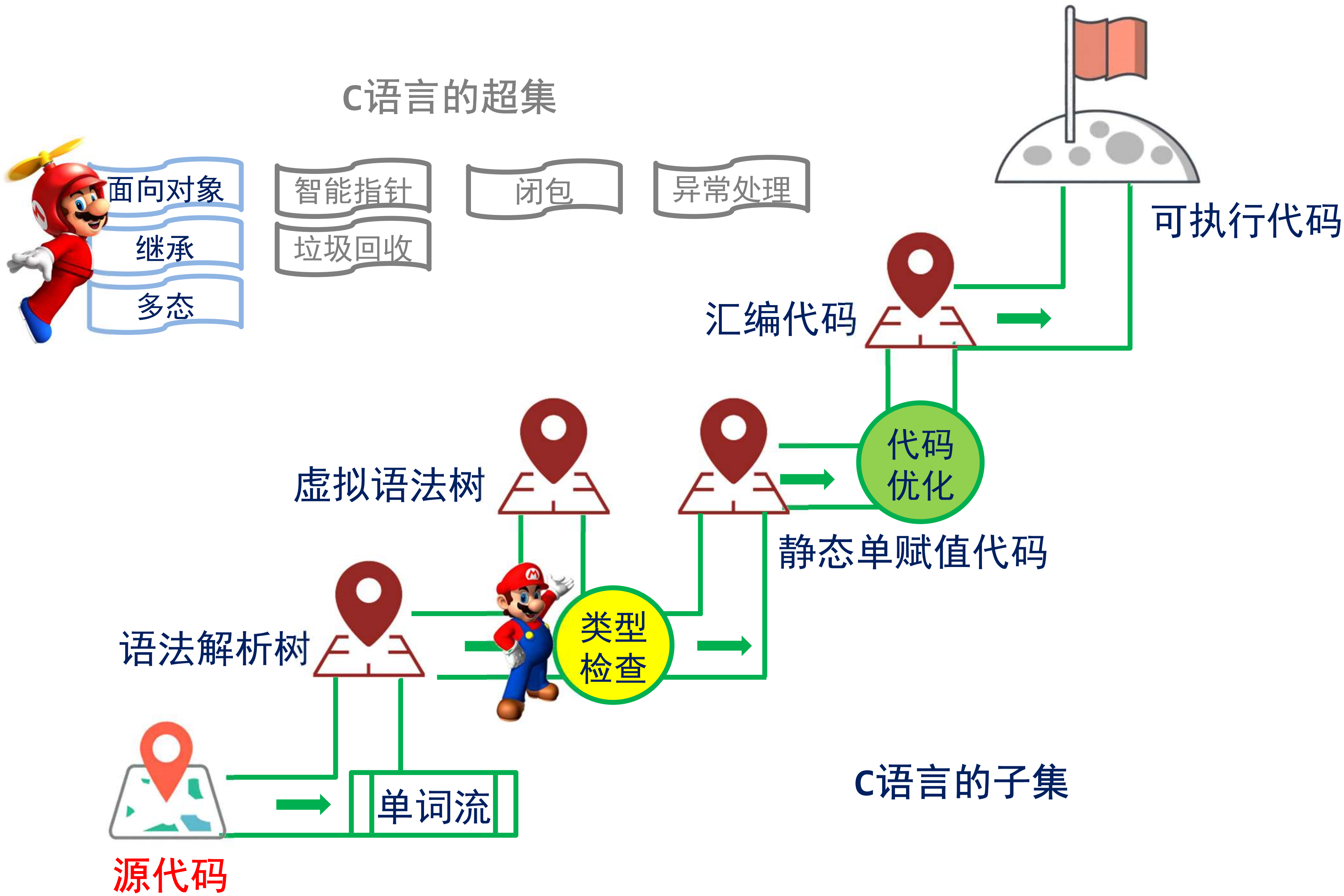
Lecture 7

类型系统

徐 辉

xuh@fudan.edu.cn

学习地图


C语言的超集
面向对象
继承
多态
智能指针
垃圾回收
闭包
异常处理
可执行代码
汇编代码
代码优化
虚拟语法树
静态单赋值代码
类型检查
语法解析树
单词流
源代码
C语言的子集

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;
}
```

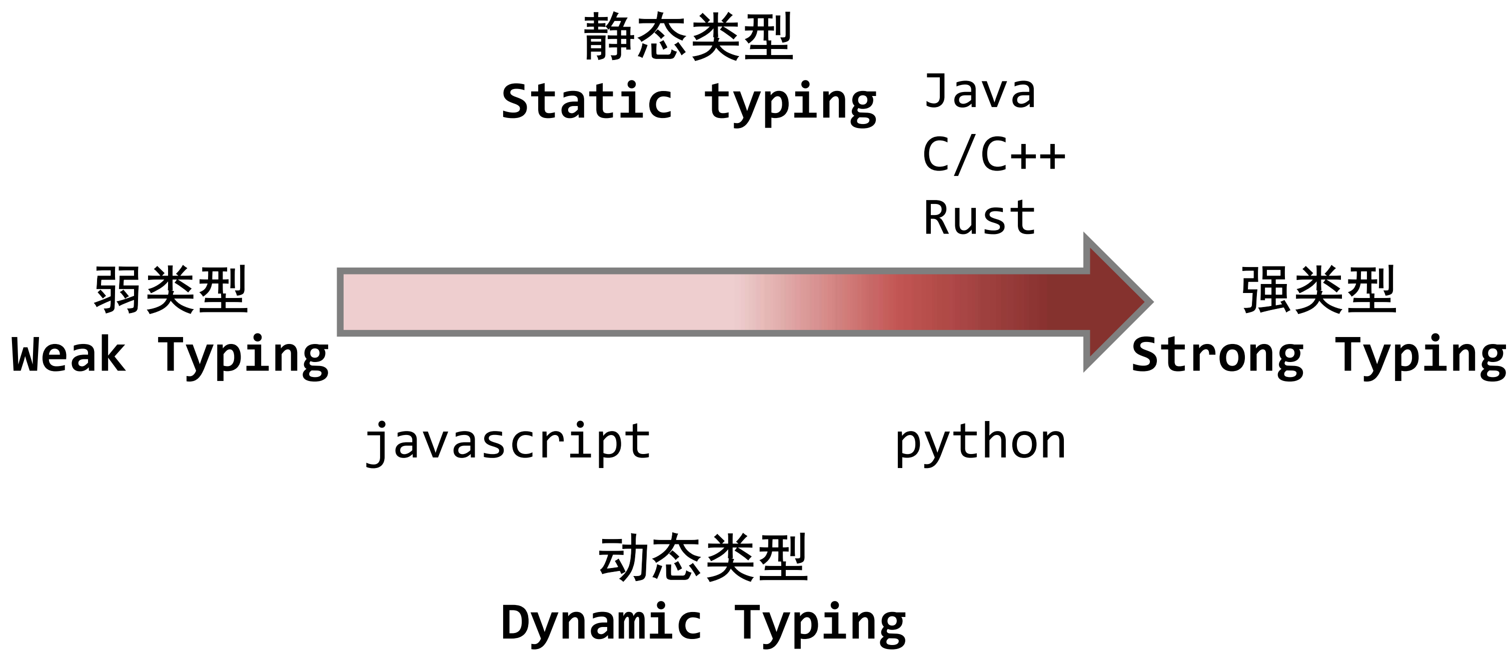
类型系统的目标

- 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



静态类型
Static typing
Java
C/C++
Rust
弱类型
Weak Typing
强类型
Strong Typing
javascript
python
动态类型
Dynamic Typing

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

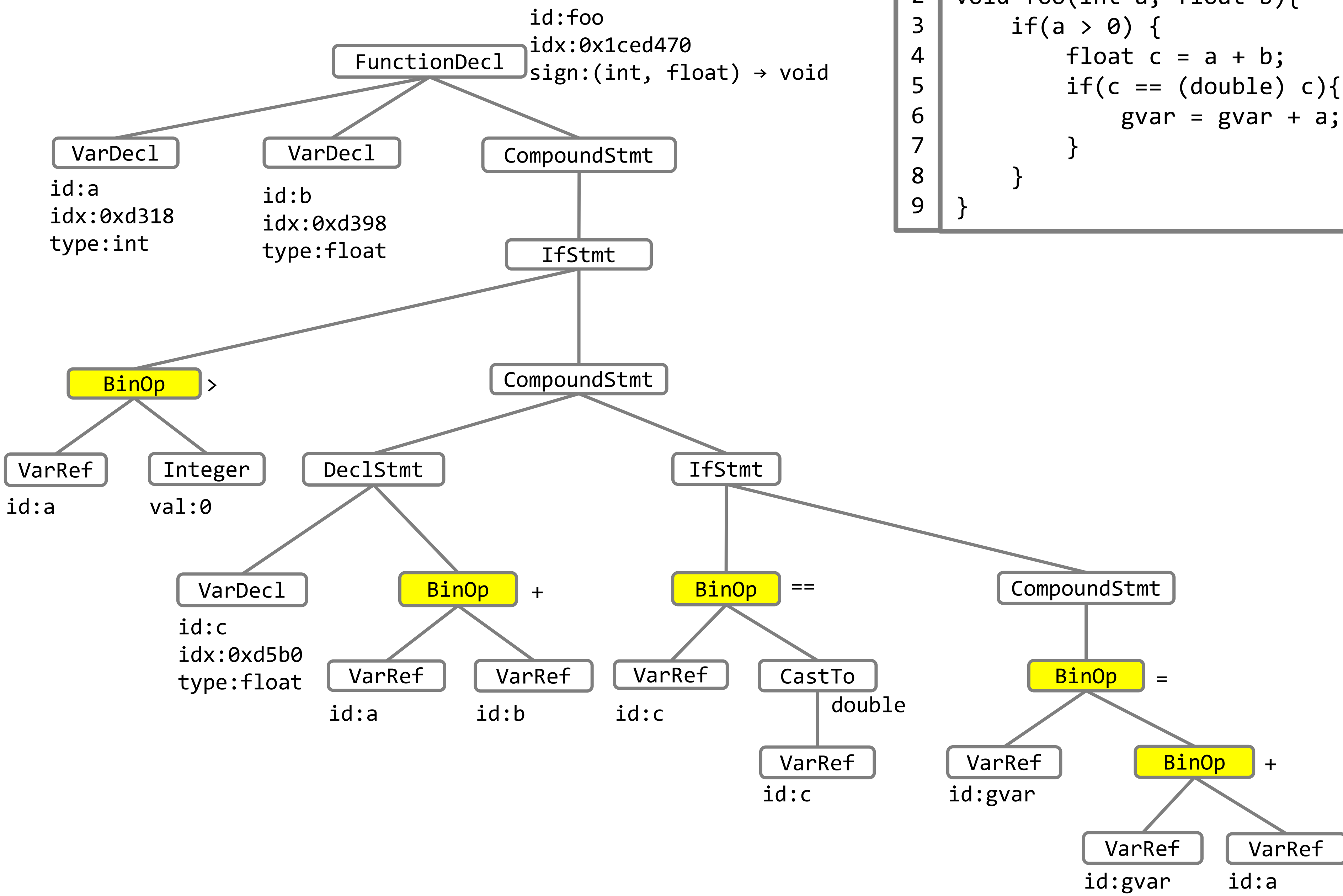
- 如何提取代码标识符类型?
  - 类型
  - 作用域
  - 其它属性

```
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

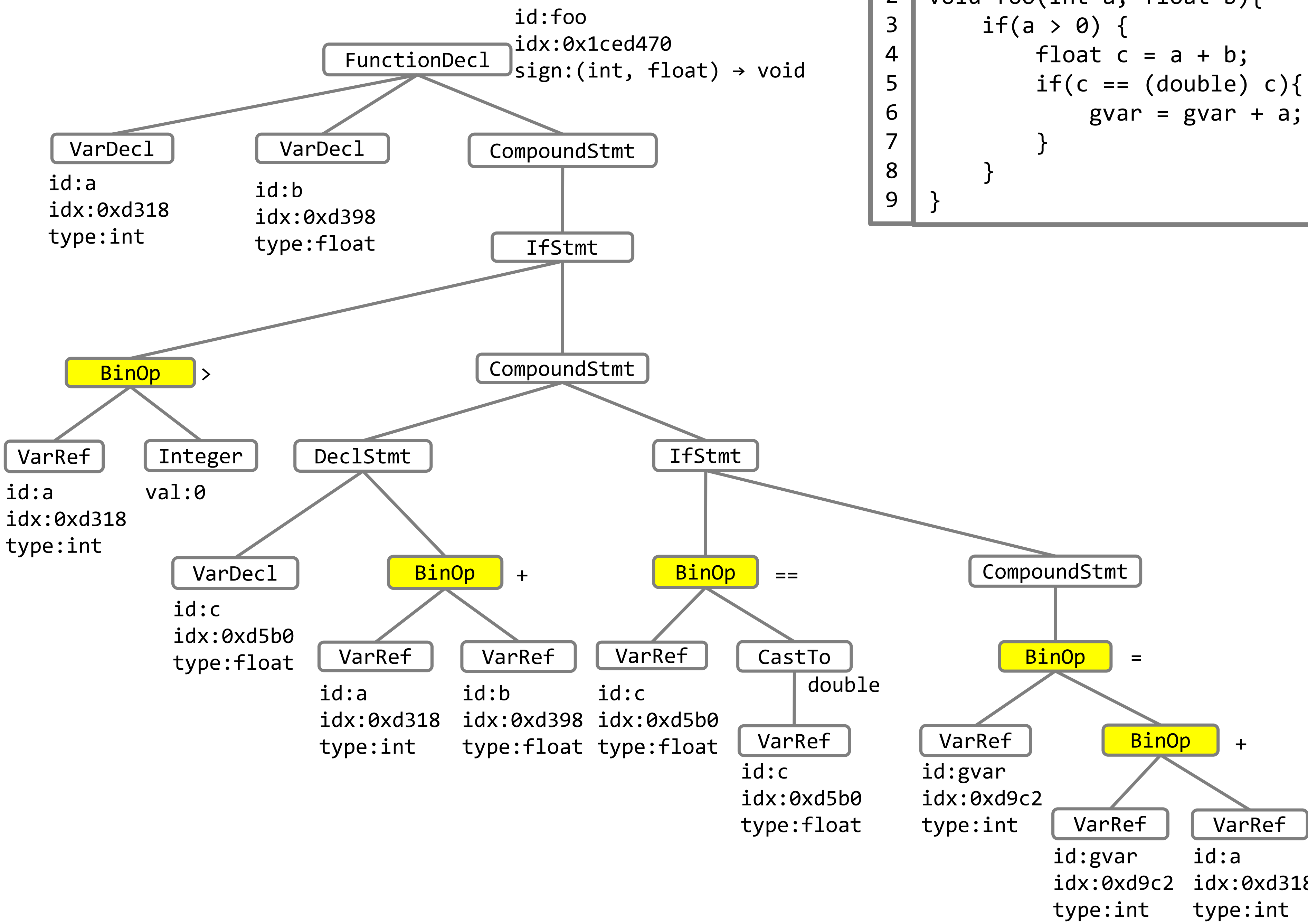
Raw AST

```
int gvar = 0;
void foo(int a, float b){
    if(a > 0) {
        float c = a + b;
        if(c == (double) c){
            gvar = gvar + a;
        }
    }
}
```



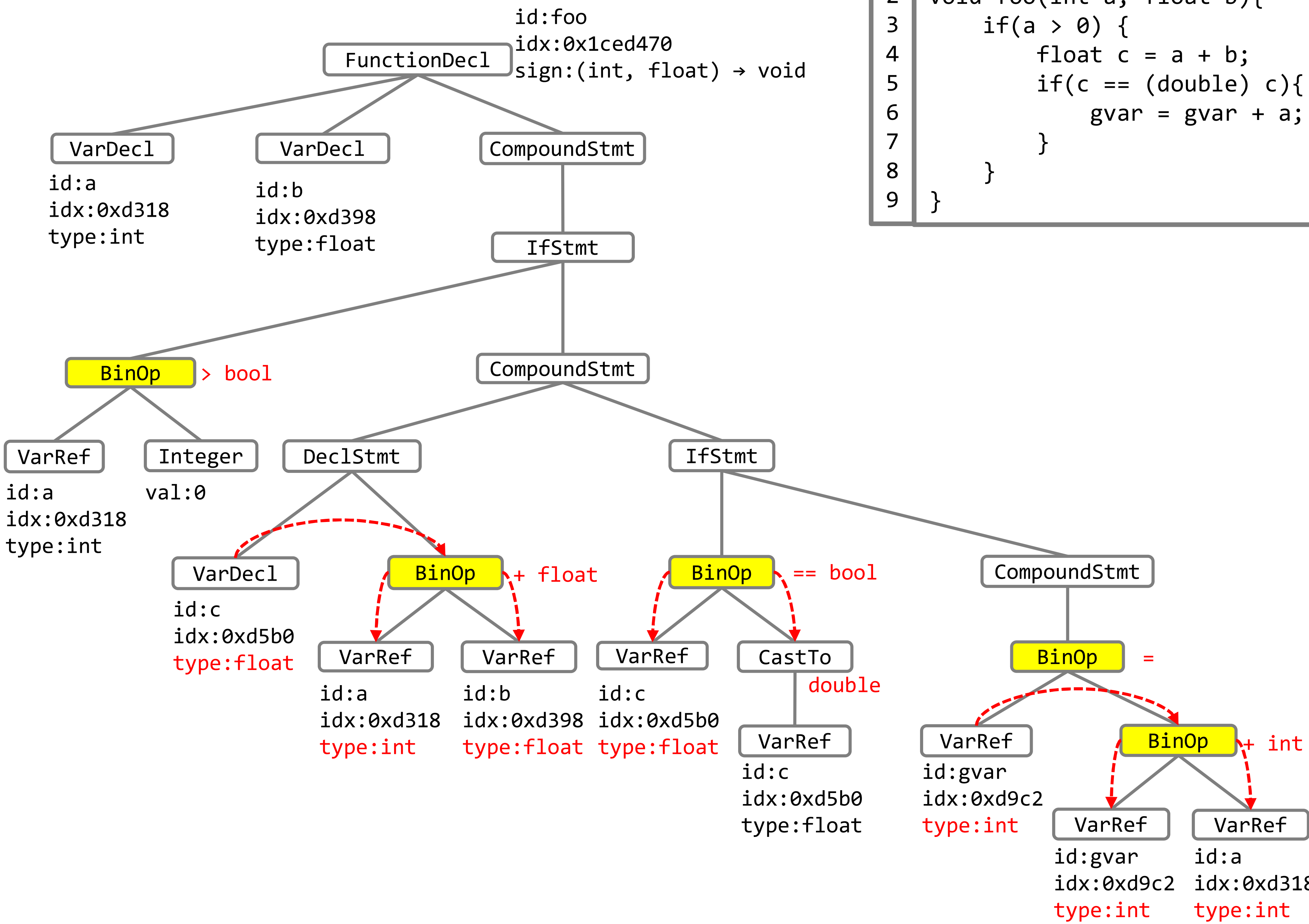
标识符解析

```
int gvar = 0;
void foo(int a, float b){
    if(a > 0) {
        float c = a + b;
        if(c == (double) c){
            gvar = gvar + a;
        }
    }
}
```

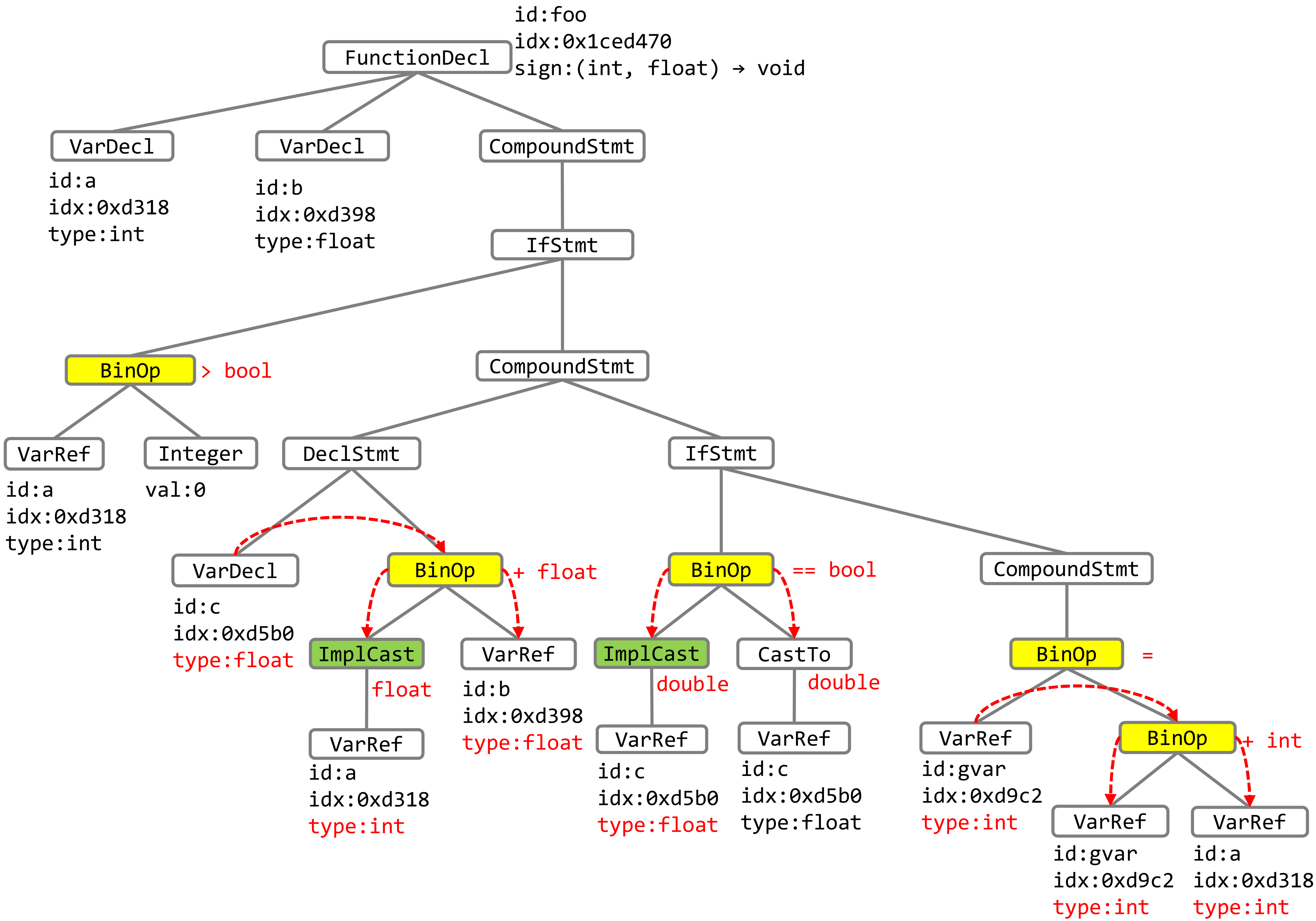


类型检查

```
int gvar = 0;
void foo(int a, float b){
    if(a > 0) {
        float c = a + b;
        if(c == (double) c){
            gvar = gvar + a;
        }
    }
}
```



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

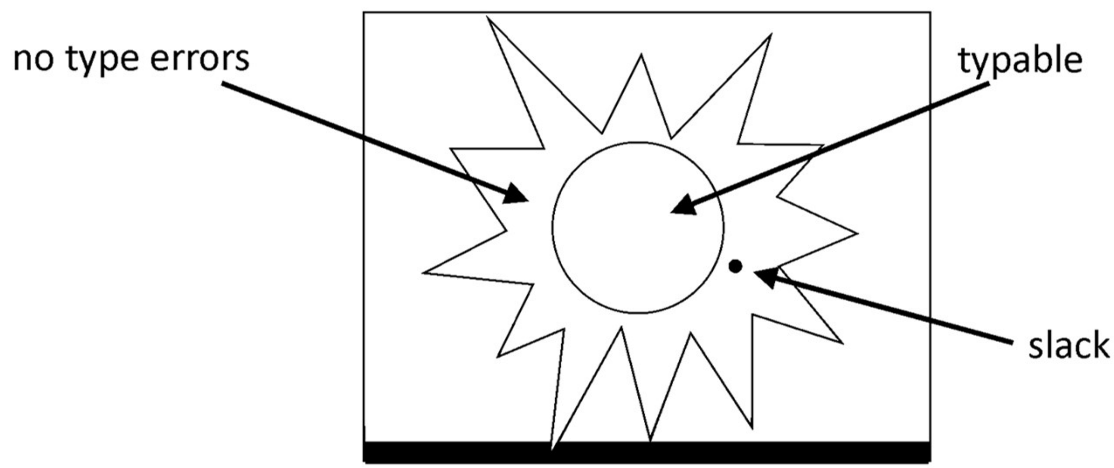
- C++11的auto、Rust的let
- C++ template、Java Interface、Rust泛型

```
//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：

```
τ → int                 整数
  | & τ                 指针
  | (τ,…,τ) → τ         函数
```

- 基于基础类型可以推导出复合类型，比如：

```
(int, &int) → &&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）	类型约束
`I //常量`	⟦I⟧ = int
`E1 op E2`	⟦E1⟧ = ⟦E2⟧ = ⟦E1 op E2⟧ = int
`E1 == E2`	⟦E1⟧ = ⟦E2⟧ = ⟦E1 == E2⟧ = int
`input`	⟦input⟧ = int
`X = E`	⟦X⟧ = ⟦E⟧
`Output E`	⟦E⟧ = int
`if(E){S}`	⟦E⟧ = int
`if(E){S1}else{S2}`	⟦E⟧ = int
`while(E){S}`	⟦E⟧ = int

类型约束生成规则举例（2/2）

程序语句	类型约束
`X(X1,...,Xn){ ... return E; }`	⟦X⟧ = (⟦X1⟧...⟦Xn⟧)→⟦E⟧
`E(E1,...,En)`	⟦E⟧ = (⟦E1⟧...⟦En⟧)→⟦E(E1...En)⟧
`alloc E`	⟦alloc E⟧ = &⟦E⟧
`&X`	⟦&X⟧ = &⟦X⟧
`*E`	⟦E⟧ = &⟦*E⟧
`*X=E`	⟦X⟧ = &⟦E⟧

举例

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

$[\![main]\!] = () \to [\![z]\!]$

$[\![x]\!] = [\![input]\!] = int$
$[\![y]\!] = [\![alloc\ x]\!] = \&[\![x]\!]$
$[\![y]\!] = \&[\![x]\!]$
$[\![z]\!] = [\![*y]\!],\ [\![y]\!] = \&[\![*y]\!]$

解约束:
$[\![x]\!] = int$
$[\![y]\!] = \&int$
$[\![z]\!] = int$
$[\![main]\!] = () \to int$

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;
}
```

⟦main⟧ = ()→⟦z⟧

⟦x⟧ = ⟦alloc 1⟧ = &⟦1⟧
⟦y⟧ = ⟦alloc x⟧ = &⟦x⟧
⟦z⟧ = &⟦x⟧

解约束:
⟦x⟧ = &int
⟦y⟧ = &&int
⟦z⟧ = &&int
⟦main⟧ = ()→&&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;
}
```

$[\![foo]\!]$ = $([\![x]\!]) \rightarrow [\![x+1]\!]$

$[\![x]\!]$ = $[\![alloc\ 10]\!]$ = $\&[\![10]\!]$ $[\![x]\!]$ = &int

$[\![x]\!]$ = $[\![1]\!]$

$[\![x]\!]$ = $[\![1]\!]$ = $[\![x+1]\!]$ $[\![x]\!]$ = 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;
}
```

⟦foo⟧ = (⟦x⟧)→⟦*x⟧
⟦x⟧ = &⟦*x⟧

解约束：
⟦x⟧ = int
    | &int
    | &&int

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

⟦x⟧ = &⟦y⟧

解约束：
⟦x⟧ = &int
⟦y⟧ = int

递归问题（可解）

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

约束:

⟦factorial⟧ = (⟦n⟧)→⟦1⟧

⟦factorial⟧ = (⟦n⟧)→⟦(* n (factorial (- n 1)))⟧

⟦n⟧ = ⟦0⟧ = ⟦(= n 0)⟧

⟦n⟧ = ⟦(factorial (- n 1))⟧ = ⟦(* n (factorial (- n 1)))⟧

⟦n⟧ = ⟦1⟧ = ⟦(- n 1)⟧

解约束:

⟦n⟧ = int

⟦factorial⟧ = (int)→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);
```

```
⟦factorial⟧ = (⟦p⟧,⟦x⟧)→⟦1⟧ = (⟦p⟧,⟦x⟧)→⟦p*x(p-1,x)⟧
⟦p⟧ = ⟦0⟧ = ⟦p==0⟧ = int
⟦p⟧ = ⟦x(p-1,x)⟧ = ⟦p*x(p-1,x)⟧ = int
⟦x⟧ = (⟦p-1⟧,⟦x⟧)→⟦x(p-1,x)⟧
⟦p⟧ = ⟦1⟧ = ⟦p-1⟧ = int
⟦factorial⟧ = (⟦10⟧,⟦factorial⟧)→⟦factorial(10, factorial)⟧
```

使用ϕ来标记Regular Type

```
⟦factorial⟧ = ϕ = (int, ϕ)→int
```

结构体递归定义

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

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

```c
//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≤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≤X；
  - 传递性：X≤Y，Y≤Z $\Longrightarrow$ X≤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（参数多态）：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

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');
```

$[\![max]\!] = (T,T)\rightarrow T$

$T = int$

$[\![max]\!] = (int,int)\rightarrow int$

```
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');
```

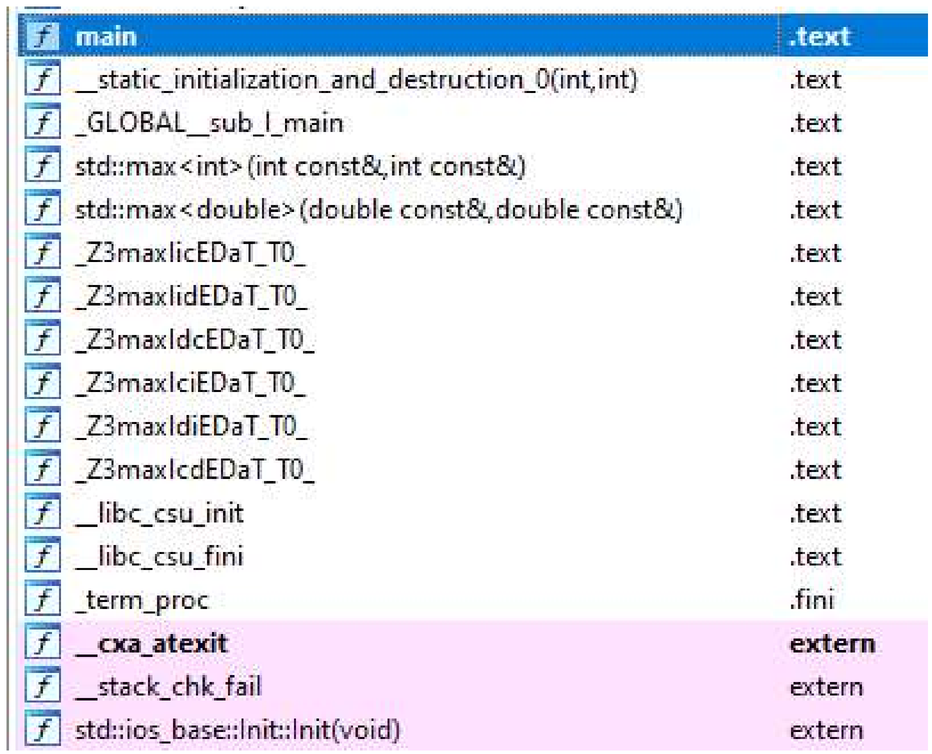
?

https://en.cppreference.com/w/cpp/language/template_argument_deduction

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);
```



Rust使用Trait作为泛型的类型约束

```
fn largest<T:PartialOrd+Copy>(list: &[T]) -> T {
    let mut largest = list[0];
    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

https://doc.rust-lang.org/book/ch10-01-syntax.html

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 { … }
```

Rust支持继承吗？

- Trait的代码可以在struct中复用；
  - “impl A for S” => S>A?
  - 但trait不是类型。
- Trait之间可以存在偏序关系；
  - “trait B:A” => B<A
  - 但非类型之间的偏序关系
  - “impl B”会作用到所有实现A的T。
    - impl<T> B for T where T:A { … }
- 基于Trait多少的subtype关系？
  - S>T?

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

```
struct S { }
trait A { … }
trait B : A { … }
impl A for S { … }
impl B for S { … }
```

```
struct S { }
struct T { }
trait A { … }
trait B { … }
impl A for T { … }
impl B for T { … }
impl A for S { … }
```

```
fn makeacall(s: &S){ }
let t = T {};
makeacall(&t);
```

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T类型
  - 类型检查，如果类型不匹配会抛出异常

```
class Base {};
class Derived : public Derived {};

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

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);
}
```

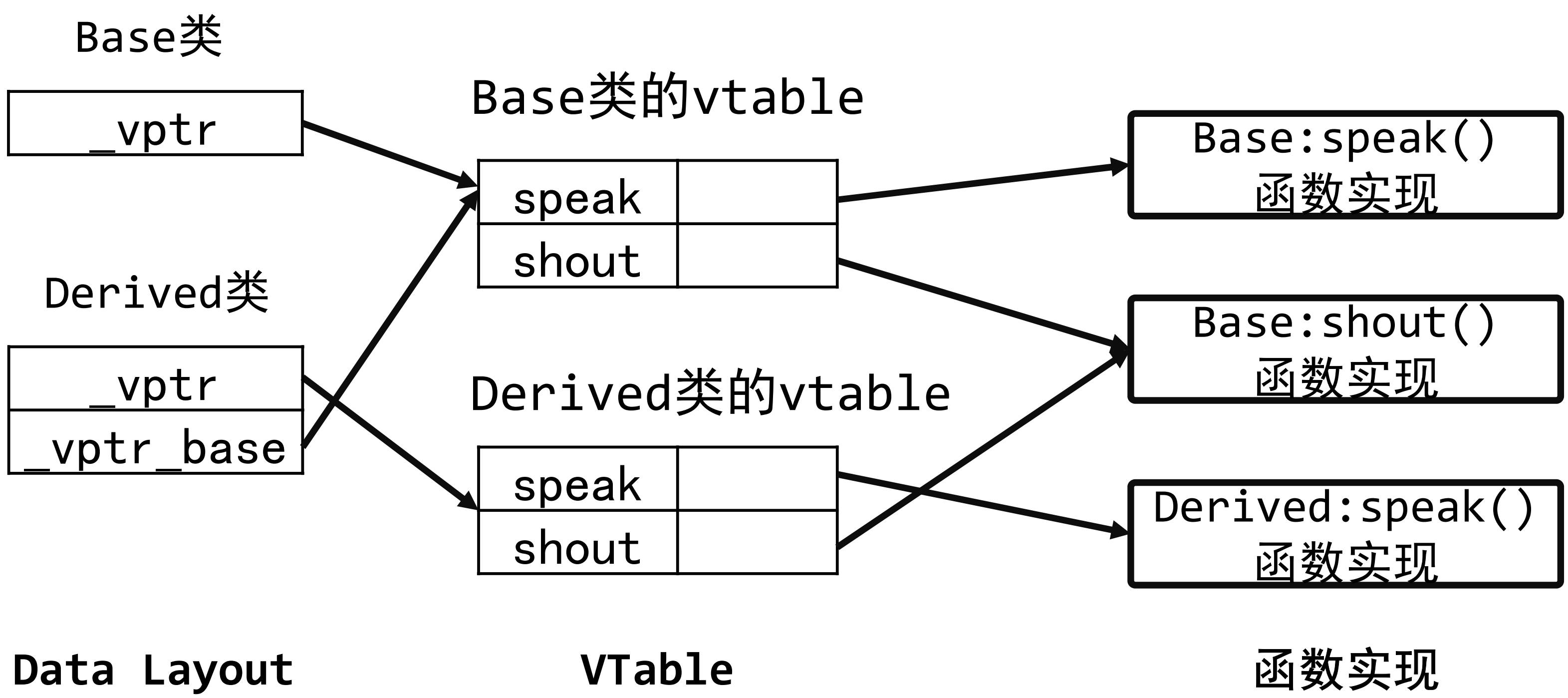
```
base print
derived speak
base shout
destroying derived
destroying base
```

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。



https://itanium-cxx-abi.github.io/cxx-abi/cxx-vtable-ex.html

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)

https://en.wikipedia.org/wiki/Run-time_type_information

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
```

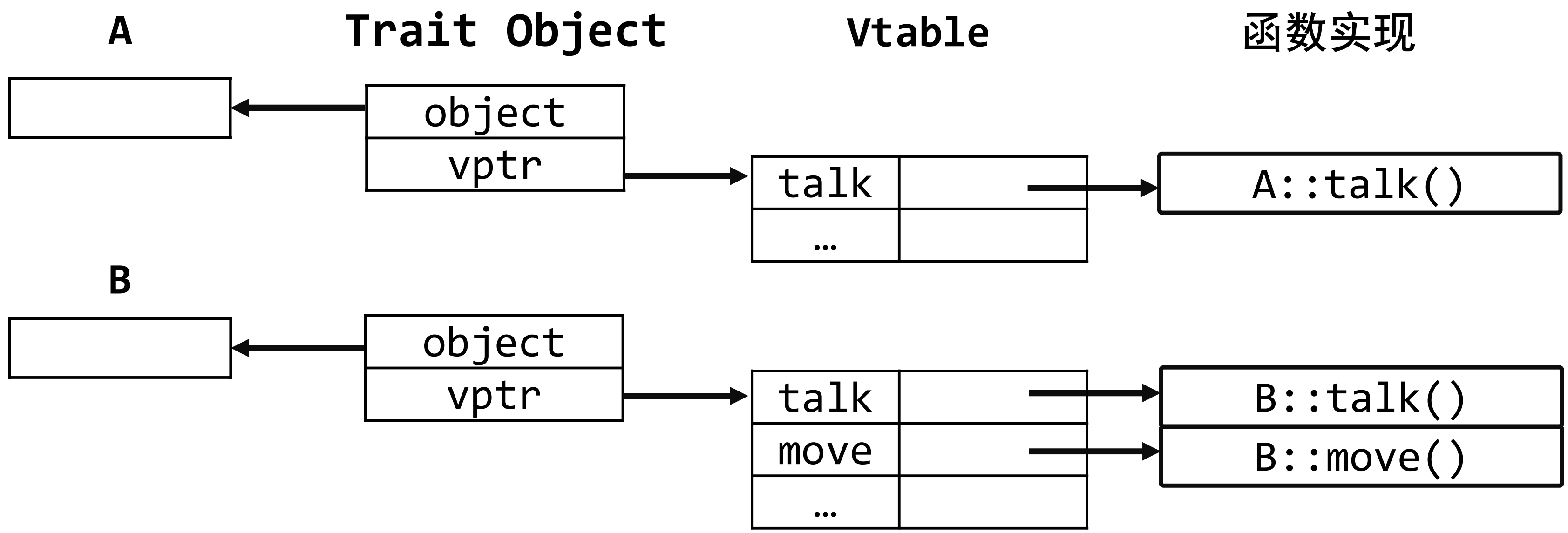
Rust Dyn Trait

```
trait Sound { fn talk(&self); }
trait Move { fn walk(&self); }
struct A { }
struct B { }
impl Sound for A {
    fn talk(&self) { println!("I am A"); }
}
impl Sound for B {
    fn talk(&self) { println!("I am B"); }
}
impl Move for B {
    fn walk(&self) { println!("Move one step"); }
}

fn makeacall(s: &dyn Sound){ s.talk(); }

fn makeanycall(s: &dyn Any){
    if let Some(a) = s.downcast_ref::<A>(){
        a.talk();
    }
}
fn main() {
  let a = A {};
  let b = B {};
  makeacall(&a);
  makeanycall(&b);
}
```

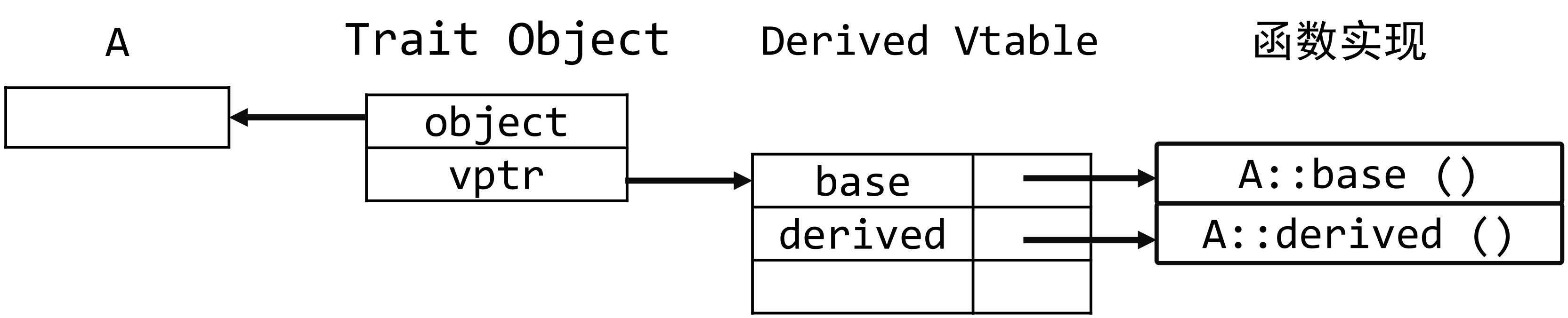
Dyn Trait vs 虚函数



Trait偏序（不支持upcast）

```
trait Base { fn base(&self); }
trait Derived : Base { fn derived(&self); }
struct A { }
impl Base for A {
    fn base(&self) { println!(“Base"); }
}
impl<T> Derived for T where T:Base {
    fn derived(&self) { println!("Derived"); }
}
fn makeacall(s: &dyn Derived){ s as &dyn Base;}

fn main() {
  let a = A {};
  a.super();
  a.derived();
  makeacall(&a);
}
```



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