)}$$

# 7.3 可扩展的和类型-标记值类型

- 类型tagged
  - 可用存在类型 $\exists(t.t\ \text{tag} \times t)$ 定义
  - **tagwith[$\tau$]($e_1$;$e_2$) = pack[$t.t$ tag × $t$ ; $\tau$](pair($e_1$,$e_2$))**
  - **lettag($e_1$;$t$ , $x$, $y.e_2$)=**

    **open[$t.t$ tag × $t$ ]($e_1$;$t$, z.[fst(z),snd(z)/$x$, $y$]$e_2$ )**

# 7.3 可扩展的和类型-安全性

## ❖ 安全性

- $T : \Theta$ iff $dom(\Theta)=T$

保持性引理(PFPL Lemma 37.1)如果$\Theta \vdash e : \tau$, $T : \Theta$ 且$(T, e) \mapsto (T', e')$，则存在$\Theta' \supseteq \Theta$ 使得$T' : \Theta'$且$\Theta' \vdash e' : \tau$.

证明:对求值转换进行归纳.(略)

范式引理(PFPL Theorem 37.2) 如果$\Theta \vdash e : tag(\tau)$, $e$ val，则$e=l$, $l \in dom(\Theta)$ 且$\Theta(l) = \tau$ .

进展性引理(PFPL Theorem 37.3) 如果$\Theta \vdash e : \tau$, $T : \Theta$，则或者$(T, e)$ final，或者存在$T' \supseteq T$和$e'$使得$(T, e) \mapsto (T', e')$.

证明:对定型规则进行归纳.利用范式引理.

# 作业

7.1 [PFPL, 35] 35.2 Exercises 1

7.2 [PFPL, 37] 37.3 Exercises

# Thanks!