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

November, 2008

第五章 动态定型

5.1 未类型化的语言[PFPL, 22]

5.2 动态定型[PFPL, 23]

5.1 未类型化的(untyped)语言

类型化的(typed)语言：变量都被给定类型

类型系统由一组定型规则组成,规定了语言的静态语义。

良类型的程序：能通过类型检查(根据定型规则来检查)的程序。

良行为的程序：运行时不会引起不能被捕获的错误。

安全语言：所有合法程序都是良行为的语言。进展性、保持性。

类型可靠的语言：良类型程序一定是良行为的。一定是安全语言。

未类型化的(untyped)语言:不限制变量值的范围

5.1.1 未类型化λ演算L{λ}

5.1.2 可表达性(expressiveness)：用L{λ}表示PCF中的基本成分

5.1.3 未类型化代表单一类型化：L{λ}嵌入到L{μ $\rightarrow$}中的形式

5.1.1 未类型化λ演算-1

- 未类型化λ演算 **L{λ}**：纯λ演算
  - 1930s，Alonzo Church & Stephen Cole Kleene
  - 语言特征
    - 只有函数，而没有其他形式的数据
    - 函数的参数和返回值可以为函数
    - 所有的数据结构必须表示成函数
- **L{λ}**的抽象语法

  λ项(terms)　　$u ::= x \mid \lambda\,(x.u) \mid \mathrm{uap}(u_1\,;\,u_2)$

  - 具体语法中，上述后两项依次记作$\lambda x.u$和$u_1(u_2)$，分别称为λ抽象和λ应用。

5.1.1 未类型化λ演算-2

❖L{λ}的静态语义

- 用来定义well-formed(合式的、良构的)项
- 假言断言 x_1 ok, ..., x_n ok $\vdash u$ ok

  表示u是包含变量$x_1, \ldots, x_n$的合式项
- 规则

变量是合式项

$$\frac{}{\Gamma, x\ \mathrm{ok} \vdash x\ \mathrm{ok}}$$

一个合式项应用到另一个合式项是合式项

$$\frac{\Gamma \vdash u_1\ \mathrm{ok} \quad \Gamma \vdash u_2\ \mathrm{ok}}{\Gamma \vdash \mathrm{uap}(u_1; u_2)\ \mathrm{ok}} \qquad (22.1)$$

一个变量在合式项u中的抽象是合式项

$$\frac{\Gamma, x\ \mathrm{ok} \vdash u\ \mathrm{ok}}{\Gamma \vdash \lambda(x.u)\ \mathrm{ok}}$$

5.1.1 未类型化λ演算-3

❖L{λ}的动态语义—结构语义

- 转换规则

可归约式

β归约：用置换定义函数应用

$$\frac{}{\mathrm{uap}(\lambda(x.u_1);u_2) \mapsto [u_2/x]u_1}$$

u_1为归约到λ抽象时，允许在λ应用下归约

$$\frac{u_1 \mapsto u_1'}{\mathrm{uap}(u_1;u_2) \mapsto \mathrm{uap}(u_1';u_2)}$$

(22.2)

❖观测等价 $u_1 \cong u_2\ [\Gamma]$

- 表示u_1和u_2 在所有上下文中是无区别的。
- 一致性、同余性、包含β归约

5.1.1 未类型化λ演算-4

❖ 观测等价

以Church布尔式为例说明。

- **Church布尔式:**

  项 $\mathbf{tt} = \lambda x.\lambda y.x$ 和 $\mathbf{ff} = \lambda x.\lambda y.y$ 分别表示真和假

  参数x 和y 都是函数

  满足 $\mathrm{tt}(u_1)(u_2) \mapsto^* u_1$ 和 $\mathrm{ff}(u_1)(u_2) \mapsto^* u_2$

- 一致性: **tt**和**ff**不等价
- 同余性: 对于项中的任何子项，用一个等价的项代替该子项，则所得的新项仍和源项等价
- 包含β归约项: $(\lambda x.u_2)(u_1) \cong [u_1 / x]u_2\ [\Gamma]$

5.1.1 未类型化λ演算-5

- 定义条件测试

  test = λ*l*.λ*m*.λ*n*.*l*(*m*)(*n*)

参数*l* 是布尔算子，*l* 是**tt** 则选择参数*m*，否则选择参数*n*。

test (b) (v) (w) 可归约到**b (v) (w)**.

例：

	test (tt) (*v*) (*w*)	
=	(λ*l*.λ*m*.λ*n*.*l* (*m*) (*n*)) (tt) (*v*) (*w*)	由定义
↦	(λ*m*.λ*n*.tt (*m*) (*n*)) (*v*) (*w*)	归约红色的可归约式
↦	(λ*n*.tt (*v*) (*n*)) (*w*)	归约红色的可归约式
↦	tt (*v*) (*w*)	归约红色的可归约式
↦	(λ*x*.λ*y*.*x*) (*v*) (*w*)	由定义
↦	(λ*y*.*v*) (*w*)	归约红色的可归约式
↦	*v*	归约红色的可归约式

5.1.1 未类型化λ演算-6

- 定义逻辑与

  and = λ*b*.λ*c*. *b*(*c*)(ff)

表示对给定的两个布尔值***b*** 和***c***，如果是**tt**，输出***c***；如果是**ff**，则输出**ff**。

- 定义序对

  pair = λ*f*.λ*s*.λ*b*. *b*(*f*)(s)

  fst = λ*p*.*p* (tt)

  snd = λ*p*.*p* (ff)

pair(v)(w)是这样的函数，当应用于一个布尔值***b*** 时，***b***会应用于**v**和**w**。

5.1.2 可表达性-1

- 未类型化λ演算具有强大的表达能力
  - 是Turing完全的语言： 自然数上的可计算函数
  - 与PCF语言一样强大：PCF中的基本成分在L{λ}中都可定义

如何在L{λ}中表示PCF中的基本成分？

- Church数值：用λ项表示自然数

$$\overline{0} = \lambda b.\, \lambda s.\, b$$

$$\overline{n+1} = \lambda b.\, \lambda s.\, s((\overline{n}(b)(s))) \qquad (22.3)$$

每个数n用一个组合式表示，它有b和s两个参数(表示零和后继)

$$\overline{n}(u_1)(u_2) \cong u_2(\ldots(u_2(u_1)))$$

$\overline{n}(u_1)(u_2)$表示u_2被重复应用n次的折叠形式，u_2最初被应用在u_1上

5.1.2 可表达性-2

❖ 一些容易定义的算术函数

- 自然数的后继函数

  succ = $\lambda x.\lambda b.\lambda s.s\ (\ x\ (b)\ (s))$

  succ表示取一个Church数值*x*,得出另一个Church数值，即重复应用*s* 到*b*，应用*x*次，然后再将*s*应用到这个结果

  另一种表示：**succ' = $\lambda x.\lambda b.\lambda s.x\ (s\ (b))\ (s)$**

- 自然数的加法函数

  用后继表示：**plus = $\lambda x.\lambda y.y\ (x)$ (succ)**

  不用后继表示：**plus' = $\lambda x.\lambda y.\lambda b.\lambda s.y\ (x\ (b)\ (s))\ (s)$**

- 自然数的乘法函数

  用加法函数表示：**times = $\lambda x.\lambda y.y\ (\overline{0})$ (plus (*x*))**

5.1.2 可表达性-3

- 定义PCF中的$\mathbf{ifz}(u, u_0, x.u_1)$
  - 定义$\mathbf{ifz}(u, u_0, u_1) = u\ (u_0)\ (\lambda x.u_1) \qquad x \# u_1$
  - 问题：如何定义求自然数前驱的项？

$$\mathtt{pred}(\overline{0}) \cong \overline{0} \qquad (22.7)$$

$$\mathtt{pred}(\overline{n+1}) \cong \overline{n} \qquad (22.8)$$

  - 方法1：直接按后继进行归纳？
    但是$\overline{0}$的前驱是$\overline{0}$，利用后继不能归纳得出
  - 方法2：考虑Church数值序对$\langle n\text{-}1, n\rangle$
    - 假设$\langle n\text{-}1, n\rangle$是取$n$时的结果，则取$n+1$时，其结果可以通过将$\langle n\text{-}1, n\rangle$的第2元左移到第1元，并且将第2元增1来得到，即为$\langle n, n+1\rangle$。
    - 基本步：$\langle 0, 0\rangle$

5.1.2 可表达性-4

- 自然数的前驱

$\langle u_1, u_2 \rangle = \lambda f.\, f\,(u_1)\,(u_2)$ (22.9)

$\mathbf{fst}(u) = u\,(\lambda x.\lambda y.x)$ (22.10)

$\mathbf{snd}(u) = u\,(\lambda x.\lambda y.y)$ (22.11)

从而有 $\mathrm{fst}(\langle u_1, u_2 \rangle) \cong u_1$ 和 $\mathrm{snd}(\langle u_1, u_2 \rangle) \cong u_2$

接下来，定义表示前驱的项**pred**

$\mathbf{prdpair} = \lambda x.x\,(\langle \overline{0}, \overline{0} \rangle)\,(\lambda y.\langle \mathbf{snd}(y), \mathbf{s}(\mathbf{snd}(y)) \rangle)$

x :是一个Church数值， y : 是取x 的前驱时所对应的序对

$\mathbf{pred} = \lambda x.\mathbf{fst}(\,\mathbf{prdpair}(x)\,)$

x :是一个Church数值

5.1.2 可表达性-5

- 定义一般递归式fix(x.u)
  - 方法：使用不动点组合式，其中之一称为Y组合式

$$\mathbf{Y} = \lambda F.\,(\lambda f.\,F\,(f\,(f\,)))\,(\lambda f.\,F\,(f\,(f\,)))$$

上式中有一个重复的结构，即$(\lambda f.\,F\,(f\,(f\,)))$。

为什么称为不动点组合式？因为 $\mathbf{Y}(F) \cong F(\mathbf{Y}(F))$

如何理解 Y ？

考虑求阶乘函数

$$\mathbf{g} = \lambda f.\,\lambda n.\ \mathbf{ifz}\ n\{\overline{0} \Rightarrow \overline{1} \mid \mathbf{s}(x) \Rightarrow \mathbf{times}\ (n)\ (f\ (\mathbf{pred}\ (n)))\}$$

n :是一个Church数值，*x* :是*n* 的前驱，即 pred (*n*) 。

$$\mathbf{fct} = \mathbf{Y}\ (\mathbf{g})$$

5.1.2 可表达性-6

$\mathbf{g = \lambda f.\ \lambda n.\ ifz\ n\{\bar{0} => \bar{1} \mid s(x) => times\ (n)\ (f\ (pred\ (n)))\}}$

$\mathbf{Y = \lambda F.\ (\lambda f.\ F\ (f\ (f\ )))\ (\lambda f.\ F\ (f\ (f\ )))}$ $\mathbf{fct = Y(g)}$

现在要求3的阶乘

$fct\ (\bar{3}) = \mathbf{Y}\ (g)\ (\bar{3})$

$\mapsto h\ (h)\ (\bar{3})$ 其中$h = \lambda f.g\ (f\ (f\ ))$

$\mapsto g\ (h\ (h))\ (\bar{3})$

$\mapsto (\lambda n.ifz\ n\{\bar{0} \Rightarrow \bar{1} \mid s(x) \Rightarrow times\ (n)\ (h\ (h)\ (pred(n)))\})\ (\bar{3})$

$\mapsto^{*} times\ (\bar{3})\ (h\ (h)\ (pred(\bar{3})))$

$\mapsto^{*} times\ (\bar{3})\ (h\ (h)\ (\bar{2}'))$ 其中$\bar{2}'$的行为等价于$\bar{2}$

注意：***h (h)***归约到***g (h (h))***，即***h (h)***是一种自重复运算符.
利用***h (h)***做一个递归调用，将展开***g*** 体的更多拷贝，并提供***h (h)***的一个新拷贝，.....

5.1.2 可表达性-7

g = λ*f*. λ*n*. ifz *n*{$\bar{0}$=> $\bar{1}$| s(*x*) => times (*n*) (*f* (pred (*n*)))}

Y = λ*F*. (λ*f*. *F* (*f* (*f*))) (λ*f*. *F* (*f* (*f*)))　　fct = Y(g)

现在要求3的阶乘

fct ($\bar{3}$) = **Y** (*g*) ($\bar{3}$)

↦ h (h) ($\bar{3}$)　　其中h = $\lambda f.g\,(f\,(f))$

↦ *g* (*h* (*h*)) ($\bar{3}$)

↦* times ($\bar{3}$) (*h* (*h*) ($\bar{2}'$))　　其中$\bar{2}'$的行为等价于$\bar{2}$

↦ times ($\bar{3}$) (*g* (*h* (*h*)) ($\bar{2}'$))

↦* times ($\bar{3}$) (times ($\bar{2}'$) (*g* (*h* (*h*)) ($\bar{1}'$)))　其中$\bar{1}'$的行为等价于$\bar{1}$

↦* times ($\bar{3}$) (times ($\bar{2}'$) (times ($\bar{1}'$) (*g* (*h* (*h*)) ($\bar{0}'$))))

其中$\bar{0}'$的行为等价于$\bar{0}$

↦* times ($\bar{3}$) (times ($\bar{2}'$) (times ($\bar{1}'$) ($\bar{1}$)))　↦* $\bar{6}$

5.1.3 未类型化是指单一类型化-1

- 未类型化λ演算可以正确地嵌入到L{μ ⇀}
  - μ包含递归类型
  - 嵌入的含义： λ 项可以表示成L{μ ⇀}中的表达式，该表达式的执行与对应的λ 项的执行行为相同。

理解这这种嵌入本质的重要性:

- 有助于为L{μ ⇀}中的λ演算编写解释器
- 更重要的是未类型化的λ项可以直接表示成L{μ ⇀}中的某种类型化表达式

未类型化λ演算是类型化语言的特例，提供了递归类型。

5.1.3 未类型化是指单一类型化-2

- 未类型化λ演算是单一类型化(uni-typed)λ演算
  - 未类型化λ演算不是没有类型，而是只有一种类型，即递归类型　　$D = \mathbf{rec}(\, t.\mathbf{arr}(t, t)\,)$
  - 类型D 的值(引入形式)：　$\mathbf{fold}[D](e)$
    e 是一个类型为$\mathbf{arr}(D, D)$的值，它是一个定义域和值域都是D 的函数。
  - 任意类型为$\mathbf{arr}(D, D)$的函数可以"折叠"成类型为D的值，任意类型为D的值可以"展开"成一个函数。
  - 同构递归：　$D \cong \mathrm{arr}(D, D)$
    类型D和D上的函数类型同构

5.1.3 未类型化是指单一类型化-3

❖ λ项u在L{μ⇀}中的嵌入形式$u^\dagger$

$$x^\dagger = x$$

$$\lambda(x.u)^\dagger = \mathbf{fold}[D](\mathbf{lam}[D]\ (x.u^\dagger)) \qquad (22.15)$$

$$\mathbf{uap}(u_1, u_2)^\dagger = \mathbf{ap}(\mathbf{unfold}(u_1^\dagger); u_2^\dagger)$$

➢ λ抽象的嵌入形式是值(递归类型的引入形式)

➢ 应用的嵌入形式指出函数通过展开递归类型而被应用

$$\begin{aligned}\mathrm{uap}(\lambda(x.u_1); u_2)^\dagger &= \mathrm{ap}(\mathrm{unfold}(\mathrm{fold}[D](\mathrm{lam}[D](x.u_1^\dagger))); u_2^\dagger)\\ &\cong \mathrm{ap}(\mathrm{lam}[D](x.u_1^\dagger); u_2^\dagger)\\ &\cong [u_2^\dagger / x]u_1^\dagger\\ &= ([u_2 / x]u_1)^\dagger\end{aligned}$$

上式表明β归约能被正确地实现

5.2 动态定型(Dynamic Typing)

未类型化的λ演算：是单一类型的语言,只有递归类型

其引入形式是函数定义,唯一的消去形式是函数应用

这种语言不会导致运行时错误,因为其消去形式的参数只能是函数.

如果未类型化的语言允许有不止一种值,就可能会导致运行时错误!

动态类型化:单一类型(type)的语言有多类别(class)的值

静态类型化:多类型化语言,如PCF、Gödel的T

5.2.1 动态类型化PCF

5.2.2 对动态定型的评论—时空开销

5.2.3 混合定型(hybrid typing)

5.2.4 动态定型的优化

5.2.5 静态定型对动态定型

5.2.6 通过递归类型进行混合定型

5.2.1 动态类型化PCF-1

❖ PCF的动态定型版本L{dyn}：抽象语法

Expr $d ::= x \mid \mathrm{num}(\overline{n}) \mid \mathrm{s}(d) \mid \mathrm{ifz}(d, d_0, x.d_1)$
$\mathrm{fun}(\lambda(x.d)) \mid \mathrm{dap}(d_1, d_2) \mid \mathrm{fix}(x.d) \mid \mathrm{error}$

L{dyn}中的每个值都被标上类别(class)，即num或fun。

- succ(*d*)是作用在类别num上的消去形式，而不是引入形式
- 未类型化λ抽象被显式标上类别fun，即fun(λ(*x*.*d*))

抽象语法	具体语法	
$\mathrm{num}(\overline{n})$	$\overline{n}$	自然数(num的引入形式)
$\mathrm{ifz}(d, d_0, x.d_1)$	$\mathrm{ifz}\ d\ \{ \mathrm{z} \Rightarrow d_0 \mid \mathrm{s}(x) \Rightarrow d_1 \}$	
$\mathrm{fun}(\lambda(x.d))$	$\lambda(x.d)$	函数定义(fun的引入形式)
$\mathrm{dap}(d_1, d_2)$	$d_1(d_2)$	函数应用(fun的消去形式)
$\mathrm{fix}(x.d)$	$\mathrm{fix}\ x\ \mathrm{is}\ d$	一般递归式，不动点

对于没有类别标记的值，必须由分析器在从具体语法分析变换到抽象语法的过程中来插入类别标记。

5.2.1 动态类型化PCF-2

- **L{dyn}的静态语义**
  - 本质上与未类型化的λ演算类似
  - 断言 x_1 ok, ..., x_n ok $\vdash d$ ok 声明d是包含出现在假言中的自由变量的合式表达式
- **L{dyn}的动态语义**
  - 值断言

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{val}} \qquad \frac{}{\mathtt{fun}(\lambda(x.d))\ \mathsf{val}} \qquad (23.1)$$

  - 断言运行时错误

$$\frac{}{\mathtt{error}\ \mathtt{err}}$$

  - 类别检查断言

$$\frac{}{\mathtt{num}(\overline{n})\ \mathsf{is\_num}\ \overline{n}} \qquad \frac{}{\mathtt{fun}(\lambda(x.d))\ \mathsf{is\_fun}\ \lambda(x.d)} \qquad (23.2)$$

  - 类别检查的否定断言

$$\frac{}{\mathtt{num}(\_)\ \mathsf{isnt\_fun}} \qquad \frac{}{\mathtt{fun}(\_)\ \mathsf{isnt\_num}} \qquad (23.3)$$

5.2.1 动态类型化PCF-3

❖L{dyn}的动态语义 (23.4)

d未完成求值,允许在后继下归约

$$\frac{d \mapsto d'}{\mathtt{s}(d) \mapsto \mathtt{s}(d')}$$

d是类别为num的值,则执行后继得到d的后继值

$$\frac{d\ \mathsf{is\_num}\ \overline{n}}{\mathtt{s}(d) \mapsto \mathtt{num}(\mathtt{s}(\overline{n}))}$$

d的类别不为num,则执行后继产生运行时错误

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{s}(d)\ \mathsf{err}}$$

d未完成求值,则允许在ifz下归约

$$\frac{d \mapsto d'}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto \mathtt{ifz}(d'; d_0; x.d_1)}$$

d是0，则归约到d0

$$\frac{d\ \mathsf{is\_num}\ \mathtt{z}}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto d_0}$$

d是自然数n的后继,则执行置换操作

$$\frac{d\ \mathsf{is\_num}\ \mathtt{s}(\overline{n})}{\mathtt{ifz}(d; d_0; x.d_1) \mapsto [\mathtt{num}(\overline{n})/x]d_1}$$

d的类别不为num,则执行ifz产生运行时错误

$$\frac{d\ \mathsf{isnt\_num}}{\mathtt{ifz}(d; d_0; x.d_1)\ \mathsf{err}}$$

5.2.1 动态类型化PCF-4

❖L{dyn}的动态语义 (23.4)

$$\frac{d_1 \mapsto d_1'}{\mathtt{dap}(d_1;d_2) \mapsto \mathtt{dap}(d_1';d_2)}$$

d1未完成求值,则允许在函数应用下归约

$$\frac{d_1\ \mathsf{val} \quad d_2 \mapsto d_2'}{\mathtt{dap}(d_1;d_2) \mapsto \mathtt{dap}(d_1;d_2')}$$

d1是值但d2未完成求值,则允许在函数应用下归约

$$\frac{d_1\ \mathsf{is\_fun}\ \lambda(x.d) \quad d_2\ \mathsf{val}}{\mathtt{dap}(d_1;d_2) \mapsto [d_2/x]d}$$

d1是λ抽象, d2是值，则执行置换

$$\frac{d_1\ \mathsf{isnt\_fun}}{\mathtt{dap}(d_1;d_2)\ \mathsf{err}}$$

d1不是函数，则执行应用将产生运行时错误

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

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

5.2.1 动态类型化PCF-5

- L{dyn}的安全性(PFPL Theorem 23.1)对于每个 d ok,或者d val,或者d err，或者存在d'使得$d \mapsto d'$.

L{dyn}与静态定型相比

- 优点：每个可分析的表达式都可以执行。
- 缺点：对于静态类型系统在编译时能排除的错误，要到执行时才会给出错误警告。

5.2.2 对动态定型的评论-1

与PCF相比，L{dyn}的动态语义表现出相当多的运行时开销——动态定型的开销！

例：加法函数

λ*x*.(fix *p* is λ*y*.ifz *y*{z=>*x* | s(*y*')=>s(*p*(*y*'))})

- 不动点表达式的体是λ抽象，分析器为之标上类别**fun**.

  fix结构的语义将*p*绑定到该λ抽象，因此为保证成功，这种递归调用会招致动态标记检查➔检查是冗余的

- 内层λ抽象应用的结果或者是*x*，或者是递归调用。

  后继操作的语义保证递归调用的结果被标上类别**num**，

  后继操作执行的标记检查可能失败的唯一原因是在最初调用时*x* 没有绑定到数值。

  ➔递归循环内的检查是冗余的

5.2.2 对动态定型的评论-2

与PCF相比，L{dyn}的动态语义表现出相当多的运行时开销——动态定型的开销！

例：加法函数

$\lambda x.(\text{fix } p \text{ is } \lambda y.\text{ifz } y\{z\Rightarrow x \mid s(y')\Rightarrow s(p(y'))\})$

- 内层λ抽象的参数y 或者出现在初始对加法函数的调用，或者是作为递归调用的结果来出现。

  如果初始调用把y 绑定到数值，则递归调用也必为数值，因为动态语义保证数值的前驱是数值。
  ——函数内部循环中的动态检查是冗余的

- 标记检查和创建标记是有时空开销的。这种检查和开销是无法避免的！

5.2.3 混合定型(hybrid typing)-1

❖L{nat dyn →}的语法

- 在L{nat →}(即PCF)的语法上扩展以下内容

  Type τ **::= dyn**

  Expr e **::= new[**l**](**e**) | cast[**l**](**e**)**

  Class l **::= num | fun**

  – **dyn**表示带类别标记的值(即**new[**l**](**e**)**)的类型(type)

  – l 表示标记的类别(class)

抽象语法	具体语法	
new[l**](**e**)**	l **!** e	**dyn**的引入形式，类别标记为l 的值
cast[l**](**e**)**	e**?**l	**dyn**的消去形式，将e 强制为类别l

cast以类别而不是以类型为参数,因为e的类型始终为**dyn.**

5.2.3 混合定型(hybrid typing)-2

❖L{nat dyn ⇀}的静态语义

- 在L{nat ⇀}的静态语义上扩展以下内容

dyn的引入规则：对类型为nat的e允许标记为num

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{new}[\mathtt{num}](e) : \mathtt{dyn}}$$

dyn的引入规则：对类型为parr的e允许标记为fun

$$\frac{\Gamma \vdash e : \mathtt{parr}(\mathtt{dyn};\mathtt{dyn})}{\Gamma \vdash \mathtt{new}[\mathtt{fun}](e) : \mathtt{dyn}}$$

(23.5)

dyn的消去规则：允许将dyn类型的e cast为num,所产生的表达式类型为nat

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{num}](e) : \mathtt{nat}}$$

dyn的消去规则：允许将dyn类型的e cast为fun,所产生的表达式类型为parr

$$\frac{\Gamma \vdash e : \mathtt{dyn}}{\Gamma \vdash \mathtt{cast}[\mathtt{fun}](e) : \mathtt{parr}(\mathtt{dyn};\mathtt{dyn})}$$

5.2.3 混合定型(hybrid typing)-3

❖L{nat dyn ⇀}的动态语义

- 在L{nat ⇀}的动态语义上扩展以下内容

e是值，则对其标记类别*l* 后仍是值

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

e未完成求值，允许在标记下归约

$$\frac{e \mapsto e'}{\mathtt{new}[l](e) \mapsto \mathtt{new}[l](e')}$$

e未完成求值，允许在cast下归约

$$\frac{e \mapsto e'}{\mathtt{cast}[l](e) \mapsto \mathtt{cast}[l](e')}$$

(23.6)

对标记为*l* 的值cast为t, 得到标记前的值

$$\frac{\mathtt{new}[l](e)\ \mathsf{val}}{\mathtt{cast}[l](\mathtt{new}[l](e)) \mapsto e}$$

对标记为*l* 的值cast为其他类别，会产生运行时错误

$$\frac{\mathtt{new}[l'](e)\ \mathsf{val} \quad l \neq l'}{\mathtt{cast}[l](\mathtt{new}[l'](e))\ \mathsf{err}}$$

5.2.3 混合定型(hybrid typing)-4

- L{nat dyn →}的安全性
  - 范式引理(Lemma23.2): 如果e:dyn且e val，则存在某个标记l和某个e' val，使得e=new[l](e').
    如果l =num,则e':nat; 如果l =fun,则e':parr(dyn,dyn).
  - 安全性(Theorem23.3):
    1.保持性（略）
    2.进展性（略）

5.2.3 混合定型(hybrid typing)-5

❖L{dyn}到L{nat dyn ⇀}的编译

*d*是L{dyn}中的表达式，则其编译到L{nat dyn ⇀}中类型为dyn 的表达式$e = d^{\dagger}$。

$x^{\dagger} = x$

$\mathbf{num}(\overline{n})^{\dagger} = \mathbf{new}[\mathbf{num}](n)$

$\mathbf{s}(d)^{\dagger} = \mathbf{new}[\mathbf{num}](\mathbf{s}(\mathbf{cast}[\mathbf{num}](d^{\dagger})))$

$\mathbf{ifz}(d; d_0; x.d_1)^{\dagger} = \mathbf{ifz}(\mathbf{cast}[\mathbf{num}](d^{\dagger}); d_0^{\dagger}; x'.[\mathbf{new}[\mathbf{num}](x')/x]d_1^{\dagger})$

$\mathbf{fun}(\lambda(x.d))^{\dagger} = \mathbf{new}[\mathbf{fun}](\mathbf{lam}[\mathbf{dyn}](x.d^{\dagger}))$

$\mathbf{dap}(d_1; d_2)^{\dagger} = \mathbf{ap}(\mathbf{cast}[\mathbf{fun}](d_1^{\dagger}); d_2^{\dagger})$

$\mathbf{fix}(x.d)^{\dagger} = \mathbf{fix}(x.d^{\dagger})$

5.2.3 混合定型(hybrid typing)-6

❖L{dyn}到L{nat dyn ⇀}的编译

*d*是L{dyn}中的表达式，则其编译到L{nat dyn ⇀}中类型为dyn 的表达式$e = d^{\dagger}$。

- 可靠性(Soundness)：如果$d \mapsto^{*} d'$，则$d^{\dagger} \mapsto^{*} (d')^{\dagger}$.
- 完备性(Completeness)：如果$d^{\dagger} \mapsto^{*} v$且$v$ val，则对于d' val且$d \mapsto^{*} d'$，有$v = (d')^{\dagger}$。

5.2.3 混合定型(hybrid typing)-7

例：加法函数在PCF、L{dyn}及L{nat dyn ⇀}中的表示

- PCF

```
fix[ parr(nat; parr(nat; nat)) ] (p.lam[nat](x.lam[nat](y.
  ifz(x; y; x'.s (ap (ap (p; y); x') ) ) ) ) )
```

- L{dyn}

```
fix( p.fun(λ(x. fun(λ(y.
  ifz(x; y; x'.s (dap (dap (p; y); x') ) ) ) ) ) )
```

- L{nat dyn ⇀}

```
fix [dyn] (p.new[fun] (lam[dyn](x. new[fun](lam[dyn](y.
  ifz(cast[num](x); y; x'.new[num]( cast[num](s (ap(cast[fun](
    ap(cast[fun](p); y)); new[num](x') ) ) ) ) ) ) ) ) ) )
```

5.2.4 动态定型的优化-1

- dyn类型：支持动态定型与静态定型的无缝集成
  - 可以充分利用静态类型的表达能力
  - 也允许在需要的时候进行灵活的动态定型
- 应用之一：利用静态显而易见的定型约束来优化动态类型化程序

例：加法函数

fun ! λ(*x*:dyn.fix *p*:dyn is fun ! λ(*y*:dyn. $e_{x,p,y}$))

$e_{x,p,y}$:dyn = ifz (*y*?num){z=>*x* |

s(*y*')=>num!(s(((*p*?fun) ((num!*y*')))?num))}

L{dyn}中：**λ*x*.(fix *p* is λ*y*.ifz *y*{z=>*x* | s(*y*')=>succ(*p*(*y*'))})**

5.2.4 动态定型的优化-2

$$\text{fun ! } \lambda(x\text{:dyn.fix } p\text{:dyn is fun ! } \lambda(y\text{:dyn. } e_{x,p,y}))$$

$$e_{x,p,y}\text{:dyn = ifz } (y?\text{num}) \{ z \Rightarrow x \mid s(y')\Rightarrow \text{num ! (s (((}p?\text{fun) ((num!}y'\text{)))?num))}\}$$

- fix体是带显式标记的函数

  当递归展开时，变量p绑定到类型为dyn的值，这时检查p是标记为fun的值的操作是冗余的，可以被消除。

可以重写加法函数为

$$\text{fun ! } \lambda(x\text{:dyn.fun ! fix } p\text{: dyn} \rightarrow \text{dyn is } \lambda(y\text{:dyn. } e'_{x,p,y}))$$

$$e'_{x,p,y} = \text{ifz } (y ? \text{num}) \{ z \Rightarrow x \mid s(y')\Rightarrow \text{num ! (s ((}p \text{ ((num!}y'\text{)))?num))}\}$$

  - 将fun标记检查提升到递归循环外;
  - 去除循环内的cast，即$(p?\text{fun})$变成p ;
  - 修改p 的类型。

5.2.4 动态定型的优化-3

fun ! λ(*x*:dyn.fun ! fix *p*: dyn⇀dyn is λ(*y*:dyn. $e'_{x,p,y}$))

$e'_{x,p,y}$ = ifz (*y*?num) {z=>*x* | s(*y'*)=>num!(s((*p* ((num!*y'*)))?num))}

- 参数*y* 的类型为dyn，每次循环迭代时，*y*在测试是否为零前被cast为num。

*y*的绑定或者出现在对加法函数的初始调用，或者出现在每次递归调用上，后者作用在 *y* 的前驱*y'*上。数值的前驱一定是数值。

为此，可以　fun ! λ(*x*:dyn.fun ! λ(*y* : dyn.e''_x (*y* ? num)))

e''_x = fix *p*: nat⇀dyn is λ(*y* : nat.ifz *y* { z=>*x* |
s(*y'*)=>num!(s((*p*(*y'*))?num)) }

- 将对*y* 的标记检查(*y*?num)提升到循环外;
- 去除循环内的cast，即(*y*?num)变成*y* ;
- 从而 *y* 的类型为nat ,函数类型变为 nat⇀dyn ;
- 但加法函数的类型仍必须为dyn⇀dyn ， 故将fix 结构处理为e''_x 。

5.2.4 动态定型的优化-4

$\mathbf{fun\ !\ \lambda(x{:}dyn.fun\ !\ \lambda(y : dyn.e''_x\ (y\ ?\ num)))}$

$\mathbf{e''_x = fix\ p{:}\ nat \rightharpoonup dyn\ is\ \lambda(y : nat.ifz\ y\ \{\ z{=>}x\ |\ s(y'){=>}num!(s(\ (p(y'))?num\ ))\ \}}$

- 检查递归调用的结果以保证类别为num(即$(p\ (y'))?num$)，如果是则将值增1，再标记为num。

  如果递归调用的结果来自x，则可能不为类别num，如果来自另一分支，则一定是num。

为此，可以 $\mathbf{fun\ !\ \lambda(x{:}dyn.fun\ !\ \lambda(y : dyn.\ num!\ e'''_x\ (y\ ?\ num)))}$

$\mathbf{e'''_x = fix\ p{:}\ nat \rightharpoonup nat\ is\ \lambda(y : nat.ifz\ y\ \{\ z{=>}x\ ?\ num\ |\ s(y'){=>}s(p(y'))\ \}}$

- 将x cast为num，它将检查x 是否是num,这样递归函数结果是nat;
- 去除对递归调用结果的cast$((p(y'))?num\ )$以及对后继的标记;
- 修改 p 的类型为$nat \rightharpoonup dyn$;
- 将标记num应用到函数结果，以恢复为类型dyn 。

5.2.5 静态定型对动态定型-1

❖ 一些误解和错误

- 静态类型系统将类型与变量相联系，动态类型系统将类型与值相联系。

  应为类别

  但L{dyn}中的λ抽象没有类型信息，而是将值标上类别

  类型和类别是不一样的！

- 动态语言在运行时检查类型，而静态语言在编译时检查类型。

  应为类别

  例如，对于函数应用，动态类别检查只检查第1个参数是否标记为**fun**，而不对函数体进行类型检查。

5.2.5 静态定型对动态定型-2

❖ 一些误解和错误

- 动态语言中的数据结构是异质的(heterogeneous)，而静态语言中的则是同质的。

例如，将list引入到L{dyn}

$$\text{cons}(\ s(z),\ \text{cons}(\lambda x.x,\ \text{nil})\ )$$

第1个元素是数值，第2个元素是函数，这样的list称为是异质的。

说静态语言只允许同质list是错误的！

在混合语言中，动态值均是类型为dyn的值，故上例的list也是同质的。

在静态和动态语言中，list是类型同质、类别异质的！

5.2.6 通过递归类型进行混合定型

- 5.2.3节中的dyn被处理为原语
  - 引入形式: **new[*l*](*e*)** **$l ! e$**
  - 消去形式: **cast[*l*](*e*)** **$e ? l$**
- 利用递归类型可以定义dyn

dyn := $\mu(t.[num:nat; fun:t \rightharpoonup t])$

dyn' := [num:nat; fun:dyn ⇀ dyn]

num!*e* := fold(in[num](*e*))

fun!*e* := fold(in[fun](*e*))

$e ?$ num := case unfold(*e*) { in[num](*x*)=>*x* | in[fun](*x*)=>error }

$e ?$ fun := case unfold(*e*) { in[num](*x*)=>error | in[fun](*x*)=>*x* }

作业

5.1利用Church布尔式，

1) 定义逻辑或函数;

2) 计算 snd(pair(v)(w))。

5.2定义以下在Church数值上的函数

1) 写出一种不用plus定义Church数值上的乘法。

2) 用pred定义一个减法函数。

5.3下面是列表在lambda演算中的一种表示，要求：

1)用该列表形式和不动点组合式Y写出一个函数，使得它能对Church数值列表中的数值求和。

2)用所写函数对列表 cons ($\overline{3}$) (cons ($\overline{4}$) (cons($\overline{2}$) (nil)))求值。

空表 **nil = pair (tt) (tt)**

非空表 **cons = λ*h*.λ*t*.pair (ff) (pair (*h*) (*t*))**

表头 **head = λz. fst(snd(z))** 表尾 **tail = λz. snd(snd(z))**

判断表是否为空 **isnil = λz.fst(z)**

作业

5.4 在L{nat dyn →}上增加布尔类型，定义静态语义和动态语义。

语法

τ ::= dyn | t | nat | parr(τ_1; τ_2) | unit | sum(τ_1; τ_2) | rec($t.\tau$)

e ::= x | z | s(e) | triv | in[l][τ](e) | in[r][τ](e) | if(e; e_1; e_2) | ifz(e; e_0; $x.e_1$)
| lam[τ]($x.e$) | ap(e_1; e_2) | fix[τ]($x.e$) | fold[$t.\tau$](e) | unfold(e) | error

l ::= num | bool | fun

Thanks!