6 类型推导

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 了解抽象语法树的概念
- ⋆⋆⋆ 掌握标识符索引化方法
- ⋆⋆⋆ 掌握 Hindley-Milner 类型规则设计与应用方法

6.1 类型系统

类型系统由类型和使用规则组成，用于保证程序在运行过程中不会发生类型错误。

- 类型：指程序中数据的分类，用于描述数据的取值范围及其可执行的操作，主要包括基础类型（如 i32、bool）、复合类型（如数组、结构体）以及函数类型（用于描述函数的参数类型和返回值类型）。
- 规则：指对类型的使用方式所施加的约束，包括类型检查、类型推导以及类型之间的兼容性与转换规则，用于确保不同类型的数据被正确、安全地使用。

根据类型检查发生的时间，类型系统可分为静态类型系统和动态类型系统：静态类型系统通常在编译阶段确定标识符类型并完成类型检查；动态类型系统则在程序运行过程中确定类型并进行类型检查。根据类型约束的严格程度或是否允许隐式类型转换，类型系统还可以分为强类型系统和弱类型系统。

Tea 语言是一种编译型语言，其类型系统设计为静态类型系统，支持的类型如表 6.1 所示。

表 6.1: Tea 语言类型分类

类型类别	具体类型	说明
基础类型	i32	整型标量类型
	bool	仅作为中间计算结果存在
复合类型	数组	同类型元素的集合
	结构体	用户自定义数据类型
函数类型	$T_1, T_2 \to T$	描述参数类型与返回类型

为了提高编程的简洁性，Tea 语言允许在部分场景中省略标识符类型标注，并由编译器自动进行类型推导。主要规则如下：

- 局部变量声明中允许省略类型，由编译器进行类型推导；
- 函数声明中，参数类型与返回值类型必须显式指定；
- 结构体字段类型必须显式声明。

类型推导一般是基于抽象语法树进行的，包括两个步骤：1）标识符索引化；2）根据类型规则从抽象语法树中提取类型约束并求解。下面分别进行讲解。

6.2 抽象语法树

语法解析的结果是语法解析树 (Parse Tree 或 Concrete Syntax Tree)。以代码 6.1为例，其对应的语法解析树如图 6.1所示。可以看出，语法解析树中包含了大量对后续分析和编译的冗余信息，因此可以对其进行进一步的抽象与化简。

```
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;
}
```

代码 6.1: Tea 语言代码示例

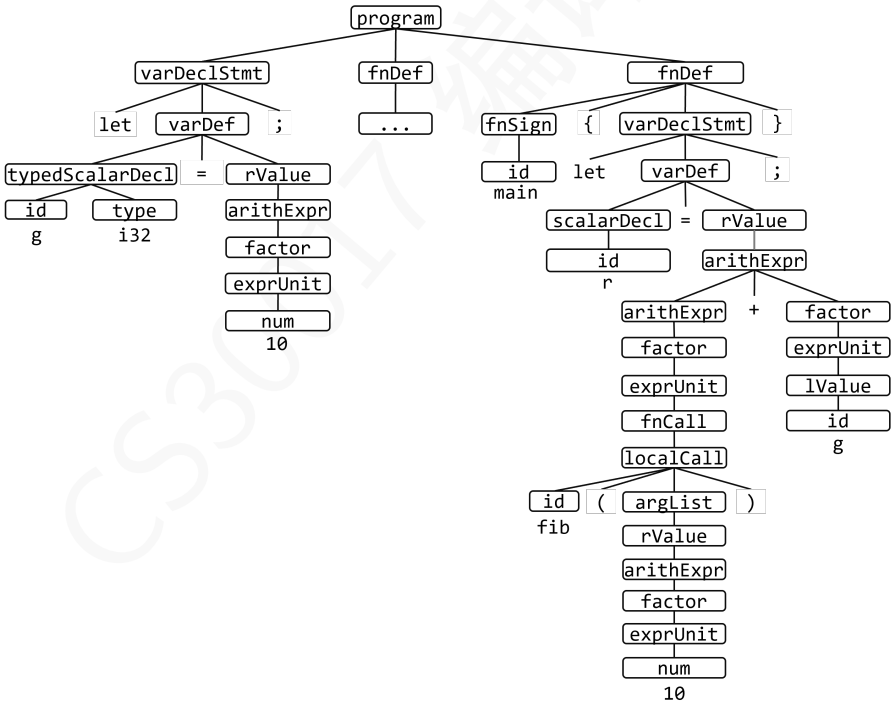

图 6.1: 代码 6.1对应的语法解析树示例（省略了 fib 函数体）

抽象语法树（Abstract Syntax Tree，AST）是语法解析树化简后的树形中间代码表示形式，在整个编译过程中可能会被多次编辑，记录代码分析和编译过程的中间结果。图 6.2展示了代码 6.1对应的一种 AST 表示。该 AST 去除了语法解析树中的括号和分号等冗余节点，并且对单一展开形式（只有一个孩子节点）的情况进行了合并处理，如将 `rValue->arithExpr->factor->exprUnit->num` 缩短为 `num`。

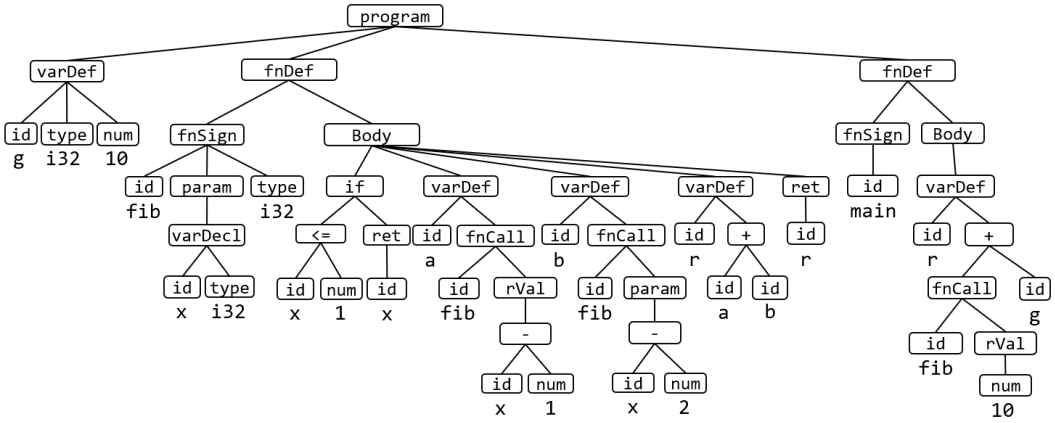

图 6.2: 代码 6.1对应的抽象语法树示例

6.3 标识符索引化

标识符索引化的目的是解决代码中的标识符指代问题。当标识符的指代对象唯一时，该问题较易处理，可以直接将其声明与使用建立对应关系；当标识符的指代对象不唯一时，则需要结合作用域等代码上下文信息进行消歧，并通过符号解析确定其具体指代对象。

Tea 语言对全局标识符在文件中的声明与使用顺序不作限制，但对象标识符的命名需遵循如下规则：

- 两个同名局部变量的作用域不能存在交集；
- 局部变量可以与全局标识符（包括变量和函数）同名。

标识符索引化的输出结果包括若干符号表以及索引化后的抽象语法树。需要注意的是，该符号表可能不包含完整的类型信息；从本质上看，类型推导过程可以视为作为标识符补充类型标注的过程。

6.3.1 创建符号表

符号表记录所有标识符的作用域和已知类型信息，每一行对应一个索引项。通过扫描 AST 中的变量和函数声明（或定义）节点，可以直接得到符号表。

```
let g:i32 = 10;
fn fib(x:i32) -> i32 { // scope fib
    if (x <= 1) {
        return x;
    }
    let a = fib(x - 1); // {  scope 1
        let b = fib(x - 2); // { scope 2
            let r = a + b; // { scope 3
                return r;
            // }
        // }
    // }
}
fn main() { // scope main
    let r = fib(10) + g; // { scope 1
    // }
}
```

代码 6.2: Tea 语言代码示例：补充了标识符作用域标注

符号表通常分为全局符号表和局部变量符号表。以代码 6.2 为例，其符号表包括一个全局符号表（见表 6.2 ）以及两个函数对应的局部变量符号表（见表 6.3 和表 6.4 ）。需要注意的是，函数参数本质上也属于局部变量的范畴。此外，在构建符号表时，无需考虑变量或函数的使用节点，仅需处理其声明信息。

表 6.2: 代码 6.2 对应的全局符号表

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

表 6.3: 代码 6.2 中 main 函数对应的局部变量符号表

标识符	作用域 (辅助信息)	索引	类型
r	main:scope1	0x1000	未知

表 6.4: 代码 6.2 中 fib 函数对应的局部变量符号表

标识符	作用域 (辅助信息)	索引	类型
x	fib	0x1100	i32
a	fib:scope1	0x1101	未知
b	fib:scope2	0x1102	未知
r	fib:scope3	0x1103	未知

6.3.2 添加标识符索引

该步骤为 AST 中的每个标识符添加索引信息。在实际编译器实现中，该过程通常与符号表的构建同步进行：在遇到标识符声明时创建新的索引，在遇到标识符引用时则关联已有索引。

图 6.3 对该问题进行了抽象表示，其中红色节点表示局部变量的声明，蓝色节点表示标识符的引用。索引化的难点在于需要严格结合标识符的作用域进行分析。

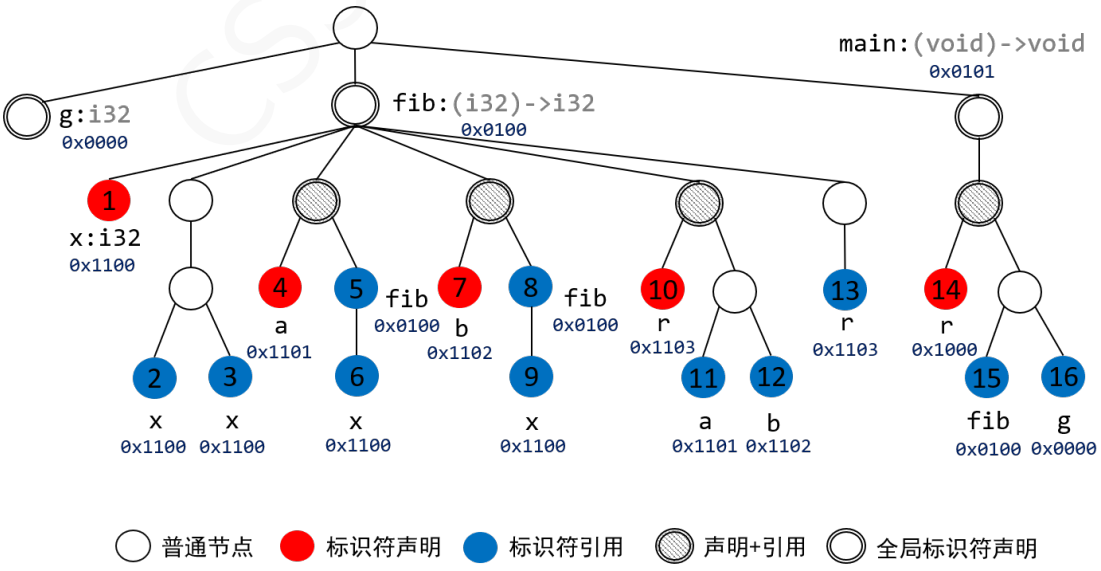

图 6.3: 标识符索引化问题

假设全局符号表已经创建完毕，算法 1 给出了对于函数内部的标识符进行索引化的具体思路。其核心思想是为每个函数维护一个标识符字典 `dict`，用于记录当前作用域中可见的标识符。在遍历 AST 时，对每个中间节点维护一个局部字典 `subdict`，用于记录该子树中新声明的标识符；当遍历退出该节点作用域时，需要将 `subdict` 中的标识符从 `dict` 中移除，从而恢复外层作用域状态。

算法 1 函数局部变量标识符索引化算法

Input: AST root of a function; Global symbol table: *gdict*

let *dict* = *gdict* // all usable identifiers of the function
procedure INDEXING(*cur*)
    subdict = ∅; // identifiers defined in the current subtree;
    for each *child* ∈ *cur.children* **do** // left to right visit in order;
        match *child.type* :
            case VarDecl ⇒ // declaration node
                dict.add(*child.id*); // add to the dictionary; If already existed, report error;
                subdict.add(*child.id*); // add to the sub dictionary;
            case VarRef ⇒ // reference node
                child.refid.index = *dict*.getIndex(*child.refid*); //this step may fail; or return none if not existed;
            case VarDeclRef ⇒ // declaration and reference that may reference multiple vars, e.g., d = a + b;
                for *refid* ∈ *child.refids* **do**
                    refid.index = *dict*.getIndex(*refid*); //this step may fail; or return none if not existed;
                end for
                dict.add(*child.id*); // add to the dictionary; If already existed, report an error;
                subdict.add(*child.id*); // add to the sub dictionary;
            case OtherLeafNode ⇒ // other leaf node that has no identifiers
                Continue;
            case NonLeafNode ⇒ // for intermediate nodes: recursively indexing the subtree;
                Indexing(*child*);
        end match
    end for
    for each *entry* ∈ *subdict* **do** // remove the identifiers defined in the current subtree;
        dict.remove(entry);
    end for
end procedure

6.4 类型约束和求解

接下来，需要为符号表中缺省类型的标识符确定具体类型。常用的方法是基于约束求解的 Hindley-Milner 类型推导方法 [1]。该方法主要包括以下三个步骤：

1) 为不同语句类型定义相应的类型约束规则；

2) 根据上述规则从程序中提取类型约束；

3) 对类型约束进行求解，从而确定标识符的具体类型。

6.4.1 类型约束规则

表 6.5 定义了 Tea 语言语言的主要类型约束规则。

表 6.5: Tea 语言中的主要类型约束规则

代码模式	**类型约束**	**含义**
`X: Ty`	$[\![X]\!] = \text{Ty}$	声明 X 的类型为 Ty
`I`	$[\![I]\!] = \text{i32}$	数字类型为 i32
`X[I]: Ty`	$[\![X]\!] = \&\text{Ty}$, $[\![I]\!] = \text{i32}$	声明 X 数组的类型为 &Ty
`{I1,...,In}`	$[\![I_1, ..., I_n]\!] = \&\text{i32}$	数组类型为 &i32
`{I; N}`	$[\![I; N]\!] = \&\text{i32}$	数组类型为 &i32
`X = Y`	$[\![X]\!] = [\![Y]\!]$	等号左右节点类型相同
`X = Y[Z]`	$[\![Z]\!] = \text{i32}$, $[\![X]\!]=[\![*Y]\!]$, $[\![Y]\!]=\&[\![*Y]\!]$	数组解引用作为右值
`X[Z] = Y`	$[\![Z]\!] = \text{i32}$, $[\![X]\!]= \&[\![Y]\!]$	数组解引用作为左值
`X bArithOp Y`	$[\![X]\!] = [\![Y]\!] = [\![X\ bArithOp\ Y]\!]$	二元算数运算操作数和运算结果同类型
`X bRelOp Y`	$[\![X]\!] = [\![Y]\!]$, $[\![X\ bRelOp\ Y]\!] = \text{bool}$	二元关系运算操作数同类型，结果为 bool
`if(X) {...}`	$[\![X]\!] = \text{bool}$	条件为 bool
`while(X) {...}`	$[\![X]\!] = \text{bool}$	条件为 bool
`X bLogOp Y`	$[\![X]\!] = [\![Y]\!] = [\![X\ bLogOp\ Y]\!] = \text{bool}$	二元逻辑运算操作数和结果均为 bool
`uLogOp X`	$[\![X]\!] = [\![uLogOp\ X]\!] = \text{bool}$	一元逻辑运算操作数和结果均为 bool

`F(X: Ty1)-> Ty2 {`
`  return Y;`
`}`
	$[\![F]\!] = (\text{Ty1}) \rightarrow \text{Ty2}$, $[\![Y]\!] = \text{Ty2}$	函数声明/定义涉及的类型约束

`F(X)`	$[\![F]\!] = ([\![X]\!]) \rightarrow [\![F(X)]\!]$	函数调用的类型约束

`struct ST {`
`  A:Ty1,`
`  B:Ty2`
`}`
	$[\![ST]\!] = (\text{Ty1, Ty2})$	结构体类型

`X.A = Y`	$[\![X.A]\!] = [\![Y]\!]$, $[\![X]\!] = [\![X.A, \_]\!]$	结构体 field 类型

注：符号 $[\![X]\!]$ 表示标识符 X 的类型

将上述规则应用到代码 6.1 的 AST 中，可以得到类型约束。

$$
\begin{aligned}
&\texttt{let g:i32 = 10}\\
&\Rightarrow [\![0x0000]\!] = [\![10]\!],\ [\![0x0000]\!] = i32,\ [\![10]\!] = i32\\
&\texttt{fn fib(x:i32) -> i32}\\
&\Rightarrow [\![0x0100]\!] = (i32) \rightarrow i32,\ [\![0x1100]\!] = i32\\
&\texttt{if (x <= 1)}\\
&\Rightarrow [\![0x1100 <= 1]\!] = bool,\ [\![0x1100]\!] = [\![1]\!],\ [\![1]\!] = i32\\
&\texttt{return x}\\
&\Rightarrow [\![0x1100]\!] = i32\\
&\texttt{let a = fib(x - 1)}\\
&\Rightarrow [\![0x1101]\!] = [\![0x0100(0x1100 - 1)]\!]\\
&\Rightarrow [\![0x0100]\!] = ([\![0x1100 - 1]\!]) \rightarrow [\![0x0100(0x1100 - 1)]\!]\\
&\Rightarrow [\![0x1100 - 1]\!] = [\![0x1100]\!] = [\![1]\!]\\
&\texttt{let b = fib(x - 2)}\\
&\Rightarrow [\![0x1102]\!] = [\![0x0100(0x1100 - 2)]\!]\\
&\Rightarrow [\![0x0100]\!] = ([\![0x1100 - 2]\!]) \rightarrow [\![0x0100(0x1100 - 2)]\!]\\
&\Rightarrow [\![0x1100 - 2]\!] = [\![0x1100]\!] = [\![2]\!]\\
&\texttt{let r = a + b}\\
&\Rightarrow [\![0x1103]\!] = [\![0x1101 + 0x1102]\!]\\
&\Rightarrow [\![0x1101 + 0x1102]\!] = [\![0x1101]\!] = [\![0x1102]\!]\\
&\texttt{return r}\\
&\Rightarrow [\![0x1103]\!] = i32\\
&\texttt{fn main()}\\
&\Rightarrow [\![0x0101]\!] = (void) \rightarrow void\\
&\texttt{let r = fib(10) + g}\\
&\Rightarrow [\![0x1000]\!] = [\![0x0100(10) + 0x0000]\!]\\
&\Rightarrow [\![0x0100(10) + 0x0000]\!] = [\![0x0100(10)]\!] = [\![0x0000]\!]\\
&\Rightarrow [\![0x0100]\!] = ([\![10]\!]) \rightarrow [\![0x0100(10)]\!]
\end{aligned}
\tag{6.1}
$$

由于上述类型约束均为等价关系，可采用并查集方法进行求解，从而得到：$[\![0x1000]\!] = i32$, $[\![0x1101]\!] = i32$, $[\![0x1102]\!] = i32$, $[\![0x1103]\!] = i32$。若类型系统中引入子类型或范型机制，则类型约束关系将由等价关系扩展为包含关系，此时需要采用更一般的约束求解方法。

需要注意的是，类型推导不一定总是存在解。例如，当代码中出现结构体递归定义时，通常会导致约束不可解，此时应报告类型推导错误。若类型约束集合存在解，则说明程序是可类型的（typable）；否则，编译器应报告类型错误，或在语言允许的情况下通过隐式类型转换消解部分不一致。因此，类型推导在一定程度上也实现了类型检查；而类型检查可以视为在类型已知前提下的类型推导特例。

参考文献

[1] Robin Milner. “A theory of type polymorphism in programming.” *Journal of Computer and System Sciences*, 1978.

练习

1) 为什么要基于 AST 而非源代码进行类型推导或检查?

2) 按步骤为下列 Tea 语言代码进行类型推导:

   (a) 画出 AST;

   (b) 创建符号表;

   (c) 提取类型约束并求解。

```
fn fac(n:i32) -> i32 {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    return r;
}
```

代码 6.3: Tea 语言代码

3) 思考: 若对 Tea 语言的类型系统规则进行扩展，以支持以下特性，应如何相应地修改类型推导方法?

- 允许同名局部变量的作用域发生重叠，并在标识符引用时优先选择作用域最内层的定义;
- 允许函数重名（函数重载），但要求其函数签名互不相同。