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

类型检查 I

《编译原理和技术(H)》

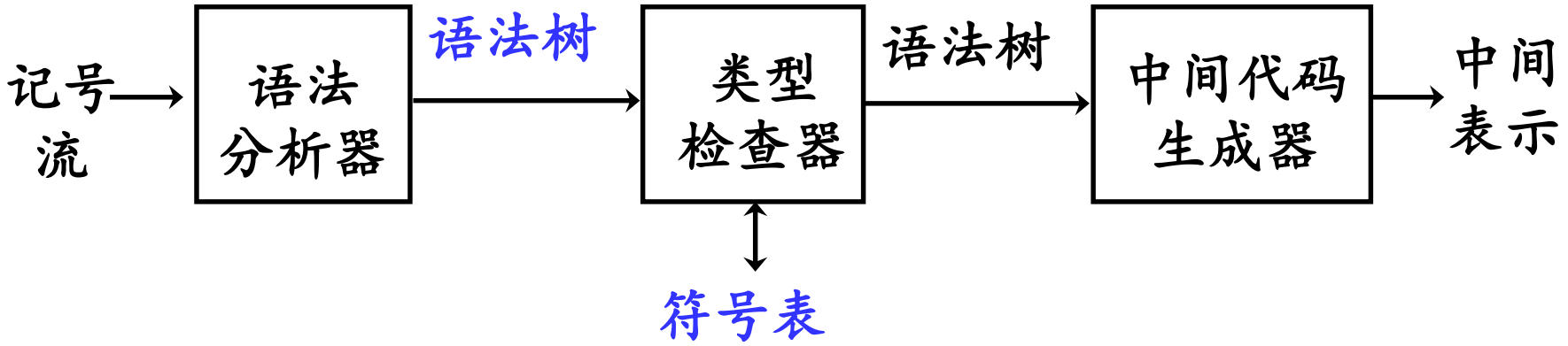
张昱

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

5.1 类型在编程语言中的作用

- 类型化语言与类型系统
- 执行错误与安全语言
- 不安全性及其检查
- 类型化语言的优点

类型化语言和未类型化语言

类型：可限定变量在程序执行期间的取值范围和能参与的操作

- **类型化的语言(typed language)**
  - **变量都被给定类型，运算都被定义了运算对象和运算结果所允许的类型**

    例如，C语言中int类型的变量x的值是4字节的整数，可以进行算术运算等
  - **静态类型语言：如C/C++、Ada、ML、Java、Go、Rust**

    对变量、表达式的类型指派是在编译时进行
  - **动态类型语言：如Lisp、Perl、PHP、Python、JavaScript、Ruby**

    在运行时确定表达式类型，变量不绑定到特定类型
- **未类型化的语言(untyped language)：如汇编语言、纯λ演算**
  - **没有显式的类型系统（缺少静态类型），或者说仅有一个泛类型**
    - 允许运算作用于任意对象，其结果可能是有意义的值、错误、异常、未定义的结果

类型系统

- **语言的一部分**，由一组定型规则（typing rule），用来给各种程序构造指派类型

  例如，如果 M 和 N 都是整型表达式，那么 $M + N$ 也是整型表达式

  - **显式类型化语言**：程序员必须在定义变量或函数时明确地声明类型，如C/C++、Java
  - **隐式类型化语言**：允许程序员在代码中忽略部分类型信息，如ML

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

- **设计目的**
  - 用类型检查的方式来保证合法程序在运行时的良行为
- **类型检查**：根据定型规则来确定程序中各语法构造的类型

  通常是静态类型检查；对于静态无法检查的，可由编译器生成动态检查代码
- **类型推断**：在缺少显式类型声明时，根据上下文推断出变量或表达式的类型

```
fun add x y = x + y;  // ML 语言编译器会根据对变量 x 和 y 做加法，推断 x 和 y 是数值类型
```

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

- 类型化语言与类型系统
- 执行错误与安全语言
- 不安全性及其检查
- 类型化语言的优点

程序运行时的执行错误

可捕获的错误（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　　执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

- 可捕获的错误（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
- 不可捕获的错误(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可捕获的错误

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可捕获的错误
- 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可捕获的错误
  - 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

类型可靠的语言

- 良类型的程序(well-typed program)
  - 良类型的程序是没有类型错误的程序
  - 在语言定义中，除文法外，若所有上下文有关的限制都由类型系统表达，则良类型的程序即是合法程序
- 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

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	良行为程序

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
- 类型化语言与类型系统
- 不安全性及其检查
- 类型化语言的优点

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可捕获的错误

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++中得以缓解
  - 更多一些问题在Java中已得到解决

安全性、灵活性与性能的权衡

- 静态类型 vs. 动态类型 vs. 无类型(唯一的泛类型)
  - 静态类型：一般都是静态检查，也需要一些运行时的检查，如数组访问越界类型检查也可以放在运行时完成，但影响效率
  - 动态类型或无类型：可通过运行时的类型推断和检查来排除禁止错误
- 安全性：语言设计中，性能的追求会伴随着不安全性的引入
- 现代程序语言进展
  - 动态类型语言的静态类型推断：如Google面向Python的PyPi
  - 引入渐进类型（2006 Jeremy Siek）：What is Gradual Typing
  - 安全性日趋重要

    Kotlin、TypeScript、Rust 在2018年GitHub贡献者数增速分别为 2.6 、1.9、1.7 倍

```
def f(x, y : int):
    return x+y

result = f(2, 4)
print(result)
```

静态类型的好处 — 性能和安全性

对语言的性能评估

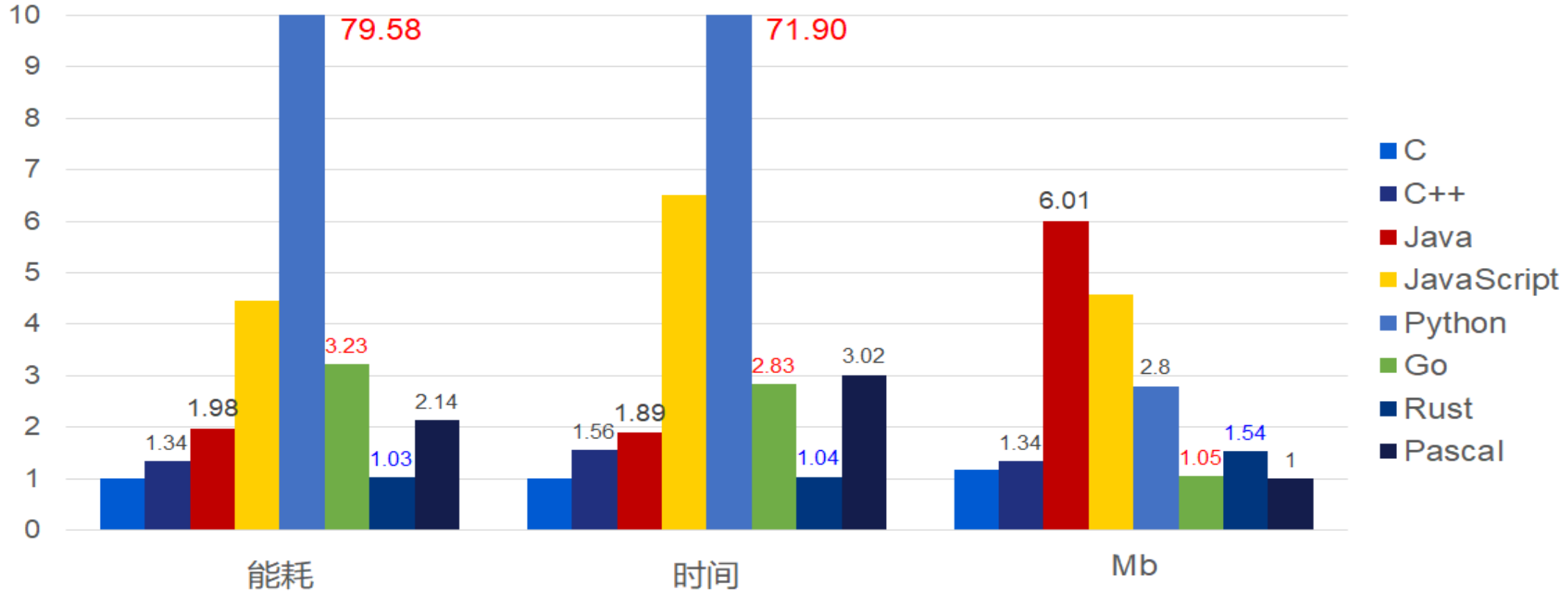
Which programming language is fastest?

source	secs	mem	gz
Intel C	23.34	11,052	427
C gcc #2	25.22	11,112	406
C gcc	26.27	11,112	427
Go #2	26.53	11,112	494
Go	27.22	11,112	462
Java	29.53	41,244	439
C# .NET	47.52	30,452	465
PHP #2	143.01	12,992	391
PHP #3	163.38	13,004	412
PHP	204.22	12,976	384
Ruby #2	19 min	25,596	307
Ruby	20 min	25,492	335
Python 3 #3	23 min	11,064	384
Python 3 #2	30 min	11,192	330
Python 3	1h 03 min	11,032	373

https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

编程语言的能效对比

编程语言的能效对比



CLBG :The Computer Language Benchmarks Game

[SLE2017] Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?

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

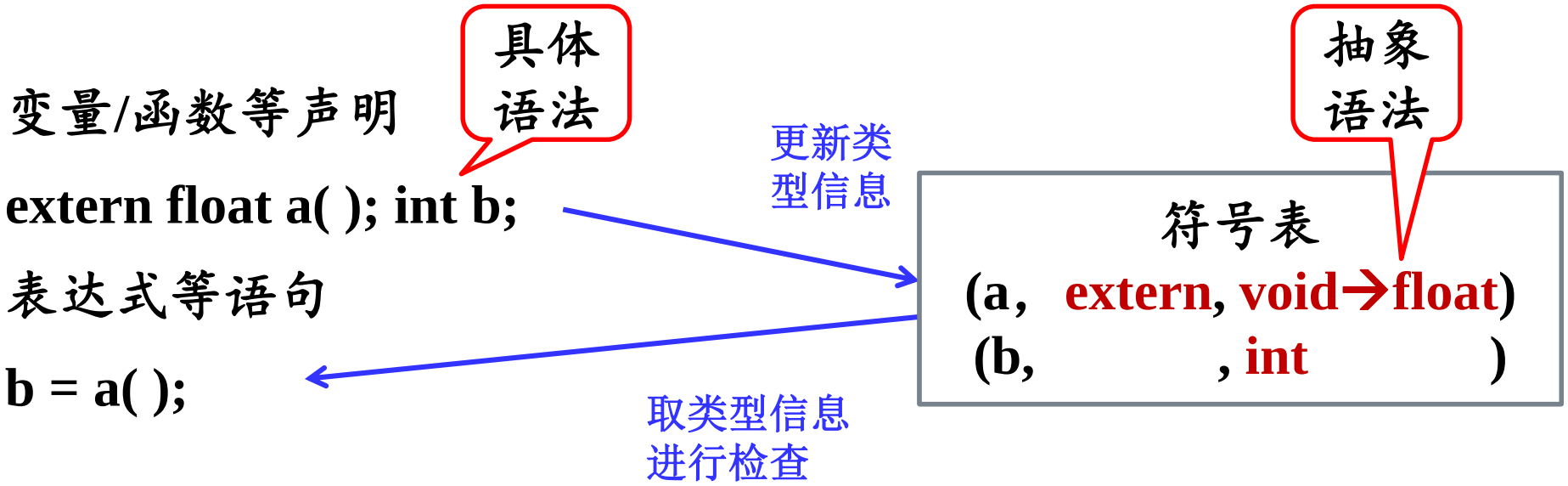
5.1 类型在编程语言中的作用

- 执行错误与安全语言
- 类型化语言与类型系统
- 不安全性及其检查
- 类型化语言的优点

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 类型系统的描述语言

- 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

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

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)

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

静态定型环境
（即符号表）

断言

断言的形式

Γ |– 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)

  $\Gamma \vdash \text{true} : \text{boolean}$

- 不合法的断言(invalid assertion)

  $\Gamma \vdash \text{true} : \text{nat}$

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

$P \rightarrow D\ ;\ S$

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

$T \rightarrow \text{boolean} \mid \text{integer} \mid \text{array [num] of } T \mid {*T} \mid T\ \text{‘}\rightarrow\text{’}T$

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

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

例

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

类型系统

环境规则

(Env ∅)

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

(Decl Var)

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

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 ∉ *dom* (Γ) ），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)

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

*void*用于表示语句类型

➔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

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

具体语法： $*T$

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

如果T是类型，则$pointer(T)$是类型

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

具体语法： **array** $[N]$ **of** T

$$\frac{\Gamma \vdash T,\ \Gamma \vdash N : integer}{\Gamma \vdash array(N, T)} \quad (N>0)$$

(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)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathbf{truth} : \mathit{boolean}}$$

(Exp Num)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathbf{num} : \mathit{integer}}$$

(Exp Id)

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

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)

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

(Exp Deref)

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

(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*)

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

(Stmt If)

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

(Stmt While)

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

(Stmt Seq)

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

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

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

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
  - *addtype*(id, type)：将id的类型type存入符号表，若出现重复定义则报错
  - *lookup*(id.entry)：从符号表中查找 id 的类型，若未找到则报错
- 考虑到报错的需要，增加了类型错误 ***type_error***
- 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*)**

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，边解析边检查

$D \rightarrow D; D$ //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

实现类型系统中的**环境规则**

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

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);
    ……
}
```

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：**在遍历AST时进行类型检查**

  可以在 *exitD2* 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）

```
visitD1(D1):
  foreach (child: D1.list)
    visitD2(child);
exitD2(D2):
  addtype (id.entry, T.type);
```

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*; }

$T \rightarrow *T_1$ {*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*)

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

类型检查——声明语句

语法制导的翻译方案

$D \rightarrow D; D$

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

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

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

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

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

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

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

(Bool,--)

(Int, --)

(Pointer,T1)

(Array,T1,num)

(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**

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

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　　{ if (*E_1.type == integer && E_2. type == integer*)

E.type = integer ;

else *E.type = type_error*; }

$E \rightarrow E_1$ [E_2]　　{ if (*E_2. type == integer && E_1. type == array*(*s, t*))

E.type = t;

else *E.type = 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)

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

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

类型检查——表达式

语法制导的翻译方案

$E \rightarrow *E_1$ { if (E_1.*type* == *pointer*(*t*)) E.*type* = *t* ;
else E.*type* = *type_error* ; }

$E \rightarrow E_1 (E_2)$ { if (E_2 . *type* == *s* && E_1. *type* == $s \rightarrow t$)
E.*type* = *t*;
else E.*type* = *type_error* ; }

(Exp Deref) $\dfrac{\Gamma \vdash E : pointer(T)}{\Gamma \vdash *E : T}$

(Exp FunCall) $\dfrac{\Gamma \vdash E_1 : T_1 \rightarrow T_2, \quad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1 (E_2) : T_2}$

类型转换

语法制导的翻译方案

$E \rightarrow E_1$ op E_2

{if (E_1.*type* == *integer* && E_2.*type* == *integer*)
E.type = *integer*;
else if (E_1.*type* == *integer* && E_2.*type* == *real*)
E.type = *real* ;
else if (E_1.*type* == *real* && E_2.*type* == *integer*)
E.type = *real* ;
else if (E_1.*type* == *real* && E_2.*type* == *real*)
E.type = *real* ;
else *E.type* = *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 { if ($E.type == boolean$) $S.type = S_1.type$;
else $S.type = type\_error$; }

$S \rightarrow$ while E do S_1 { if ($E.type == boolean$) $S.type = S_1.type$;
else $S.type = type\_error$; }

$S \rightarrow S_1; S_2$ { if ($S_1.type == void$ && $S_2.type == void$)
$S.type = void$;
else $S.type = type\_error$; }

类型检查——程序

语法制导的翻译方案

$P \rightarrow D; S$ { if ($S.type == void$) $P.type = void$;

else $P.type = type\_error$; }

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：

$$\text{pointer}(\text{array}(0..\,i_2-1,\ \ldots\ \text{array}(0..\,i_n-1,\,t)\ldots))$$

表达式&a的类型是：

$$\text{pointer}(\text{array}(0..\,i_1-1,\ \ldots\ \text{array}(0..\,i_n-1,\,t)\ldots))$$