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

类型检查

《编译原理和技术》

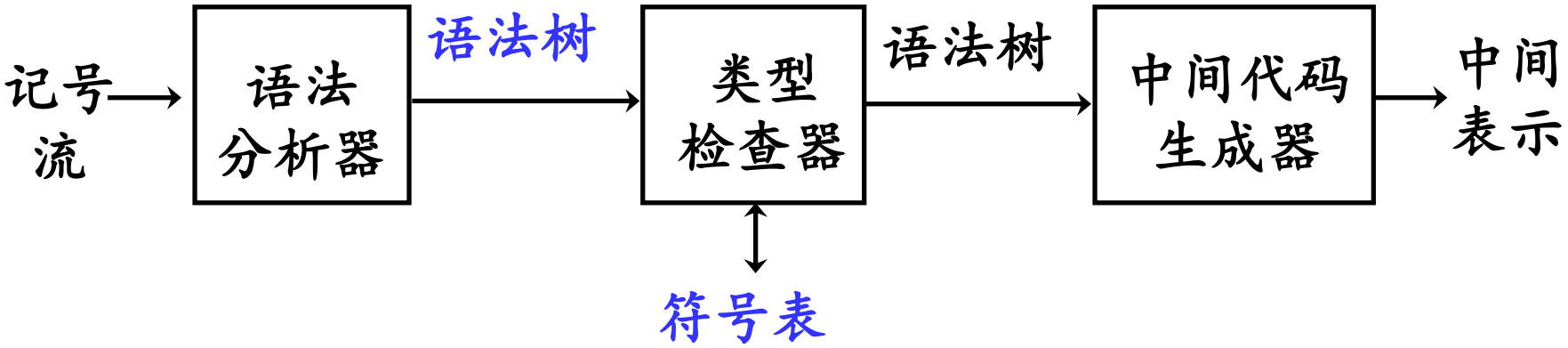
张昱

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

5.1 类型在编程语言中的作用

- 执行错误与安全语言
- 类型与类型系统
- 类型的作用

程序运行时的执行错误

会被捕获的错误（trapped error）

- 例：非法指令错误、非法内存访问、除数为零
- 捕获到错误，会使计算立即停止

divzero.c

```
#include <stdio.h>
int main(){
        printf("%d", 10/0);
}
```

除数为零

gcc编译

```
$ gcc divzero.c  -o divzero
divzero.c: In function ‘main’:
divzero.c:3:17: warning: division by zero [-Wdiv-by-zero]
  printf("%d", 10/0);
                 ^
```

生成可执行文件divzero

```
$ ./divzero
浮点数例外 (核心已转储)
```

执行到除零运算，立即停止

Floating point exception(core dumped)

出现core dumped时可调试

```
$ulimit -c 1024     设置存储core的大小
$./a.out            行会引起异常的可执行程序
$.gdb --core=core   调试core
```

注意：源文件编译时加上 -g 选项以生成调试信息

程序运行时的执行错误

- 会被捕获的错误（trapped error）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止
- 不会被捕获的错误(untrapped error)
  - 例：下标变量的访问越过了数组的末端；
    跳到一个错误的地址，该地址开始的内存正好代表一个指令序列
  - 错误可能会有一段时间未引起注意

希望可执行的程序不存在不会被捕获的错误

安全语言

- 良行为的(well-behaved)程序
  - 没有统一的定义
  - 如: 良行为的程序定义为没有任何不会被捕获的错误
- 安全语言(safe language)
  - 定义: 安全语言的任何合法程序都是良行为的
  - 设计类型系统,通过静态类型检查拒绝不会被捕获的错误
  - 设计正好只拒绝不会被捕获错误的类型系统是困难的
- 实际往往是拒绝禁止错误(*forbidden error*)
  - 不会被捕获错误集合 + 会被捕获错误的一个子集

类型及语言

- 变量的类型
  - 限定了变量在程序执行期间的取值范围
- 类型化的语言(typed language)
  - 变量都被给定类型的语言，如C/C++、Java、Go
  - 表达式、语句等程序构造的类型都可以静态确定

例如，类型*boolean*的变量*x*在程序每次运行时的值只能是布尔值，*not* (*x*)总有意义

no static types,
而非没有类型

- 未类型化的语言(untyped language)
  - 不限制变量值范围的语言, 如 LISP、JavaScript、Perl

```
var ab;
```

类型化的语言

显式类型化语言

- 类型是语法的一部分, 如

  int a, b;

隐式类型化的语言

- 不存在纯隐式类型化的主流语言，但可能存在忽略类型信息的程序片段, 如不需要程序员在声明函数中给出参数类型

  float f( ); // 仅说明函数的返回类型而忽略对参数的声明

  … f(3)+3.0…

  float f(int a) { return (float)a; } // 实际函数定义包含参数

  设计理由：编译器需要在函数调用处检查其返回值是否允许参加相应的计算

类型系统

- 语言的组成部分，其构成成分是

  一组定型规则（*typing rule*），用来给各种程序构造指派类型

- 设计目的

  用静态检查的方式来保证合法程序在运行时的良行为

- 类型系统的形式化

  类型表达式、定型断言、定型规则

- 类型检查算法

  通常是静态地完成类型检查

类型可靠的语言

- 良类型的程序(well-typed program)

  没有类型错误的程序，也称合法程序

- 类型可靠（type sound）的语言
  - 所有良类型程序（合法程序）都是良行为的
  - 类型可靠的语言一定是安全的语言

若语言定义中，除类型系统外，没有用其它方式表示对程序的约束

语法的和静态的概念	动态的概念
类型化语言	安全语言
良类型程序	良行为的程序

类型检查

未类型化语言

可以通过运行时的类型推断和检查来排除禁止错误

类型化语言

- 类型检查也可以放在运行时完成，但影响效率
- 一般都是静态检查，类型系统被用来支持静态检查
- 通常也需要一些运行时的检查，如数组访问越界检查

一些实际的编程语言并不安全

禁止错误集合没有囊括所有不会被捕获的错误

例 C语言的共用体

```
int main() {
        union U { int u1; int *u2;} u;
        int *p;
        u.u1 = 10;
        p = u.u2;
        *p = 0;

}
```

Segmentation Fault
段错误 (核心已转储)

原因：地址为10的内存单元不是用户态能访问存储单元

一些实际的编程语言并不安全

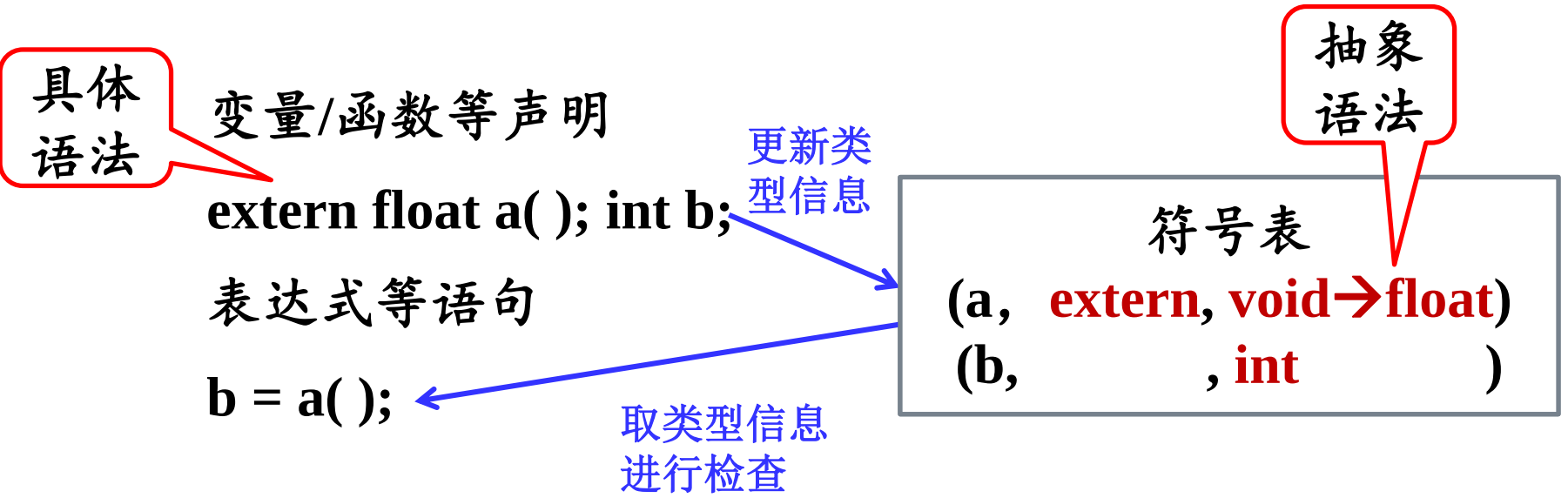
- C语言
  - 有很多不安全但被广泛使用的特征，如：

    指针算术运算、类型强制、参数个数可变
  - 在语言设计的历史上，安全性考虑不足是因为当时强调代码的执行效率
- 在现代语言设计上，安全性的位置越来越重要
  - C的一些问题已经在C++中得以缓和
  - 更多一些问题在Java中已得到解决

类型化语言的优点

从工程的观点看

- 开发的实惠：较早发现错误、类型信息具有文档作用
- 编译的实惠：程序模块可以相互独立地编译
- 运行的实惠：可得到更有效的空间安排和访问方式



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

5.2 描述类型系统的语言

- 类型系统的形式化
  - 断言、推理规则
- 类型检查和类型推断

类型系统的形式化

类型系统是一种逻辑系统

有关自然数的逻辑系统

- 自然数表达式（需要定义它的语法）

  a+b, 3

- 良形公式　　（逻辑断言，需要定义它的语法）

  a+b=3, (d=3)∧(c<10)

- 推理规则

$$\frac{a < b, \quad b < c}{a < c}$$

前提

结论

类型系统的形式化

类型系统是一种逻辑系统

有关自然数的逻辑系统

- 自然数表达式

  a+b, 3

- 良形公式

  a+b=3, (d=3)∧(c<10)

- 推理规则

类型系统

- 类型表达式

  int, int → int

- 定型断言(typing assertion)

  x:int |– x+3 : int

- 定型规则(typing rules)

$$\frac{a < b,\quad b < c}{a < c}$$

$$\frac{\Gamma \mid\!- M : \text{int}, \Gamma \mid\!- N : \text{int}}{\Gamma \mid\!- M + N : \text{int}}$$

定型环境
（即符号表）

断言

断言的形式

$\Gamma \vdash S$　　S的所有自由变量都声明在Γ中

其中

- Γ是一个静态定型环境（在编译器实现中，为符号表）

  如$x_1:T_1, \ldots, x_n:T_n$
- S的形式随断言形式的不同而不同
- 断言有三种具体形式

断言的种类

- **定型环境的断言**

  $\Gamma \vdash \Diamond$　　该断言表示Γ是良形的定型环境

  - 将用推理规则来定义环境的语法(而不是用文法)

- **类型表达式的语法断言**

  $\Gamma \vdash \mathbf{nat}$　　在定型环境Γ下，nat是类型表达式

  - 将用推理规则来定义类型表达式的语法

- **语法项的定型断言**

  $\Gamma \vdash M : T$　　在定型环境Γ下， 语法项M具有类型T

  例：$\varnothing \vdash \mathbf{true} : \mathbf{boolean}$　　$x : \mathbf{nat} \vdash x+1 : \mathbf{nat}$

  - 将用推理规则来确定程序构造实例的类型

断言的有效性、推理规则

断言的有效性

- 合法的断言(valid assertion)

  $\Gamma \vdash$ true : boolean

- 不合法的断言(invalid assertion)

  $\Gamma \vdash$ true : nat

推理规则(inference rules)

$$\frac{\Gamma_1 \vdash S_1, \ \ldots, \ \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

- 前提(premise)、结论(conclusion)
- 公理(axiom)（前提为空）、推理规则

推理规则

（规则名） （注释） 推理规则 （注释）

- 环境规则

(Env ∅)

$$\frac{}{\varnothing \vdash \Diamond}$$

空环境是良形的环境

- 语法规则

(Type Bool)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \text{boolean}}$$

boolean是类型表达式

- 定型规则

(Val +)

$$\frac{\Gamma \vdash M : \text{int}, \Gamma \vdash N : \text{int}}{\Gamma \vdash M + N : \text{int}}$$

在环境Γ下，$M + N$是int类型

类型检查

类型检查(type checking)

- 用语法制导的方式，根据上下文有关的定型规则来判定程序构造是否为良类型的程序构造的过程

可以边解析边检查，也可以在访问AST时进行检查

类型推断(type inference)

在类型信息不完整的情况下的定型判定问题

例如：$f(x:t)=E$ 和 $f(x)=E$的区别

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

5.3 简单类型检查器的说明

- 一个简单的语言及其类型系统
- 类型检查

一个简单的语言

$P \rightarrow D\ ;\ S$

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

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

$\uparrow T \mid T\ \text{`}\rightarrow\text{'}\ T$

$S \rightarrow \text{id} := E \mid \text{if } E \text{ then } S \mid \text{while } E \text{ do } S \mid S\ ;\ S$

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

$E\uparrow \mid E\ (E\ )$

例

```
i : integer;
j : integer;
j := i mod 2000
```

类型系统

环境规则

(Env ∅)

$$\frac{}{\varnothing \vdash \Diamond}$$

(Decl Var)

$$\frac{\Gamma \vdash T,\ id \notin dom(\Gamma)}{\Gamma, id : T \vdash \Diamond}$$

id不在Γ的定义域中

其中$id : T$是该简单语言的一个声明语句

遇到一个变量声明语句，若该变量在此之前未被声明过（即$id \notin dom(\Gamma)$），则向定型环境（符号表）中增加一个符号定型，即$id : T$

类型系统

- 语法规则：哪些是合法的类型表达式

(Type Bool)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{boolean}}$$

(Type Int)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{integer}}$$

(Type Void)

*void*用于表示语句类型

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{void}}$$

➔编程语言和定型断言的类型表达式并非完全一致

基本类型是合法的类型表达式
如*boolean*、 *integer*、 *void*

类型系统

语法规则：哪些是合法的类型表达式

如果T是类型，则$pointer(T)$是类型

(Type Ref) $(T \neq void)$

具体语法：$\uparrow T$

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

(Type Array) $(T \neq void)$ $(N>0)$

具体语法：array [N] of T

$$\frac{\Gamma \vdash T,\ \Gamma \vdash N : integer}{\Gamma \vdash array(N, T)}$$

(Type Function) $(T_1, T_2 \neq void)$

T_1和T_2分别是函数的参数类型和返回类型

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

定型断言中的类型表达式用的是抽象语法

将类型构造算子作用于类型表达式可以构造新的类型表达式
如*pointer*、 *array*、 → 是类型构造算子

类型系统 -- 定型规则

定型规则——表达式

(Exp Truth)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textbf{truth} : \textit{boolean}}$$

(Exp Num)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textbf{num} : \textit{integer}}$$

(Exp Id)

$$\frac{\Gamma_1, \textbf{id} : T, \Gamma_2 \vdash \Diamond}{\Gamma_1, \textbf{id} : T, \Gamma_2 \vdash \textbf{id} : T}$$

类型系统 -- 定型规则

定型规则——表达式

(Exp Mod)

$$\frac{\Gamma \vdash E_1: integer,\ \Gamma \vdash E_2: integer}{\Gamma \vdash E_1 \bmod E_2: integer}$$

(Exp Index)

$$\frac{\Gamma \vdash E_1: array(N,T),\ \Gamma \vdash E_2: integer}{\Gamma \vdash E_1[E_2] : T}$$

$$(0 \le E_2 \le N-1)$$

(Exp Deref)

$$\frac{\Gamma \vdash E : pointer(T)}{\Gamma \vdash E\uparrow : T}$$

(Exp FunCall)

$$\frac{\Gamma \vdash E_1: T_1 \to T_2, \qquad \Gamma \vdash E_2: T_1}{\Gamma \vdash E_1\ (E_2) : T_2}$$

类型系统 -- 定型规则

定型规则——语句

(Stmt Assign) (*T*=*boolean* or *T*= *integer*)

$$\frac{\Gamma \vdash \mathrm{id} : T,\ \Gamma \vdash E : T}{\Gamma \vdash \mathrm{id} := E : \mathit{void}}$$

(Stmt If)

$$\frac{\Gamma \vdash E : \mathit{boolean},\ \Gamma \vdash S : \mathit{void}}{\Gamma \vdash \mathrm{if}\ E\ \mathrm{then}\ S : \mathit{void}}$$

(Stmt While)

$$\frac{\Gamma \vdash E : \mathit{boolean},\ \Gamma \vdash S : \mathit{void}}{\Gamma \vdash \mathrm{while}\ E\ \mathrm{do}\ S : \mathit{void}}$$

(Stmt Seq)

$$\frac{\Gamma \vdash S_1 : \mathit{void},\ \Gamma \vdash S_2 : \mathit{void}}{\Gamma \vdash S_1; S_2 : \mathit{void}}$$

类型系统—扩展

更多的类型构造算子

- **积类型**构造算子×: $T_1 \times T_2$

  可用于表示实际编程语言中的列表和元组，T_1、T_2为成员类型

  如果成员有名字，如 $\mathrm{f}_1{:}T_1 \times \mathrm{f}_2{:}T_2$

  则用于表示结构体类型、记录类型

- **和类型**构造算子+: $T_1 + T_2$

  可用于表示实际编程语言中的共用体

类型检查

设计语法制导的类型检查器

- 设计依据：前面定义的类型系统
- 定型环境 Γ 的信息存入编译器的符号表
- 对类型表达式采用抽象语法

  具体：

  数组类型 array [*N*] of *T*　抽象：*array* (*N*, *T*)

  指针类型　↑*T*　　*pointer* (*T*)
- 考虑到报错的需要，增加了类型错误 *type_error*

类型检查——声明语句

- 方法1：用语法制导的翻译方案实现类型检查

$D \rightarrow D; D$ //D1

$D \rightarrow \mathrm{id} : T$ {*addtype* (id.*entry*, *T.type*)} //D2

addtype：把符号id的类型信息*T.type*填入符号表

实现类型系统中的**环境规则**

$$\text{(Decl Var)} \quad \frac{\Gamma \vdash T,\ \mathrm{id} \notin dom(\Gamma)}{\Gamma, \mathrm{id} : T \vdash \Diamond}$$

类型检查——声明语句

- 方法1：用语法制导的翻译方案实现类型检查

  $D \rightarrow D; D$ //D1

  $D \rightarrow \text{id} : T$ {*addtype* (id.*entry*, *T.type*)} //D2

  addtype：把符号id的类型信息*T.type*填入符号表

- 方法2：**在访问AST时进行类型检查**

  可以在 *exitD2* (ast) 中增加对*addtype*的调用

如何表达多个声明*D1* 呢?

将多个声明组织成 list （可以用表示线性表的容器类）

如何处理多个声明*D1* 呢?

对list 中元素的迭代访问 （可以用现成的Iterator类）

类型检查——声明语句

语法制导的翻译方案

$D \rightarrow D; D$

$D \rightarrow \text{id} : T$ {*addtype* (id.*entry*, *T.type*)}

$T \rightarrow \text{boolean}$ {*T.type* = *boolean*}

$T \rightarrow \text{integer}$ {*T.type* = *integer*}

$T \rightarrow \uparrow T_1$ {*T.type* = *pointer*(T_1.*type*)}

$T \rightarrow \text{array [num] of } T_1$

{*T.type* = *array*(num.*val*, T_1.*type*)}

$T \rightarrow T_1$ '$\rightarrow$' T_2 {*T.type* = T_1.*type* $\rightarrow$ T_2.*type* }

实现类型系统中的**语法规则**

(Type Function)
($T_1, T_2 \neq void$)

$$\frac{\Gamma \mid\!- T_1,\ \Gamma \mid\!- T_2}{\Gamma \mid\!- T_1 \rightarrow T_2}$$

类型检查——声明语句

语法制导的翻译方案

$D \rightarrow D; D$

$D \rightarrow \text{id} : T$ $\quad \{addtype(\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean}$ $\quad \{T.type = boolean\}$ (Bool,--)

$T \rightarrow \text{integer}$ $\quad \{T.type = integer\}$ (Int, --)

$T \rightarrow \uparrow T_1$ $\quad \{T.type = pointer(T_1.type)\}$ (Pointer,T1)

$T \rightarrow \text{array [num] of } T_1$

$\{T.type = array(\text{num}.val, T_1.type)\}$ (Array,T1,num)

$T \rightarrow T_1$ '$\rightarrow$' T_2 $\quad \{T.type = T_1.type \rightarrow T_2.type\}$ (Fun,T1, T2)

如何实现对各种类型的表示?

用记录类型: (类型的类别kind，该类别的类型的其他信息)

类型检查——表达式

语法制导的翻译方案

$E \rightarrow \text{truth}$ $\{E.type = boolean\}$

$E \rightarrow \text{num}$ $\{E.type = integer\}$

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

查符号表，获取 id的类型

实现类型系统中的**定型规则**

(Exp Id)
$$\frac{\Gamma_1, \text{id} : T, \Gamma_2 \vdash \Diamond}{\Gamma_1, \text{id} : T, \Gamma_2 \vdash \text{id} : T}$$

类型检查——表达式

语法制导的翻译方案

$E \rightarrow$ truth　　{*E.type* = *boolean* }

$E \rightarrow$ num　　{*E.type* = *integer*}

$E \rightarrow$ id　　{*E.type* = *lookup*(id.*entry*)}

$E \rightarrow E_1$ mod E_2　　{*E.type* = if E_1.*type* == *integer* and E_2. *type* == *integer* then *integer* else *type_error* }

$E \rightarrow E_1 [E_2]$　　{*E.type* = if E_2. *type* == *integer* and E_1. *type* == *array*(*s*, *t*) then *t* else *type_error* }

(Exp Mod)

$$\frac{\Gamma \mid\!- E_1 : integer,\ \Gamma \mid\!- E_2 : integer}{\Gamma \mid\!- E_1 \bmod E_2 : integer}$$

(Exp Index)

$$\frac{\Gamma \mid\!- E_1 : array(N,T),\ \Gamma \mid\!- E_2 : integer}{\Gamma \mid\!- E_1[E_2] : T} \quad (0 \le E_2 \le N-1)$$

类型检查——表达式

语法制导的翻译方案

$$E \rightarrow E_1\uparrow \{\ E.type = \text{if } E_1.type == pointer(t) \text{ then } t$$

$$\text{else } type\_error\ \}$$

$$E \rightarrow E_1\,(E_2\,)\ \{E.\ type = \text{if } E_2\ .\ type == s \text{ and}$$

$$E_1.\ type == s \rightarrow t \text{ then } t$$

$$\text{else } type\_error\ \}$$

(Exp Deref)
$$\frac{\Gamma \mid\!- E : pointer(T)}{\Gamma \mid\!- E\uparrow : T}$$

(Exp FunCall)
$$\frac{\Gamma \mid\!- E_1 : T_1 \rightarrow T_2, \qquad \Gamma \mid\!- E_2 : T_1}{\Gamma \mid\!- E_1\,(E_2) : T_2}$$

类型转换

语法制导的翻译方案

$E \rightarrow E_1$ op E_2

{*E.type* = if E_1.*type* == *integer* and E_2.*type* == *integer*
then *integer*
else if E_1.*type* == *integer* and E_2.*type* == *real*
then *real*
else if E_1.*type* == *real* and E_2.*type* == *integer*
then *real*
else if E_1.*type* == *real* and E_2.*type* == *real*
then *real*
else *type_error* }

类型检查——语句

语法制导的翻译方案

$S \rightarrow$ id := E { if (*id.type* == *E.type* &&
E.type $\in$ {*boolean*, *integer*}) *S.type* = *void*;
else *S.type* = *type_error*;}

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

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

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

类型检查——程序

语法制导的翻译方案

$$P \rightarrow D; S \quad \{P.type = \text{if } S.type == void \text{ then } void$$

$$\text{else } type\_error \}$$

现代编译器的主流实现

- 主流过程：[ParseTree]➔ AST ➔ 类型检查
- 类型检查器的实现
  - 一般是对语法树进行类型检查

设计实现的关键：

- 符号表的设计：如何表示不同的类型
- 语法树的Visitor设计

回顾：ANTLR会生成与标签对应的语法结构的enter和exit方法

- 可以带标签(#标签名，后跟空格或换行)

```
e : e '*' e # Mult | e '+' e # Add | INT # Int ;
```

ANTLR为每个标签产生规则上下文类 XXXParser.MultContext

- 有何用处？

ANTLR会生成与该标签对应的语法结构的enter和exit方法

```
public interface XXXListener extends ParseTreeListener {
    void enterMult(XXXParser.MultContext ctx);
    void exitMult(XXXParser.MultContext ctx);
    ……
}
```

现代编译器的主流实现

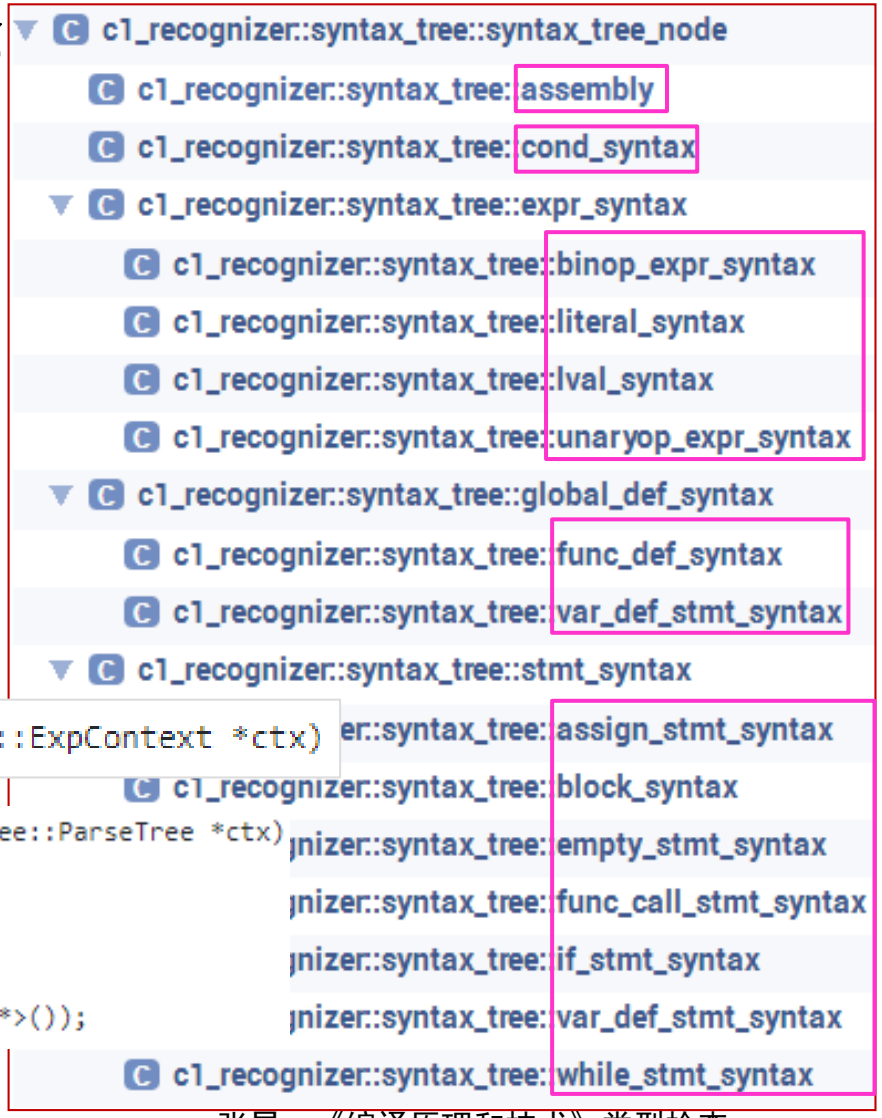
[ParseTree]➔ AST ➔ 类型检查

- 用ANTLR构造分析器

  ParseTree➔AST

  - syntax_tree_node
  - 访问者

    syntax_tree_builder




```
antlrcpp::Any syntax_tree_builder::visitExp(C1Parser::ExpContext *ctx)
```

```
ptr<syntax_tree_node> syntax_tree_builder::operator()(antlr4::tree::ParseTree *ctx)
{
    auto result = visit(ctx);
    if (result.is<syntax_tree_node *>())
        return ptr<syntax_tree_node>(result.as<syntax_tree_node *>());
```

现代编译器的主流实现

[ParseTree]➔ AST ➔ 类型检查

- **AST的定义**
  - syntax_tree_node

Public Member Functions

virtual void **accept** (syntax_tree_visitor &visitor)=0

Public Attributes

int **line**

int **pos**

  - syntax_tree
- **访问者**

syntax_tree_visitor

- c1_recognizer::syntax_tree::syntax_tree_node
  - c1_recognizer::syntax_tree::assembly
  - c1_recognizer::syntax_tree::cond_syntax
  - c1_recognizer::syntax_tree::expr_syntax
    - c1_recognizer::syntax_tree::binop_expr_syntax
    - c1_recognizer::syntax_tree::literal_syntax
    - c1_recognizer::syntax_tree::lval_syntax
    - c1_recognizer::syntax_tree::unaryop_expr_syntax
  - c1_recognizer::syntax_tree::global_def_syntax
    - c1_recognizer::syntax_tree::func_def_syntax
    - c1_recognizer::syntax_tree::var_def_stmt_syntax
  - c1_recognizer::syntax_tree::stmt_syntax
    - c1_recognizer::syntax_tree::assign_stmt_syntax
    - c1_recognizer::syntax_tree::block_syntax
    - c1_recognizer::syntax_tree::empty_stmt_syntax
    - c1_recognizer::syntax_tree::func_call_stmt_syntax
    - c1_recognizer::syntax_tree::if_stmt_syntax
    - c1_recognizer::syntax_tree::var_def_stmt_syntax
    - c1_recognizer::syntax_tree::while_stmt_syntax

例题 1

编译时的控制流检查的例子

```
main() {
    printf("\n%ld\n",gcd(4,12));
    continue;
}
```

编译时的报错如下：

```
continue.c:  In function 'main':
continue.c:3: continue statement not within a loop
```

例题 2

编译时的唯一性检查的例子

```
main() {
int i;
    switch(i){
    case 10: printf("%d\n", 10); break;
    case 20: printf("%d\n", 20); break;
    case 10: printf("%d\n", 10); break;
    }
}
```

编译时的报错如下：

```
switch.c: In function 'main':
switch.c:6: duplicate case value
switch.c:4: this is the first entry for that value
```

例题 3

C语言

- 称&为地址运算符，&a为变量a的地址
- 数组名代表数组第一个元素的地址

问题：

如果*a*是一个数组名，那么表达式*a*和&*a*的值都是数组*a*第一个元素的地址，它们的使用是否有区别？

用四个C文件的编译报错或运行结果来提示

例题 3


```
typedef  int  A[10][20];
A  a;

A  *fun() {
     return(a);
}
```

该函数在Linux上用gcc编译，报告的错误如下：

第5行：warning: return from incompatible pointer type

例题 3


```
typedef  int  A[10][20];
A  a;

A  *fun() {
     return(&a);
}
```

该函数在Linux上用gcc编译时，没有错误

例题 3


```
typedef  int  A[10][20];
typedef  int  B[20];
A  a;

B  *fun() {
     return(a);
}
```

该函数在Linux上用gcc编译时，没有错误

例题 3


```
typedef  int  A[10][20];
A  a;

fun() { printf("%d,%d,%d\n", a, a+1, &a+1);}

main() { fun(); }
```

该程序的运行结果是：

134518112, 134518192, 134518912

例题 3

结论

对一个t 类型的数组$a[\,i_1\,][\,i_2\,]\ldots[\,i_n\,]$来说，

表达式a的类型是：

pointer(array(0.. i_2–1, … array(0.. i_n–1, t)…))

表达式&a的类型是：

pointer(array(0.. i_1–1, … array(0.. i_n–1, t)…))

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 then
    return true
else if s == *array*(s_1, s_2) and t == *array*(t_1, t_2) then
    return sequiv(s_1, t_1) and sequiv(s_2, t_2)
else if s == $s_1 \times s_2$ and t == $t_1 \times t_2$ then
    return sequiv(s_1, t_1) and sequiv(s_2, t_2)
else if s == *pointer* (s_1) and t == *pointer*(t_1) then
    return sequiv(s_1, t_1)
else if s == $s_1 \rightarrow s_2$ and t == $t_1 \rightarrow t_2$ then
    return squiv(s_1, t_1) and sequiv(s_2, t_2)
else return false

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)

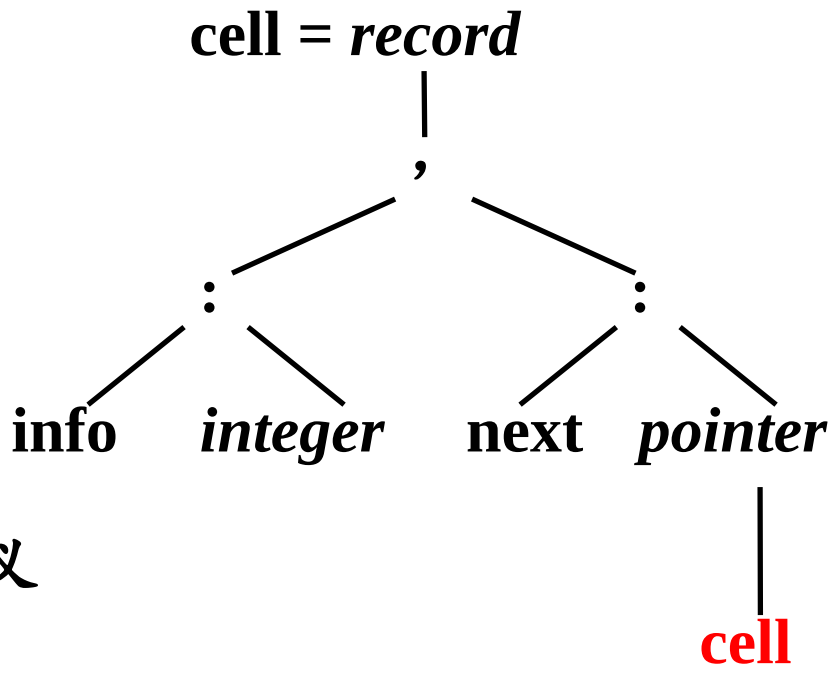
$$\frac{\Gamma \mid\!- M_1{:}T_1, \ldots, \Gamma \mid\!- M_n{:}\, T_n}{\Gamma \mid\!- \text{record}\ (l_1{=}M_1, \ldots, l_n{=}M_n) : \text{record}\ (l_1{:}T_1, \ldots, l_n{:}T_n)}$$

(Val Record Select)

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

类型表示中的环

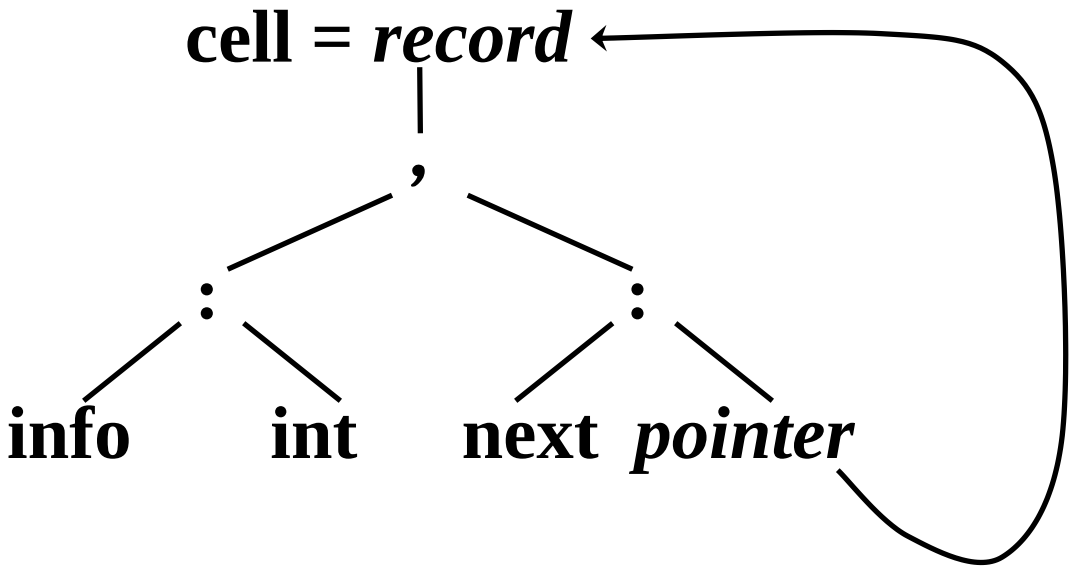
```
type link = ↑ cell ;
cell = record
        info : integer ;
        next : link
end;
```



引入环的话，递归定义的类型名可以替换掉

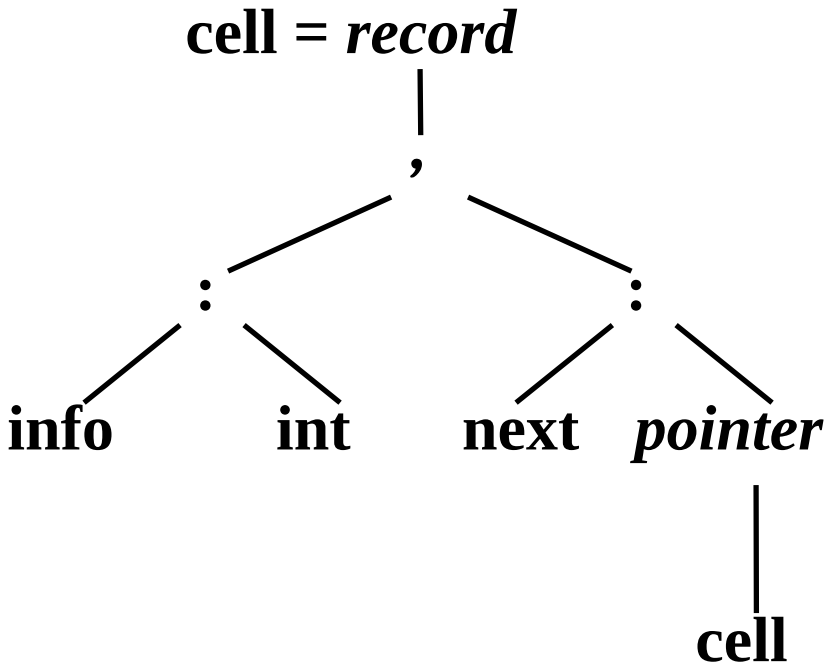
类型表示中的环

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



类型表示中的环

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;
```

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;
```

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

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

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

- 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

- 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
  - 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，在不要求标识符的声明先于使用的语言中，可以通过使用类型变量来确定程序变量的类型

```
function deref (p);        -- 对p的类型一无所知：β
begin
    return p↑              -- β = pointer (α)
end;
```

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

这是一个抽象语言，忽略了函数定义的函数体

$P \rightarrow D; E$

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

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

$\mid T$

$T \rightarrow T\ \text{`}\rightarrow\text{'}\ T$

$\mid T \times T$ 笛卡儿积类型

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

$\mid \text{basic-type}$

$\mid \text{type-variable}$ 引入类型变量

$\mid (\ T\ )$

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

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

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

begin
if *t* 是基本类型 then return *t*
else if *t* 是类型变量 then return *Sv*(*t*)
else if *t* 是$t_1 \rightarrow t_2$ then return
subst(t_1, *Sv*) → *subst*(t_2, *Sv*)
end

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

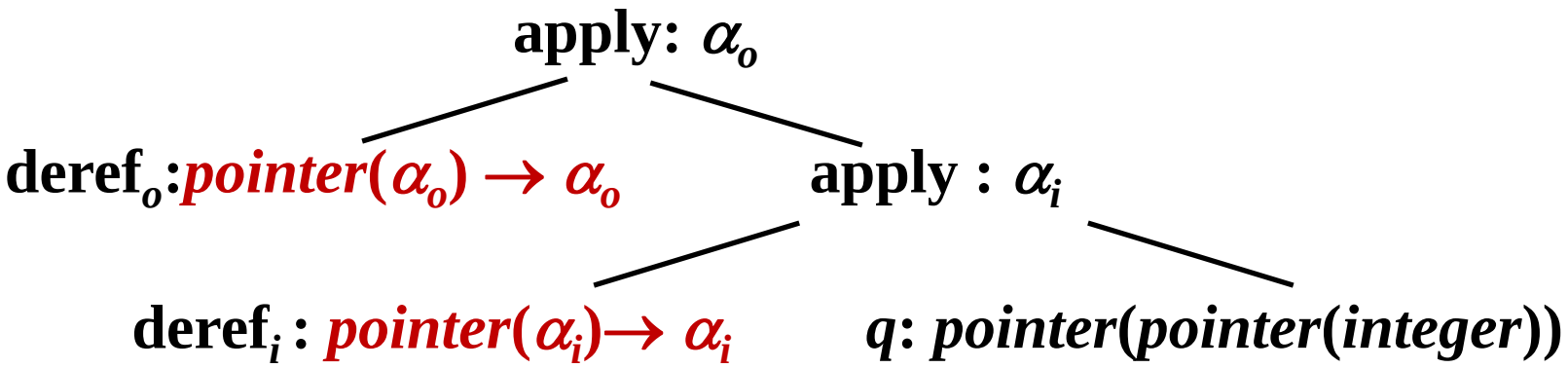
多态函数的类型检查

- 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}

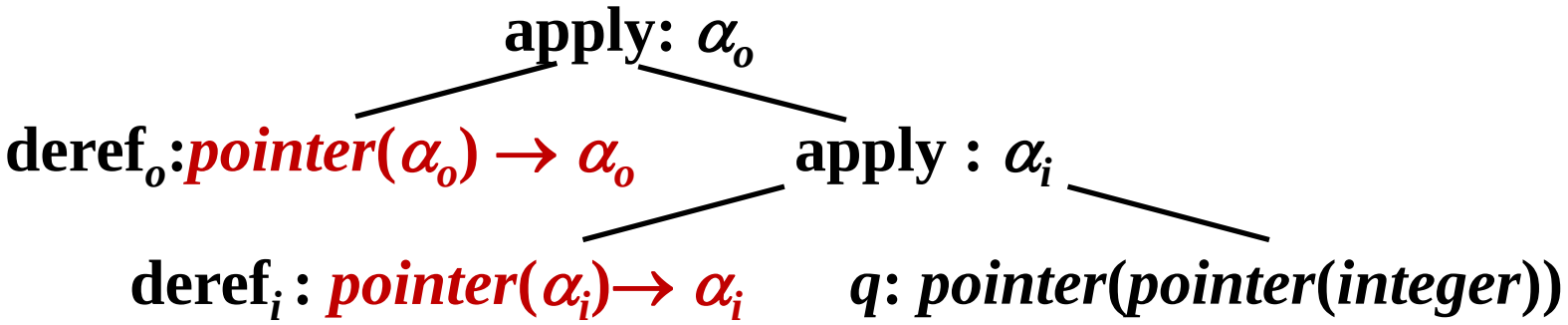
$E \rightarrow E_1, E_2$

{*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 : $\forall\alpha$. $\alpha\times\alpha\rightarrow\alpha$;

类型声明部分

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

```
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)
)
```

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* × *integer* → *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;

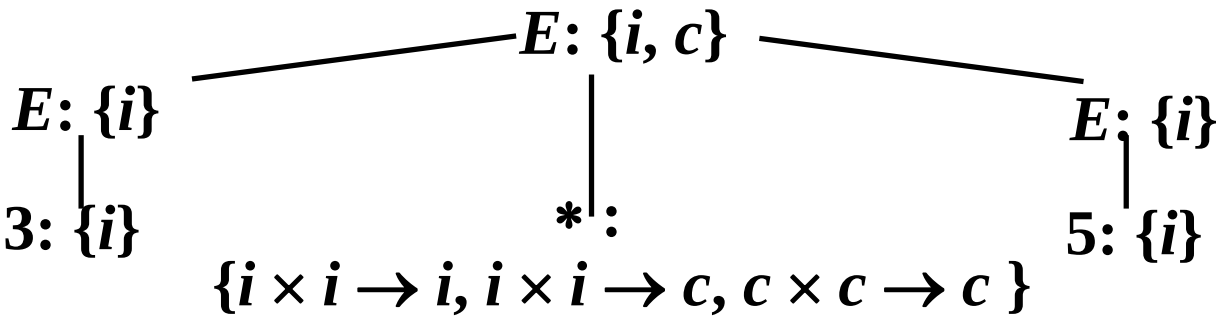
使得算符*重载，可能的类型包括：

$integer \times integer \rightarrow integer$ --这是预定义的类型

$integer \times integer \rightarrow complex$

$complex \times complex \rightarrow complex$

(3 * 5) * z（*z:complex*）



重载函数的应用

缩小可能类型的集合

- $E' \rightarrow E$ $\quad$ *E. unique* = if *E. types* == { *t* } then *t* else *type_error*
- $E \rightarrow$ id $\quad$ *E. types* = *lookup*(id. *entry*)
- $E \rightarrow E_1(E_2)$ $\quad$ *E. types* = {*s′* | E_2. *types*中存在一个*s*，使得*s* $\rightarrow$ *s′*属于E_1. *types*}

  t = *E. unique*

  S = {*s* | *s* $\in$ E_2. *types* and *s* $\rightarrow$ *t* $\in$ E_1.*types* }

  E_2. *unique* = if *S* == { *s* } then *s* else *type_error*

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

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));
}
```