



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

# 类型检查 II

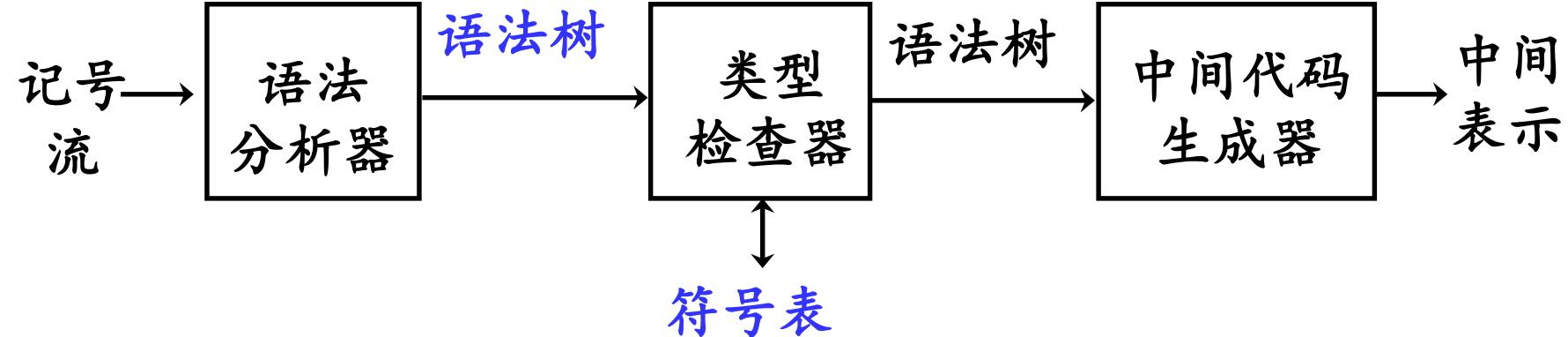
《编译原理和技术(H)》

张昱

0551-63603804, [yuzhang@ustc.edu.cn](mailto: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



## 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 $\Rightarrow$ 如何解释类型表达式相同？

- 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;
```

- 引入类型名后，类型表达式的等价就有**结构等价、名字等价**两种不同的定义



## □ 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 sequiv(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

- 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 $\uparrow$ cell都隐含不同的类型名

type link = $\uparrow$ cell;	type link = $\uparrow$ cell;
var next : link;	np = $\uparrow$ cell;
last : link;	nqr = $\uparrow$ cell;
p : $\uparrow$ cell;	var next : link;
q, r : $\uparrow$ cell;	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) )*



## □ 定型规则

(Type Record)

( $l_i$ 是有区别的)

$$\frac{\Gamma \vdash T_1, \dots, \Gamma \vdash T_n}{\Gamma \vdash \text{record}(l_1:T_1, \dots, l_n:T_n)}$$

(Val Record) ( $l_i$ 是有区别的)

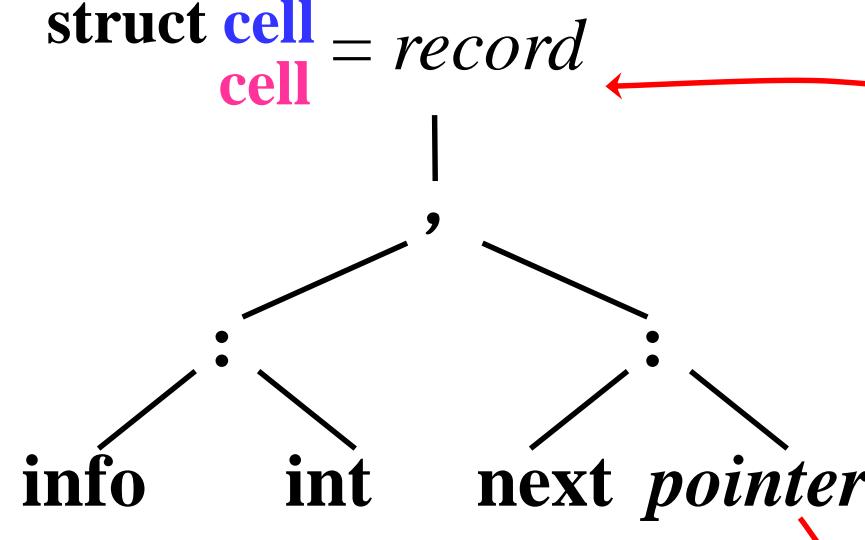
$$\frac{\Gamma \vdash M_1:T_1, \dots, \Gamma \vdash M_n: T_n}{\Gamma \vdash \text{record } (l_1=M_1, \dots, l_n=M_n) : \text{record } (l_1:T_1, \dots, l_n:T_n)}$$

(Val Record Select)

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

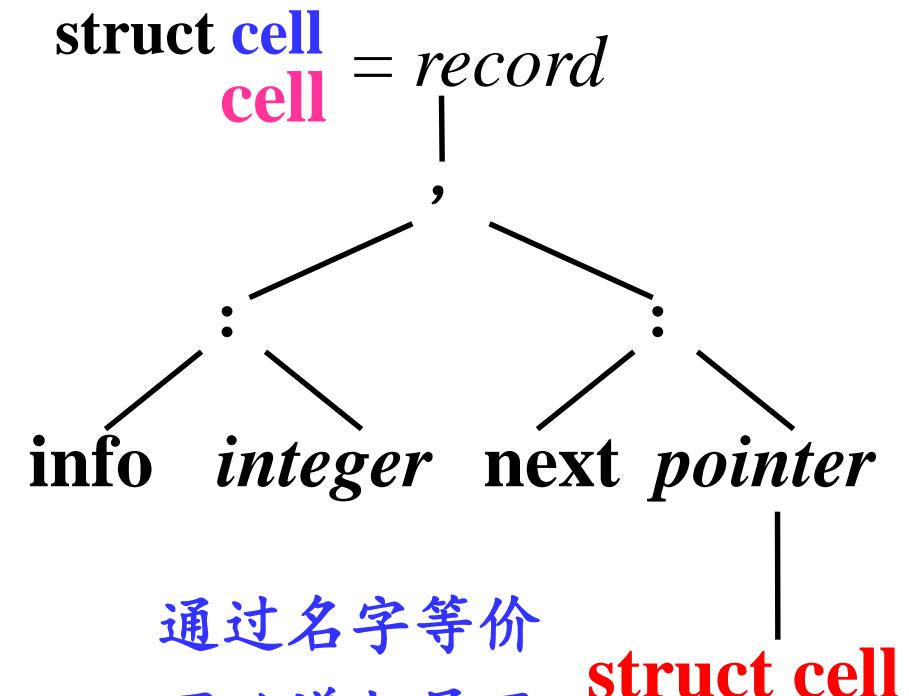
# 类型表示中的环

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



struct cell 自引用  
自身，形成环

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？



## \*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;
```



编译没报错？

那用选项 -Wall 再试试  
<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

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



## □ 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

- 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:  $\beta$

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

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

$P \rightarrow D; E$

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

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

$/ T$

$T \rightarrow T' \rightarrow' T$

$/ T \times T$  笛卡儿积类型

$/ \text{unary-constructor ( } T \text{ )}$

$/ \text{basic-type}$

$/ \text{type-variable}$  引入类型变量

$/ ( T )$

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

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

$P \rightarrow D; E$

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

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

$/ T$

$T \rightarrow T ' \rightarrow ' T$

$/ T \times T$

$/ \text{unary-constructor} ( T )$

$/ \text{basic-type}$

$/ \text{type-variable}$

$/ ( T )$

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

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 $\alpha$ 加到定型环境中

(Env Var)

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

■ 语法规则

(Type Var)

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

(Type Product)

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



## □ 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 ( $\alpha_i$  不在  $\Gamma$  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)

$$\frac{\Gamma \vdash E_1 : T_1, \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1, E_2 : T_1 \times T_2}$$

## (Exp FunCall)

$$\frac{\Gamma \vdash E_1 : T_1 \rightarrow T_2, \quad \Gamma \vdash E_2 : T_3}{\Gamma \vdash E_1(E_2) : S(T_2)}$$

(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

$\text{subst}(t:\text{type\_exp}, S_v: \text{type\_var} \rightarrow \text{type\_exp}): \text{type\_exp}$

实例：把  $\text{subst}$  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( $t_1$ ,  $Sv$ )  $\rightarrow$  subst( $t_2$ ,  $Sv$ );  
}
```

例子 ( $s < t$  表示  $s$  是  $t$  的实例,  $\alpha$ 、 $\beta$  是类型变量)

$pointer(\text{integer}) < pointer(\alpha)$

$pointer(\text{real}) < pointer(\alpha)$

$\text{integer} \rightarrow \text{integer} < \alpha \rightarrow \alpha$

$pointer(\alpha) < \beta$

$\alpha < \beta$



# 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 → real*

$\alpha \rightarrow \alpha$

$\alpha$  的代换不一致

*integer → α*

$\alpha \rightarrow \alpha$

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

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

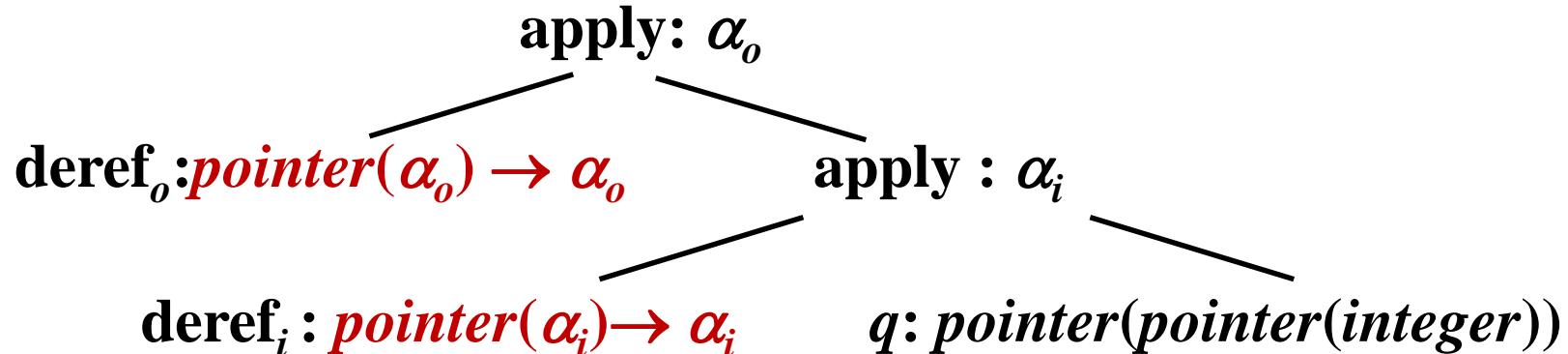
代换  $Sv': \alpha \rightarrow \text{integer}, \beta \rightarrow \text{list}(\text{integer})$ , 使得  $S'(t_1) = S'(t_2) = \text{pointer}(\text{list}(\text{integer}))$

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 (E1.type, mknoden ('→', E2.type, p));  
E.type = p; }
```

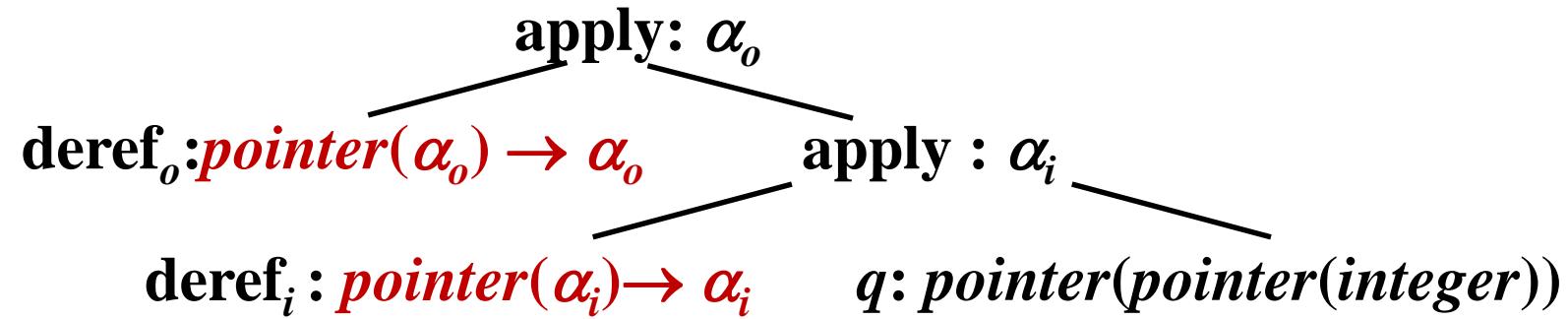
$E \rightarrow E_1, E_2$

```
{E.type = mknoden ('×', E1.type , E2.type); }
```

$E \rightarrow \text{id}$

```
{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
$q : \text{pointer(pointer(integer))}$	
$\text{deref}_i : \text{pointer}(\alpha_i) \rightarrow \alpha_i$	
$\text{deref}_i(q) : \text{pointer(integer)}$	$\alpha_i = \text{pointer(integer)}$
$\text{deref}_o : \text{pointer}(\alpha_o) \rightarrow \alpha_o$	
$\text{deref}_o(\text{deref}_i(q)) : \text{integer}$	$\alpha_o = \text{integer}$



# 求表长的函数的检查

length :  $\beta$  ;      lptr :  $\gamma$  ;  
if :  $\forall \alpha. boolean \times \alpha \times \alpha \rightarrow \alpha$  ;  
null :  $\forall \alpha. list(\alpha) \rightarrow boolean$  ;  
tl :  $\forall \alpha. list(\alpha) \rightarrow list(\alpha)$  ;  
0 : integer ; 1 : integer ;  
+ : integer  $\times$  integer  $\rightarrow$  integer ;  
match :  $\forall \alpha. \alpha \times \alpha \rightarrow \alpha$  ;  
match (  
    length (lptr),                          -- 表达式，匹配length函数的  
    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

-- 表达式，匹配length函数的

-- 函数头部和函数体的类型



# 求表长的函数的检查

行	定型断言	代换	规则
(1)	$\text{lptr} : \gamma$		(Exp Id)
(2)	$\text{length} : \beta$		(Exp Id)
(3)	$\text{length}(\text{lptr}) : \delta$	$\beta = \gamma \rightarrow \delta$	(Exp FunCall)
(4)	$\text{lptr} : \gamma$		从(1)可得
(5)	$\text{null} : \text{list}(\alpha_n) \rightarrow \text{boolean}$		(Exp Id)和 (Type Fresh)
(6)	$\text{null}(\text{lptr}) : \text{boolean}$	$\gamma = \text{list}(\alpha_n)$	(Exp FunCall)
(7)	$0 : \text{integer}$		(Exp Num)
(8)	$\text{lptr} : \text{list}(\alpha_n)$		从(1)可得



# 求表长的函数的检查

行	定型断言	代换	规则
(9)	$\text{tl} : \text{list}(\alpha_t) \rightarrow \text{list}(\alpha_t)$		(Exp Id)和 (Type Fresh)
(10)	$\text{tl(lptr)} : \text{list}(\alpha_n)$	$\alpha_t = \alpha_n$	(Exp FunCall)
(11)	$\text{length} : \text{list}(\alpha_n) \rightarrow \delta$		从(2)可得
(12)	$\text{length}(\text{tl(lptr)}) : \delta$		(Exp FunCall)
(13)	$1 : \text{integer}$		(Exp Num)
(14)	$+ : \text{integer} \times \text{integer} \rightarrow \text{integer}$		(Exp Id)



# 求表长的函数的检查

行	定型断言	代换	规则
(15)	$\text{length}(\text{tl(lptr)}) + 1 : \text{integer}$	$\delta = \text{integer}$	(Exp FunCall)
(16)	$\text{if} : \text{boolean} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$		(Exp Id)和 (Type Fresh)
(17)	$\text{if}(\dots) : \text{integer}$	$\alpha_i = \text{integer}$	(Exp FunCall)
(18)	$\text{match} : \alpha_m \times \alpha_m \rightarrow \alpha_m$		(Exp Id)和 (Type Fresh)
(19)	$\text{match}(\dots) : \text{integer}$	$\alpha_m = \text{integer}$	(Exp FunCall)

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



## 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;**

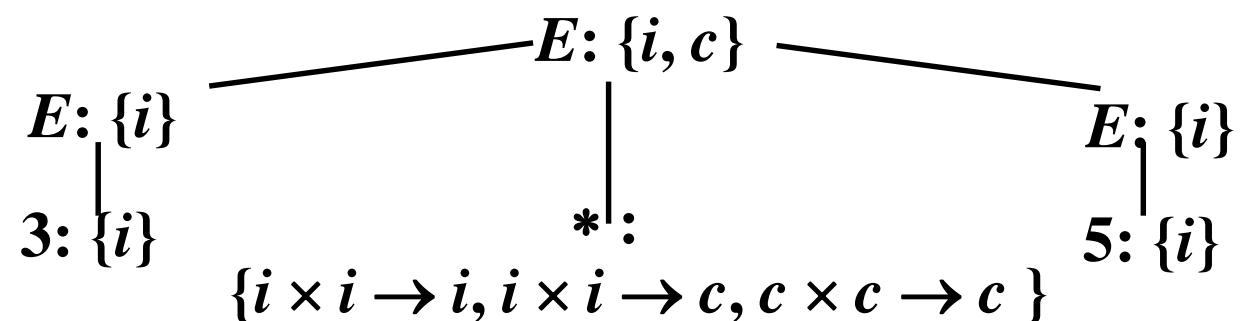
使得算符\*重载， 可能的类型包括：

*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：

■  $E' \rightarrow E$

$E'.types = E.types;$

综合属性types：可能的类型集合

■  $E \rightarrow id$

$E.types = lookup(id.entry);$

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



# 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

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

$E'.code = E.code;$

■  $E \rightarrow \text{id}$

$E.types = \text{lookup}(\text{id}.entry);$

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

■  $E \rightarrow E_1(E_2)$

$E.types = \{s' | E_2.types \text{ 中存在一个 } s, \text{ 使得 } s \rightarrow s' \text{ 属于 } E_1.types\};$

$t = E.unique;$

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

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

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

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

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



## □ 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;
```

■  $E \rightarrow \text{id}$

```
E.types = lookup(id.entry);
E.code = gen(id.lexeme, ':', E.unique);
```

■  $E \rightarrow E_1(E_2)$

```
E.types = {s' | E_2.types 中存在一个 s, 使得 s → s' 属于 E_1.types};
t = E.unique;
```

```
S = {s / s ∈ E_2.types && s → t ∈ E_1.types} ;
if (S == { s }) E_2.unique = s; else E_2.unique = type_error;
if ( S == { s }) E_1.unique = s → 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  $\tau$
- 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));  
}
```