

Lecture 7

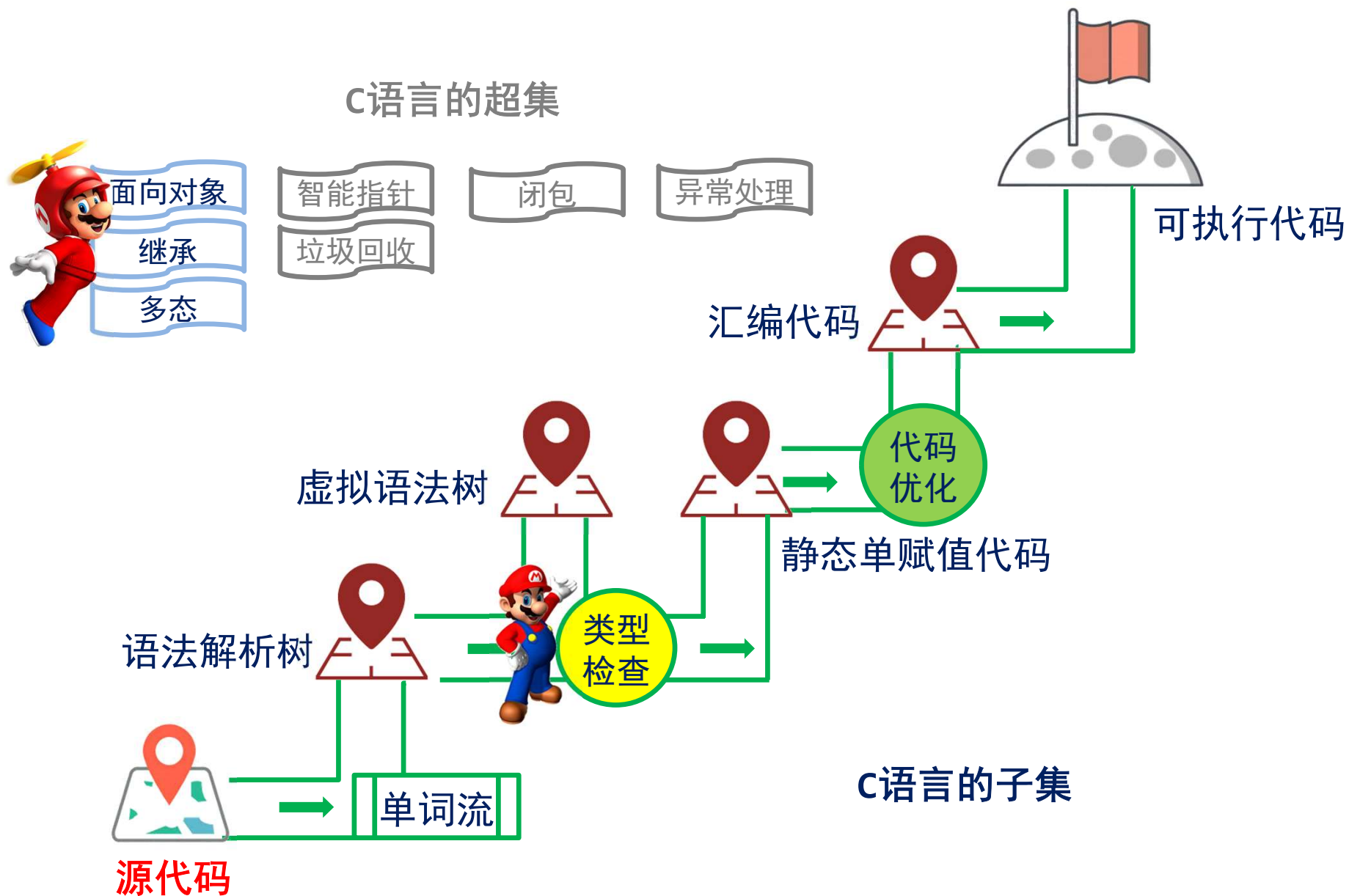
类型系统

徐 辉

xuh@fudan.edu.cn



学习地图



大纲

- 一、基本概念
- 二、类型检查
- 三、类型推断
- 四、子类型和泛型
- 五、动态类型

一、基本概念

未知程序

- 并非所有可解析的代码都是well defined。
 - 除数为0?
 - “abc” + 123
 - 12(34)
- 类型问题是一类主要问题
- 如何解决这类问题?
 - 需要借助上下文信息
 - 语义推敲/semantic elaboration



类型系统：类型和规则

- 类型包括语言预定义类型或程序员构建的类型，分为：
 - 基础类型（语言预定义）
 - 数字：整数（有符号、无符号）、浮点数
 - 字符：ASCII、Unicode、UTF8
 - 布尔值：true/false。
 - 复合类型
 - 数组
 - 串：字符串
 - 枚举类型（enum）：值的枚举（C）？ Rust则不同
 - 并集（union）：类型的枚举（C）？

类型系统：规则集合

- 规则举例
 - 运算符&的返回值是指针，指向的值和操作数的类型相同。
 - 运算符+、-、*的返回值与运算数相同
- 如何判断类型是否等价？
 - 名字相同
 - 结构相同
 - MyString vs String

```
struct MyString {  
    char* val;  
    len n;  
}  
struct String {  
    char* val;  
    len n;  
}
```

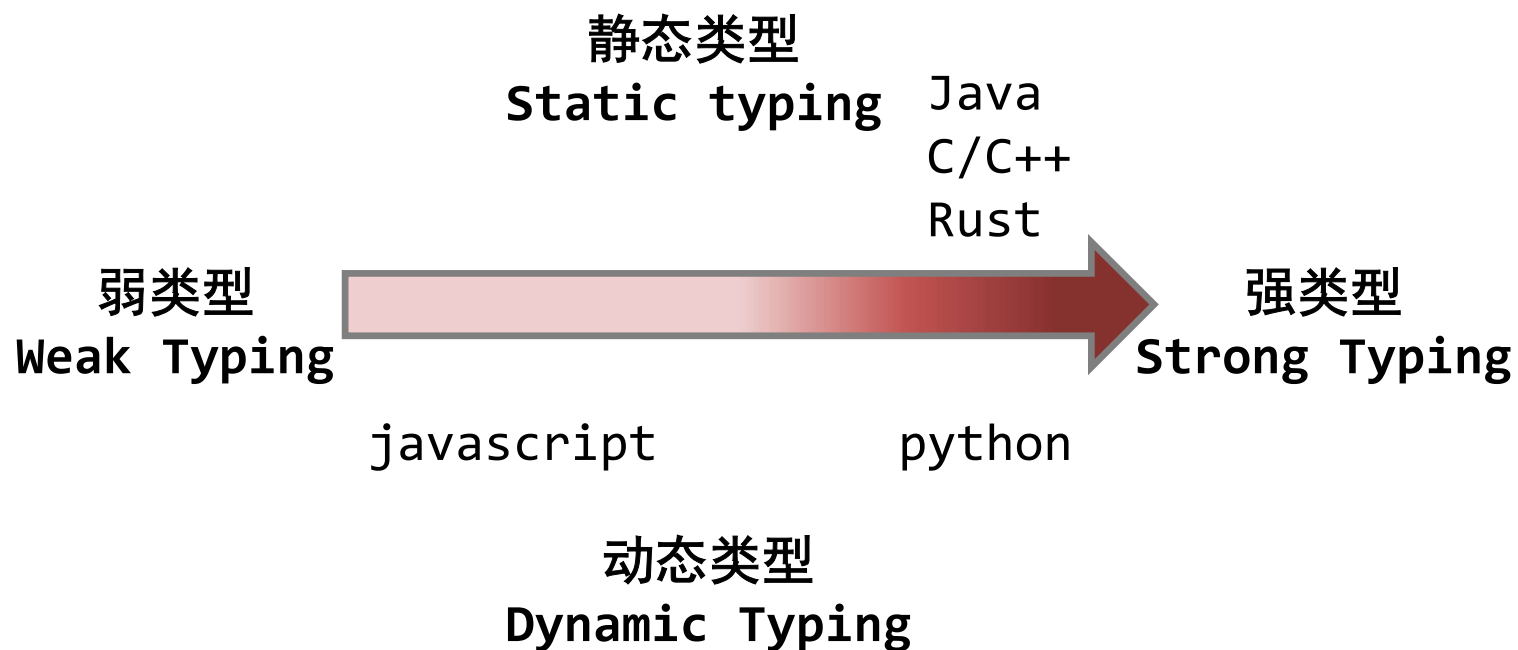
类型系统的目标

- 安全性（type soundness）：一个well-typed程序在运行时不会遇到未定义操作undefined operation。
 - 类型错误：float=>int
 - 内存越界
- 表达能力：不因严格的类型要求增加语言的使用难度
 - 加入上下文相关的特性，如运算符重载
 - 隐式转换：编译器按需插入类型转换操作

如何实现安全性？

- 类型检查（type checking）：分析每一个运算的参数类型是否与运算符要求一致。
 - 需要显示声明变量类型。
- 类型推断（type inference）：为代码中的每个标识符和表达式确定类型。
 - 无需显示声明变量类型。
 - 可类型（typeability）：类型推断是有解？

类型系统分类



动态类型的例子

- 静态类型系统：编译阶段检查类型的一致性，避免运行时错误，一般偏向强类型。
- 动态类型系统：一般不显示定义变量类型，在运行时检查类型的一致性。

```
//python代码，foo的类型是什么？  
def foo(x):  
    if x == 1:  
        return "bingo!"  
    return x  
  
print(foo(10))  
print(foo(1))  
print(foo(10) + foo(1))
```

```
#: python factorial.py  
10  
bingo!  
Traceback (most recent call last):  
  File "factorial.py", line 11, in <module>  
    print(foo(10) + foo(1))  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

弱类型语言

- 类型会发生隐式转换，可能会造成意想不到的错误
- 代表语言：JavaScript
 - Javascript Equality Game: <https://eqeq.js.org/>

```
1 + '2';  
1 + true;  
'1' + true;
```

*ToString(number) + string
number + ToNumber(boolean)
string + ToString(boolean)*

```
var a = 42;  
var b = "42";  
var c = [42];  
a === b;  
a == b;  
a == c;
```

*false
true
true*

```
if (a == 1 && a == 2) {  
  alert('hello world!');  
}
```

```
var i = 1;  
Number.prototype.valueOf = function() {  
  return i++;  
};  
  
var a = new Number(1);
```

强类型语言

- 变量必须先定义后使用，不允许隐式类型转换。
- 代表语言：Python、Java
 - C/C++?

//python代码

```
a = 1 + '2';  
b = 1 + True;  
c = '1' + True;
```

TypeError

2

TypeError

//C代码

```
int a = 1 + '2';  
int b = 1 + "2";  
int c = 1 + true;  
int d = '1' + true;  
int e = "1" + true;
```

51

4202501

2

50

4202503

二、类型检查

类型检查问题

- 已知类型系统中运算符的类型定义或函数签名；
- 分析当前语句中的变量、常量是否满足类型约束；
- 变量必须先声明后使用。

运算符类型定义

```
+ = (int, int) → int  
    | float + float → float  
    | float + int → float
```

```
int a = 1 + 2;  
int b = c + d;
```

函数类型定义

```
Tf = (T1, T2) → TR
```

```
T1 p1;
```

```
T2 p2;
```

```
TR r = f(p1, p2,);
```

类型检查的主要思路

- 提取AST上每一个运算符的类型定义或函数签名；
- 提取每一个参数标识符的类型；
 - 考虑作用域
- 检查参数类型是否满足类型约束。
- 如果不满足？
 - 隐式转换
 - 报错

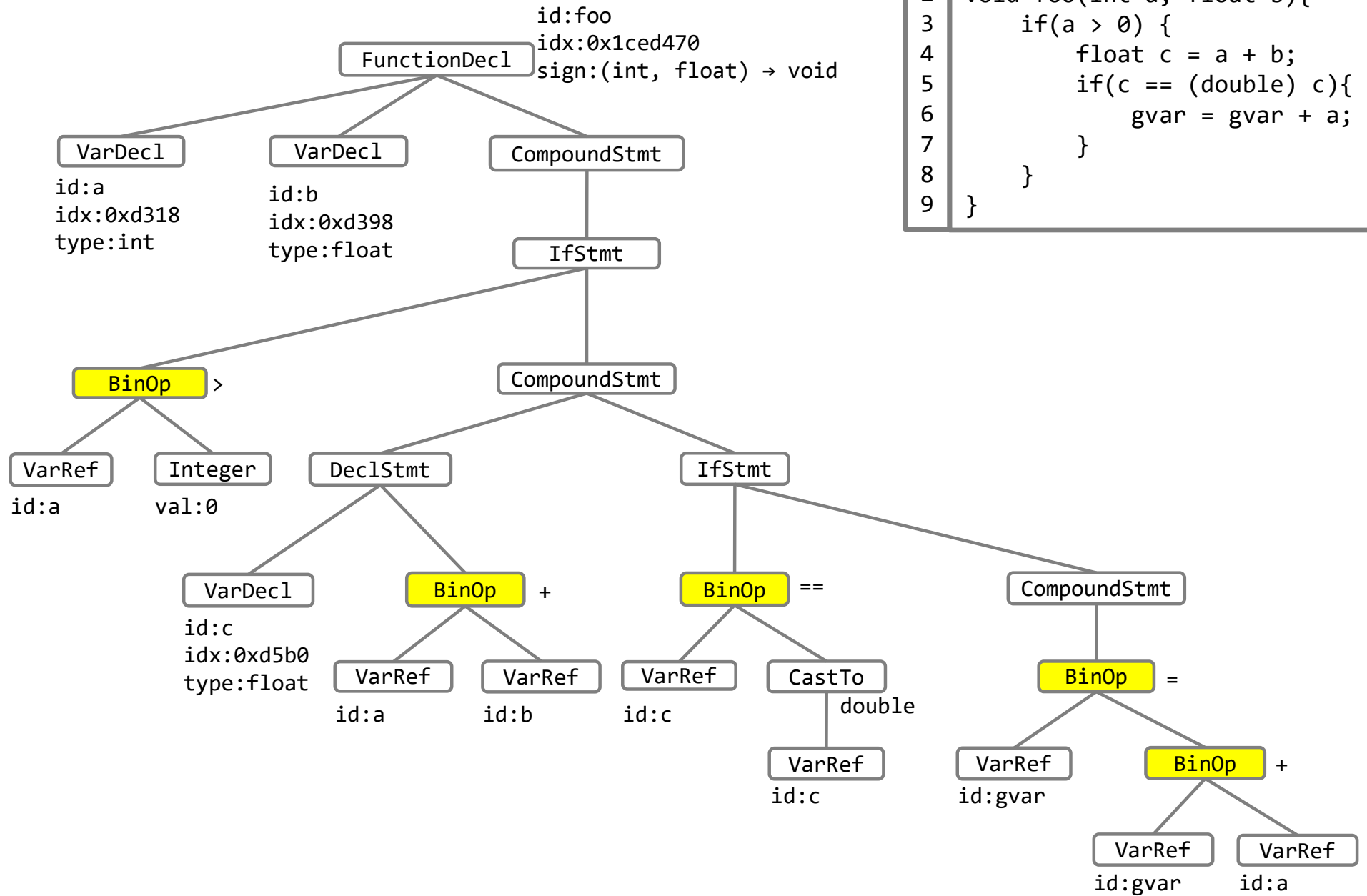
标识符类型

- 如何提取代码标识符类型?
 - 类型
 - 作用域
 - 其它属性

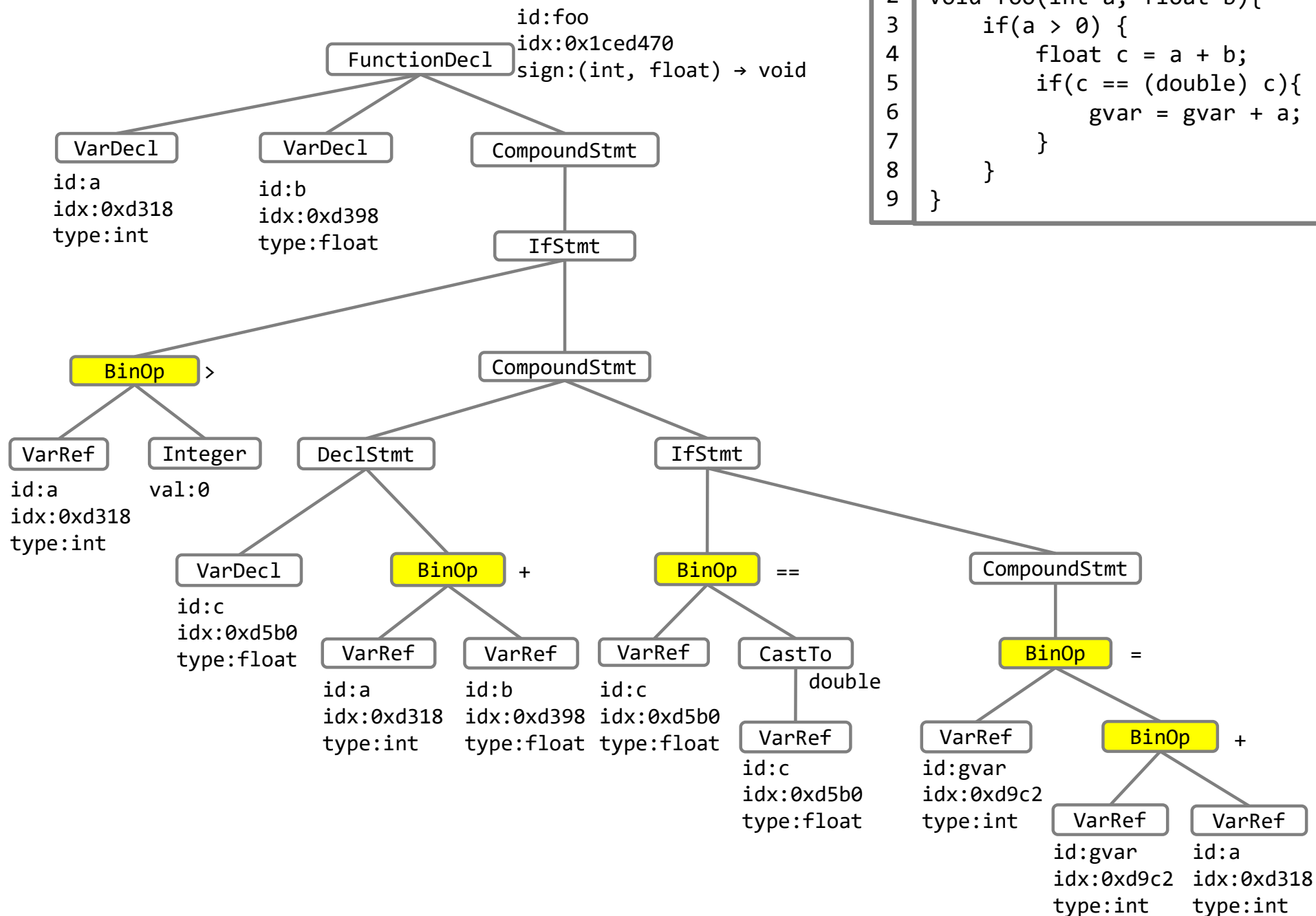
```
1 int gvar = 0;
2 void foo(int a, float b){
3     if(a > 0) {
4         float c = a + b;
5         if(c == (double) c){
6             gvar = gvar + a;
7         }
8     }
9 }
```

标识符	索引	类型	作用域
gvar	0xd9c2	int	global
foo	0xd470	(int,float) → void	global
a	0xd318	int	foo
b	0xd398	float	foo
c	0xd5b0	float	foo:line4-7

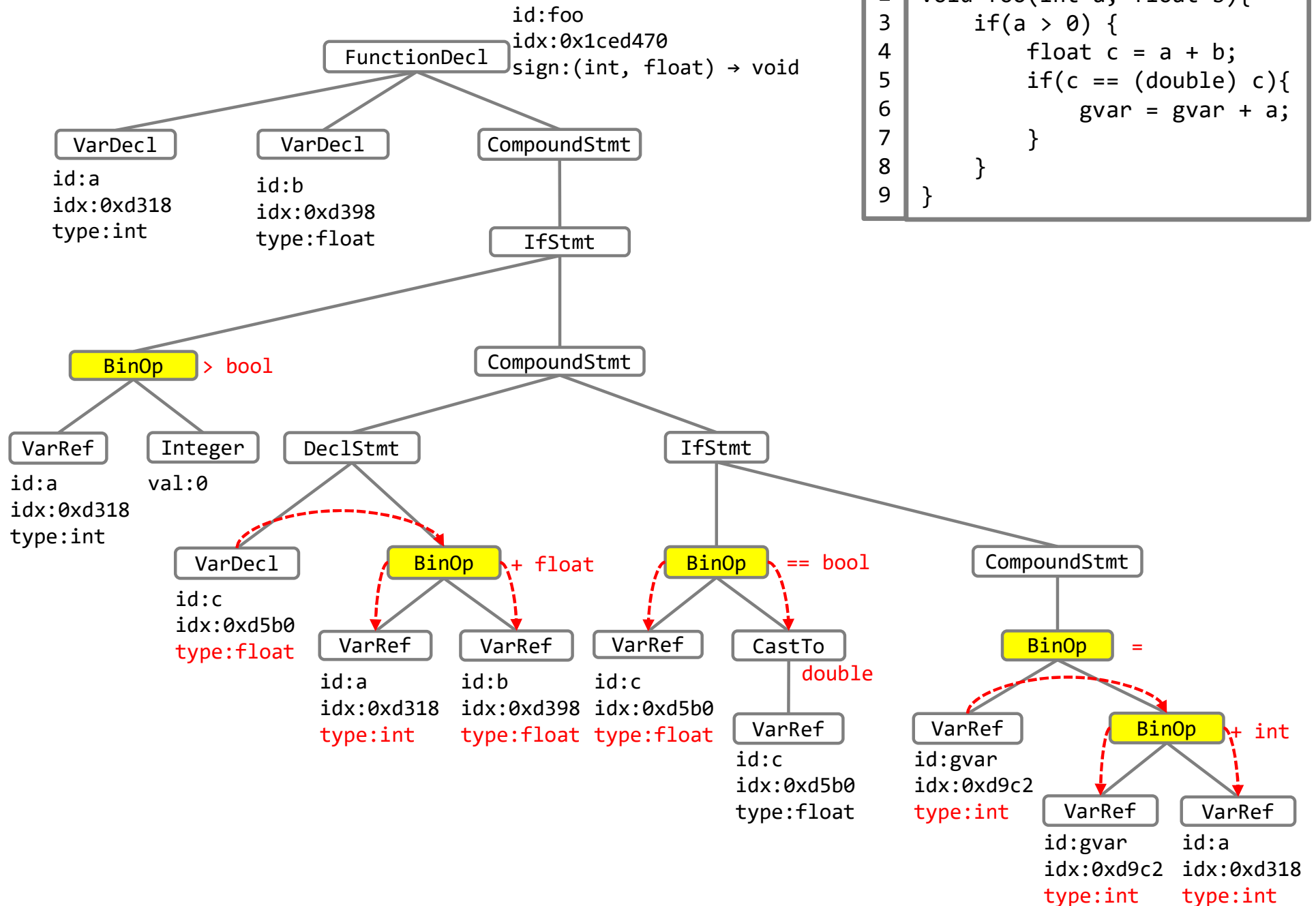
Raw AST



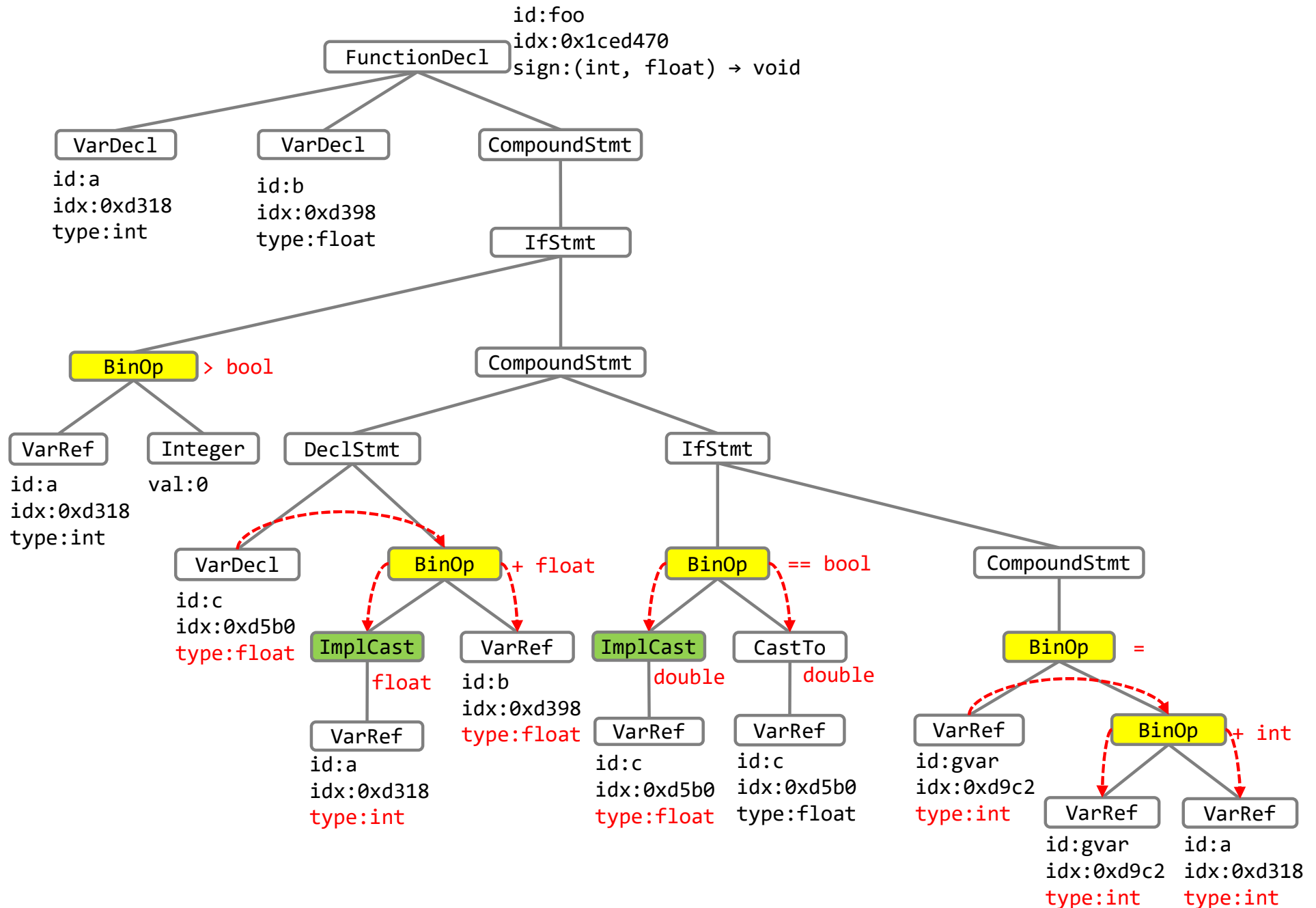
标识符解析



类型检查



不匹配的如何处理？隐式转换？



小心疏漏！

- 可能会有False Negatives?
- 比如Java的Array是Covariant Type。
 - 如果X是Y的子类型，那么X[]也是Y[]的子类型。
- Type soundness vs Representativeness

```
Integer[] myInts = {1,2,3,4};  
Number[] myNumber = myInts;  
myNumber[0] = 3.14;
```

ArrayStoreException
(动态类型检查)

思考：如何实现Variadic function?

- 可变参数的函数？如C语言的printf等函数

```
//c语言程序  
int printf(const char *format,...);
```

```
//c语言程序  
int sum(int num,...)  
{  
    va_list ap;  
    int sum = 0;  
    va_start(ap,num);  
    for(int i=0; i<num; i++){  
        sum += va_arg(ap,int);  
    }  
    va_end(ap);  
    return sum;  
}
```

三、类型推断

Damas-Hindley-Milner算法

类型推断的典型场景

- 缺省变量类型定义的语言，如
 - Scheme等函数式编程语言
 - Python（动态类型）

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Scheme程序样例：

- n的类型？
- factorial(n)的类型？

```
def factorial(n):
    if n == 1:
        return n
    else:
        return n*factorial(n-1)
```

Python程序样例：

- n的类型？
- factorial(n)的类型？

其它场景

- C++11的auto、Rust的let
- C++ template、Java Interface、Rust泛型

//c++代码

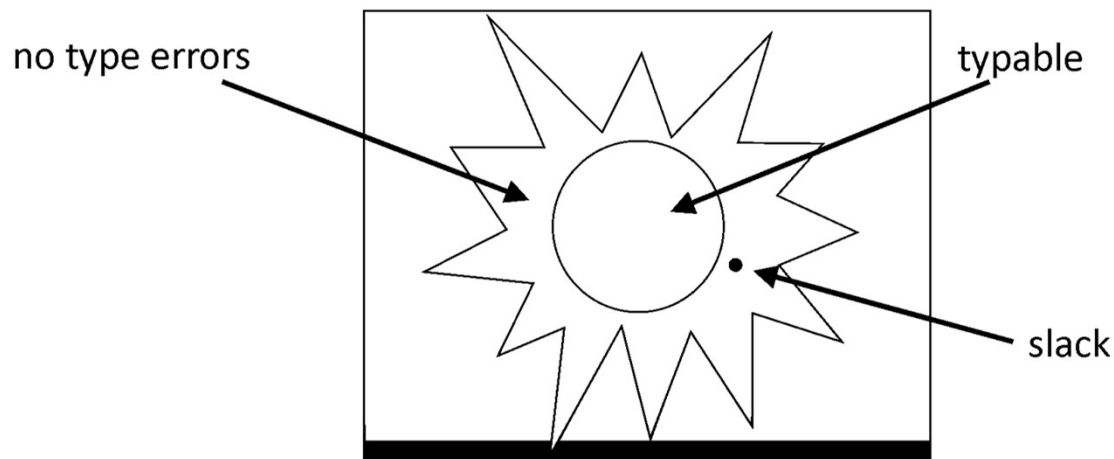
```
auto a = 1 + 2;  
auto b = add(1, 1.2);  
auto d = {1, 2};  
auto (*p)() -> int;  
auto lambda = [](int x) { return x + 3; };
```

//Rust代码

```
let i1 = 10u8;  
let i2 = 1024;  
let f = 5.;  
let mut vec = Vec::new();  
vec.push(i1);  
vec.push(i2);  
vec.push(f); //Error
```

类型推断

- Damas-Hindley-Milner类型推断方法
 - 基于约束求解的方法；
 - ML、Haskell、Ocaml等语言中使用
- 使用保守的推断策略
 - 根据虚拟语法树获得类型约束；
 - 如果可类型，则不应出现运行时错误；
 - 有些程序可能被错误拒绝 (slack/false positive)



假设存在以下类型系统

- 基础的值类型：

$\tau \rightarrow \text{int}$	整数
$\quad \quad \& \tau$	指针
$\quad \quad (\tau, \dots, \tau) \rightarrow \tau$	函数

- 基于基础类型可以推导出复合类型，比如：

$(\text{int}, \&\text{int}) \rightarrow \&\&\text{int}$

类型约束

- 基于AST生成类型约束：
 - 约束一般都为等价关系
 - 通过合一算法（unification algorithm）求解
- 类型变量：
 - 变量X的类型变量用 $[[X]]$ 表示
 - 所有的变量标识符都是唯一的
 - 表达式E（非标识符的）的类型变量用 $[[E]]$ 表示
 - E为一个AST节点

类型约束生成规则举例（1/2）

程序语句（AST节点）

I // 常量

E1 op E2

E1 == E2

input

X = E

Output E

if(E){S}

if(E){S1}else{S2}

while(E){S}

类型约束

$\llbracket I \rrbracket = \text{int}$

$\llbracket E1 \rrbracket = \llbracket E2 \rrbracket = \llbracket E1 \text{ op } E2 \rrbracket = \text{int}$

$\llbracket E1 \rrbracket = \llbracket E2 \rrbracket = \llbracket E1 == E2 \rrbracket = \text{int}$

$\llbracket \text{input} \rrbracket = \text{int}$

$\llbracket X \rrbracket = \llbracket E \rrbracket$

$\llbracket E \rrbracket = \text{int}$

$\llbracket E \rrbracket = \text{int}$

$\llbracket E \rrbracket = \text{int}$

$\llbracket E \rrbracket = \text{int}$

类型约束生成规则举例（2/2）

程序语句

```
X(X1, ..., Xn){  
    ...  
    return E;  
}  
E(E1, ..., En)  
alloc E  
&X  
*E  
*X=E
```

类型约束

$$\llbracket X \rrbracket = (\llbracket X1 \rrbracket \dots \llbracket Xn \rrbracket) \rightarrow \llbracket E \rrbracket$$
$$\llbracket E \rrbracket = (\llbracket E1 \rrbracket \dots \llbracket En \rrbracket) \rightarrow \llbracket E(E1 \dots En) \rrbracket$$
$$\llbracket \text{alloc } E \rrbracket = \&\llbracket E \rrbracket$$
$$\llbracket \&X \rrbracket = \&\llbracket X \rrbracket$$
$$\llbracket E \rrbracket = \&\llbracket *E \rrbracket$$
$$\llbracket X \rrbracket = \&\llbracket E \rrbracket$$

举例

```
main(){  
    var x, y, z;  
    x = input;  
    y = alloc x;  
    *y = x;  
    z = *y;  
    return z;  
}
```

$\llbracket \text{main} \rrbracket = () \rightarrow \llbracket z \rrbracket$

$\llbracket x \rrbracket = \llbracket \text{input} \rrbracket = \text{int}$

$\llbracket y \rrbracket = \llbracket \text{alloc } x \rrbracket = \&\llbracket x \rrbracket$

$\llbracket y \rrbracket = \&\llbracket x \rrbracket$

$\llbracket z \rrbracket = \llbracket *y \rrbracket, \llbracket y \rrbracket = \&\llbracket *y \rrbracket$

解约束:

$\llbracket x \rrbracket = \text{int}$

$\llbracket y \rrbracket = \&\text{int}$

$\llbracket z \rrbracket = \text{int}$

$\llbracket \text{main} \rrbracket = () \rightarrow \text{int}$

练习

推导变量x、y、z的类型

```
main(){  
    var x, y, z;  
    x = alloc 1;  
    y = alloc x;  
    *z = x;  
    return z;  
}
```

$\llbracket \text{main} \rrbracket = () \rightarrow \llbracket z \rrbracket$

$\llbracket x \rrbracket = \llbracket \text{alloc } 1 \rrbracket = \&\llbracket 1 \rrbracket$

$\llbracket y \rrbracket = \llbracket \text{alloc } x \rrbracket = \&\llbracket x \rrbracket$

$\llbracket z \rrbracket = \&\llbracket x \rrbracket$

解约束:

$\llbracket x \rrbracket = \&\text{int}$

$\llbracket y \rrbracket = \&\&\text{int}$

$\llbracket z \rrbracket = \&\&\text{int}$

$\llbracket \text{main} \rrbracket = () \rightarrow \&\&\text{int}$

Slack: 无法求解?

- 由于Flow-sensitivity导致无解

```
foo(x) {  
  var x;  
  x = alloc 10;  
  x = 1  
  return x+1;  
}
```

$\llbracket \text{foo} \rrbracket = (\llbracket x \rrbracket) \rightarrow \llbracket x+1 \rrbracket$

$\llbracket x \rrbracket = \llbracket \text{alloc } 10 \rrbracket = \&\llbracket 10 \rrbracket$

$\llbracket x \rrbracket = \llbracket 1 \rrbracket$

$\llbracket x \rrbracket = \llbracket 1 \rrbracket = \llbracket x+1 \rrbracket$

$\llbracket x \rrbracket = \&\text{int}$

$\llbracket x \rrbracket = \text{int}$

如果存在多个可行解？

- 不支持？
- 或取较安全的类型？

```
foo(x) {  
    return *x;  
}
```

$$\llbracket \text{foo} \rrbracket = (\llbracket x \rrbracket) \rightarrow \llbracket *x \rrbracket$$
$$\llbracket x \rrbracket = \&\llbracket *x \rrbracket$$

解约束：

$$\begin{array}{l} \llbracket x \rrbracket = \text{int} \\ \quad | \quad \&\text{int} \\ \quad | \quad \&\&\text{int} \end{array}$$

```
foo(x,y) {  
    *x = y;  
}
```

$$\llbracket x \rrbracket = \&\llbracket y \rrbracket$$

解约束：

$$\begin{array}{l} \llbracket x \rrbracket = \&\text{int} \\ \llbracket y \rrbracket = \text{int} \end{array}$$

递归问题（可解）

```
//Scheme代码
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

约束:

$[[\text{factorial}]] = ([n]) \rightarrow [1]$

$[[\text{factorial}]] = ([n]) \rightarrow [* \ n \ (\text{factorial} \ (- \ n \ 1))]$

$[n] = [0] = [(= \ n \ 0)]$

$[n] = [(\text{factorial} \ (- \ n \ 1))] = [* \ n \ (\text{factorial} \ (- \ n \ 1))]$

$[n] = [1] = [(- \ n \ 1)]$

解约束:

$[n] = \text{int}$

$[[\text{factorial}]] = (\text{int}) \rightarrow \text{int}$

递归问题（不可解）

```
factorial(p, x) {  
  if (p == 0)  
    return 1;  
  else  
    return p * x(p-1, x);  
}  
factorial(10, factorial);
```

$\llbracket \text{factorial} \rrbracket = (\llbracket p \rrbracket, \llbracket x \rrbracket) \rightarrow \llbracket 1 \rrbracket = (\llbracket p \rrbracket, \llbracket x \rrbracket) \rightarrow \llbracket p * x(p-1, x) \rrbracket$

$\llbracket p \rrbracket = \llbracket 0 \rrbracket = \llbracket p == 0 \rrbracket = \text{int}$

$\llbracket p \rrbracket = \llbracket x(p-1, x) \rrbracket = \llbracket p * x(p-1, x) \rrbracket = \text{int}$

$\llbracket x \rrbracket = (\llbracket p-1 \rrbracket, \llbracket x \rrbracket) \rightarrow \llbracket x(p-1, x) \rrbracket$

$\llbracket p \rrbracket = \llbracket 1 \rrbracket = \llbracket p-1 \rrbracket = \text{int}$

$\llbracket \text{factorial} \rrbracket = (\llbracket 10 \rrbracket, \llbracket \text{factorial} \rrbracket) \rightarrow \llbracket \text{factorial}(10, \text{factorial}) \rrbracket$

使用 ϕ 来标记Regular Type

$\llbracket \text{factorial} \rrbracket = \phi = (\text{int}, \phi) \rightarrow \text{int}$

结构体递归定义

```
//Java代码
class List<T> {
    T value;
    List<T> next;
}
```

```
//Haskell代码
data List a = Nil
            | Cons a (List a)
```

使用 ϕ 来标记Regular Type
[[List<T>]] = ϕ = (T, ϕ)

```
//C代码
struct List{
    int data;
    struct List next;
};
```




```
//C代码
struct List{
    int data;
    struct List *next;
};
```




Rust中的递归结构

- 需要将递归结构放入Box中，为什么？
 - Box将数据放在堆上

```
//Rust代码
struct MyList{
    val: u32,
    next: Option<MyList>,
}
```



```
//Rust代码
struct MyList{
    val: u32,
    next: Option<Box<MyList>>,
}
```



```
[_] impl<T> Box<T, Global>
```

```
[_] pub fn new(x: T) -> Box<T, Global> ⓘ
```

Allocates memory on the heap and then places x into it.


This doesn't actually allocate if T is zero-sized.

Examples


```
let five = Box::new(5);
```

更多例子：C++中的递归函数？

```
//C++代码
auto factorial(int i) {
    if(i == 1)
        return i;
    else
        return factorial(i-1)*i;
}
```



```
//C++代码
auto factorial(int i) {
    return (i == 1) ? i : factorial(i-1)*i;
}
```



```
#: clang++ autofunc.cpp
autofunc.cpp:12:25: error: function 'factorial' with deduced return
type cannot be used before it is defined
    return (i == 1) ? i : factorial(i-1)*i;
```


四、子类型和泛型

子类型

- 类型之间存在偏序关系，如 $X \leq Y$ 表示：
 - X 是 Y 的子类型；
 - Y 是的父类型。
- 偏序的特性：
 - 自反性： $X \leq X$ ；
 - 传递性： $X \leq Y, Y \leq Z \Rightarrow X \leq Z$ ；
- 当类型约束为父类型时，可用子类型的对象；
- 子类型的数据结构可兼容父类型。

子类型的应用

- 泛型编程：Genetic Programming
 - 将参数类型设为通用类型，编译时确定具体类型。
 - 如C++ template、Rust Generic
- 继承关系
 - C++类的继承
 - Rust Trait的继承（复用）

如何编写可实现下列功能API?

- 比较两个参数的大小，并返回其中较大的一个。
 - 支持int、float、double、char等参数类型
 - 支持自定义参数类型

//C++代码

```
int max(int x, int y) {  
    return (x > y) ? x : y;  
}  
  
double max(double x, double y) {  
    return (x > y) ? x : y;  
}  
  
char max(char x, char y) {  
    return (x > y) ? x : y;  
}  
  
max(3, 7);  
max(3.0, 7.0);  
max('g', 'e');
```



//C++代码

```
template <typename T>  
T max(T x, T y) {  
    return (x > y) ? x : y;  
}  
  
max(3, 7);  
max(3.0, 7.0);  
max('g', 'e');
```



//C++代码

```
template <typename T, typename G>  
auto max(T x, G y) {  
    return (x > y) ? x : y;  
}  
  
max(3, 'g');  
max(3, 1.5);  
max(3.0, 'g');
```

泛型的实现

- 编译阶段推导确定具体类型；
- 也可以通过属性指定泛型的具体类型；

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
max(3, 7);
max(3.0, 7.0);
max('g', 'e');
```

$\llbracket \text{max} \rrbracket = (T, T) \rightarrow T$

$T = \text{int}$

$\llbracket \text{max} \rrbracket = (\text{int}, \text{int}) \rightarrow \text{int}$

```
template <typename T, typename G>
auto max(T x, G y) {
    return (x > y) ? x : y;
}
```

















```
max<int, char>(123, 'g');
max(3, 1.5);
max(3.0, 'g');
```

?

汇编代码

```
template <typename T, typename G>
auto max(T x, G y) {
    return (x > y) ? x : y;
}
```

```
max(3, 7);
max(3.0, 7.0);
max(3, 'g');
max(3, 7.0);
max(3.0, 'g');
max('g', 3);
max(7.0, 3);
max('g', 3.0);
```

main	.text
 _static_initialization_and_destruction_0(int,int)	.text
 _GLOBAL__sub_I_main	.text
 std::max<int>(int const&,int const&)	.text
 std::max<double>(double const&,double const&)	.text
 _Z3maxlicEDaT_T0_	.text
 _Z3maxlidEDaT_T0_	.text
 _Z3maxldcEDaT_T0_	.text
 _Z3maxlciEDaT_T0_	.text
 _Z3maxldiEDaT_T0_	.text
 _Z3maxlcdEDaT_T0_	.text
 _libc_csu_init	.text
 _libc_csu_fini	.text
 _term_proc	.fini
 _cxa_atexit	extern
 _stack_chk_fail	extern
 std::ios_base::Init::Init(void)	extern

C语言有泛型吗？

- 基于void可以实现类似的功能。
- C语言的运算符也支持多种参数
 - 编译器自身支持的功能；
 - 不支持运算符重载。

```
int a = 1 + 2;  
int = 'a' + 'b';
```

```
void *max(void *x, void *y, int* (*f)(void *, void *)) {  
    if (f(x, y) > 0)  
        return x;  
    else  
        return y;  
}  
  
int* compare(void *x, void *y) {  
    return (* (int *) x > * (int *) y) ? 1 : 0;  
}  
  
int *a = 123;  
int *b = 234;  
int *r = (int *)max(&a, &b, compare);  
printf("max = %d\n", *r);
```

继承

- 类型关系：父类>子类

B>S, A>S

```
//C++代码  
class A { ... }  
class B { ... }  
class S : public B, public A { ... }
```

A>B>S

```
//C++代码  
class A { ... }  
class B : public A { ... }  
class S : public B { ... }
```


Upcast和Downcast

- Upcasting: 如果 $X > Y$, 将Y类型转换为X类型
 - 一般不存在风险, 默认都允许
 - Rust Trait不支持Upcast
- Downcasting: 如果 $X > Y$, 将X类型转换为Y类型
 - 类型检查, 如果类型不匹配会抛出异常

```
class Base {};  
class Derived : public Base {};  
  
int main(int argc, const char** argv) {  
    Base* base = new Base();  
    if(Derived* derived = dynamic_cast<Derived *>(base)){  
        ...  
    }  
}
```

Rust使用Trait作为泛型的类型约束

```
trait Countable{ fn getcount(&self) -> u32; }
struct MyList{ val:u32, next:Option<Box<MyList>>, }

impl Countable for MyList {
    fn getcount(&self) -> u32 {
        let mut r = self.val;
        let mut cur = &self.next;
        loop {
            match cur {
                Some(x) => { r = r+x.val; cur = &x.next}
                _ => {break;}
            }
        }
        return r;
    }
}

fn foo<T:Countable>(t: T) { println!("Count: {:?}", t. getcount()); }

fn main() {
    let l = MyList{val:1, next:Some(Box::new(MyList{val:2, next:None}))};
    foo(l);
}
```

Rust支持继承吗？

- Trait的功能可以在struct中复用；

- “impl A for S” => S>A?

- 但trait不是类型。


```
struct S { }  
trait A { }  
impl A for S { }
```

- Trait的代码可以被其它Trait复用；

- impl<T> B for T where T:A { } => A < B ?

- 有点反直觉


```
trait A { }  
trait B { }  
struct S { }  
struct T { }  
  
impl A for S { }  
impl<T> B for T where T:A { }  
fn makeacall<T:B>(s: &T){ }  
  
fn main() {  
    let a = S {};  
    makeacall(&a);  
}
```




Rust支持继承吗？

- Trait之间可以存在偏序关系；
 - “trait B:A” \Rightarrow B<A
 - 但非类型之间的偏序关系
- 基于Trait多少的subtype关系？
 - S>T?

```
struct S { }  
trait A { }  
trait B : A { }  
impl A for S { }  
impl B for S { }  
  
fn makeacall<T:A>(s: &T){ s.a() }  
  
fn main() {  
    let a = S {};  
    makeacall(&a);  
}
```



```
struct S { }  
struct T { }  
trait A { }  
trait B { }  
impl A for T { }  
impl B for T { }  
impl A for S { }  
fn makeacall(s: &S){ }  
  
fn main() {  
    let t = T {};  
    makeacall(&t);  
}
```



Rust基于生命周期的subtype

- 如果s的生命周期大于t，则s是t的subtype;

```
fn main() {  
    let s: &'static str = "hi";  
    let t: &'a str = s;  
}}
```

五、动态类型

下面这段代码输出什么？

```
class Base {
public:
    void print(){ cout << "base print" << endl;}
    virtual void speak(){ cout << "base speak" << endl;}
    virtual void shout(){ cout << "base shout" << endl;}
    virtual ~Base(){ cout << "destroying base" << endl;}
};

class Derived : public Base {
public:
    void print(){ cout << "derived print" << endl;}
    virtual void speak(){ cout << "derived speak" << endl;}
    virtual ~Derived(){ cout << "destroying derived" << endl;}
};

void test(Base* bptr){
    bptr->print();
    bptr->speak();
    bptr->shout();
}

int main(){
    Derived dobj;
    test(&dobj);
}
```

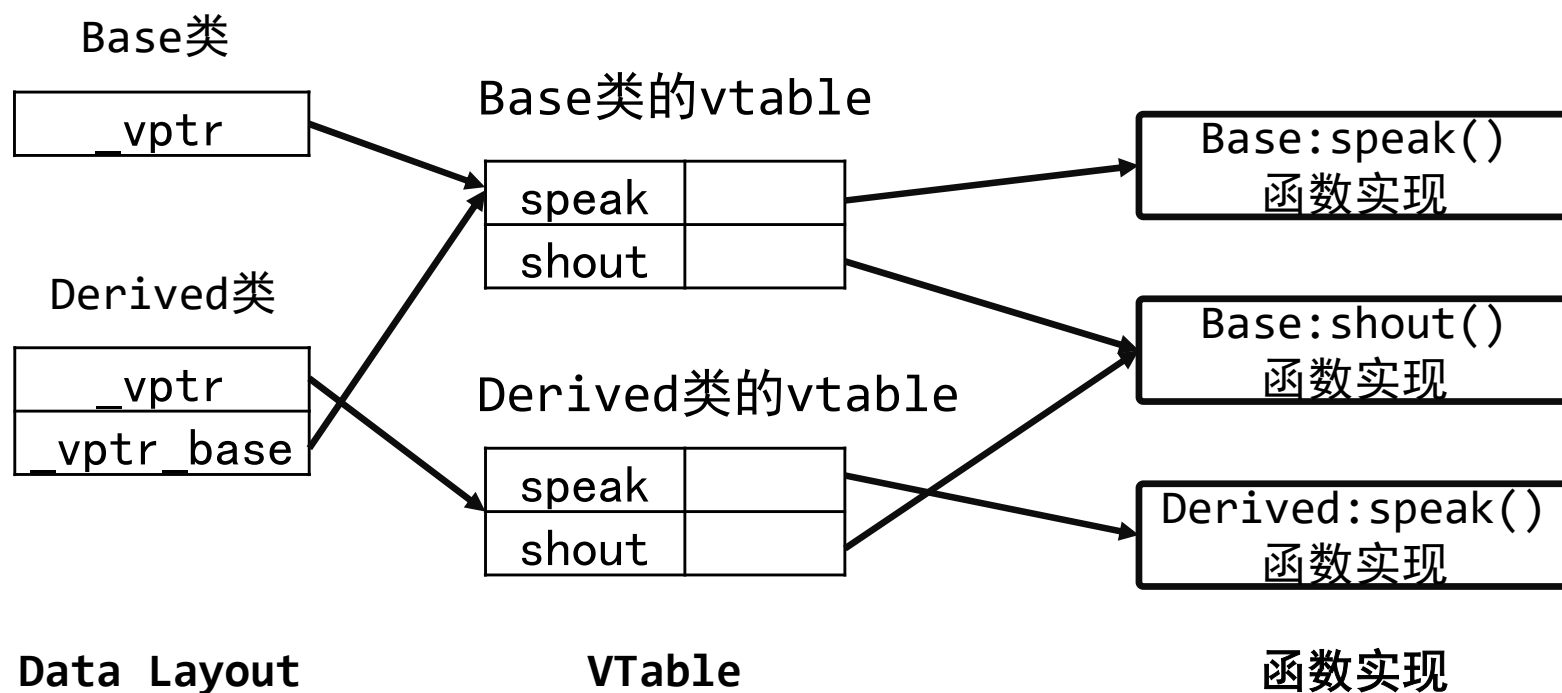
```
base print
derived speak
base shout
destroying derived
destroying base
```

虚函数和动态绑定

- 静态绑定：可在编译时确定执行版本
 - 通过对象调用任意函数
 - 调用非虚函数
- 动态绑定：直到运行时才能确定执行版本
 - C++虚函数
 - Rust `dynamic trait`

C++如何实现动态分发

- 编译器为每个类创建一个虚拟指针（vptr）指向虚拟方法表格（vtable: virtual method table）
- vtable包含每一个可用虚函数以及指向其具体函数实现的指针。



Clang++的VTable

```
clang++ -Xclang -fdump-vtable-layouts
```

Vtable for 'Base' (6 entries).

```
0 | offset_to_top (0)
1 | Base RTTI
  -- (Base, 0) vtable address --
2 | void Base::speak()
3 | void Base::shout()
4 | Base::~~Base() [complete]
5 | Base::~~Base() [deleting]
```

VTable indices for 'Base' (4 entries).

```
0 | void Base::speak()
1 | void Base::shout()
2 | Base::~~Base() [complete]
3 | Base::~~Base() [deleting]
```

Vtable for 'Derived' (6 entries).

```
0 | offset_to_top (0)
1 | Derived RTTI
  -- (Base, 0) vtable address --
  -- (Derived, 0) vtable address --
2 | void Derived::speak()
3 | void Base::shout()
4 | Derived::~~Derived() [complete]
5 | Derived::~~Derived() [deleting]
```

VTable indices for 'Derived' (3 entries).

```
0 | void Derived::speak()
2 | Derived::~~Derived() [complete]
3 | Derived::~~Derived() [deleting]
```

- RTTI(run-time type identification)

LLVM IR + Assembly Code

```
%class.Derived = type { %class.Base }
%class.Base = type { i32 (...)** }

%1 = alloca %class.Derived, align 8
%2 = alloca %class.Base*, align 8
%3 = alloca i8*
%4 = alloca i32
call void @_ZN7DerivedC2Ev(%class.Derived* %1) #3
%5 = bitcast %class.Derived* %1 to %class.Base*
store %class.Base* %5, %class.Base** %2, align 8
invoke void @_ZN7Derived5printEv(%class.Derived* %1)
    to label %6 unwind label %21

6:                                     ; preds = %0
%7 = load %class.Base*, %class.Base** %2, align 8
invoke void @_ZN4Base5printEv(%class.Base* %7)
    to label %8 unwind label %21

8:                                     ; preds = %6
%9 = load %class.Base*, %class.Base** %2, align 8
%10 = bitcast %class.Base* %9 to void (%class.Base*)***
%11 = load void (%class.Base*)**, void (%class.Base*)*** %10, align 8
%12 = getelementptr inbounds void (%class.Base*)*,
    void (%class.Base*)** %11, i64 0
%13 = load void (%class.Base*)*, void (%class.Base*)** %12, align 8
invoke void %13(%class.Base* %9)
    to label %14 unwind label %21

14:                                    ; preds = %8
%15 = load %class.Base*, %class.Base** %2, align 8
%16 = bitcast %class.Base* %15 to void (%class.Base*)***
%17 = load void (%class.Base*)**, void (%class.Base*)*** %16, align 8
%18 = getelementptr inbounds void (%class.Base*)*,
    void (%class.Base*)** %17, i64 1
%19 = load void (%class.Base*)*, void (%class.Base*)** %18, align 8
invoke void %19(%class.Base* %15)
    to label %20 unwind label %21
```

```
# %bb.0:pushq    %rbp
        movq     %rsp, %rbp
        subq     $48, %rsp
        leaq     -8(%rbp), %rax
        movq     %rax, %rdi
        movq     %rax, -40(%rbp)
        callq    @_ZN7DerivedC2Ev
        movq     -40(%rbp), %rax
        movq     %rax, -16(%rbp)
.Ltmp0: movq     %rax, %rdi
        callq    @_ZN7Derived5printEv
.Ltmp1: jmp      .LBB1_1
.LBB1_1:movq     -16(%rbp), %rdi
.Ltmp2: callq    @_ZN4Base5printEv
.Ltmp3: jmp      .LBB1_2
.LBB1_2:movq     -16(%rbp), %rax
        movq     (%rax), %rcx
        movq     (%rcx), %rcx
.Ltmp4: movq     %rax, %rdi
        callq    %rcx
.Ltmp5: jmp      .LBB1_3
.LBB1_3:movq     -16(%rbp), %rax
        movq     (%rax), %rcx
        movq     8(%rcx), %rcx
.Ltmp6: movq     %rax, %rdi
        callq    %rcx
.Ltmp7: jmp      .LBB1_4
.LBB1_4:leaq     -8(%rbp), %rdi
        callq    @_ZN7DerivedD2Ev
        xorl     %eax, %eax
        addq     $48, %rsp
        popq     %rbp
        retq
```

Rust Dyn Trait

- dyn Trait表示任意实现了trait的类型
 - 类似T:Base
 - 但当成动态类型编译，使用vtable寻址

```
trait A {  
    fn a(&self) { println!("super a"); }  
}  
trait B : A{  
    fn b(&self) { println!("sub b"); }  
}  
struct S { }  
  
impl A for S { }  
impl B for S { }  
  
//fn makeacall1<T:A>(s: &T){ s.a() }  
fn makeacall2(s: &dyn A){ s.a() }  
  
fn main() {  
    let s = S {};  
    makeacall2(&s);  
}
```

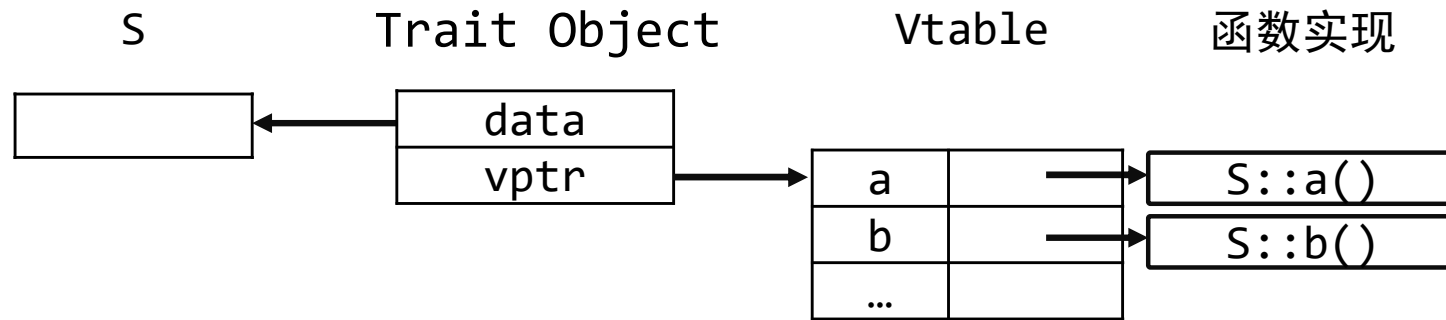
```
; subtype::makeacall1::hb802f2baed532665  
_ZN7subtype10makeacall117hb802f2baed532665E proc n  
; __unwind {  
push    rax  
call    _ZN7subtype4Base4base17h9f72e927683f8486E  
pop     rax  
retn  
; } // starts at 5450  
_ZN7subtype10makeacall117hb802f2baed532665E endp
```

```
; subtype::makeacall2::h6da6d010eef52869  
_ZN7subtype10makeacall1217h6da6d010eef52869E proc  
; __unwind {  
push    rax  
call    qword ptr [rsi+18h]  
pop     rax  
retn  
; } // starts at 5460  
_ZN7subtype10makeacall1217h6da6d010eef52869E endp
```



Dyn Trait vs 虚函数

- Dyn Trait 只有一个 vtable, 不支持 upcast



参考资料

- 《编译器设计（第2版）》
 - 第4章：上下文相关分析
- 《Static Program Analysis》，Anders Møller等著
 - 第3章 Type Analysis