7　抽象语法树和类型检查

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 了解抽象语法树
- 基于 AST 进行类型检查
- 基于 AST 进行类型推导

7.1　抽象语法树

相对于语法解析树 (Parse Tree 或 Concrete Syntax Tree)，抽象语法树（Abstract Syntax Tree）是一种更精简的表示，一般只保留编译器后续分析所需要的内容。精简内容包括：1）去除括号等冗余符号节点；2）将运算符等一些叶子节点作为父节点的属性；3）单一展开形式塌陷，如 A->B->C->D 变为 A->D。

7.2　类型检查和推导

类型检查和推导是基于抽象语法树进行的，一般分为两个步骤：1）确定所有标识符的作用域，将变量引用关联到其声明信息（或索引化）；2）根据类型约束规则分析抽象语法树中所有标识符的类型，并检查类型的正确性。

7.2.1　变量索引

给定一个 AST，确定每个变量名的索引：1）变量声明 (varDecl) 时创建新索引；2）变量引用 (varRef) 时关联已创建索引。

图 7.1对该问题进行了抽象表示，其中红色节点表示声明一个变量，蓝色节点表示引用一个变量，紫色节点表示声明一个变量并使用其它变量的引用初始化该变量。上述有色节点在抽象语法树上都是叶子节点，另外还是有一些其它无颜色的节点。算法 1描述了如何对其中的变量进行索引。

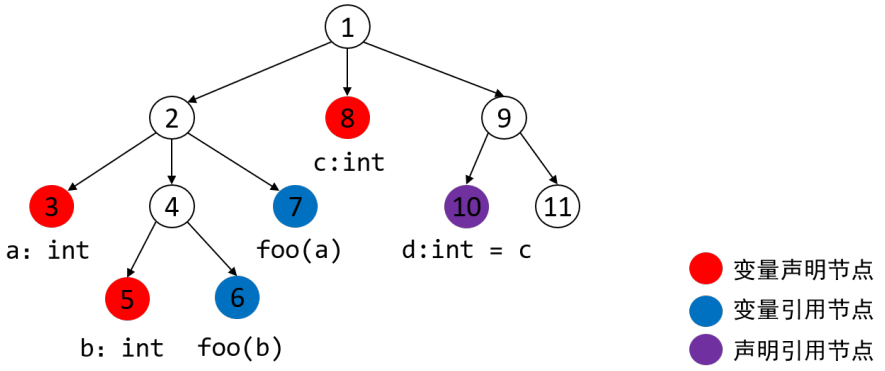

图 7.1: 变量索引问题举例

算法 1 变量索引算法

```
Input: AST root of a function;
let dict = ∅ // all variables of the function
procedure VarIndexing(root)
    let cur = root
    while cur do
        let children = cur.children;
        subdict = ∅; // variables defined in the current subtree;
        for child ∈ 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;
                    child.id.index = dict.getIndex(child.id); // obtain the unique index from the dict;
                    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 error;
                    child.id.index = dict.getIndex(child.id); // obtain the unique index from the dict;
                    subdict.add(child.id); // add to the sub dictionary;
                case OtherLeafNode ⇒ // other leaf node that has no idenfier
                    Continue;
                case NonLeafNode ⇒ // for intermidiate nodes: recursively indexing the subtree;
                    VarIndexing(child);
            end match
        end for
        for entry ∈ subdict do // remove the variables defined in the current subtree;
            dict.remove(entry);
        end for
    end while
end procedure
```

7.2.2 TeaPL 的类型约束规则

类型推导指的是为变量声明时缺省类型的情况分配具体类型；类型检查则是检查已知类型是否满足要求。这两种方法本质上都是根据语言的类型约束分析代码的类型信息。表 7.1定义了 TeaPL 语言的主要类型约束规则。

表 7.1: TeaPL 中的主要类型约束规则

代码语句	举例	AST 节点名称	类型约束	含义
赋值语句	x = y	assignStmt	[x] = [y]	左右子节点类型相同
二元算数运算	x + 1	arithBinOpExpr	[x] = [y] = int	左右子节点均为 int
一元算数运算	-x	exprUnit	[x] = int	子节点为 int
比较运算	x > y	cmpExpr	[x] = [y] = int	左右子节点均为 int
二元逻辑运算	x && y	boolBiOpExpr	[x] = [y] = bool	左右子节点均为 bool
一元逻辑运算	!x	boolUnit	[x] = bool	子节点为 bool
函数调用	y = foo(x)	fnCall	[foo] = ([x]) -> [foo(x)], [y] = [foo(x)]	参数和返回值类型与函数签名一致

注：符号 [x] 表示标识符 x 的类型