CS30017 编译

第六讲：类型推导

徐 辉

xuh@fudan.edu.cn

大纲

一、类型推导问题

二、标识符作用域

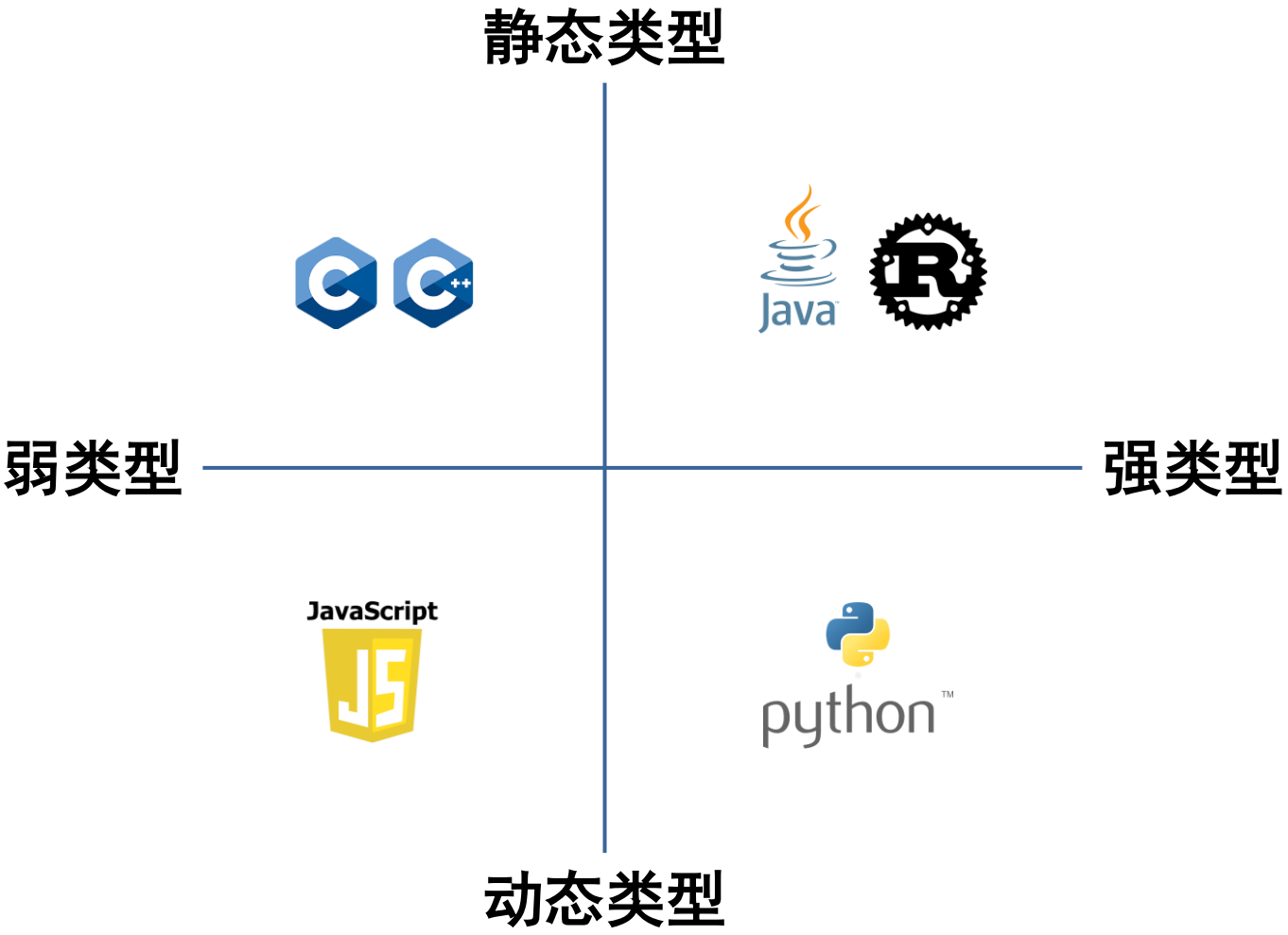
三、类型约束和求解

一、类型推导问题

类型系统基本概念

- 类型系统包括由类型和规则组成
- 类型：
  - 基础类型（Primitive Type）
    - 标量类型（Scalar Types）：bool、char、int、float
    - 复合类型（Compound Type）：数组、元组
  - 自定义类型：结构体、枚举
- 类型规则：
  - 类型推导和检查规则
  - 隐式类型转换

类型系统分类



动态类型 vs 静态类型

- 静态类型系统：编译时检查类型的一致性
- 动态类型系统：运行时检查类型的一致性

```
//python代码
def foo(x):
    if x == 1:
        return "bingo!"
    return x
```

```
//foo的类型是什么？
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'
```

强类型 vs 弱类型

- 强类型系统：一般不允许隐式类型转换
- 弱类型类型：自动隐式转换，灵活但易出错

```python
//python代码
b = 1 + True;
a = 1 + '2';
c = '1' + True;
```

```
2
类型错误
类型错误
```

```c
//C代码
int a = 1 + true;
int b = '1' + true;
int c = 1 + '2';
int d = 1 + "2";
```

```
2
50
51
4202501
```

```javascript
//Javascript代码
1 + true;
1 + '2';
'1' + true;
```

```
2
'12'
'1true'
```

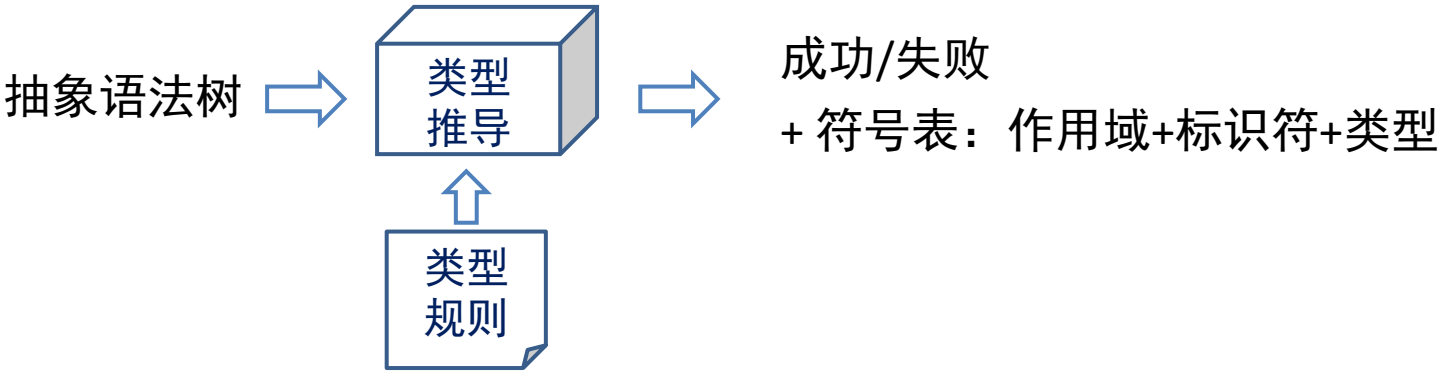
```javascript
//Javascript代码
var a = 42;
var b = "42";
var c = [42];
a === b;
a == b;
a == c;
```

```
false
true
true
```

Javascript Equality Game：https://eqeq.js.org/

类型推导问题

- 已知源代码（抽象语法树）和类型规则
- 为所有标识符找到满足类型规则的唯一解
  - 如满足运算符、函数签名等要求
- 类型检查是类型推导问题的特例



类型推导思路

- 基于抽象语法树对标识符进行作用域识别和类型分析
  - 声明新标识符：确定作用域，建立索引
  - 使用标识符：确定索引，推导或检查类型

```
let g:i32 = 10;

fn fib(x:i32) -> int {
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1);
    let b = fib(x - 2);
    let r = a + b;
    return r;
}

fn main() {
    let r = fib(10) + g;
}
```

标识符	作用域（粗）	索引	类型
g	global	0x0000	i32
fib	global	0x0100	(i32) → i32
main	global	0x0101	(void) → void
x	fib	0x1100	i32
a	fib	0x1101	i32
b	fib	0x1102	i32
r	fib	0x1103	i32
r	main	0x82d0	i32

二、标识符作用域

作用域分析（细粒度）

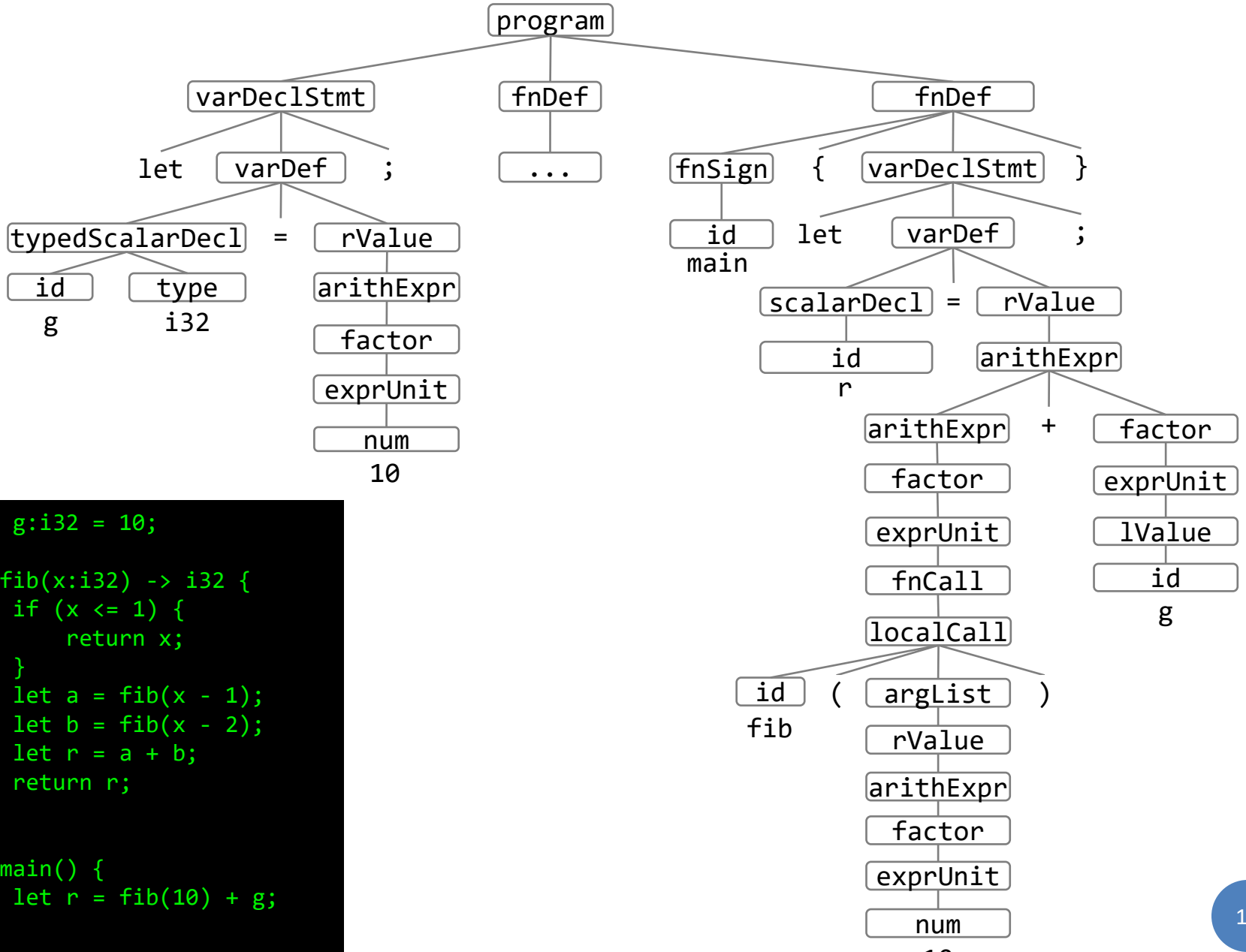
```
let g:i32 = 10;

fn fib(x:i32) -> int { //scope fib: available {g, x}
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1); //{ scope 1: available {g, x, a}
        let b = fib(x - 2); //{ scope 2: available {g, x, a, b}
            let r = a + b; //{ scope 3:  available {g, x, a, b, r}
                return r;
            // end scope 3 }
        // end scope 2 }
    // end scope 1 }
}

fn main() { //scope main
    let r = fib(10) + g;
}
```

作用域之间的偏序关系：fib > 1 > 2 > 3

抽象语法树：Abstract Syntax Tree

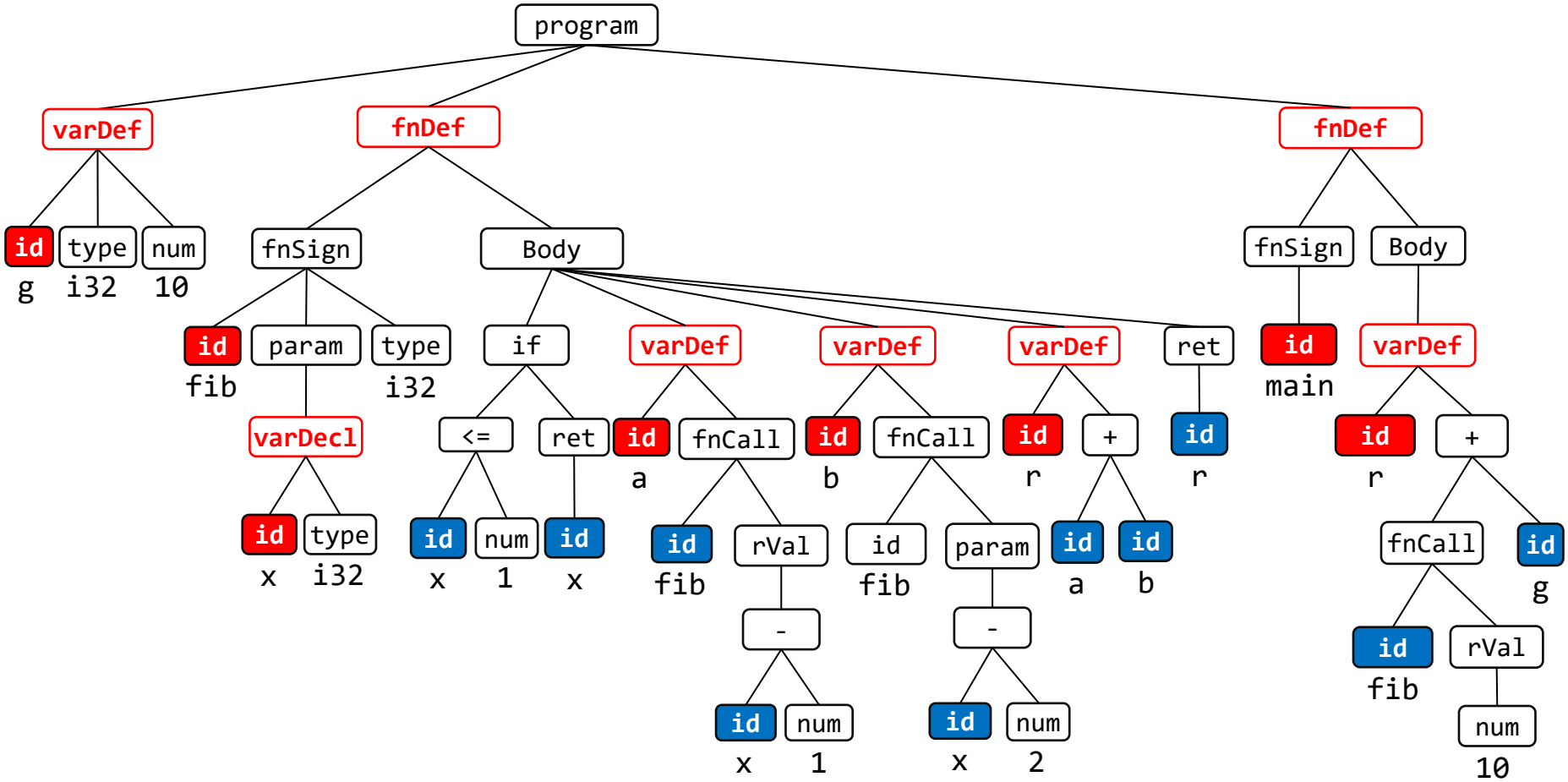
- 具体语法：程序员实际写的代码
  - 语法解析树是对源代码的完整表示
- 抽象语法树：消除解析过程中的一些步骤或节点
  - 单一展开形式塌陷，如E1->E2->E3->NUM
  - 去除括号等冗余信息
  - 避免在叶子结点使用运算符和保留字
- AST记录编译器的阶段性分析结果，会被持续编辑

示例：Tea语言编译器生成的语法树




```
let g:i32 = 10;

fn fib(x:i32) -> i32 {
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1);
    let b = fib(x - 2);
    let r = a + b;
    return r;
}

fn main() {
    let r = fib(10) + g;
}
```

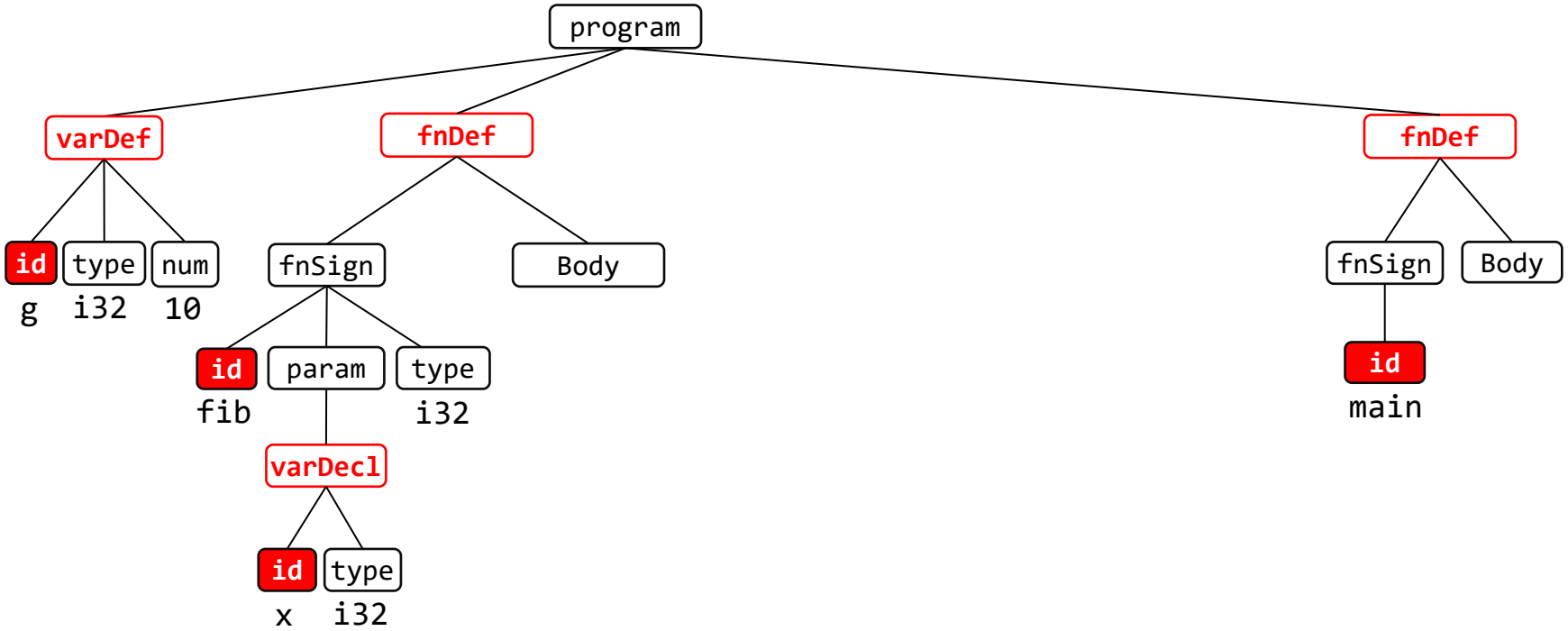
语法树化简 => 问题抽象



变量声明

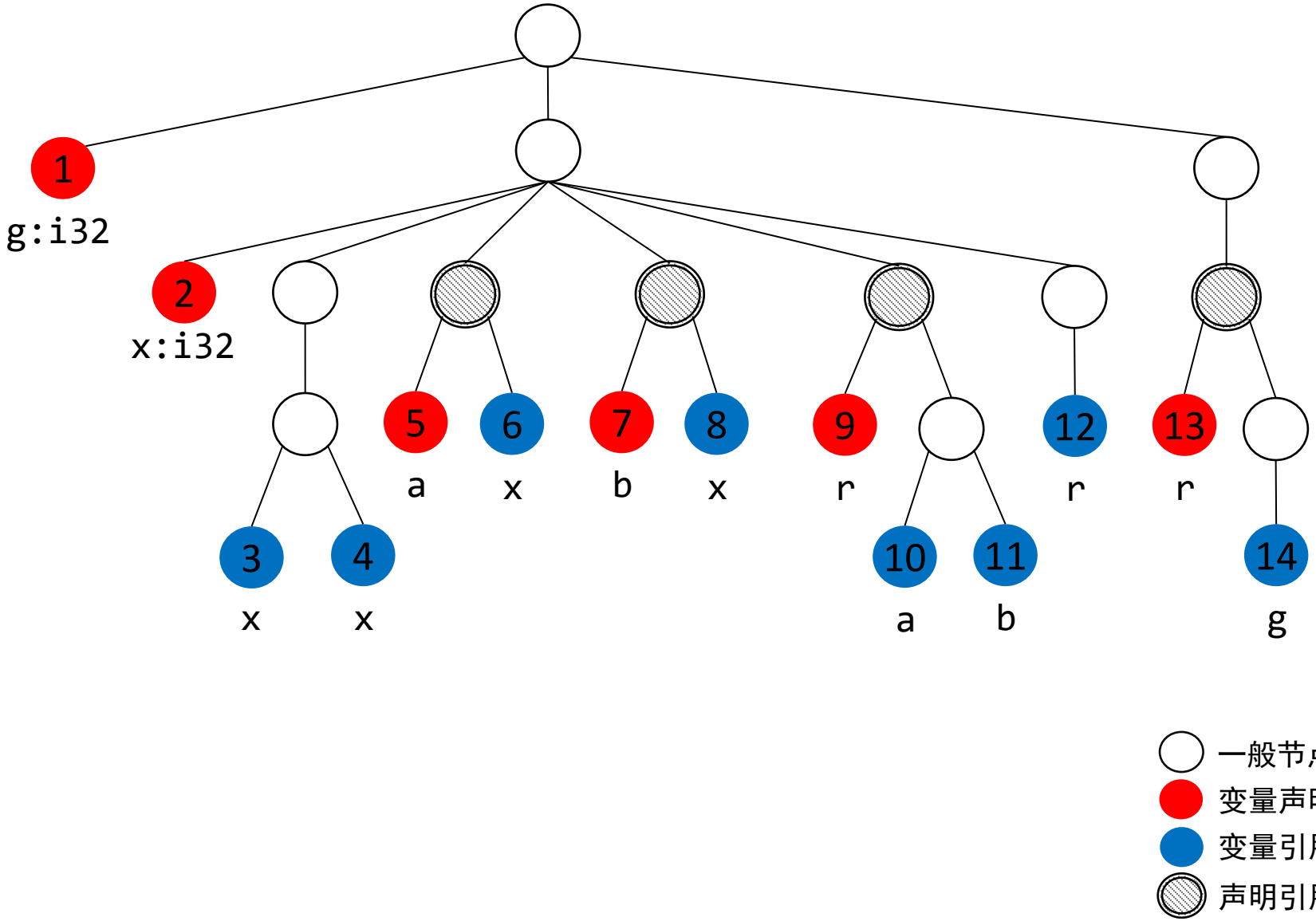
变量引用

全局标识符符号表

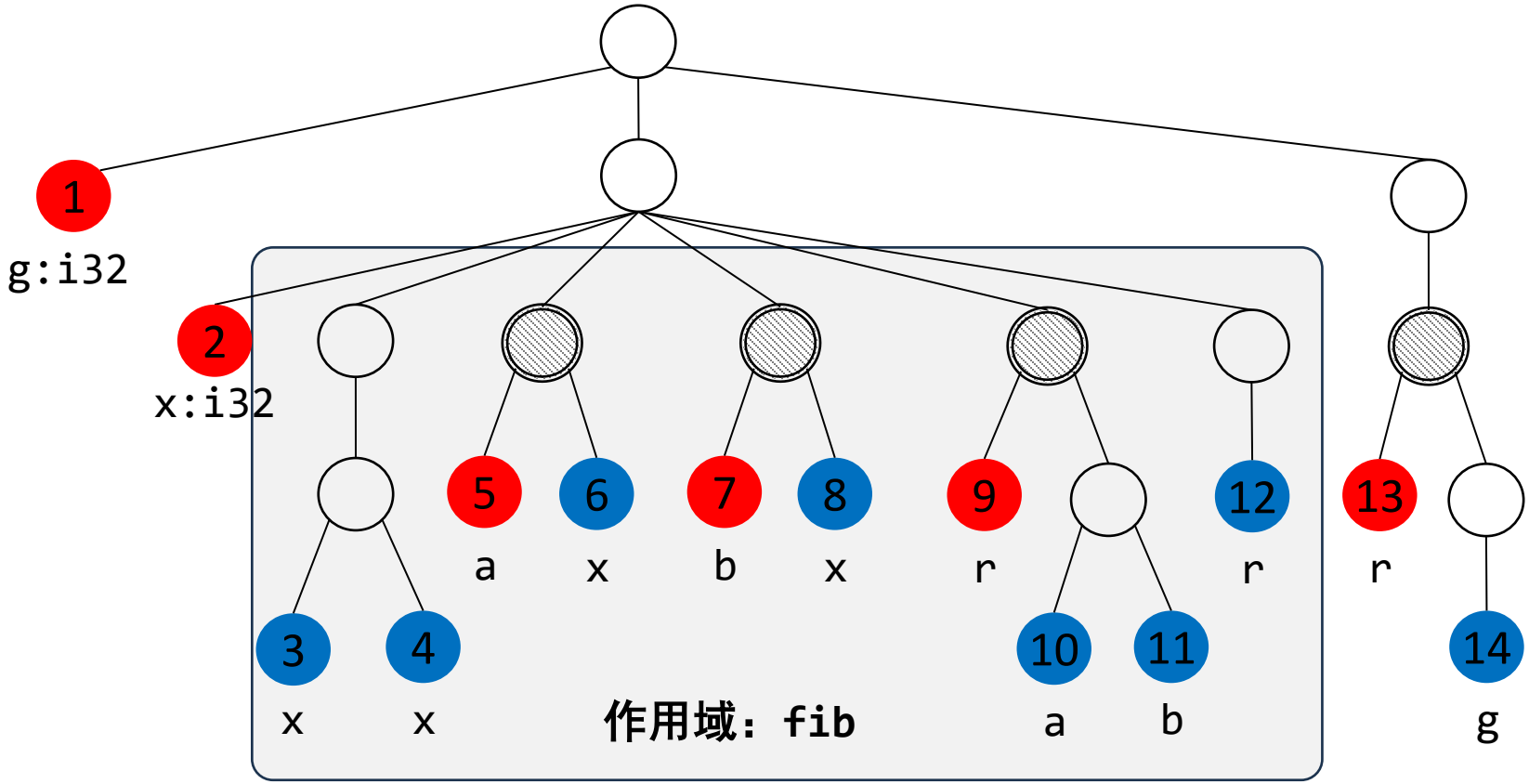


标识符	作用域	索引	类型
g	global	0x0000	i32
fib	global	0x0100	(i32) → i32
main	global	0x0101	(void) → void

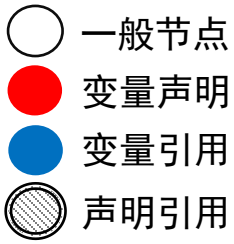
局部变量类型分析


1
g:i32
2
x:i32
5
a
6
x
7
b
8
x
9
r
12
r
13
r
3
x
4
x
10
a
11
b
14
g
一般节点
变量声明
变量引用
声明引用

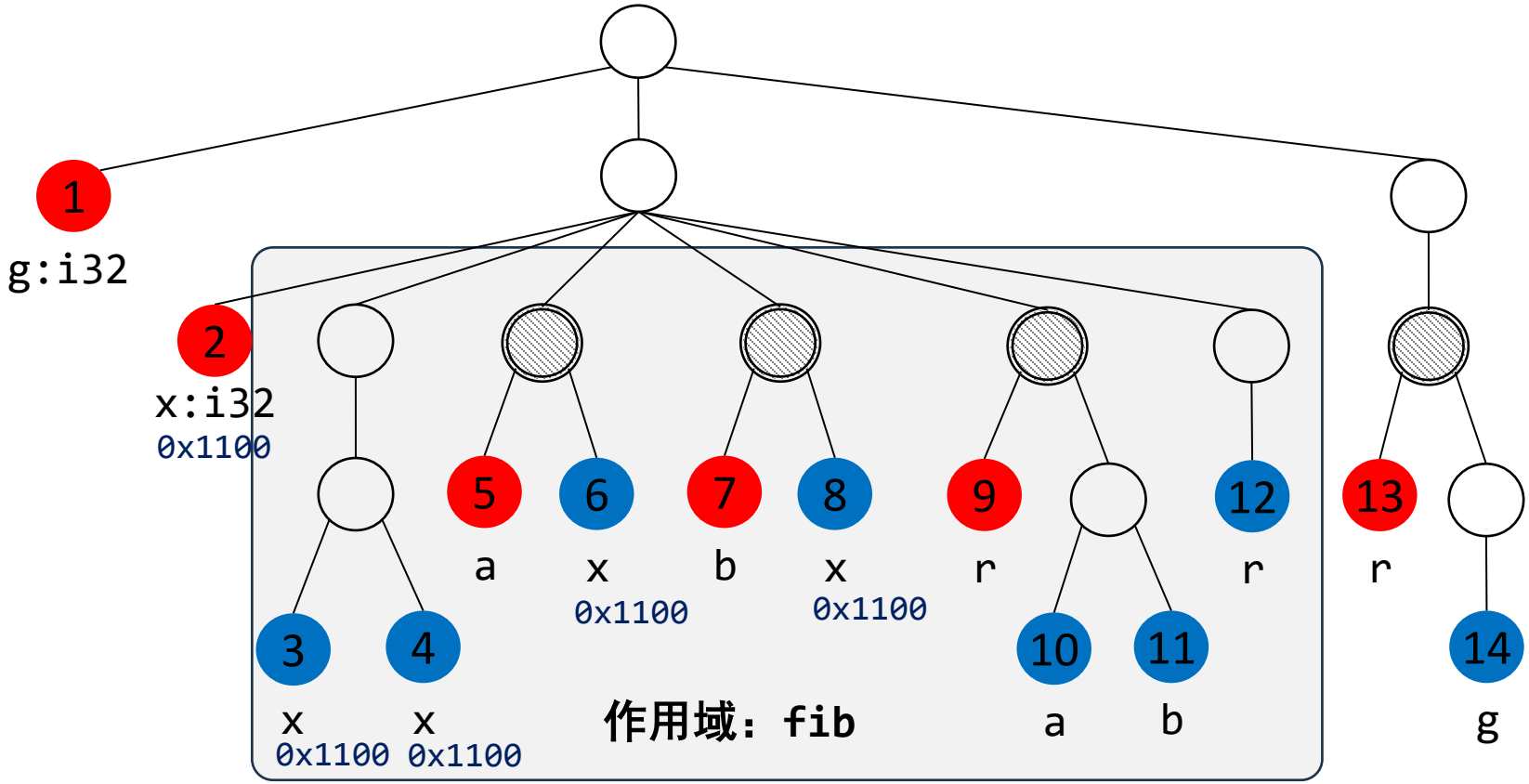
局部变量类型分析：x



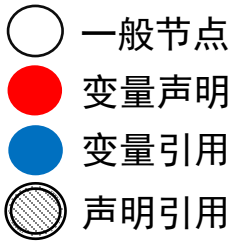
标识符	作用域	索引	类型
x	fib	0x1100	i32
a			
b			
r			



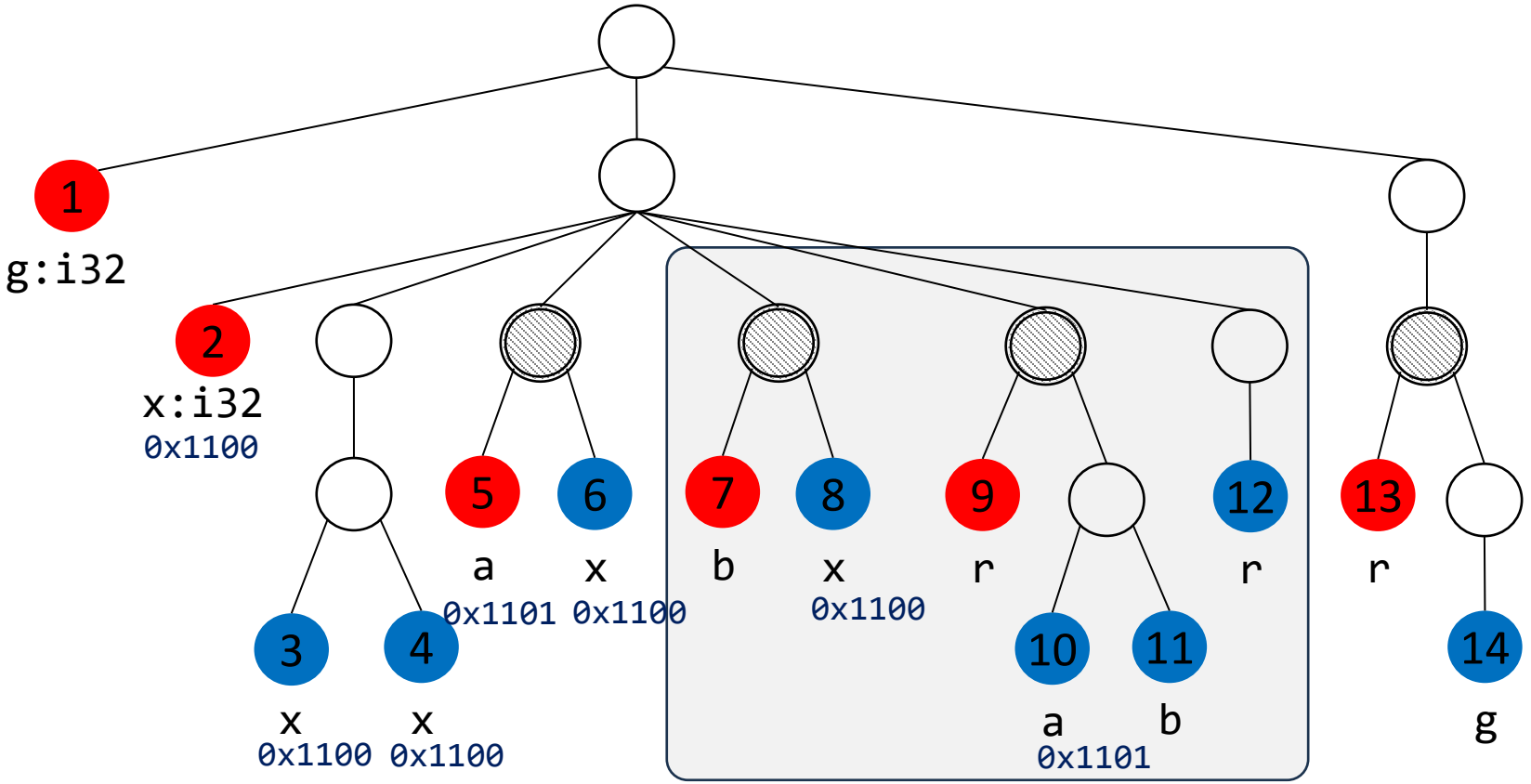
标识符索引化：x



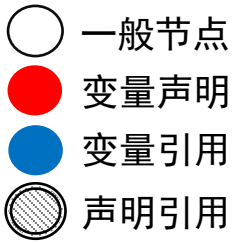
标识符	作用域	索引	类型
x	fib	0x1100	i32
a			
b			
r			



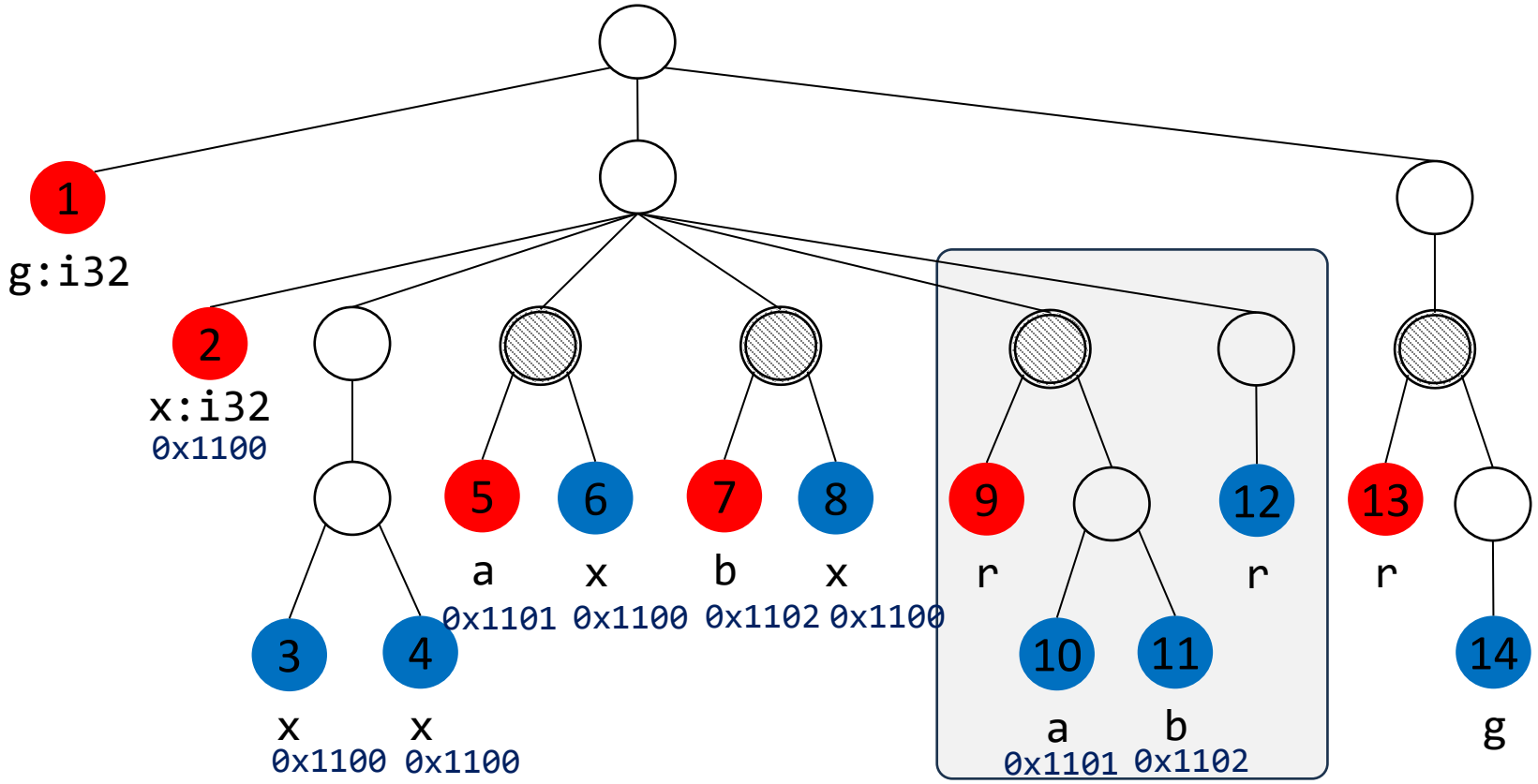
标识符索引化：a



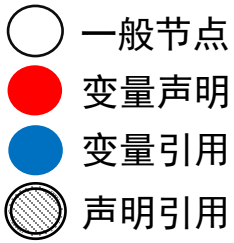
标识符	作用域	索引	类型
x	fib	0x1100	i32
a	**fib:scope1**	**0x1101**	
b			
r			



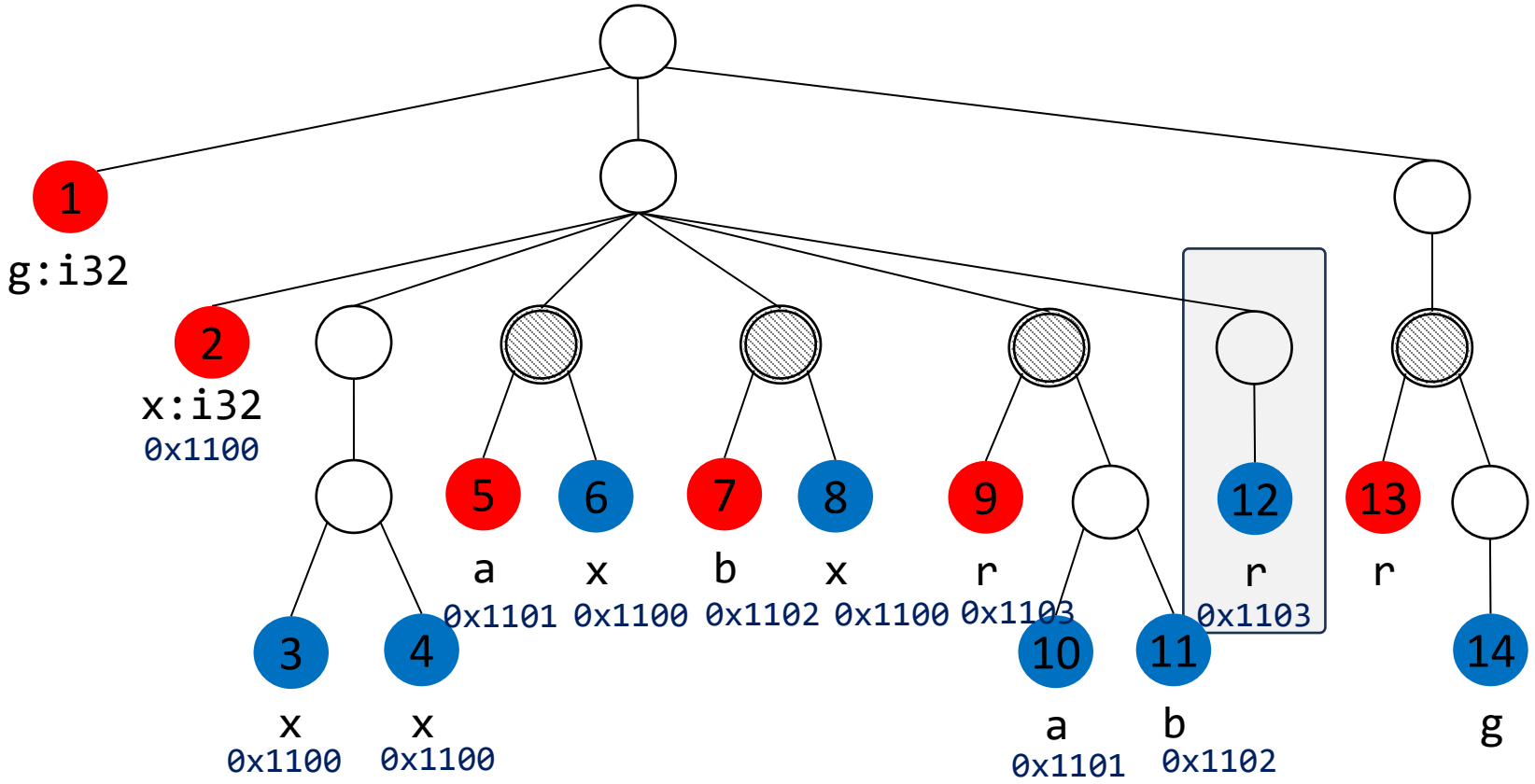
标识符索引化：b



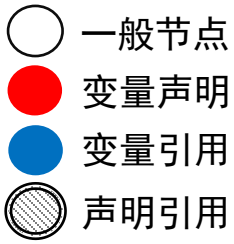
标识符	作用域	索引	类型
x	fib	0x1100	i32
a	fib:scope1	0x1101	
b	**fib:scope2**	**0x1102**	
r			



标识符索引化：r



标识符	作用域	索引	类型
x	fib	0x1100	i32
a	fib:scope1	0x1101	
b	fib:scope2	0x1102	
r	**fib:scope3**	**0x1103**	



算法实现思路

- 动态维护两个哈希表记录标识符作用域和类型
  - 全局符号表
  - 局部符号表
- 遍历抽象语法树对标识符索引化
  - 标识符声明时将其加入符号表
  - 离开作用域时将其移出符号表

对应项目代码：全局变量符号表定义

```rust
23      /// Represents a compiled module containing all global variables and functions.
24      /// This is the top-level container for the generated IR output.
25      pub struct Module {
26          /// A map of global variable names to their definitions.
27          pub global_list: IndexMap<String, GlobalVariable>,
28          /// A map of function names to their compiled function representations.
29          pub function_list: IndexMap<String, Function>,
30      }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/module.rs

对应项目代码：遍历全局变量

```rust
41      fn generate(&mut self) -> Result<(), Error> {
42          let input = self.input;
43
44          // Pass 1: handle `use` statements so imported symbols are available.
45          for use_stmt in input.use_stmts.iter() {
46              self.handle_use_stmt(use_stmt)?;
47          }
48
49          // Pass 2: register all declarations and definitions (signatures only).
50          for elem in input.elements.iter() {
51              use ast::ProgramElementInner::*;
52              match &elem.inner {
53                  VarDeclStmt(stmt) => self.handle_global_var_decl(stmt)?,
54                  FnDeclStmt(fn_decl) => self.handle_fn_decl(fn_decl)?,
55                  FnDef(fn_def) => self.handle_fn_def(fn_def)?,
56                  StructDef(struct_def) => self.handle_struct_def(struct_def)?,
57              }
58          }
59
60          // Pass 3: generate IR bodies for every function definition.
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/module_gen.rs

对应项目代码：将全局变量加入符号表

```
317     fn handle_global_var_decl(&mut self, stmt: &ast::VarDeclStmt) -> Result<(), Error> {
318         let identifier = match stmt.identifier() {
319             Some(id) => id,
320             None => return Err(Error::SymbolMissing),
321         };
322
323         let dtype = Dtype::try_from(stmt)?;
324         let initializers = if let ast::VarDeclStmtInner::Def(d) = &stmt.inner {

345         self.module
346             .global_list
347             .insert(
348                 identifier.clone(),
349                 GlobalVariable {
350                     dtype,
351                     identifier,
352                     initializers,
353                 },
354             )
355             .map_or(Ok(()), |v| {
356                 Err(Error::VariableRedefinition {
357                     symbol: v.identifier,
358                 })
359             })
360     }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/module_gen.rs

对应项目代码：函数符号表定义

```rust
pub struct FunctionGenerator<'ir> {
    /// Shared type registry containing struct and function type definitions for
    /// the whole module.
    pub registry: &'ir Registry,
    /// Reference to the module's global variable map, used during variable lookup.
    pub global_variables: &'ir IndexMap<String, GlobalVariable>,
    /// Map of currently visible local variables, keyed by their source-level name.
    /// Variables are inserted when declared and removed when their enclosing scope
    /// exits.
    pub local_variables: IndexMap<String, LocalVariable>,
    /// A stack of scopes. Each entry is the list of local variable names introduced
    /// in that scope, enabling bulk removal when the scope exits via [`exit_scope`].
    ///
    /// [`exit_scope`]: FunctionGenerator::exit_scope
    scope_locals: Vec<Vec<String>>,
    /// The flat list of IR statements being accumulated for the current function.
    pub irs: Vec<Stmt>,
    /// The function's formal parameters as local variables.
    pub arguments: Vec<LocalVariable>,
    /// Counter for allocating unique virtual register indices; incremented by
    /// [`alloc_vreg`].
    ///
    /// [`alloc_vreg`]: FunctionGenerator::alloc_vreg
    pub next_vreg: usize,
    /// Counter for allocating unique basic block label indices; starts at `1`
    /// because index `0` is reserved for the implicit function-entry block.
    pub next_basic_block: usize,
}
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

对应项目代码：标识符作用域管理

```rust
176         /// Pushes a new empty lexical scope onto the scope stack.
177         ///
178         /// Call this before entering any block (e.g., `{` in the source language) so
179         /// that variables declared inside can be tracked and later removed by
180         /// [`exit_scope`].
181         ///
182         /// [`exit_scope`]: FunctionGenerator::exit_scope
183         pub fn enter_scope(&mut self) {
184             self.scope_locals.push(Vec::new());
185         }
186
187         /// Pops the innermost lexical scope from the scope stack and removes all local
188         /// variables that were introduced in that scope from [`local_variables`].
189         ///
190         /// [`local_variables`]: FunctionGenerator::local_variables
191         pub fn exit_scope(&mut self) {
192             if let Some(locals) = self.scope_locals.pop() {
193                 for id in locals {
194                     self.local_variables.shift_remove(&id);
195                 }
196             }
197         }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

对应项目代码：while作用域处理示例

```rust
462      pub fn handle_while_stmt(&mut self, stmt: &ast::WhileStmt) -> Result<(), Error> {
463          let test_label = self.alloc_basic_block();
464          let true_label = self.alloc_basic_block();
465          let false_label = self.alloc_basic_block();
466
467          // Jump unconditionally into the loop test from the predecessor block.
468          self.emit_jump(test_label.clone());
469
470          // Emit the loop condition test.
471          self.emit_label(test_label.clone());
472          self.handle_bool_unit(&stmt.bool_unit, true_label.clone(), false_label.clone())?;
473
474          // Loop body; `continue` → test_label, `break` → false_label.
475          self.emit_label(true_label);
476          self.enter_scope();
477          for s in stmt.stmts.iter() {
478              self.handle_block(s, Some(test_label.clone()), Some(false_label.clone()))?;
479          }
480          self.exit_scope();
481          // Back-edge: jump back to the loop condition.
482          self.emit_jump(test_label);
483
484          self.emit_label(false_label);
485          Ok(())
486      }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

作用域导致的类型错误举例

```
fn foo(n:i32) -> i32 {
    while (n>0) {
        ...
        n = n-1;
    }
    return x;
}
```

错误：变量x未声明

当前使用变量不在符号表中

```
fn foo(n:i32) -> i32 {
    while (n>0) {
        let x:i32;
        ...
        n = n-1;
    }
    return x;
}
```

声明变量x，但作用域太小

错误：变量x未声明

作用域导致的类型检查问题

```
fn foo(n:i32) -> i32 {
    let x:i32;
    if (n>0) {
        let x:i32;
        ...
        x = n-1;
    }
    return x;
}
```

声明变量x

错误：重复声明

当前变量名已经在局部变量表中

```
fn foo(n:i32) -> i32 {
    if (n>0) {
        let x:i32;
        ...
    }
    else {
        let x:i32;
        ...
    }
}
```

正确：声明变量x

正确：声明变量x

Tea语言中的全局变量使用要求

```
fn foo(n:i32) -> i32 {
    x = x + n;
    return x;
}
let x:i32 = 0;
```

全局变量声明和在函数中的使用顺序无关

```
let a:i32 = 5;
let x:i32 = a + 5;
```

语法规则支持，翻译IR时不支持

```
let x:i32 = 0;
fn foo(n:i32) -> i32 {
    let x = x + n;
    return x;
}
```

全局变量和局部变量可以重名

对应项目代码：符号查询

```rust
164     pub fn lookup_variable(&self, id: &str) -> Result<Operand, Error> {
165         if let Some(local) = self.local_variables.get(id) {
166             Ok(Operand::from(local))
167         } else if let Some(global) = self.global_variables.get(id) {
168             Ok(Operand::Global(global.clone()))
169         } else {
170             Err(Error::VariableNotDefined {
171                 symbol: id.to_string(),
172             })
173         }
174     }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

Tea语言中的函数重名问题：是否允许重载/多态？

```
fn foo(x: i32);

fn foo() -> i32;
```

Tea语言暂时不允许重载

支持方法：在全局符号表中增添项目即可

标识符	作用域	索引	类型
foo	global	0xadc2	(i32) → void
foo	global	0xbc70	(void) → i32

对应项目代码：函数注册

```
375      fn handle_fn_decl(&mut self, decl: &ast::FnDecl) -> Result<(), Error> {
376          let identifier = decl.identifier.clone();

398          if let Some(ftype) = self
399              .registry
400              .function_types
401              .insert(identifier.clone(), function_type.clone())
402          {
403              if ftype != function_type {
404                  return Err(Error::ConflictedFunction { symbol: identifier });
405              }
406          }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

三、类型约束和求解

Tea语言的类型系统

- 基础类型
  - 标量类型：i32、bool
  - 复合类型：数组
  - 函数类型
- 自定义类型：使用struct定义
- 规则（静态类型系统）
  - 类型推导：为代码中的每个标识符和表达式确定类型
  - 类型检查：分析每个参数类型是否符合运算符或函数签名要求
  - 类型转换：允许隐式类型转换？

类型问题举例

```
fn foo(n:i32) -> i32;

fn main() {
    foo(foo);
}
```

参数类型错误

```
fn main() {
    let a;
    let b;
    b = a;
}
```

无法确定a和b的类型

类型推导/检查思路

Damas–Hindley–Milner方法

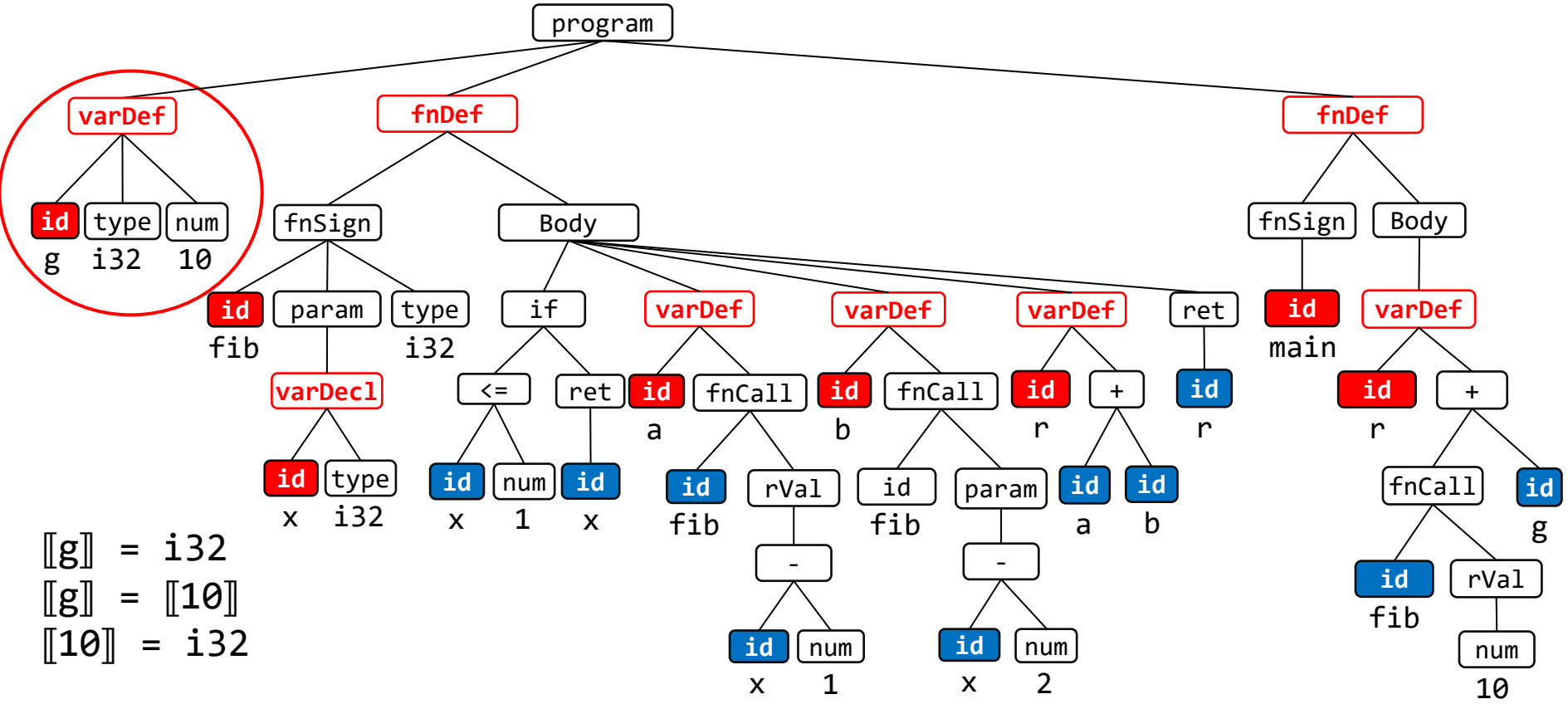
- 根据符号表确定变量类型
- 基于类型规则提取类型约束
  - 类型表示：用⟦X⟧表示变量X的类型
  - 约束提取：一般都为等价关系，如果支持子类型和范型除外
- 约束求解

类型规则设计

- 为不同的语法制定相应的推断规则

代码示例	代码模式	约束
`a: i32`	`X: Ty`	⟦X⟧ = Ty
`0`	`N`	⟦N⟧ = i32
`a = b;`	`X = Y`	⟦X⟧ = ⟦Y⟧
`a + b;`	`X bop Y`	⟦X⟧ = ⟦Y⟧ = ⟦X+Y⟧
`c = a + b;`	`Z = X bop Y`	⟦X⟧ = ⟦Y⟧ = ⟦X+Y⟧ = ⟦Z⟧

示例：约束提取

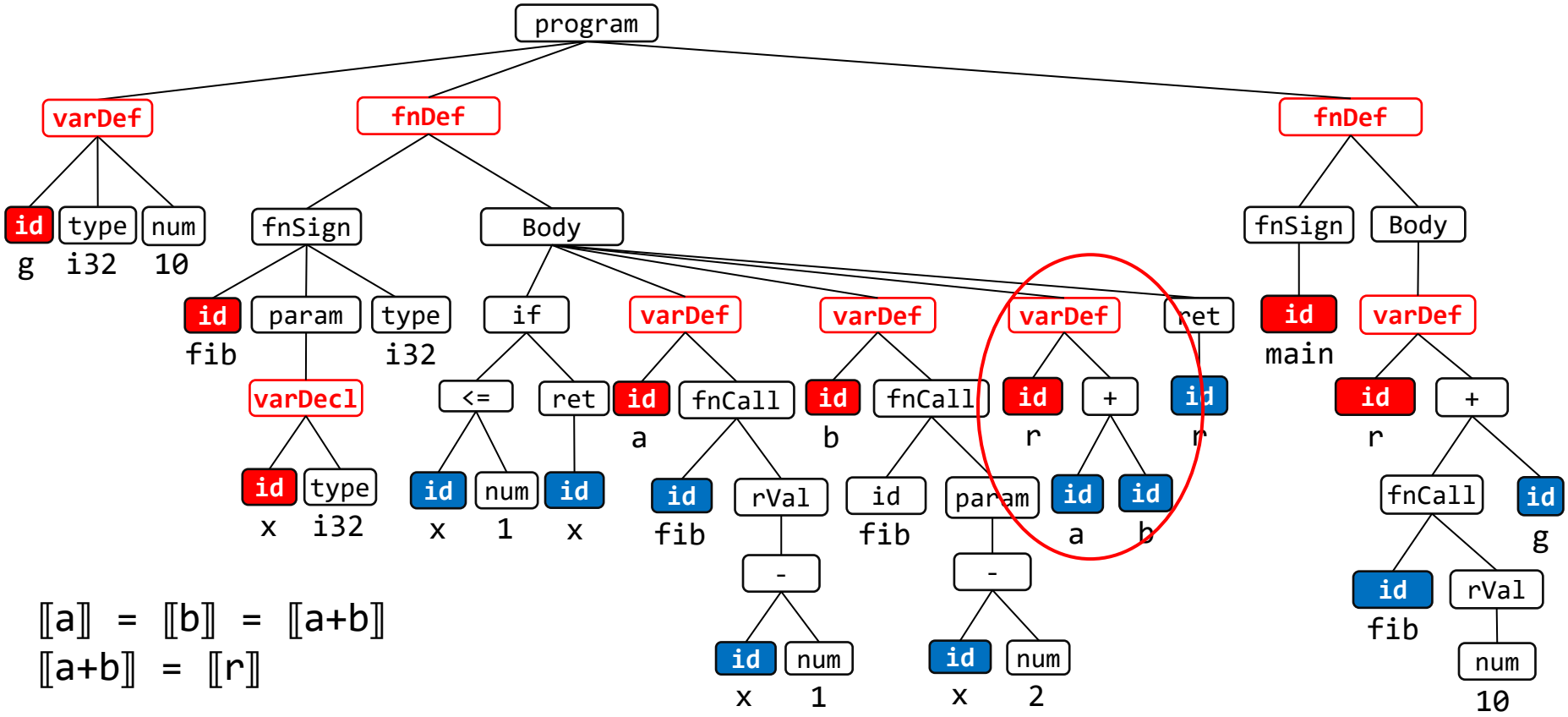


⟦g⟧ = i32
⟦g⟧ = ⟦10⟧
⟦10⟧ = i32

标识符	作用域	索引	类型
g	global	0x0000	i32
fib	global	0x0100	(i32) → i32
main	global	0x0101	(void) → void

变量声明
变量引用

示例：约束提取



⟦a⟧ = ⟦b⟧ = ⟦a+b⟧
⟦a+b⟧ = ⟦r⟧

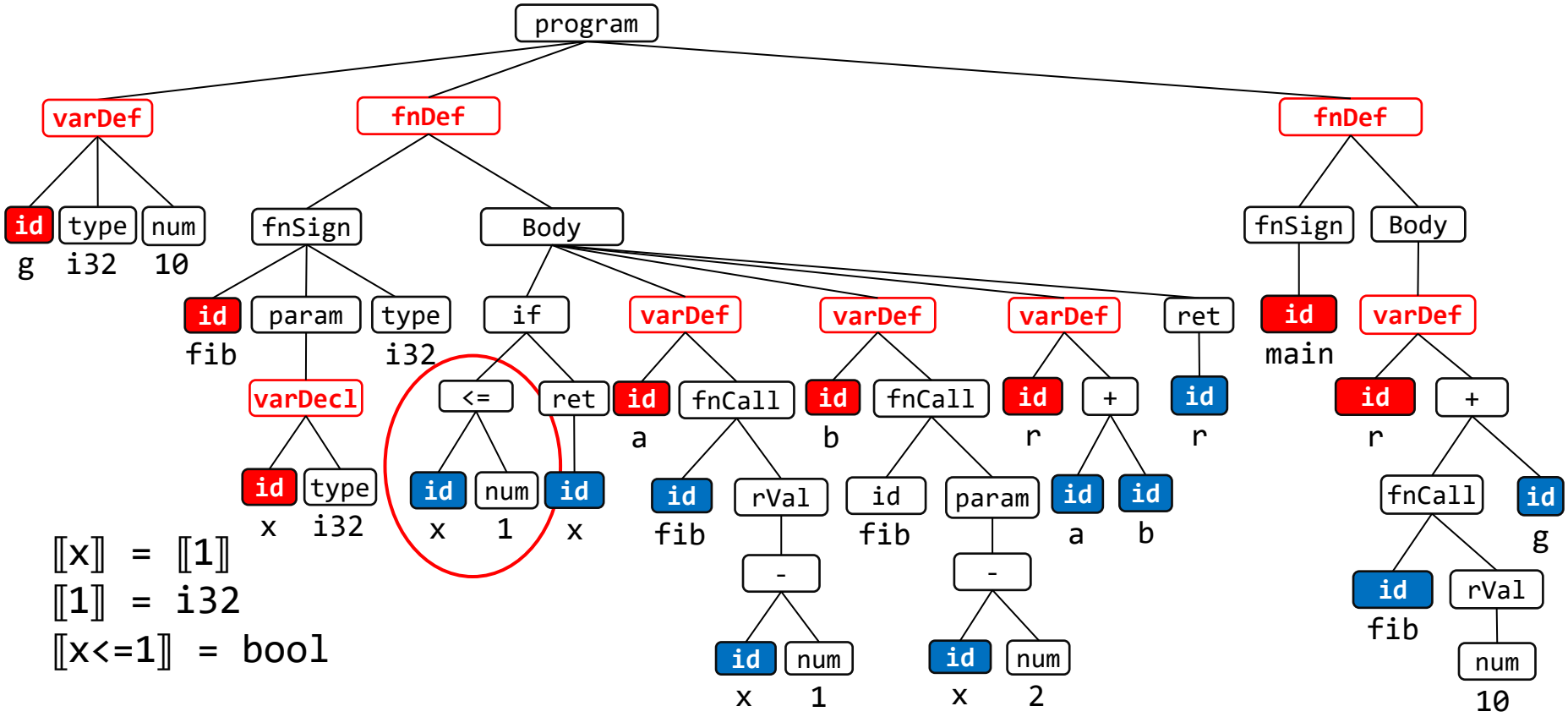
标识符	作用域	索引	类型
x	fib	0x1100	i32
a	fib:scope1	0x1101	
b	fib:scope2	0x1102	
r	fib:scope3	0x1103	

变量声明
变量引用

更多类型规则

代码示例	代码模式	约束
`a > b`	`X cmp Y`	⟦X⟧ = ⟦Y⟧, ⟦X cmp Y⟧ = bool
`a && b`	`X lop Y`	⟦X⟧ = ⟦Y⟧ = ⟦X lop Y⟧ = bool
`if(a){...}`	`if(X)`	⟦X⟧ = bool
`while(a){...}`	`while(X)`	⟦X⟧ = bool

示例：约束提取



标识符	作用域	索引	类型
x	fib	0x1100	i32
a	fib:scope1	0x1101	
b	fib:scope2	0x1102	
r	fib:scope3	0x1103	

变量声明

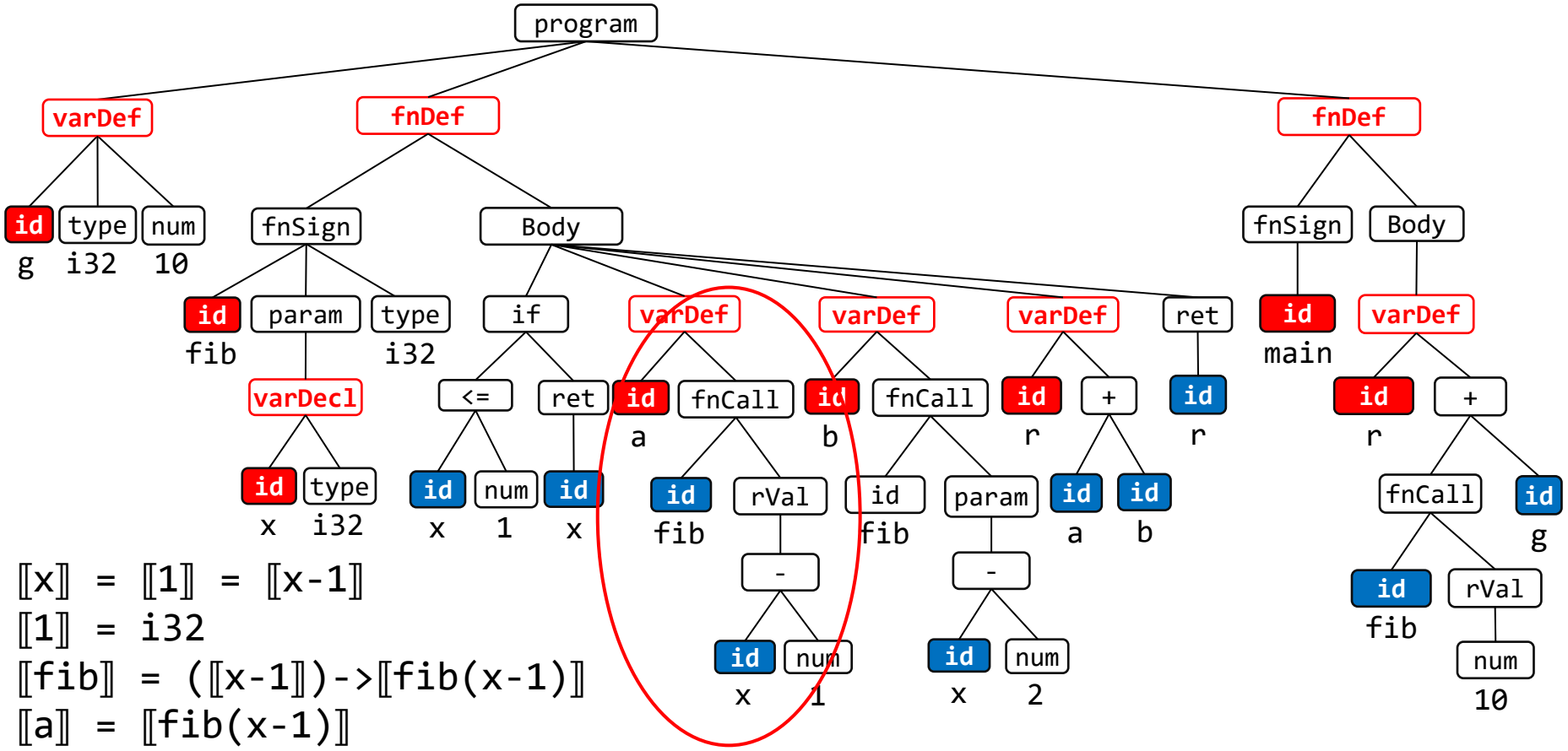
变量引用

更多类型规则

代码示例	代码模式	约束
`foo(a, b);`	`F(X, Y)`	`⟦F⟧ = (⟦X⟧, ⟦Y⟧) -> ⟦F(X, Y)⟧`

`return a;`	`F(X) -> Ty {`
	`  ...`
	`  return Y;`
	`}`
		`⟦F⟧ = (⟦X⟧)->Ty, ⟦Y⟧ = Ty`

示例：约束提取



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

$[\![1]\!] = i32$

$[\![fib]\!] = ([\![x-1]\!])\text{->}[\![fib(x-1)]\!]$

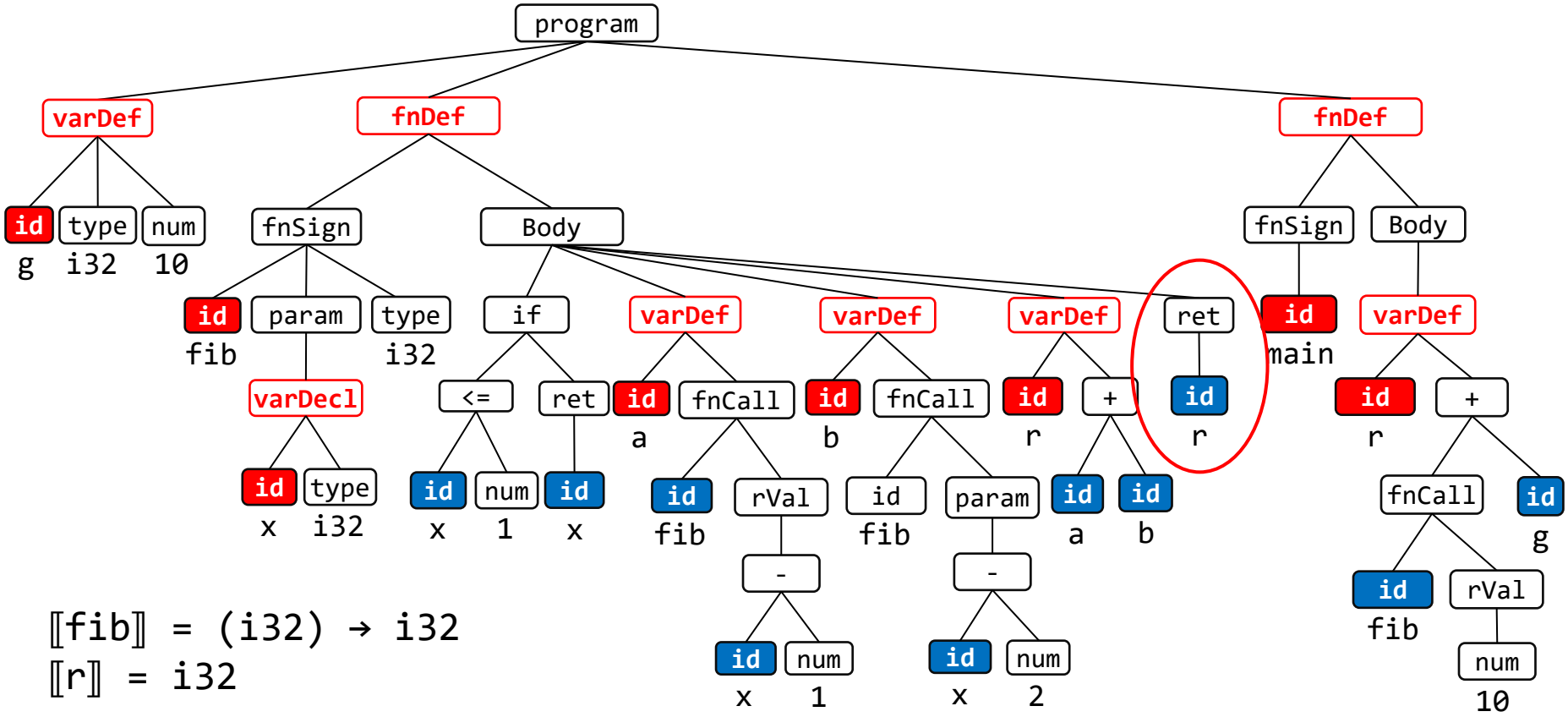
$[\![a]\!] = [\![fib(x-1)]\!]$

标识符	作用域	索引	类型
g	global	0x0000	i32
fib	global	0x0100	(i32) → i32
main	global	0x0101	(void) → void

变量声明

变量引用

示例：约束提取



⟦fib⟧ = (i32) → i32

⟦r⟧ = i32

标识符	作用域	索引	类型
g	global	0x0000	i32
fib	global	0x0100	(i32) → i32
main	global	0x0101	(void) → void

变量声明

变量引用

约束提取结果

全局变量：

⟦g⟧ = i32
⟦g⟧ = ⟦10⟧
⟦10⟧ = i32

main函数：

⟦main⟧ = (void)->void

⟦fib⟧ = (⟦10⟧)->⟦fib(10)⟧
⟦fib(10)⟧ = ⟦g⟧ = ⟦r_{main}⟧

```
let g:i32 = 10;

fn fib(x:i32) -> i32 {
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1);
    let b = fib(x - 2);
    let r = a + b;
    return r;
}

fn main() {
    let r = fib(10) + g;
}
```

fib函数：

⟦fib⟧ = (⟦x⟧)->i32
⟦r_{fib}⟧ = i32
⟦x⟧ = i32

⟦x⟧ = ⟦1⟧
⟦1⟧ = i32
⟦x<=1⟧ = bool
⟦x⟧ = i32

⟦x⟧ = ⟦1⟧ = ⟦x-1⟧
⟦1⟧ = i32
⟦fib⟧ = (⟦x-1⟧)->⟦fib(x-1)⟧
⟦a⟧ = ⟦fib(x-1)⟧

⟦x⟧ = ⟦2⟧ = ⟦x-2⟧
⟦2⟧ = i32
⟦fib⟧ = (⟦x-2⟧)->⟦fib(x-2)⟧
⟦b⟧ = ⟦fib(x-2)⟧

⟦a⟧ = ⟦b⟧ = ⟦a+b⟧ = ⟦r_{fib}⟧

基于并查集方法求解

- 维护不存在相交关系的集合，支持查找和联合两种操
  - Find(x)：返回包含变量x的集合
  - Union(x, y)：联合包含x和y的两个集合

```
while(getPair()!=NULL){
  [p,q] = readPair(p,q);
  pset = find(p);
  qset = find(q);
  if(pset == qset)
      continue;
  else union(p,q);
}
```

应用并查集方法求解

⟦g⟧ = i32 ⟦g⟧ = ⟦10⟧ ⟦10⟧ = i32	**S1**:{i32, ⟦10⟧, ⟦g⟧}
⟦fib⟧ = (⟦x⟧)->i32 ⟦r_{fib}⟧ = i32 ⟦x⟧ = i32	**S1**:{i32, ⟦10⟧, ⟦g⟧, ⟦x⟧, ⟦r_{fib}⟧} **S2**:{⟦fib⟧, (⟦x⟧)->i32}
⟦x⟧ = ⟦1⟧ ⟦1⟧ = i32 ⟦x<=1⟧ = bool ⟦x⟧ = i32	**S1**:{i32, ⟦10⟧, ⟦1⟧, ⟦g⟧, ⟦x⟧, ⟦r_{fib}⟧} **S3**:{bool, ⟦x<=1⟧}
⟦x⟧ = ⟦1⟧ = ⟦x-1⟧ ⟦1⟧ = i32 ⟦fib⟧ = (⟦x-1⟧)->⟦fib(x-1)⟧ ⟦a⟧ = ⟦fib(x-1)⟧	**S1**:{i32, ⟦10⟧, ⟦1⟧, ⟦g⟧, ⟦x⟧, ⟦r_{fib}⟧, ⟦x-1⟧} **S2**:{⟦fib⟧, (⟦x⟧)->i32, (⟦x-1⟧)->⟦fib(x-1)⟧} **S4**:{⟦a⟧, ⟦fib(x-1)⟧}
...	...

应用并查集方法求解

⟦x⟧ = ⟦2⟧ = ⟦x-2⟧ ⟦2⟧ = i32 ⟦fib⟧ = (⟦x-2⟧)->⟦fib(x-2)⟧ ⟦b⟧ = ⟦fib(x-2)⟧	**S1**:{i32, ⟦10⟧, ⟦1⟧, ⟦2⟧, ⟦g⟧, ⟦x⟧, ⟦r_{fib}⟧, ⟦x-1⟧, ⟦x-2⟧} **S2**:{⟦fib⟧, (⟦x⟧)->i32, (⟦x-1⟧)->⟦fib(x-1)⟧, (⟦x-2⟧)->⟦fib(x-2)⟧ } **S4**:{⟦a⟧, ⟦b⟧, ⟦fib(x-1)⟧, ⟦fib(x-2)⟧}
⟦a⟧ = ⟦b⟧ = ⟦a+b⟧ = ⟦r_{fib}⟧	**S1=S1US4:** {i32, ⟦10⟧, ⟦1⟧, ⟦2⟧, ⟦g⟧, ⟦x⟧, ⟦r_{fib}⟧, ⟦x-1⟧, ⟦x-2⟧, ⟦a⟧, ⟦b⟧, ⟦fib(x-1)⟧, ⟦fib(x-2)⟧}
⟦main⟧ = (void)->void	**S5**:{(void)->void, ⟦main⟧}
⟦fib⟧ = (⟦10⟧)->⟦fib(10)⟧ ⟦fib(10)⟧ = ⟦g⟧ = ⟦r_{main}⟧	**S1**:{i32, ⟦10⟧, ⟦1⟧, ⟦2⟧, ⟦g⟧, ⟦x⟧, ⟦r_{fib}⟧, ⟦r_{main}⟧, ⟦x-1⟧, ⟦x-2⟧, ⟦a⟧, ⟦b⟧, ⟦fib(10)⟧, ⟦fib(x-1)⟧, ⟦fib(x-2)⟧} **S2**:{⟦fib⟧, (⟦x⟧)->i32, (⟦x-1⟧)->⟦fib(x-1)⟧, (⟦x-2⟧)->⟦fib(x-2)⟧, (⟦10⟧)->⟦fib(10)⟧ }

最终解

S1:{i32, ⟦10⟧, ⟦1⟧, ⟦2⟧, ⟦g⟧, ⟦x⟧, $⟦r_{fib}⟧$, $⟦r_{main}⟧$, ⟦x-1⟧, ⟦x-2⟧, ⟦a⟧, ⟦b⟧, ⟦fib(10)⟧, ⟦fib(x-1)⟧, ⟦fib(x-2)⟧}

S2:{⟦fib⟧, (⟦x⟧)->i32, (⟦x-1⟧)->⟦fib(x-1)⟧, (⟦x-2⟧)->⟦fib(x-2)⟧, (⟦10⟧)->⟦fib(10)⟧ }

S3:{bool, ⟦x<=1⟧}

S5:{(void)->void, ⟦main⟧}

⟦g⟧ = i32

⟦a⟧ = i32

⟦b⟧ = i32

$⟦r_{fib}⟧$ = i32

$⟦r_{main}⟧$ = i32

⟦x<=1⟧ = bool

⟦fib⟧ = (i32) → i32

⟦main⟧ = (void)->void

⟦fib(T3)⟧ = i32

更多类型规则：数组

代码示例	代码模式	约束
`{0, 1}`	`{M, N}`	`⟦{M, N}⟧ = &i32`
`{0; 1}`	`{M; N}`	`⟦{M; N}⟧ = &i32`
`let a[10]: i32`	`X[I]: Ty`	`⟦X⟧ = &Ty`
`b = a[i];`	`Y = X[Z]`	`⟦Z⟧ = i32, ⟦Y⟧ = ⟦*X⟧, ⟦X⟧ = &⟦*X⟧`
`a[i] = b;`	`X[Z] = Y`	`⟦Z⟧ = i32, ⟦X⟧ = &⟦Y⟧`

更多类型规则：结构体

代码示例	代码模式	约束

`struct Foo {`
`  a: i32,`
`  b: i32,`
`}`
	`struct ST {`
	`  A: Ty1,`
	`  B: Ty2,`
	`}`
		⟦ST⟧ = ⟦A, B⟧ = (Ty1, Ty2)

`foo.a = d;`	`X.A = Y`	⟦X.A⟧ = ⟦Y⟧, ⟦X.A, _⟧ = ⟦X⟧

类型推断可能存在的问题

- 解不唯一的情况：优先选择哪些类型？
- 无解的情况：是否允许隐式类型转换？
- 如何判断类型是否等价：名字相同 vs 结构相同

```
struct Position {
    x:i32,
    y:i32
}
```

```
struct Location {
    x:i32,
    y:i32
}
```

递归问题

```
// Tea语言代码
fn fac(n:i32) -> i32 {
    if (n == 0) {
        return 1;
    } else {
        let r = n * fac(n-1);
        return r;
    }
}
```

⟦fac⟧ = (i32)->i32

```
// Tea语言代码
struct List {
    v:i32,
    next List,
}
```

⟦List⟧ = ϕ = (i32, ϕ)

```
// C代码
struct List {
    i32 data;
    struct List* next;
};
```

⟦List⟧ = ϕ = (i32, &ϕ)
= (i32, usize)

缺少返回语句

```
fn foo(n:i32) -> i32 {
    if (n <= 1) {
        return n;
    }
    let x:i32 = foo(n-1);
}
```

类型为i32/void时自动添加

思考

- 假如Tea语言函数声明可缺省类型，设计方法分析下列代码中的类型信息

```
fn f(n) {
    if (n == 0) {
        return 1;
    } else {
        let r = n * f(n-1);
        return r;
    }
}
```

```
fn f(f, n) {
    if (n == 0) {
        return 1;
    } else  {
        let r = n * f(f, n-1);
        return r;
    }
}
```