中国科学技术大学
University of Science and Technology of China

类型检查II

《编译原理和技术(H)》、《编译原理(H)》

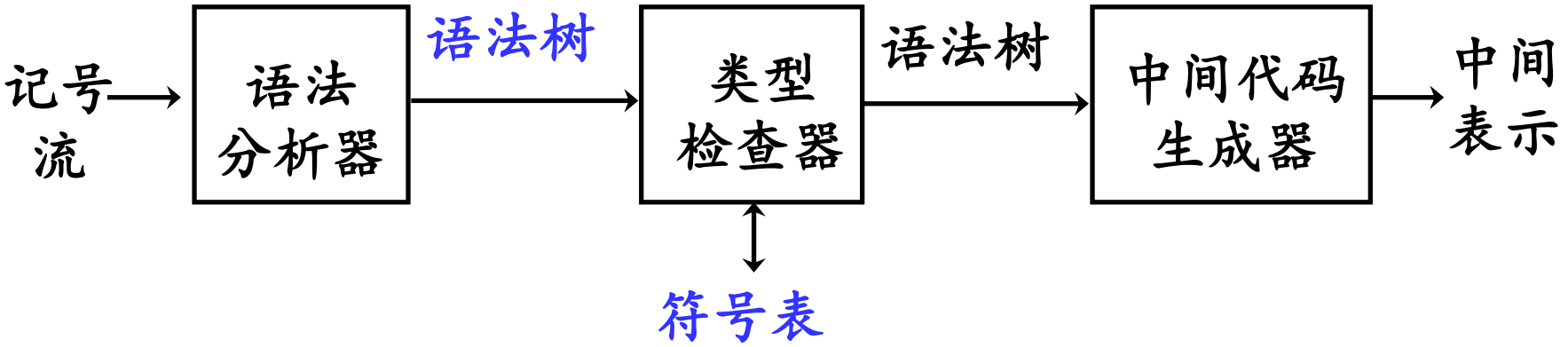
张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容



- 语义检查中最典型的部分——类型检查
  - 类型系统、类型检查、符号表的作用
  - 多态函数、重载
- 其他的静态检查（不详细介绍）
  - 控制流检查、唯一性检查、关联名字检查

中国科学技术大学
University of Science and Technology of China

5.4 类型表达式的等价

- 类型表达式的命名
- 名字等价、结构等价
- 记录类型的定义

类型表达式的等价

- 对类型表达式命名=〉如何解释类型表达式相同？
  - 结构等价、名字等价
  - 是类型表达式的一个语法约定，而不是引入新的类型

```
typedef cell *link;
link next;
link last;
cell  *p;
cell  *q, *r;
```

结构等价

结构等价

- 无类型名时，两个类型表达式完全相同
- 有类型名时，用类型名所定义的类型表达式代换它们，所得表达式完全相同（类型定义无环时）

```
typedef cell *link;
link next;
link last;
cell  *p;
cell  *q, *r;
```

next, last, p, q和r的类型是结构等价的

结构等价测试

- sequiv(s, t) (无类型名时)

```
if (s 和 t 是相同的基本类型)
    return true ;
else if (s == array(s1, s2) && t == array(t1, t2) )
    return sequiv(s1, t1) && sequiv(s2, t2) ;
else if (s == s1 × s2 && t == t1× t2 )
    return sequiv(s1, t1) && sequiv(s2, t2) ;
else if (s == pointer (s1) && t == pointer(t1) )
    return sequiv(s1, t1) ;
else if (s == s1 → s2 && t == t1 → t2 )
    return squiv(s1, t1) && sequiv(s2, t2) ;
else return false;
```

红色部分去掉则表示
忽略对数组的界的检查

名字等价

名字等价

- 把每个类型名看成是一个可区别的类型
- 两个类型表达式不做名字代换就结构等价

```
typedef cell *link;
link next;
link last;
cell  *p;
cell  *q, *r;
```

next和last的类型是名字等价的

p, q和r的类型是名字等价的

类型表达式的等价

Pascal语言的许多实现用隐含的类型名和每个声明的标识符联系起来，例如下面每个↑cell都隐含不同的类型名

```
type    link = ↑cell;
var     next   : link;
        last   : link;
        p      : ↑cell;
        q, r   : ↑cell;
```

```
type    link = ↑cell;
        np = ↑cell;
        nqr = ↑cell;
var     next : link;
        last : link;
        p : np;
        q : nqr;
        r : nqr;
```

p的类型与q和r的类型不是名字等价的

记录类型

记录类型

- 记录类型可看成其各个域类型的积类型
- 记录和积之间的主要区别是记录的域被命名

例如，C语言的记录类型

```
typedef struct {
        int address;
        char lexeme [15 ];
}row;
```

的类型表达式是

record(address : int, lexeme : *array*(15, char))

记录类型

定型规则

(Type Record)
(l_i是有区别的)

$$\frac{\Gamma \mid\!- T_1, \ldots, \Gamma \mid\!- T_n}{\Gamma \mid\!- \mathbf{record}(l_1{:}T_1, \ldots, l_n{:}T_n)}$$

(Val Record) (l_i是有区别的)

$$\frac{\Gamma \mid\!- M_1{:}T_1, \ldots, \Gamma \mid\!- M_n{:}\, T_n}{\Gamma \mid\!- \text{record}\ (l_1{=}M_1, \ldots, l_n{=}M_n) : \text{record}\ (l_1{:}T_1, \ldots, l_n{:}T_n)}$$

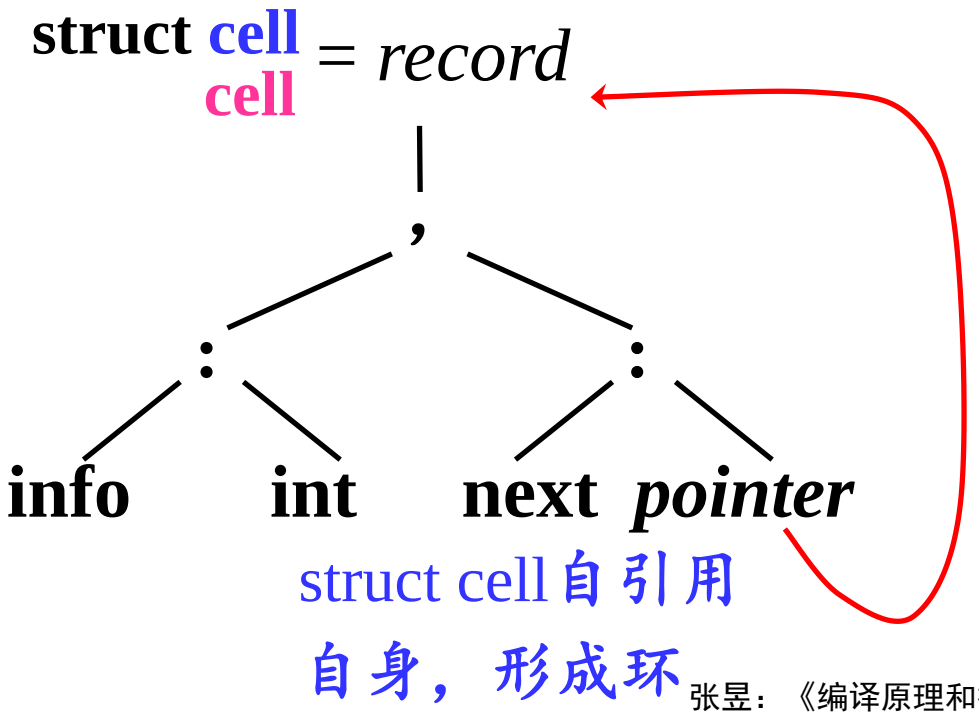
(Val Record Select)

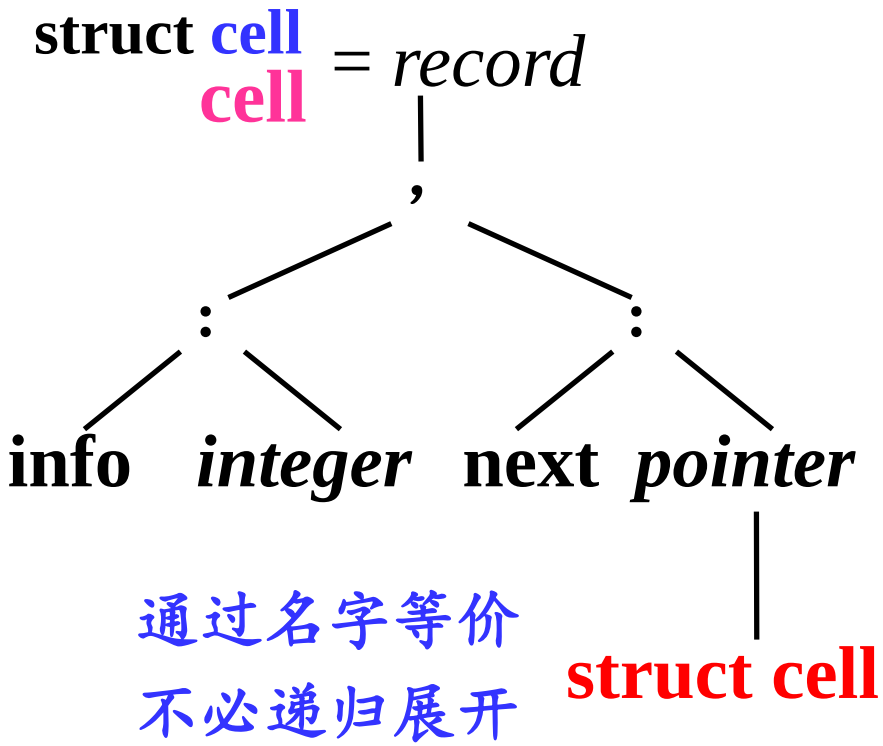
$$\frac{\Gamma \mid\!- M : \text{record}(l_1{:}T_1, \ldots, l_n{:}T_n)}{\Gamma \mid\!- M.l_i : T_i \quad (i \in 1..n)}$$

类型表示中的环

```
typedef struct cell {
        int info;
        struct cell *next;
} cell;
```

C语言对除**记录（结构体）**、**共用体**以外的所有类型使用结构等价，而对记录和共用体类型用的是名字等价，以避免类型图中的环。





例题 4

在X86/Linux机器上，编译器报告下面代码中蓝色行有错误：

incompatible types in return

```
typedef int A1[10];          | A2 *fun1( ) {
typedef int A2[10];          |     return(&a);
A1 a;                        | }
typedef struct {int i;}S1;   | S2  fun2( ) {
typedef struct {int i;}S2;   |     return(s);
S1 s;                        | }
```

S1和S2是不同的类型

在C语言中，数组和结构体都是构造类型，为什么上面第2个函数有类型错误，而第1个函数却没有？

中国科学技术大学
University of Science and Technology of China

*5.5 多态函数

- 参数化多态
- 类型系统的定义
- 类型检查

多态函数的引出

例 如何编写求表长的通用程序?

```
typedef struct {
    int info;
    link next;
} cell, *link;
```

unknown type name 'link'

因为link的定义在后

多态函数的引出

例 如何编写求表长的通用程序?

```
typedef struct cell{
    int info;
    struct cell *next;
} cell, *link;
```

计算过程与表元的数据类型无关，但语言的类型系统使该函数不能通用

```
int length(link lptr) {
    int len = 0;
    link p = lptr;
    while (p != NULL) {
        len++;
        p = p->next;
    }
    return len;
}
```

多态函数的引出

例　如何编写求表长的通用程序?

用ML语言很容易写出求表长的程序而不必管表元的类型

```
fun length (lptr) =
    if null (lptr) then 0
    else length (tl (lptr)) + 1;
```

tl- 返回表尾　　null-测试表是否为空

length ([“sun”, “mon”, “tue”])

length ([10, 9, 8])

都等于3

参数化多态 与 Ad-hoc多态

- 多态函数(polymorphic functions) 参数化多态
  - 允许函数参数的类型有多种不同的情况
  - 函数体中语句的执行能适应参数为不同类型的情况——相同的实现

- 多态算符(polymorphic operators) Ad-hoc多态
  - 例如：数组索引、函数应用、通过指针间接访问相应操作的代码段接受不同类型的数组、函数等
  - 多态算符针对不同的参数类型有不同的实现
  - C语言手册中关于取地址算符&的论述是：

如果运算对象的类型是‘...’，那么结果类型是指向‘...’的指针”

类型变量及其应用

类型变量

- length的类型可以写成$\forall\alpha.list(\alpha) \rightarrow integer$
- 类型变量的引入便于讨论未知类型
  - 如，在不要求标识符先声明再使用的语言中，可通过类型变量确定程序变量的类型

例如，如下假想的程序

```
function deref (p){
    return *p;
}
```

-- 对p的类型一无所知：β

-- $\beta = pointer(\alpha)$

deref 的类型是$\forall\alpha.\ pointer(\alpha) \rightarrow \alpha$

多态函数的类型系统

一个含多态函数的语言

$P \rightarrow D; E$

$D \rightarrow D; D \mid \text{id} : Q$

$Q \rightarrow \forall \text{type-variable}.\ Q$ 多态函数

$\quad \mid T$

$T \rightarrow T \text{ ‘}\rightarrow\text{’ } T$

$\quad \mid T \times T$ 笛卡儿积类型

$\quad \mid \text{unary-constructor} (T)$

$\quad \mid \text{basic-type}$

$\quad \mid \text{type-variable}$ 引入类型变量

$\quad \mid (T)$

$E \rightarrow E (E) \mid E, E \mid \text{id}$

这是一个抽象语言，忽略了函数定义的函数体

多态函数的类型系统

一个含多态函数的语言

$P \rightarrow D; E$

$D \rightarrow D; D \mid \text{id} : Q$

$Q \rightarrow \forall \text{type-variable}.\ Q$

$\quad \mid T$

$T \rightarrow T \text{ ‘}\rightarrow\text{’ } T$

$\quad \mid T \times T$

$\quad \mid \text{unary-constructor} (T)$

$\quad \mid \text{basic-type}$

$\quad \mid \text{type-variable}$

$\quad \mid (T)$

$E \rightarrow E (E) \mid E, E \mid \text{id}$

一个程序：

$\text{deref} : \forall \alpha.\ \text{pointer}(\alpha) \rightarrow \alpha ;$

$q : \text{pointer} (\text{pointer} (\text{integer}));$

$\text{deref} (\text{deref} (q))$

多态函数的类型系统

类型系统中增加的推理规则

- 环境规则　　类型变量α加到定型环境中

(Env Var)

$$\frac{\Gamma \vdash \Diamond,\ \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma, \alpha \vdash \Diamond}$$

- 语法规则

(Type Var)

$$\frac{\Gamma_1, \alpha, \Gamma_2 \vdash \Diamond}{\Gamma_1, \alpha, \Gamma_2 \vdash \alpha}$$

(Type Product)

$$\frac{\Gamma \vdash T_1,\ \Gamma \vdash T_2}{\Gamma \vdash T_1 \times T_2}$$

多态函数的类型系统

类型系统中增加的推理规则

语法规则

(Type Parenthesis)

$$\frac{\Gamma \vdash T}{\Gamma \vdash (T)}$$

(Type Forall)

$$\frac{\Gamma, \alpha \vdash T}{\Gamma \vdash \forall \alpha.T}$$

(Type Fresh) 类型变量换名（α_i 不在Γ中）

$$\frac{\Gamma \vdash \forall \alpha.T, \quad \Gamma, \alpha_i \vdash \Diamond}{\Gamma, \alpha_i \vdash [\alpha_i / \alpha]\, T}$$

多态函数的类型系统

■ 定型规则

(Exp Pair)
$$\frac{\Gamma \mid\!- E_1 : T_1,\ \Gamma \mid\!- E_2 : T_2}{\Gamma \mid\!- E_1, E_2 : T_1 \times T_2}$$

(Exp FunCall)
$$\frac{\Gamma \mid\!- E_1 : T_1 \to T_2,\ \Gamma \mid\!- E_2 : T_3}{\Gamma \mid\!- E_1\ (E_2) : S(T_2)}$$

(其中S是T_1和T_3的最一般的合一代换)

代换：类型表达式中的类型变量用其所代表的类型表达式去替换

subst(*t*:*type_exp*, *Sv*: *type_var*→ *type_exp*):*type_exp*

实例：把*subst*函数用于*t*后所得的类型表达式是*t* 的一个实例，用*S*(*t*)表示

代换和实例

typeExpression *subst* (typeExpression *t*, *typeVar*→typeExpression ***Sv***) {

if (*t* 是基本类型) return *t* ;

else if (*t* 是类型变量) return *Sv*(*t*);

else if (*t* 是$t_1 \rightarrow t_2$) return *subst*(t_1, ***Sv***) → *subst*(t_2, ***Sv***);

}

例子（$s < t$ 表示s是t 的实例，α、β是类型变量）

pointer(*integer*) < *pointer*(α)　　*pointer*(*real*) < *pointer*(α)

integer → *integer* < $\alpha \rightarrow \alpha$　　*pointer*(α) < β

$\alpha < \beta$

不合法的实例

例 下面左边的类型表达式不是右边的实例

integer	*real*
	代换不能用于基本类型
$integer \rightarrow real$	$\alpha \rightarrow \alpha$
	α的代换不一致
$integer \rightarrow \alpha$	$\alpha \rightarrow \alpha$
	α的所有出现都应该代换

合一

- 合一(unify)
  - 如果存在某个代换 Sv 使得 $S(t_1) = S(t_2)$，那么这两个表达式t_1和 t_2能够合一
- 最一般的合一代换(the most general unifier) S
  - $S(t_1) = S(t_2)$;
  - 对任何其它满足$S'(t_1) = S'(t_2)$的代换Sv'，代换$S'(t_1)$是$S(t_1)$的实例

例如， $t_1 = pointer(\text{list}(\alpha))$ $t_2 = pointer(\beta)$

代换Sv: $\alpha \rightarrow \alpha$, $\beta \rightarrow \text{list}(\alpha)$，使得$S(t_1) = S(t_2) = pointer(\text{list}(\alpha))$

代换Sv' : $\alpha \rightarrow \text{integer}$, $\beta \rightarrow \text{list(integer)}$，使得$S'(t_1) = S'(t_2) = pointer(\text{list(integer)})$

代换Sv是最一般的合一代换

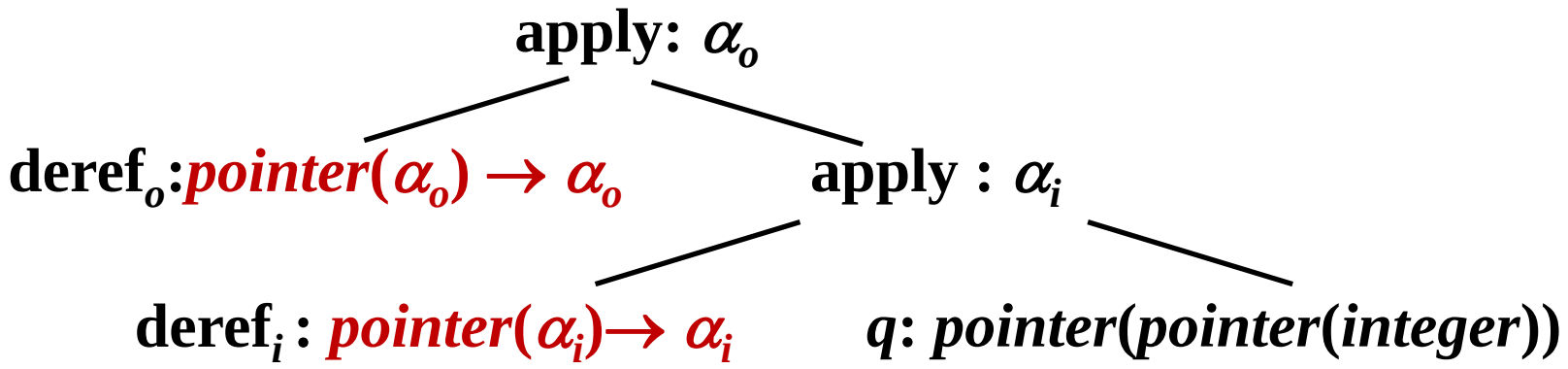
多态函数的类型检查

多态函数和普通函数在类型检查上的区别

(1) 同一多态函数的不同出现不要求变元/参数有相同类型

(2) 必须把类型相同的概念推广到类型合一

(3) 要记录类型表达式合一的结果



deref(deref (*q*))的带标记的语法树

检查多态函数的翻译方案

$E \rightarrow E_1 (E_2)$

{ p = *mkleaf* (*newtypevar*); // 返回类型

unify (E_1. *type*, *mknode* ('→', E_2.*type*, p));

E. *type* = p; }

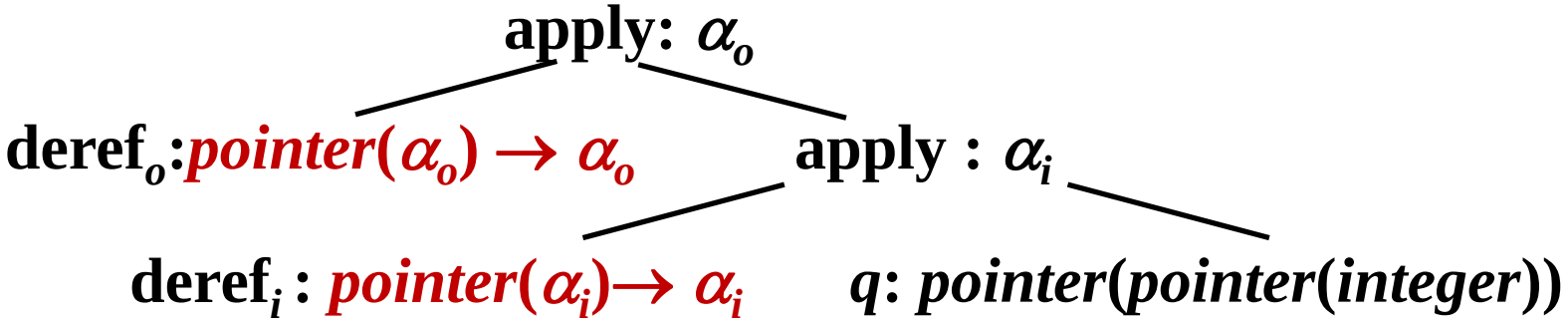
$E \rightarrow E_1, E_2$

{E. *type* = *mknode* ('×', E_1.*type* , E_2.*type*); }

$E \rightarrow$ id

{E. *type* = *fresh* (*lookup*(id.*entry*)); } // 类型变量

例：多态函数的检查



表达式：类型	代换
q : pointer(pointer(integer))	
deref$_i$: pointer(α_i) → α_i	
deref$_i$(q) : pointer(integer)	α_i = pointer(integer)
deref$_o$: pointer(α_o) → α_o	
deref$_o$(deref$_i$ (q)) : integer	α_o = integer

求表长的函数的检查

length : β ;　　lptr : γ ;

if : $\forall\alpha$. *boolean* $\times\alpha\times\alpha\rightarrow\alpha$;

null : $\forall\alpha$. *list* (α) $\rightarrow$ *boolean* ;

tl : $\forall\alpha$. *list* (α) $\rightarrow$ *list* (α) ;

0 : *integer* ; 1 : *integer* ;

+ : *integer* $\times$ *integer* $\rightarrow$ *integer* ;

match : $\forall\alpha$. $\alpha\times\alpha\rightarrow\alpha$;

```
match (                          -- 表达式，匹配length函数的
    length (lptr),               -- 函数首部和函数体的类型
    if (null (lptr), 0, length (tl(lptr)) +1)
)
```

```
fun length (lptr) =
    if null (lptr) then 0
    else length (tl (lptr)) + 1;
```

类型声明部分

求表长的函数的检查

行	定 型 断 言	代 换	规 则
(1)	lptr : γ		(Exp Id)
(2)	length : β		(Exp Id)
(3)	length(lptr) : δ	$\beta = \gamma \rightarrow \delta$	(Exp FunCall)
(4)	lptr : γ		从(1)可得
(5)	null : $list(\alpha_n) \rightarrow boolean$		(Exp Id)和 (Type Fresh)
(6)	null(lptr) : *boolean*	$\gamma = list\ (\alpha_n)$	(Exp FunCall)
(7)	0 : *integer*		(Exp Num)
(8)	lptr : $list(\alpha_n)$		从(1)可得

求表长的函数的检查

行	定 型 断 言	代 换	规 则
(9)	tl : $list(\alpha_t) \rightarrow list(\alpha_t)$		(Exp Id)和 (Type Fresh)
(10)	tl(lptr) : $list(\alpha_n)$	$\alpha_t = \alpha_n$	(Exp FunCall)
(11)	length : $list(\alpha_n) \rightarrow \delta$		从(2)可得
(12)	length(tl(lptr)) : δ		(Exp FunCall)
(13)	1 : *integer*		(Exp Num)
(14)	+ : $integer \times integer \rightarrow integer$		(Exp Id)

求表长的函数的检查

行	定 型 断 言	代 换	规 则
(15)	length (tl(lptr)) +1 : *integer*	δ = *integer*	(Exp FunCall)
(16)	if : *boolean* $\times\alpha_i \times\alpha_i \rightarrow\alpha_i$		(Exp Id)和 (Type Fresh)
(17)	if (...) : *integer*	α_i = *integer*	(Exp FunCall)
(18)	match : $\alpha_m \times \alpha_m \rightarrow\alpha_m$		(Exp Id)和 (Type Fresh)
(19)	match (…) : *integer*	α_m=*integer*	(Exp FunCall)

length函数的类型是 $\forall\alpha.\ list(\alpha) \rightarrow integer$

中国科学技术大学
University of Science and Technology of China

5.6 函数和算符重载

- Ad-hoc多态
- 可能的类型集合及其缩小
- 附加：子类型关系引起的协变和逆变

重载

重载符号

- 有多个含义，但在每个引用点的含义都是唯一的

例如：

- 加法算符+可用于不同类型，“+”是多个函数的名字，而不是一个多态函数的名字
- 在Ada中，( ) 是重载的，*A*(*I*) 有不同含义

重载的消除

- 在重载符号的引用点，其含义能确定到唯一

表达式的可能类型集合

例 Ada语言

声明：

function "*" (*i, j* : integer) return complex;

function "*" (*x, y* : complex) return complex;

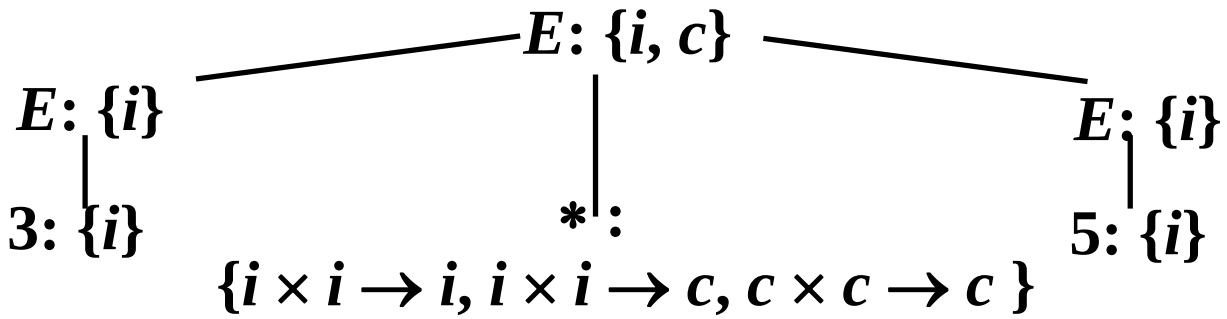
使得算符*重载，可能的类型包括：

integer* × *integer* → *integer　　--这是预定义的类型

integer* × *integer* → *complex

complex* × *complex* → *complex

(3 * 5) * *z*（*z:complex*）



重载函数的应用

- 缩小可能类型的集合：

综合属性types：可能的类型集合

- $E' \rightarrow E$　　$E'.types = E.types$;
- $E \rightarrow \mathrm{id}$　　$E.\ types = lookup(\mathrm{id}.\ entry)$;
- $E \rightarrow E_1(E_2)$　　$E.\ types = \{s' \mid E_2.\ types$中存在一个$s$，使得$s \rightarrow s'$属于$E_1.\ types\}$;

重载函数的应用

- 缩小可能类型的集合：Ada 要求完整的表达式有唯一的类型

  - $E' \rightarrow E$ 　　$E'.types = E.types;$

    继承属性

    if ($E'.types == \{ t \}$) $E.unique = t$; else $E.unique = type\_error$;

    $E'.code = E.code;$

  - $E \rightarrow \mathrm{id}$ 　　$E.types = lookup(\mathrm{id}.entry);$

    $E.code = gen(\mathrm{id}.lexeme, ':', E.unique);$

  - $E \rightarrow E_1(E_2)$ 　　$E.types = \{s' \mid E_2.types$中存在一个$s$，使得$s \rightarrow s'$属于$E_1.types\}$;

    $t = E.unique;$

    $S = \{s \mid s \in E_2.types \ \&\& \ s \rightarrow t \in E_1.types \}$;

    if ($S == \{ s \}$) $E_2.unique = s$; else $E_2.unique = type\_error$;

    if ($S == \{ s \}$) $E_1.unique = s \rightarrow t$; else $E_1.unique = type\_error$;

    $E.code = E_1.code \ || \ E_2.code \ || \ gen('apply', ':', E.unique)$;

非L属性定义

只有归约到开始非终结符时，才能确定唯一类型信息

不能边解析边类型检查

重载函数的应用

- 缩小可能类型的集合：Ada 要求完整的表达式有唯一的类型

$E' \rightarrow E$	$E'.types = E.types;$ if ($E'.types == \{ t \}$) $E.unique = t$; else $E.unique = type\_error$; $E'.code = E.code;$
$E \rightarrow \mathrm{id}$	$E.types = lookup(\mathrm{id}.entry);$ $E.code = gen(\mathrm{id}.lexeme, ':', E.unique);$

$E \rightarrow E_1(E_2)$	$E.types = \{s' \mid E_2.types$中存在一个$s$，使得$s \rightarrow s'$属于$E_1.types\}$;
	$t = E.unique;$
	$S = \{s \mid s \in E_2.types \;\&\&\; s \rightarrow t \in E_1.types\}$;
	if ($S == \{ s \}$) $E_2.unique = s$; else $E_2.unique = type\_error$;
	if ($S == \{ s \}$) $E_1.unique = s \rightarrow t$; else $E_1.unique = type\_error$;
	$E.code = E_1.code \,\|\, E_2.code \,\|\, gen('apply', ':', E.unique);$

基于树访问的类型检查

简单地，可以采取两遍访问

①在exit时计算*types*

②在enter时计算*unique*，在exit时计算*code*

附加：子类型 - 协变和逆变

- 子类型关系 <
  - 类型上的偏序关系 τ
  - 满足包含原理：如果s是t的子类型，则需要类型为t的值时，都可以将类型为s的值提供给它
- 协变（covariant）t<t', 则c(t)<c(t')
  - 函数类型在值域上是协变的

    假设$e:\sigma\rightarrow\tau$，$e1:\sigma$，则$e(e1):\tau$. 如果$\tau<\tau'$，则$e(e1):\tau'$.
- 逆变（contravariant）t<t', 则c(t')<c(t)
  - 函数类型在定义域上是逆变的

    假设$e:\sigma\rightarrow\tau$，$e1:\sigma'$，如果$\sigma'<\sigma$，则$e(e1):\tau$.

例　题　5

编译器和连接装配器未能发现下面的调用错误

```
long gcd (p, q) long p, q;{/*这是参数声明的传统形式*/
    /*参数声明的现代形式是long gcd ( long p, long q) { */
    if (p%q == 0)
            return q;
    else
            return gcd (q, p%q);
}
main() {
    printf("%ld,%ld\n", gcd(5), gcd(5,10,20));
}
```