

第六章 类型检查



内容

- 类型系统
- 类型表达式的等价
- 类型转换
- 函数和运算符的重载
- 多态函数
- 一致化算法



静态检查 (static checking)

1. 类型检查 (type check)

- 操作对象必须与操作符匹配：函数名相加×

2. 控制流检查 (flow-of-control check)

- break必须退出while、for、switch...

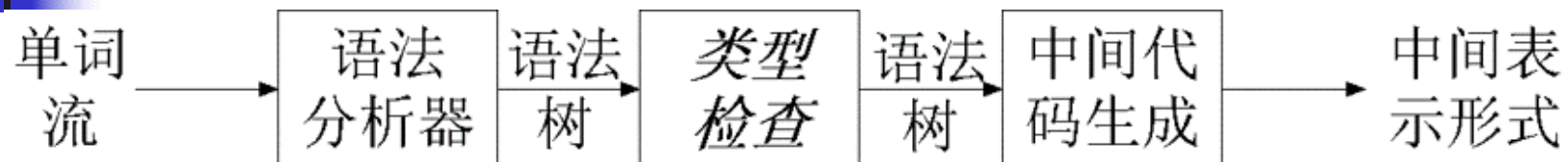
3. 唯一性检查 (uniqueness check)

- 对象（变量、标号...）定义必须唯一

4. 名字关联检查 (name-related check)

- 相同名字在不同位置

类型检查



- 检查语法结构的类型与上下文匹配
 - ▣ 简单的类型检查
 - ▣ 两个类型的匹配
- 代码生成要利用类型信息
- 重载，多态



6.1 类型系统

- 语法结构、类型、将类型赋予语法结构的规则
 - $+$, $-$, $*$ 的两个运算数均为整数，结果为整数
 - $\&$ 的结果为指向操作对象的指针，若操作对象类型为T，结果类型为“指向T的指针”
- 每个表达式都有一个相关联的类型
- 类型是有结构的！——指针
- 基本类型：语言内部支持类型
- 结构类型：组合基本类型构成新类型



6.1.1 类型表达式

- **type expression**——用以表示语言结构的类型
- **基本类型**或用类型构造符组合基本类型
 1. 基本类型: *boolean, char, integer, real, type_error, void*
 2. 类型名



类型表达式（续）

3. 类型构造符

- a. 数组：T是类型表达式，I为索引集合（整数范围），则`array(I, T)`是一个类型表达式，表示元素为类型T的数组类型

`int A[10];`——`array({0, ..., 9}, integer)`

- b. 笛卡儿积： T_1 、 T_2 为类型表达式，则 $T_1 \times T_2$ 为类型表达式



类型表达式（续）

- c. 记录：与笛卡儿积的不同之处仅在于记录的域有名字。<域名，域类型>元组

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

```
row table[101];
```

类型表达式为：

```
record((address×integer)×  
        (lexeme×array({0, ..., 15}, char)))
```




类型表达式（续）

- d) 指针：T为类型表达式，则`pointer(T)`为类型表达式，表示“指向类型为T的对象的指针”类型

`row *p;——pointer(row)`

- e) 函数：数学上，一个集合“定义域”到另一个集合“值域”的映射。程序语言，定义域类型D到值域类型R的映射： $D \rightarrow R$ 。

%运算符—— $(\text{int} \times \text{int}) \rightarrow \text{int}$

`int *f(char a, char b);——`

$(\text{char} \times \text{char}) \rightarrow \text{pointer}(\text{integer})$

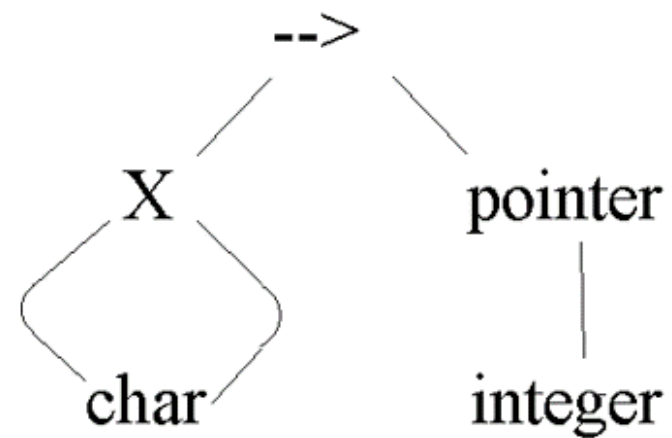
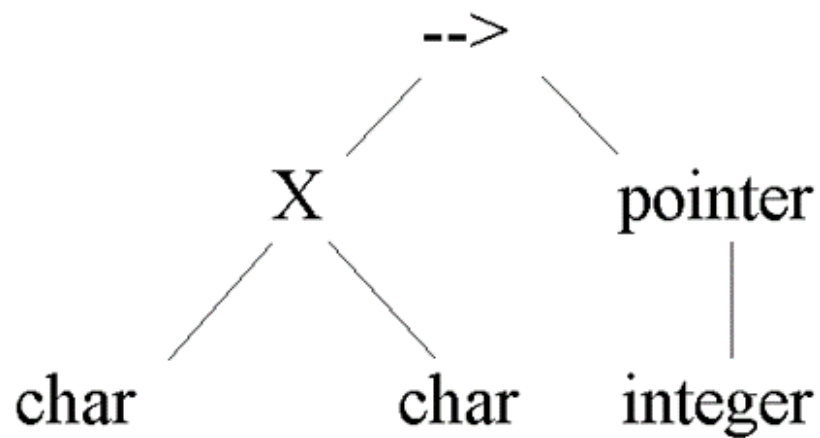
不考虑函数返回数组、函数类型的情况

$(\text{integer} \rightarrow \text{integer}) \rightarrow (\text{integer} \rightarrow \text{integer})$

4. 可使用类型表达式变量

图表示类型表达式

○ $(\text{char} \times \text{char}) \rightarrow \text{pointer}(\text{integer})$





6.1.2 类型系统

- **type system**: 规则的集合
规则——将类型表达式赋予程序的不同部分
- 类型检查程序：实现一个类型系统
- 语法制导方式实现——嵌入语义规则



6.1.2 静态/动态检查

- 静态——编译器进行
动态——运行时进行
- 可靠类型系统，强类型语言——编译器
无type_error→运行时无类型错误
- `int a[10], i; b=a[i];`——需动态检查
- 安全领域也涉及类型检查（缓冲溢出问题）



6.1.4 错误恢复

- 最低要求：报告类型错误位置
- 错误处理应融入规则中
- 错误修正比描述正确程序更困难
 - 根据错误的程序、处理缺失信息，来推测正确类型 \leftrightarrow 在变量使用之前无需定义它
 - 类型变量可用来处理这种问题



6.2 一个简单的类型检查器

6.2.1 一种简单语言

- 用综合属性type保存类型表达式

$P \rightarrow D ; E$

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

$T \rightarrow \text{char} \mid \text{integer} \mid \text{array} [\text{num}] \text{ of } T \mid ^T$

$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E [E] \mid E^E$

- 基本类型: char、integer、type_error

- 例

CODE

key:integer;

key mod 1999

Some Types

array[256] of char

^integer

Expressions

array(1..256, char)

pointer(integer)

翻译模式

$P \rightarrow D ; E$	$\{ \}$
$D \rightarrow D ; D$	$\{ \}$
$D \rightarrow \text{id} : T$	$\{ \text{addtype}(\text{id.entry}, T.type) \}$
$T \rightarrow \text{char}$	$\{ T.type = \text{char} \}$
$T \rightarrow \text{integer}$	$\{ T.type = \text{integer} \}$
$T \rightarrow \text{array} [\text{num}] \text{ of } T$	$\{ T.type = \text{array}(1.. \text{num.val}, T.type) \}$
$T \rightarrow ^T$	$\{ T.type = \text{pointer}(T.type) \}$

- $D \rightarrow \text{id} : T$ 的规则在符号表保存标识符类型
- $T.type$ 由后几个产生式语义规则计算
- $P \rightarrow D ; E$, D 在 E 之前, 保证在检查表达式类型之前类型已经保存入符号表

6.2.2 表达式类型检查

$E \rightarrow \text{literal}$	$\{E.type = char\}$	
$E \rightarrow \text{num}$	$\{E.type = integer\}$	
$E \rightarrow \text{id}$	$\{E.type = lookup(\text{id}.entry)\}$	
$E \rightarrow E_1 \text{ mod } E_2$	$\{E.type = \text{if } (E_1.type == integer) \\ \text{and } (E_2.type == integer) \\ \text{then } integer \text{ else } type_error\}$	
$E \rightarrow E_1 [E_2]$	$\{E.type = \text{if } (E_2.type == integer) \\ \text{and } (E_1.type == array(s,t)) \\ \text{then } t \text{ else } type_error\}$	$E_1 \rightarrow \text{数组}$ $E_2 \rightarrow int$
$E \rightarrow E_1^{\wedge}$	$\{E.type = \text{if } (E_1.type == pointer(t)) \\ \text{then } t \text{ else } type_error\}$	

○ 可添加其他类型和运算



6.2.3 语句的类型检查

○ 赋值、条件、while

○ 无错误, void; 错误, type_error

$S \rightarrow \text{id} := E$ $\{S.type = \text{if } (lookup(id.entry) == E.type) \text{ then void else type_error} \}$

$S \rightarrow \text{if } E \text{ then } S_1$ $\{S.type = \text{if } (E.type == \text{boolean}) \text{ then } S_1.type \text{ else type_error} \}$

$S \rightarrow \text{while } E \text{ do } S_1$ $\{S.type = \text{if } (E.type == \text{boolean}) \text{ then } S_1.type \text{ else type_error} \}$

$S \rightarrow S_1 ; S_2$ $\{S.type = \text{if } (S_1.type == \text{void}) \text{ and } (S_2.type == \text{void}) \text{ then void else type_error} \}$



6.2.4 函数的类型检查

○ 函数定义

$T \rightarrow T_1 \xrightarrow{\text{blue}} T_2 \quad \{T.type = T_1.type \rightarrow T_2.type\}$

○ 函数调用

$E \rightarrow E_1(E_2)$
 $\{E.type = \text{if } (E_2.type == s) \\ \text{and } (E_1.type == s \rightarrow t) \\ \text{then } t \text{ else } type_error\}$

○ 多参数: $T_1, \dots, T_n \longrightarrow T_1 \times \dots \times T_n$

○ 更复杂例子: $\text{root}: (\text{real} \rightarrow \text{real}) \times \text{real} \rightarrow \text{real}$
 $\text{function root}(\text{function } f(\text{real}): \text{real}; x: \text{real}): \text{real}$

6.3 类型表达式的等价

- 两个类型表达式等价的精确定义？
- 用类型表示方式可快速确定等价性
- 结构等价和名字等价
- 类型表达式的基本类型
- 递归定义





6.3.1 类型表达式的结构等价

- 结构等价 (**structural equivalence**)
 1. 相同的基本类型
 2. 对子表达式施加的类型构造符相同
 - 两个类型表达式结构等价——dag中对应相同结点
- 有时要放松条件——数组参数忽略界限
 - 等价性检查算法稍做修改



等价性检查算法

```
bool sequiv(s, t)
{
    if (s和t为相同基本类型)
        return true;
    else if (s == array(s1, s2) and t == array(t1, t2))
        return sequiv(s1, t1) and sequiv(s2, t2);
    else if (s == s1 × s2 and t = t1 × t2)
        return sequiv(s1, t1) and sequiv(s2, t2);
    else if (s == pointer(s1) and t == pointer(t1))
        return sequiv(s1, t1);
    else if (s == s1 → s2 and t == t1 → t2)
        return sequiv(s1, t1) and sequiv(s2, t2);
    else return false;
}
```



例6.1： 编码类型表达式

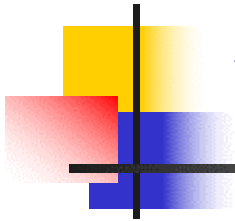
- 用二进制码表示类型和构造符
- 节省内存，加速检测——二进制码不同的类型表达式肯定不等价
- D. M. Ritchie，C编译器
- 忽略数组界限和函数参数

`char`

`freturns(char)`

`pointer(freturns(char))`

`array(pointer(freturns(char)))`



例6.1（续）

- 构造符的编码

pointer	01
---------	----

array	10
-------	----

freturns	11
----------	----

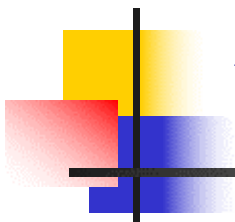
- 基本类型编码

boolean	0000
---------	------

char	0001
------	------

integer	0010
---------	------

real	0011
------	------

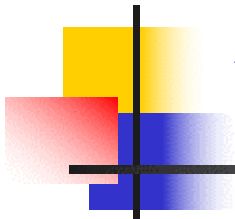


例6.1（续）

○ 编码方法

- 最右端四位二进制位表示基本类型
- 它前面两位表示第一个构造符
- 再前面两位表示第二个构造符

类型表达式	编码
char	000000 0001
freturns(char)	000011 0001
pointer(freturns(char))	000111 0001
array(pointer(freturns(char)))	100111 0001



例6.1（续）

- 加速等价性检查

- 不同二进制串不可能表示相同类型
- 不同类型可能表示为相同二进制串——数组

界限、函数参数

- 记录的编码

- 在类型表达式中记录作为基本类型
- 用另一个二进制串编码它的域



6.3.2 名字等价

```
type link = ^cell;
```

```
var next : link;
```

```
    last : link;
```

```
    p : ^cell;
```

```
    q, r : ^cell;
```

- 5个变量类型是否都相同？——依赖实现
- 允许类型表达式命名，但不允许回路
- 名字等价：完全相同
- 结构等价：名字被替换后完全相同



例6.2

变量	类型表达式
next	link
last	link
p	pointer(cell)
q	pointer(cell)
r	pointer(cell)

- 名字等价：next, last类型相同，p, q, r类型相同，p, next类型不同
- 结构等价：5个变量类型都相同



例6.3

- 许多Pascal编译器会隐含地为每条标识符定义语句涉及到的类型关联一个类型名

```
type link = ^cell;  
      np = ^cell;  
      nqr = ^cell;
```

```
var next : link;  
      last : link;  
      p : np;  
      q, r : nqr;
```

- 名字等价：next, last类型相同，q, r类型相同，

p, next, q类型不同

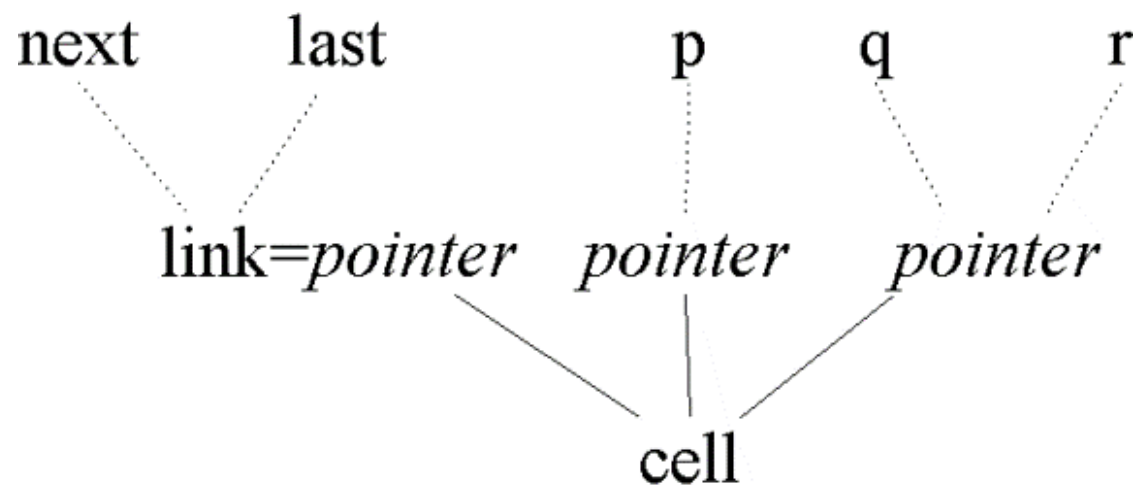
例6.3（续）

- 实现方式——构造类型图

- 对基本类型和类型构造符——创建新结点

- 对新类型名——创建叶结点，保存与类型表达式的
链接

- 名字等价——相同结点





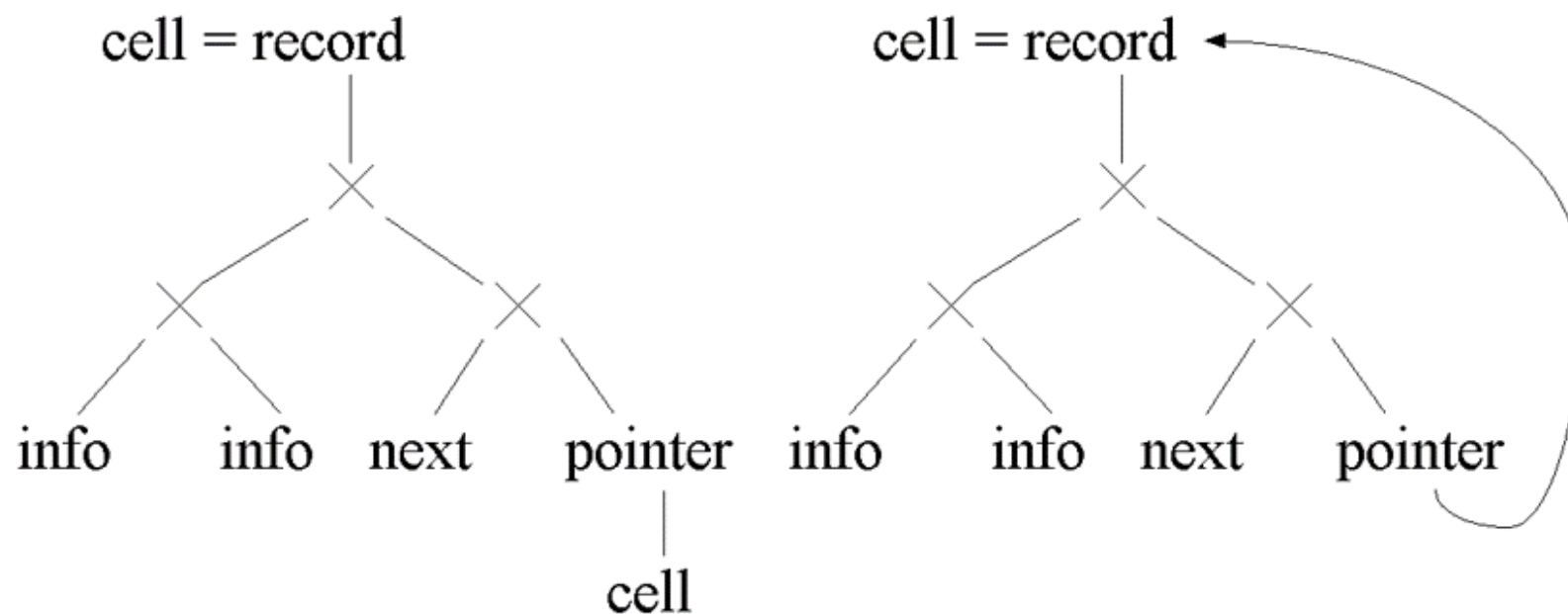
6.3.3 回路问题

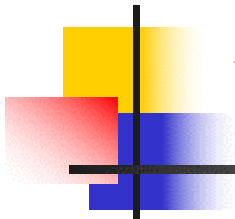
- 链表、树：递归定义
- 实现：记录——数据、指向同一记录类型的指针→回路

```
type link = ^cell;  
    cell = record  
        info : integer;  
        next : link;  
    end;
```

回路问题（续）

- 将递归定义的类型名替换为类型表达式，类型图中会产生回路





例6.4 C语言避免回路

- 对除记录之外的类型使用结构等价

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

- C语言要求类型名在使用前声明
- 例外：未定义记录类型的指针
- 回路：只可能是记录指针引起
- 结构等价判定遇到记录类型停止：只有名字相同的记录才认为类型相同



6.4 类型转换

- $x+i$, x 为实型, i 为整型
 - 不同保存格式, 不同加法指令
 - 转换为相同类型进行运算
- 语言定义指定转换方法
 - 赋值: 转换为左部类型
 - 表达式: $\text{int} \rightarrow \text{real}$
 - 类型检查器完成转换操作的生成
 - $x \text{ i } \text{intto real } \text{real}+$
- 类型转换与重载紧密相连



强制类型转换（**coercion**）

- 编译器自动进行的隐含的类型转换
- 一般要求：不能丢失信息
- 显式（**explicit**）类型转换：程序员
 - C: (float)10
 - Pascal: ord('a')——字符型→整型
chr(65)——整型→字符型
- 常量类型转换编译时进行，提高性能
 - for I := 1 to N do X[I] := 1——48.4N ms
 - for I := 1 to N do X[I] := 1.0——5.4N ms



例6.5

$E \rightarrow \text{num} \quad \{E.type = integer \}$
 $E \rightarrow \text{num.num} \quad \{E.type = real \}$
 $E \rightarrow \text{id} \quad \{E.type = lookup(\text{id}.entry)\}$
 $E \rightarrow E_1 \text{ op } E_2 \quad \{E.type = \text{if } (E_1.type == integer) \\ \text{and } (E_2.type == integer) \text{ then } integer \\ \text{else if } (E_1.type == integer) \\ \text{and } (E_2.type == real) \text{ then } real \\ \text{else if } (E_1.type == real) \\ \text{and } (E_2.type == integer) \text{ then } real \\ \text{else if } (E_1.type == real) \\ \text{and } (E_2.type == real) \text{ then } real \\ \text{else } type_error \\ \}$

lcc的类型

```
typedef struct type *Type;
```

```
struct type {
```

```
    int op;
```

```
    Type type;
```

```
    int align;
```

```
    int size;
```

```
    union {
```

```
        Symbol sym;
```

```
        struct {
```

```
            unsigned oldstyle:1;
```

```
            Type *proto;
```

```
        } f;
```

```
    } u;
```

```
    Xtype x;
```

```
};
```

类型构造符（基本类型）

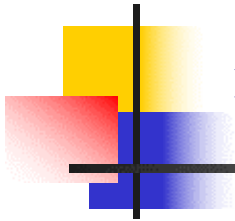
子类型表达式（用链表保存
类型表达式）

类型构造

```
static Type type(int op, Type ty, int size, int align, void *sym) {  
    unsigned h = (op^((unsigned long)ty>>3))  
    &(NELEMS(tyetable)-1);  
    struct entry *tn;  
  
    if (op != FUNCTION && (op != ARRAY || size > 0))  
        for (tn = tyetable[h]; tn; tn = tn->link)  
            if (tn->type.op == op && tn->type.type == ty  
                && tn->type.size == size && tn->type.align == align  
                && tn->type.u.sym == sym)  
                return &tn->type;
```

函数类型和不完全数组类型无
重复概念，总是创建新类型

重复类型检查



类型构造

```
NEW0(tn, PERM);  
tn->type.op = op;  
tn->type.type = ty;  
tn->type.size = size;  
tn->type.align = align;  
tn->type.u.sym = sym;  
tn->link = typetable[h];  
typetable[h] = tn;  
return &tn->type;  
}
```



指针

```
Type ptr(Type ty) {  
    return type(POINTER, ty, IR->ptrmetric.size,  
                IR->ptrmetric.align, pointersym);  
}
```

```
Type deref(Type ty) {  
    if (isptr(ty))  
        ty = ty->type;  
    else  
        error("type error: %s\n", "pointer expected");  
    return isenum(ty) ? unqual(ty)->type : ty;  
}
```

脱掉指针



数组

```
Type array(Type ty, int n, int a) {
```

```
    assert(ty);
```

```
    if (isfunc(ty)) {
```

不允许函数数组

```
        error("illegal type `array of %t'\n", ty);
```

```
        return array(inttype, n, 0);
```

数组的数组，低维必须大小

```
    }
```

```
    if (isarray(ty) && ty->size == 0)
```

已知

```
        error("missing array size\n");
```

```
    if (ty->size == 0) {
```

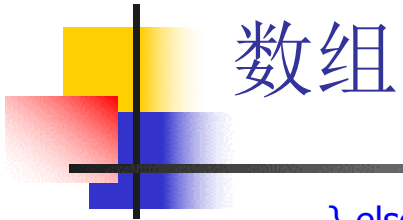
```
        if (unqual(ty) == voidtype)
```

```
            error("illegal type `array of %t'\n", ty);
```

```
        else if (Aflag >= 2)
```

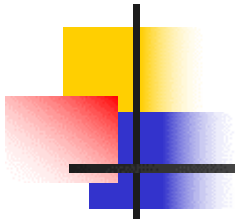
```
            warning("declaring type array of %t' is
```

```
undefined\n", ty);
```

数组

```
    } else if (n > INT_MAX/ty->size) {  
        error("size of `array of %t' exceeds %d bytes\n",  
            ty, INT_MAX);  
        n = 1;  
    }  
    return type(ARRAY, ty, n*ty->size,  
        a ? a : ty->align, NULL);  
}
```

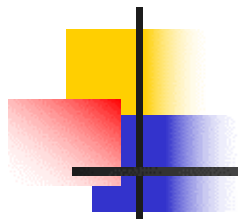


结构和联合

```
Type newstruct(int op, char *tag) {  
    Symbol p;
```

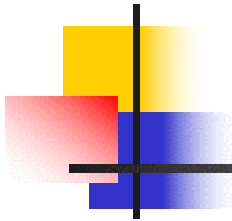
创建一个结构类型，但只是一个空架子

```
    assert(tag);  
    if (*tag == 0)  
        tag = stringd(genlabel(1));  
    else  
        if ((p = lookup(tag, types)) != NULL && (p->scope == level  
            || p->scope == PARAM && level == PARAM+1)) {  
            if (p->type->op == op && !p->defined)  
                return p->type;  
            error("redefinition of `%' previously defined at %w\n",  
                p->name, &p->src);  
        }  
}
```



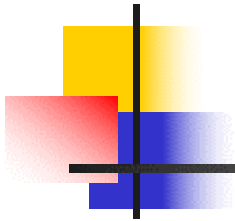
结构和联合

```
p = install(tag, &types, level, PERM);  
p->type = type(op, NULL, 0, 0, p);  
if (p->scope > maxlevel)  
    maxlevel = p->scope;  
p->src = src;  
return p->type;  
}
```



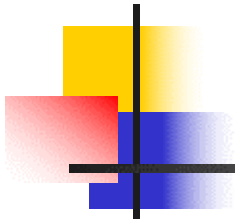
结构和联合

```
Field newfield(char *name, Type ty, Type fty) {  
    Field p, *q = &ty->u.sym->u.s.flst;  
  
    if (name == NULL)  
        name = stringd(genlabel(1));  
    for (p = *q; p; q = &p->link, p = *q)  
        if (p->name == name)  
            error("duplicate field name '%s' in '%t'\n",  
                name, ty);  
    NEW0(p, PERM);  
    *q = p;  
    p->name = name;  
    p->type = fty;  
}
```



结构和联合

```
if (xref) {
    /* omit */
    if (ty->u.sym->u.s.ftab == NULL) /* omit */
        ty->u.sym->u.s.ftab = table(NULL, level); /* omit */
    install(name, &ty->u.sym->u.s.ftab, 0, PERM)->src =
src; /* omit */
}
/* omit */
return p;
}
```



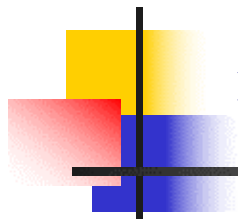
结构和联合

```
static Type structdcl(int op) {  
    ...  
    ty = newstruct(op, tag);  
    ...  
    fields(ty);  
    ...  
}  
static void fields(Type ty) {  
    ...  
    p = newfield(id, ty, fty);  
    ...  
}
```



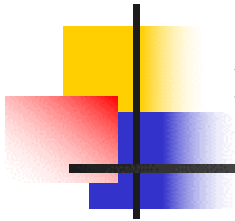
类型等价

```
int eqtype(Type ty1, Type ty2, int ret) {  
    if (ty1 == ty2)  
        return 1;  
    if (ty1->op != ty2->op)  
        return 0;  
    switch (ty1->op) {  
        case ENUM: case UNION: case STRUCT:  
        case UNSIGNED: case INT: case FLOAT:  
            return 0;  
        case POINTER: return eqtype(ty1->type, ty2->type, 1);  
        case VOLATILE: case CONST+VOLATILE:  
        case CONST: return eqtype(ty1->type, ty2->type, 1);  
    }
```



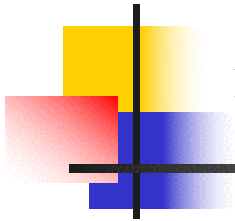
类型等价

```
case ARRAY:  if (eqtype(ty1->type, ty2->type, 1)) {  
              if (ty1->size == ty2->size)  
                return 1;  
              if (ty1->size == 0 || ty2->size == 0)  
                return ret;  
            }  
            return 0;
```

类型等价

```
case FUNCTION: if (eqtype(ty1->type, ty2->type, 1)) {  
    Type *p1 = ty1->u.f.proto, *p2 = ty2->u.f.proto;  
    if (p1 == p2)  
        return 1;  
    if (p1 && p2) {  
        for ( ; *p1 && *p2; p1++, p2++)  
            if (eqtype(unqual(*p1),  
unqual(*p2), 1) == 0)  
                return 0;  
        if (*p1 == NULL && *p2 == NULL)  
            return 1;  
    }
```



类型等价

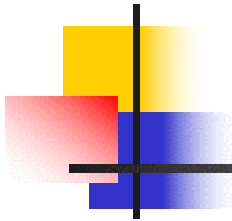
```
    } else {  
        if (variadic(p1 ? ty1 : ty2))  
            return 0;  
        if (p1 == NULL)  
            p1 = p2;  
        for ( ; *p1; p1++) {  
            Type ty = unqual(*p1);  
            if (promote(ty) != (isenum(ty) ?  
ty->type : ty))  
                return 0;  
        }  
        return 1;  
    }  
}  
}  
return 0;  
}  
assert(0); return 0;  
}
```



TinyC的类型检查

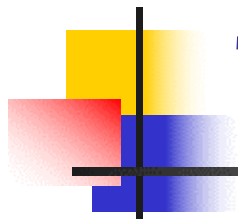
```
void typeCheck(TreeNode * syntaxTree)
{
    traverse(syntaxTree,nullProc,checkNode);
}

static void traverse( TreeNode * t, void (* preProc) (TreeNode *),
                    void (* postProc) (TreeNode *) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc,postProc);
    }
    postProc(t);
    traverse(t->sibling,preProc,postProc);
  }
}
```



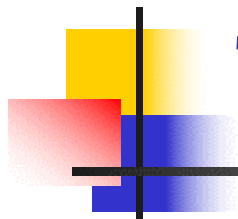
TinyC的类型检查

```
static void checkNode(TreeNode * t)
{ switch (t->nodekind)
  { case ExpK:
    switch (t->kind.exp)
    { case OpK:
      if ((t->child[0]->type != Integer) ||
          (t->child[1]->type != Integer))
        typeError(t,"Op applied to non-integer");
      if ((t->attr.op == EQ) || (t->attr.op == LT))
        t->type = Boolean;
      else
        t->type = Integer;
      break;
    case ConstK:
    case IdK:
      t->type = Integer;
      break;
```



TinyC的类型检查

```
    default:
        break;
}
break;
case StmtK:
    switch (t->kind.stmt)
    { case IfK:
        if (t->child[0]->type == Integer)
            typeError(t->child[0], "if test is not Boolean");
        break;
    case AssignK:
        if (t->child[0]->type != Integer)
            typeError(t->child[0], "assignment of non-integer value");
        break;
```



TinyC的类型检查

```
case WriteK:
    if (t->child[0]->type != Integer)
        typeError(t->child[0], "write of non-integer value");
    break;
case RepeatK:
    if (t->child[1]->type == Integer)
        typeError(t->child[1], "repeat test is not Boolean");
    break;
default:
    break;
}
break;
default:
    break;

}

}
```



6.5 函数和操作符重载

- 重载（**overloaded**）符号：根据上下文，具有不同的意义
 - +：整型加法，浮点型加法
 - $A(I)$ ：数组A的第I个元素，以参数I调用A，将I转换为类型A的显式类型转换
- 重载的解析（**resolved**）：在某个特定上下文，确定符号的唯一意义
 - $1+2$ ：+为整型加法
 - 运算符识别，operator identification



6.5.1 子表达式可能类型集合

○ 例6.6

○ Ada允许对乘法运算符“*”进行重载

□ `function “*” (i, j : integer) return complex;`

□ `function “*” (x, y : complex) return complex;`

○ *具有三种可能类型

□ *integer* × *integer* → *integer*

□ *integer* × *integer* → *complex*

□ *complex* × *complex* → *complex*

□ $2*(3*5) \rightarrow 3*5$ ——类型1

□ $z*(3*5)$, z 为复数类型 $\rightarrow 3*5$ ——类型2



可能类型集合情况的处理

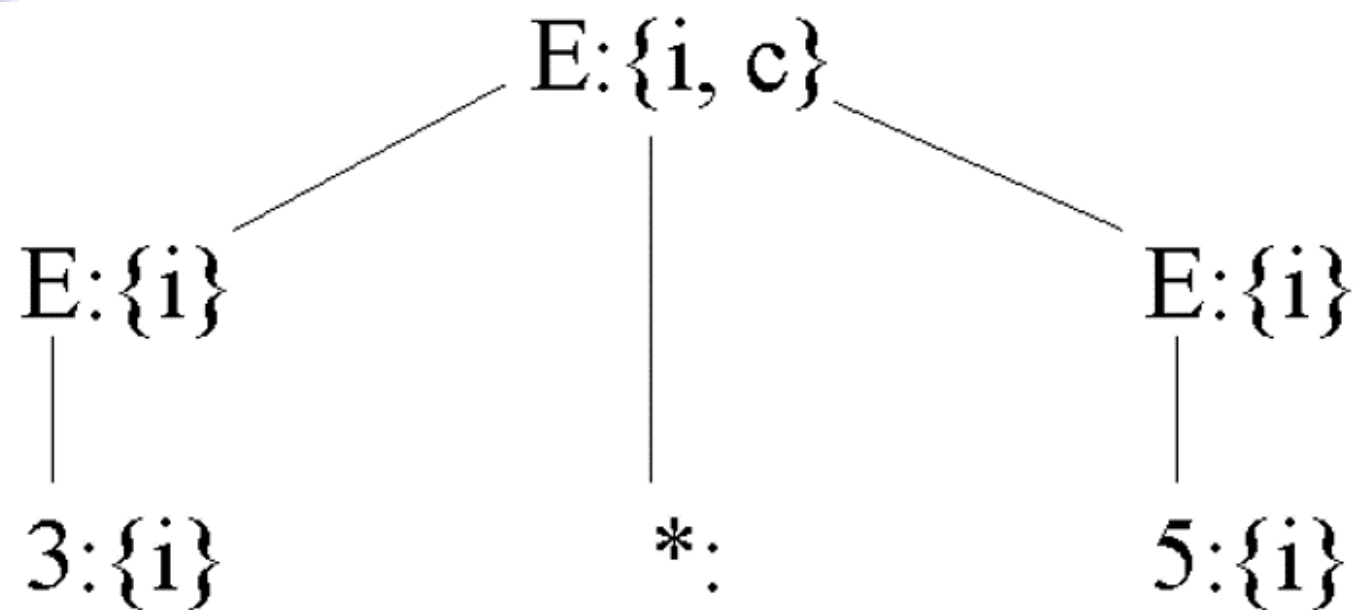
$E \rightarrow \mathbf{id}$ $E.types = lookup(\mathbf{id}.entry)$

$E \rightarrow E_1(E_2)$ $E.types = \{ t \mid E_2.types \text{ 中} \\ \text{存在 } s \text{ 使得 } s \rightarrow t \text{ 属于} \\ E_1.types \}$

○ 单一类型运算 \rightarrow 类型集合运算



例6.7


$$\{i \times i \rightarrow i, i \times i \rightarrow c, c \times c \rightarrow c\}$$



6.5.2 确定唯一类型

- 完整表达式须有唯一类型 \rightarrow 确定每个子表达式的唯一类型，或 `type_error`
- 自顶向下
- `E.types` 中每个类型 `t` 均为可行（**feasible**）类型——确定 `E` 中重载标识符的类型，某种情况下，`E` 的类型为 `t`
 - 对标识符成立，`id.types` 中类型均可行
 - 归纳法：`E` 为 `E1(E2)`，`s` 为 `E2` 可行类型，`s \rightarrow t` 为 `E1` 可行类型，因此 `t` 为 `E` 可行类型

确定唯一类型的语义规则

$E' \rightarrow E$

$E'.types = E.types$

$E.unique = \text{if } E'.types == \{t\} \text{ then } t \text{ else type_error}$

$E'.code = E.code$

$E \rightarrow \text{id}$

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

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

$E \rightarrow E_1(E_2)$

$E.types = \{ s' \mid E_2.types \text{ 中存在 } s \text{ 使得 } s \rightarrow s' \}$

属于 $E_1.types$ }

$t = E.unique$

$S = \{ s \mid s \in E_2.types \text{ and } s \rightarrow t \in E_1.types \}$

$E_2.unique = \text{if } S == \{s\} \text{ then } s \text{ else type_error}$

$E_1.unique = \text{if } S == \{s\} \text{ then } s \rightarrow t \text{ else type_error}$

$E.code = E_1.code \parallel E_2.code \parallel$

$\text{gen}(\text{'apply' ':' } E.unique)$



6.6 多态（polymorphic）函数

- 普通函数：参数类型固定
- 多态函数：对不同调用，参数可为不同类型
- 某些内置操作符——多态操作符
 - 数组索引符[]，函数调用符()，指针操作符&
 - &: “若操作对象类型为…，则操作结果的类型为指向…的指针”



6.6.1 为什么使用多态函数？

- 实现算法 → 处理数据结构，而不必管其内部元素的类型

```
type link = ^cell;  
    cell = record  
        info : integer;  
        next : link  
    end;  
function length(lptr : link) : integer;  
var len : integer;  
begin  
    len := 0;  
    while lptr <> nil do begin  
        len := len + 1;  
        lptr := lptr^.next;  
    end;  
    length := len  
end;
```

求任何类型列表长度的ML程序

fun length(lptr) =

递归函数

if null(lptr) then 0

列表为空?

else length(tl(lptr)) + 1;

列表剩余部分

○ 可应用于任何类型的列表

□ length([“sun” , “mon” , “tue”]);

□ length([10, 9, 8]);



6.6.2 类型变量

- α, β, \dots , 表示未知类型
- 重要应用：不要求标识符先声明后使用的语言中，检查标识符使用的一致性
- 类型变量表示未声明标识符的类型
 - 若类型变量发生变化，不一致！
 - 若一直未变化，一致！同时得到标识符类型
- 类型推断，**type inference**
 - 根据语言结构的使用方式判定其类型



例6.8

```
type link = ^cell;  
procedure mlist(lptr : link; procedure p);  
begin  
    while lptr <> nil do begin  
        p(lptr);  
        lptr := lptr^.next  
    end  
end;
```

- p: 参数情况未知, 不完整的过程类型
- 由p(lptr)→
 - p参数为link类型, p的类型为link→void
 - 其他使用方式均会导致type_error



例6.9

```
function deref(p);  
begin  
    return p^;  
end;
```

- 扫描第一行， p 的类型未知，用 β 表示
- 第三行， $^$ 应作用于指针，因此 p 为某未知基本类型 α 的指针类型， $\beta = \text{pointer}(\alpha)$
- 因此函数deref类型为： $\text{pointer}(\alpha) \rightarrow \alpha$



6.6.3 包含多态函数的语言

- 用 \forall 表示 “对所有类型”
- `deref`类型的精确描述: $\forall \alpha. \text{pointer}(\alpha) \rightarrow \alpha$
- `length`: $\forall \alpha. \text{list}(\alpha) \rightarrow \text{integer}$
 - $\text{list}(\text{integer}) \rightarrow \text{integer}$
 - $\text{list}(\text{list}(\text{char})) \rightarrow \text{integer}$
- \forall : 全称量词 (universal quantifier) \rightarrow
它所施用的类型变量称为由它约束 (bound)



文法定义

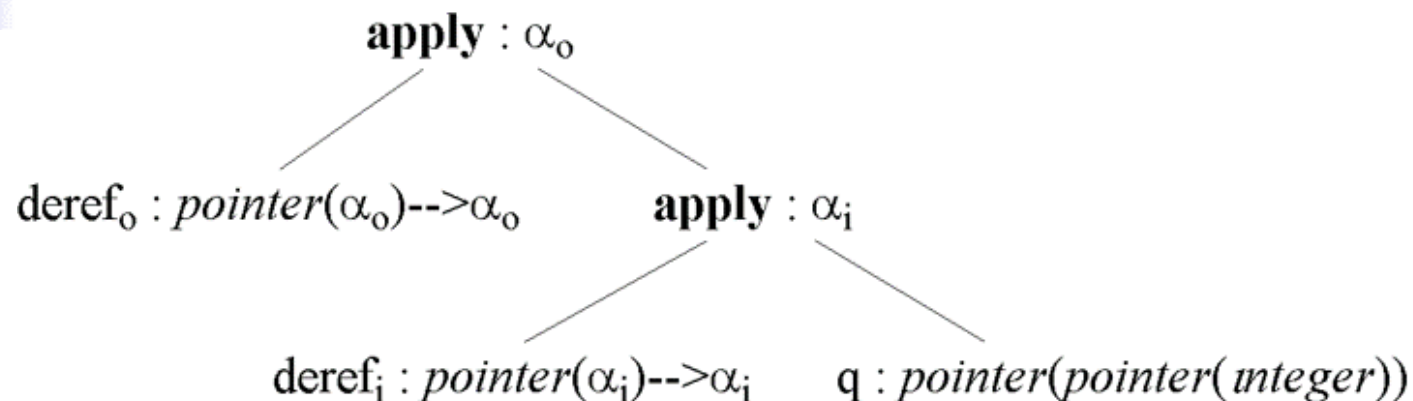
$P \rightarrow D ; E$
 $D \rightarrow D ; D \mid \mathbf{id} : Q$
 $Q \rightarrow \forall \mathbf{type_variable} . Q \mid T$
 $T \rightarrow T ' \rightarrow ' T$
 $\mid T \times T$
 $\mid \mathbf{unary_constructor} (T)$
 $\mid \mathbf{basic_type}$
 $\mid \mathbf{type_variable}$
 $\mid (T)$

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

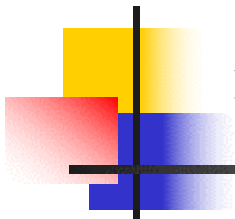
程序例:

$\mathbf{deref} : \forall \alpha . \mathbf{pointer}(\alpha) \rightarrow \alpha;$
 $\mathbf{q} : \mathbf{pointer}(\mathbf{pointer}(\mathbf{integer}));$
 $\mathbf{deref}(\mathbf{deref}(\mathbf{q}))$

多态函数类型检查



- 每个结点两个标签：子表达式和类型表达式
- 下标o：外层deref，i：内层deref



类型检查规则的特点

1. 不同位置出现的同一多态函数可具有不同类型的参数。

- deref_0 参数为二重指针，而 deref_i 的参数为一重指针。
- 关键在于 α 的解释，对不同 deref ，约束的类型变量 α 所代表的意义是不同的。
- 对多态函数的每次出现，都赋予不同的类型变量，而不再使用 —— 上例中用 α_0 、 α_i 分别对应外层和内层 deref



类型检查规则的特点（二）

2. 类型表达式中存在变量，因此要重新考虑类型等价的概念

- 类型为 $s \rightarrow s'$ 的函数 E_1 施用于类型为 t 的 E_2 ，仅判定 s 和 t 是否等价是不够的，要对它们进行“一致化”（unify）
- 适当替换类型变量，检查是否有可能达到结构等价
- 上例：将 α_i 替换为 $pointer(integer)$ ，则 $pointer(\alpha_i)$ 与 $pointer(pointer(integer))$ 等价



类型检查规则的特点（三）

3. 需要某种机制记录一致化的影响

- 一个类型变量可出现在表达式的多个位置
- 若 s 和 s' 的一致化使得变量 α 表示类型 t ，则
在整个类型表达式中，它都应表示 t
- 上例： α_i 作用域为 $\text{deref}_i(q)$ ，因此可用来一
致化 deref_i 和 q
- 而 α_o 为不同类型变量，因此表示不同类型
不违反本规则



6.6.4 代换，实例和合一

- 变量表示实际类型的形式化定义
 - 类型变量 \rightarrow 类型表达式的映射，称为替换，
substitution

```
subst(t : type_expression) : type_expression
{
    //利用代换（映射）S将类型表达式t中变量代换
    if (t为基本类型) return t;
    else if (t为类型变量) return S(t);
    else if (t为 $t_1 \rightarrow t_2$ ) return subst( $t_1$ )  $\rightarrow$  subst( $t_2$ );
}
```

- S(t)表示代换t的类型表达式，称为t的实例，
instance
- 若无法代换， $S(\alpha) = \alpha$ ，恒等映射



例6.10

○ $s < t$ 表示 s 是 t 的一个实例

□ $pointer(integer) < pointer(\alpha)$

□ $pointer(real) < pointer(\alpha)$

□ $integer \rightarrow integer < \alpha \rightarrow \alpha$

□ $pointer(\alpha) < \beta$

□ $\alpha < \beta$

○ 不是实例的情况

□ $integer$ $real$

□ $integer \rightarrow real$ $\alpha \rightarrow \alpha$

□ $integer \rightarrow \alpha$ $\alpha \rightarrow \alpha$



合一方法

- 类型表达式 t_1 和 t_2 能合一的条件→
存在代换 S , $S(t_1) = S(t_2)$
- 最一般的合一代换, **most general unifier**——
最少变量约束的代换
 - 类型表达式 t_1 和 t_2 的最一般的合一代换是一个代换 S , 它满足以下条件
 1. $S(t_1) = S(t_2)$
 2. 对任何其他代换 S' , $S'(t_1) = S'(t_2)$, S' 是 S 的一个实例, 即, 对任意 t , $S'(t)$ 是 $S(t)$ 的一个实例



6.6.5 多态函数类型检查

- 检查规则由下列对类型图的操作构成
 - `fresh(t)`——将类型表达式`t`中约束变量代换为新变量，返回代换后类型表达式结点指针→消除 \forall
 - `unify(m, n)`——将结点`m`、`n`表示的类型表达式合一，会修改、保存对应的代换。若失败，则整个类型检查过程失败
- 图的创建仍可用`mkleaf`、`mknode`完成
- 每个类型变量创建不同结点



unify操作

- 合一、代换——基于图论的方法
- m 、 n 为类型图结点，分别表示表达式 e 、 f ，若 $S(e)=S(f)$ ，称 m 、 n 在代换 S 下等价
- 求最一般的合一代换
 - 转化为 \rightarrow 在 S 下必须等价的结点的集合划分问题
 - 类型表达式等价 \rightarrow 根必须等价
 - m 、 n 等价 $\leftarrow\rightarrow$ 表示相同操作且孩子结点等价

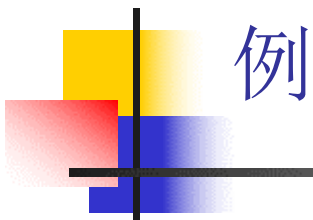
类型检查规则

$E \rightarrow E_1(E_2) \quad \{ p = \text{mkleaf}(\text{newtypevar});$
 $\text{unify}(E_1.\text{type}, \text{mknode}(' \rightarrow ', E_2.\text{type}, p));$
 $E.\text{type} = p; \}$

$E \rightarrow E_1, E_2$
 $\{ E.\text{type} = \text{mknode}(' \times ', E_1.\text{type}, E_2.\text{type}); \}$

$E \rightarrow \text{id} \quad \{ E.\text{type} = \text{fresh}(\text{id.type}); \}$

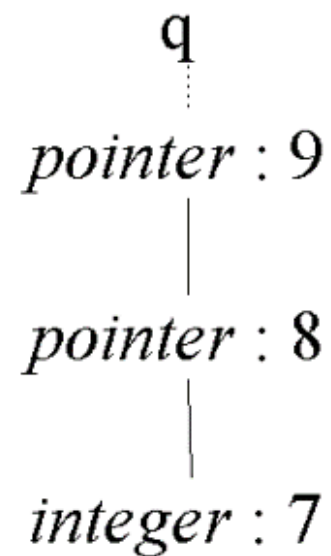
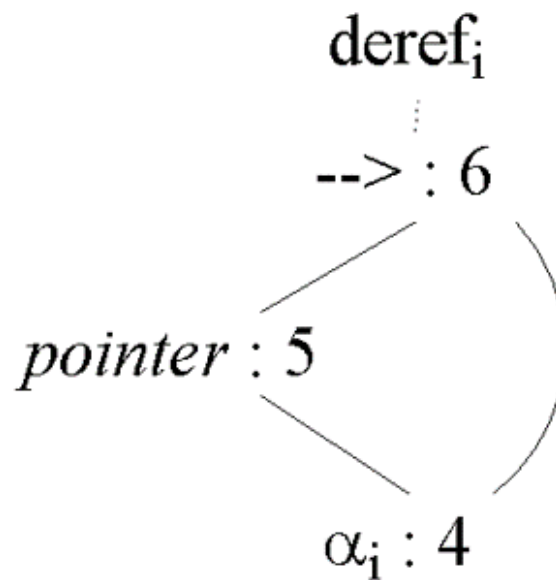
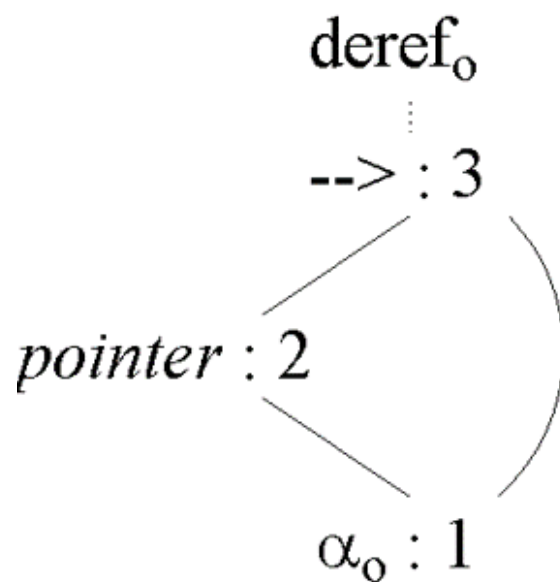
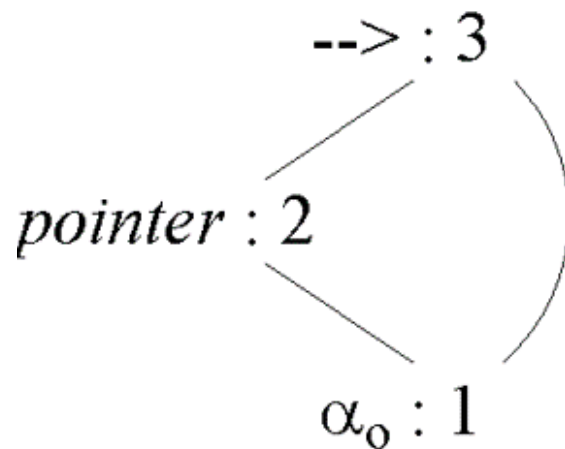
- $E_1.\text{type} = \alpha, E_2.\text{type} = \beta$, 都是类型变量
- 前者是函数, 寻求两者等价, 必有类型变量 γ , 使得 $\alpha = \beta \rightarrow \gamma$



例

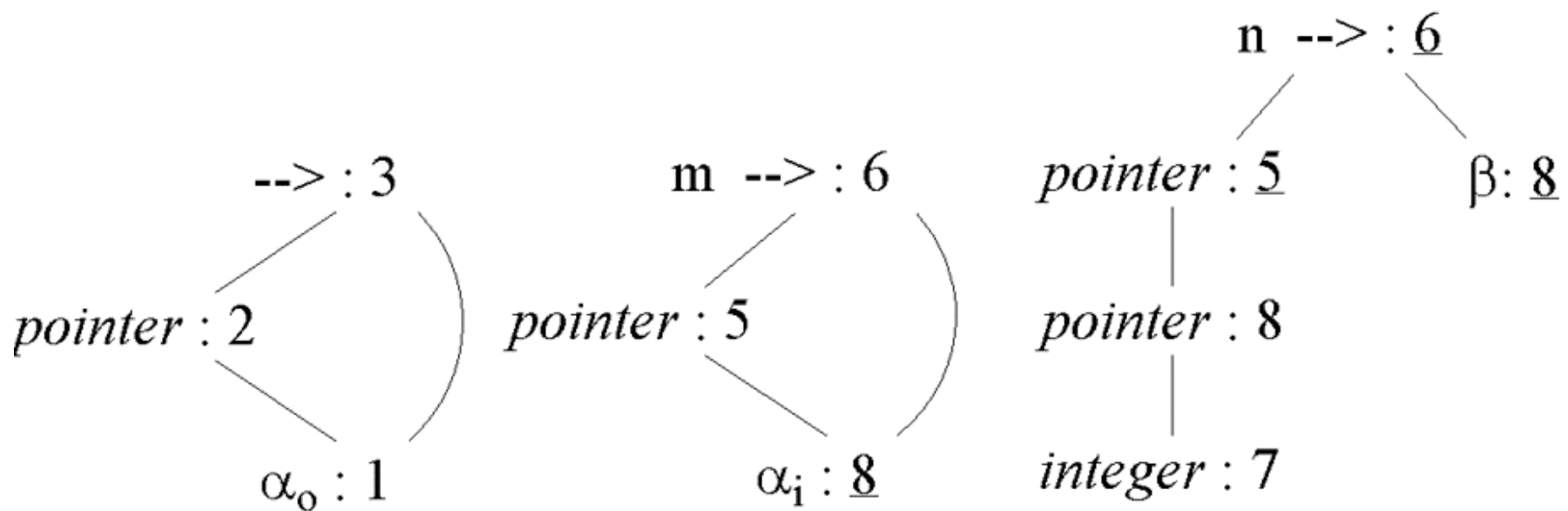
表达式	类型	代换
q	$\text{pointer}(\text{pointer}(\text{integer}))$	
deref_i	$\text{pointer}(\alpha_i) \rightarrow \alpha_i$	
$\text{deref}_i(q)$	$\text{pointer}(\text{integer})$	$\alpha_i = \text{pointer}(\text{integer})$
deref_o	$\text{pointer}(\alpha_o) \rightarrow \alpha_o$	
$\text{deref}_o(\text{deref}_i(q))$	integer	$\alpha_o = \text{integer}$

例6.11



例6.11（续）

○ deref_i 的合一过程





例6.12 ML语言多态函数检查

fun $\text{id}_0(\text{id}_1, \dots, \text{id}_k) = E;$

- id_0 : 函数名, $\text{id}_1, \dots, \text{id}_k$: 参数, E 遵从前面定义的用于多态函数类型检查的文法, 其中的标识符只可能是函数名、参数或内置函数
- 方法: 例6.9方法的形式化
 - 为函数名和参数创建新类型变量
 - 多态函数的类型变量由 \forall 约束
 - 检查 $\text{id}_0(\text{id}_1, \dots, \text{id}_k)$ 和 E 类型是否匹配
 - 若成功, 则推断出函数类型



例6.12（续）

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

- 类型变量： $\text{length} \longrightarrow \beta$, $\text{lptr} \longrightarrow \gamma$
- 为匹配函数体： $\beta = \forall \alpha. \text{list}(\alpha) \rightarrow \text{integer}$
- 按多态函数类型检查语言重写程序

$\text{length} : \beta;$

$\text{lptr} : \gamma;$

$\text{if} : \forall \alpha. \text{boolean} \times \alpha \times \alpha \rightarrow \alpha;$



例6.12（续）

$\text{null} : \forall \alpha. \text{list}(\alpha) \rightarrow \text{boolean} ;$



$\text{tl} : \forall \alpha. \text{list}(\alpha) \rightarrow \text{list}(\alpha);$

$0 : \text{integer};$

$1 : \text{integer};$

$+: \text{integer} \times \text{integer} \rightarrow \text{integer};$

$\text{match} : \forall \alpha. \alpha \times \alpha \rightarrow \alpha ;$

$\text{match} ($  检查 $\text{length}(\text{lptr})$ 与
函数体匹配
 $\text{length}(\text{lptr}),$ 
 $\text{if} (\text{null}(\text{lptr}), 0, \text{length}(\text{tl}(\text{lptr})) + 1)$
 $)$

例6.12（续）

length参数为 $lptr-\gamma$
返回类型为 δ

null参数类型为
 $list(\alpha_n) \rightarrow lptr$
类型 $\gamma = list(\alpha_n)$
length类型为
 $list(\alpha_n) \rightarrow \delta$

表达式:

$lptr :$
 $length :$
 $length(lptr) :$
 $lptr :$
 $null :$
 $null(lptr) :$
 $0 :$
 $lptr :$
 $tl :$
 $tl(lptr) :$

类型

γ
 β
 δ
 γ
 $list(\alpha_n) \rightarrow \text{boolean}$
 boolean
 integer
 $list(\alpha_n)$
 $list(\alpha_t) \rightarrow list(\alpha_t)$
 $list(\alpha_n)$

替换

$\beta = \gamma \rightarrow \delta$
 $\gamma = list(\alpha_n)$
 $\alpha_t = \alpha_n$

例6.12 (续)

表达式:	类型	替换
$\text{length} :$	$\text{list}(\alpha_n) \rightarrow \delta$	
$\text{length}(\text{tl}(\text{lptr})) :$	δ	
$1 :$	integer	
$+$	$\text{integer} \times \text{integer} \rightarrow \text{integer}$	
$\text{length}(\text{tl}(\text{lptr})) + 1 :$	integer	$\delta = \text{integer}$
$\text{if} :$	$\text{boolean} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
$\text{if}(\cdots) :$	integer	$\alpha_i = \text{integer}$
$\text{match} :$	$\alpha_m \times \alpha_m \rightarrow \alpha_m$	
$\text{match}(\cdots) :$	integer	$\alpha_m = \text{integer}$

α_n 最终未被替代,
 length 类型为
 $\forall \alpha_n. \text{list}(\alpha_n) \rightarrow \text{integer}$

length 的返回类型, 因此 $\delta = \text{integer}$



6.7 合一算法

- 合一：类型表达式 e 、 f 通过变量代换，是否可达到类型等价
 - 特殊情况：等价性检查，无变量
 - 本节算法适用类型图有回路情况
 - 算法寻找最一般的合一代换 S

例6.13

○ e、f: $((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$

$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$

○ 两个合一替换 S, S' :

x	$S(x)$	$S'(x)$
α_1	α_3	α_1
α_2	α_2	α_1
α_3	α_3	α_1
α_4	α_2	α_1
α_5	$list(\alpha_2)$	$list(\alpha_1)$

最一般的合一替换
 $S'(e)$ 是 $S(e)$ 的实例,
 反之不成立

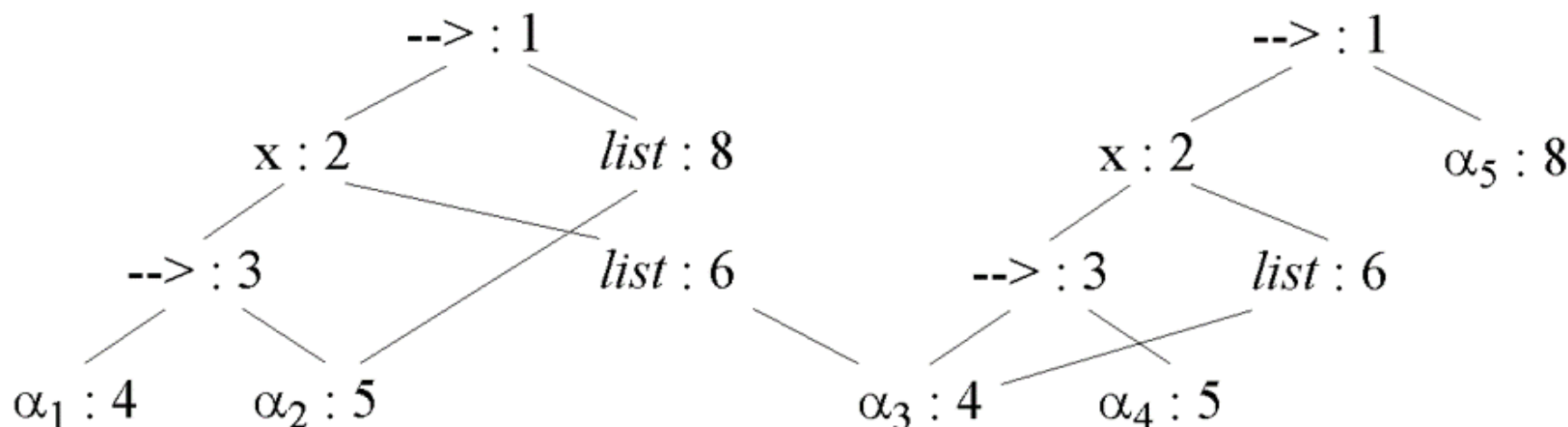
○ 代换结果:

$S(e) = S(f) = ((\alpha_3 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$

$S'(e) = S'(f) = ((\alpha_1 \rightarrow \alpha_1) \times list(\alpha_1)) \rightarrow list(\alpha_1)$

算法思想

- 用树表示表达式—— $S(e)$ 的结点数目可能与 e 、 f 的结点数目呈指数关系→图表示
- 图论方法：在最一般的合一代换下必须等价的结点，进行集合划分



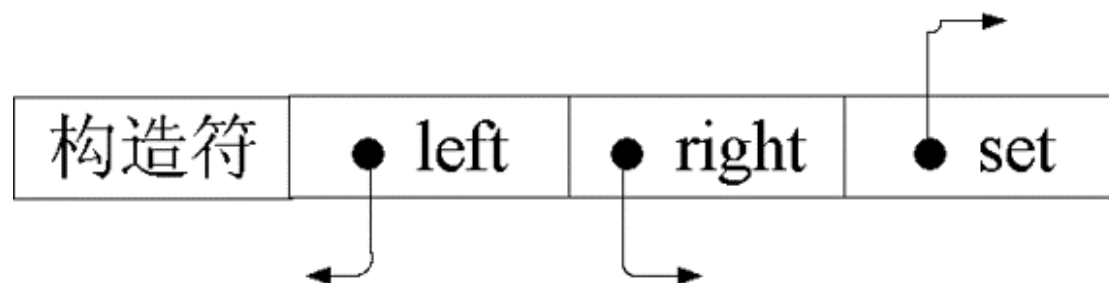
算法6.1 合一算法

输入：一个类型图和要进行合一的结点m、n

输出：若一致，返回true；否则，返回false。本

算法的函数可修改为前面给出的多态函数类型检查规则所需的unify函数

方法：



结点用上图所示记录实现

set域维护等价结点集合

每个等价类选出一个代表结点——set域为空指针

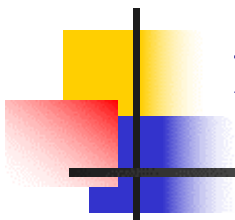
等价类内其他结点指向代表结点

初始，每个结点形成一个等价类



算法6.1（续）

```
bool unify(node m, n)
{
    s = find(m);
    t = find(n);
    if (s == t) return true;
    else if (s和t为相同基本类型结点) return true;
    else if (s为操作符结点, 孩子结点为 $s_1, s_2$  且
             t为操作符结点, 孩子结点为 $t_1, t_2$ ) {
        union(s, t);
        return unify( $s_1, t_1$ ) and unify( $s_2, t_2$ );
    } else if (s或t表示类型变量) {
        union(s, t);
        return true;
    } else return false;
}
```



find和union操作

1. find(n)

- 返回结点n等价类的代表结点

2. union(m, n)

- 将结点m、n所在等价类合并
- 若两个等价类代表结点中某个不是变量结点，则将其作为合并等价类的代表结点。
- 否则，其中任一个作为新代表结点。
- 原因：若等价类包含类型构造符或基本类型，变量结点不能做为代表结点
- 否则，（纯变量）表达式可通过变量达到合一

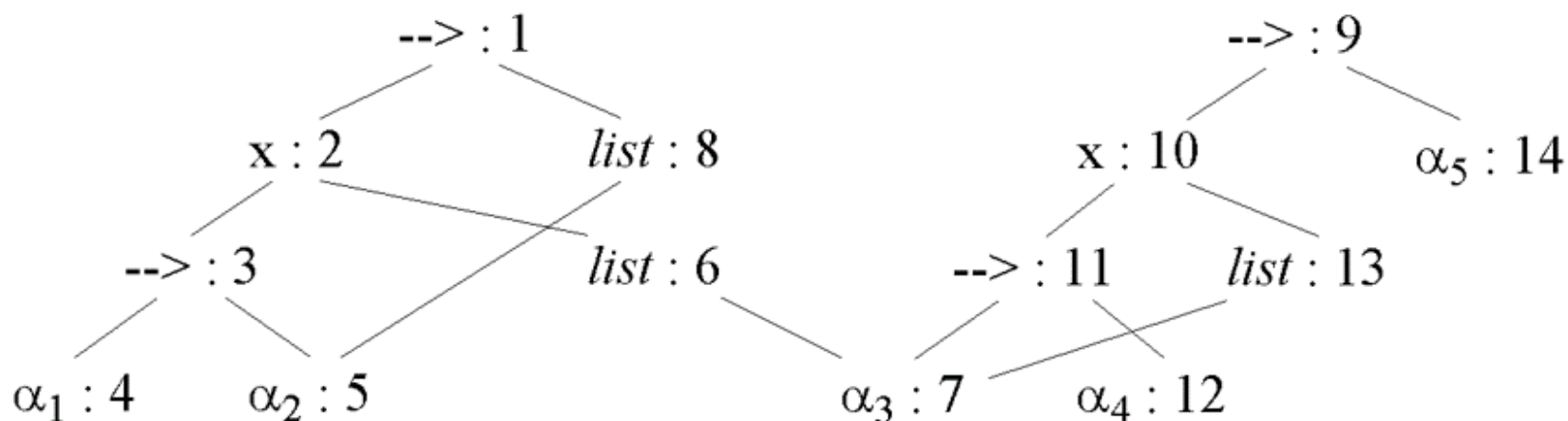


算法（续）

- union的实现：将一个等价类代表结点的set指针指向另一个的代表结点
- find：沿set链遍历，直到空指针
- 算法6.1，使用 $s = \text{find}(m)$ 和 $t = \text{find}(n)$
 - s 、 t 相等 $\rightarrow m$ 、 n 已在相同等价类中
 - s 、 t 为相同基本类型 \rightarrow 返回真
 - s 、 t 为相同类型操作符 \rightarrow 合并两个等价类，继续递归检查他们孩子结点等价性
 - s 、 t 其中一个为变量 \rightarrow 合并，类型操作符或基本类型成为代表结点 \rightarrow 变量不会与两个不同的表达式合一

例6.14

- 考虑例6.13的表达式，初始dag为

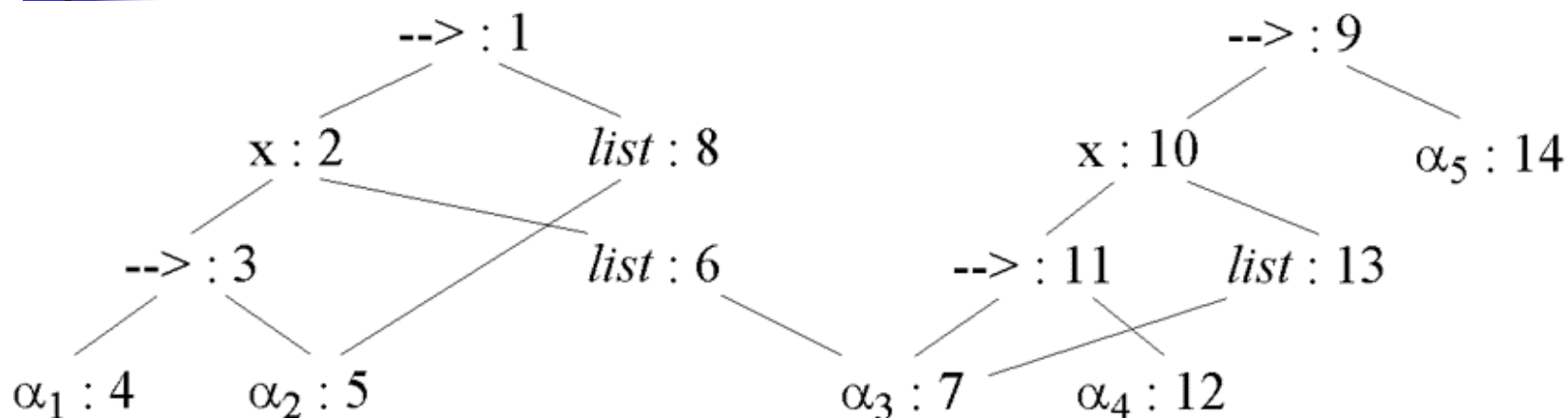


- 合一过程

$\text{unify}(1, 9)$: 相同操作符，合并， $\text{unify}(2, 10)$, $\text{unify}(8, 14)$

$\text{unify}(2, 10)$: 相同操作符，合并， $\text{unify}(3, 11)$, $\text{unify}(6, 13)$

例6.14（续）



$\text{unify}(1, 9)$: 相同操作符, 合并, $\text{unify}(2, 10)$, $\text{unify}(8, 14)$

$\text{unify}(2, 10)$: 相同操作符, 合并, $\text{unify}(3, 11)$, $\text{unify}(6, 13)$

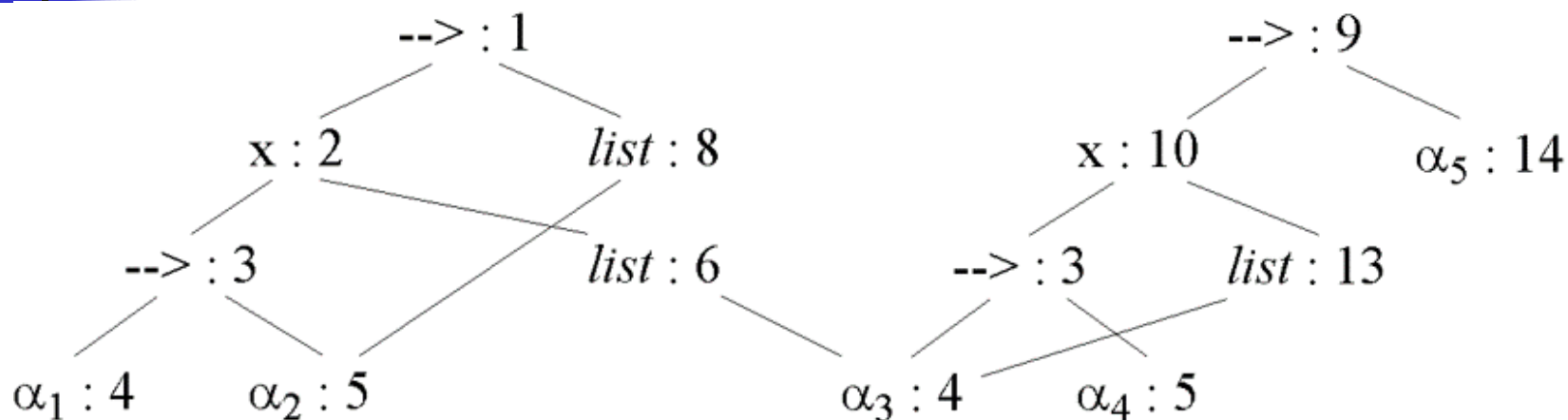
$\text{unify}(3, 11)$: 相同操作符, 合并, $\text{unify}(4, 7)$, $\text{unify}(5, 12)$

$\text{unify}(4, 7)$: 两个变量, 合并, 4作为代表结点, 返回真

$\text{unify}(5, 12)$: 两个变量, 合并, 5作为代表结点, 返回真

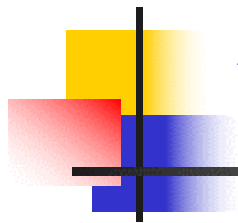
$\text{unify}(3, 11)$ 为真, 3作为代表结点

例6.14（续）



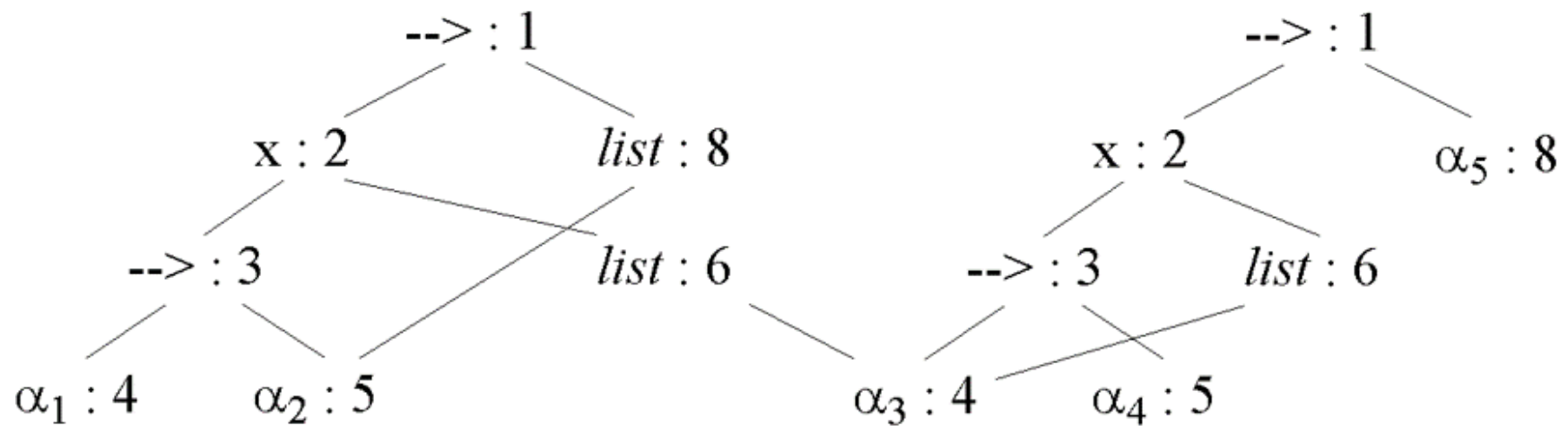
unify(6, 13): 相同操作符, 合并, unify(4, 4)—真
→unify(6, 13)为真, 6作为代表结点
→unify(2, 10)为真, 2作为代表结点

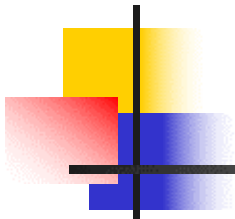
unify(8, 14): 一个变量, 合并, 8作为代表结点, 返回真
→unify(1, 9)为真, 1作为代表结点



例6.14（续）

最终结果





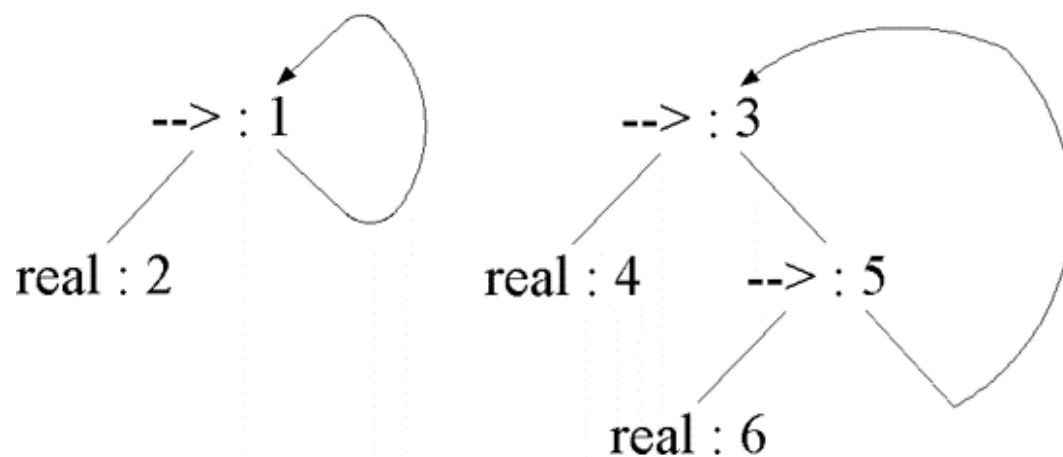
构造合一代换

- 算法6.1最终得到的类型图中，每个结点 n 表示与 $\text{find}(n)$ 相关联的类型表达式
- 对每个变量 α ， $\text{find}(\alpha)$ 表示 α 的等价类的代表结点 n
- 因此， n 表示的表达式实际上就是 $S(\alpha)$
- 如上例
 - ▣ α_3 的代表结点为4，表示 α_1
 - ▣ α_5 的代表结点为8，表示 $\text{list}(\alpha_2)$

例6.15

$e : \text{real} \rightarrow e$

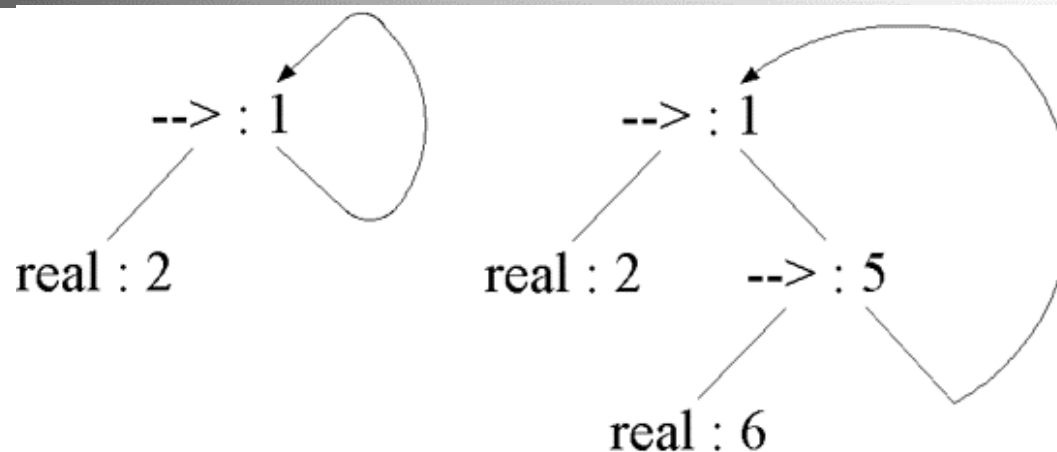
$f : \text{real} \rightarrow (\text{real} \rightarrow f)$



$\text{unify}(1, 3)$: 相同操作符, 合并, $\text{unify}(2, 4)$, $\text{unify}(1, 5)$

$\text{unify}(2, 4)$: 相同基本类型, 合并, 返回真

例6.15（续）



`unify(1, 5)`: 相同操作符，合并，`unify(2, 6)`, `unify(1, 1)`
`unify(2, 6)`: 相同基本类型，合并，返回真
 \rightarrow `unify(1, 5)`为真 \rightarrow `unify(1, 3)`为真

