6 类型推导

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 了解抽象语法树
- 掌握类型约束规则设计方法
- 掌握类型推导或检查方法

6.1 概述

类型推导一般是基于抽象语法树进行的。抽象语法树（Abstract Syntax Tree）是相较于语法解析树(Parse Tree 或 Concrete Syntax Tree) 的一种更精简的树形中间代码，如去除括号等冗余节点，单一展开形式塌陷（如 A->B->C->D 变为 A->D）等。在编译过程中，AST 可能会被编译器多次更新或编辑，生成后续分析所需要的内容。

类型推导一般分为两个步骤：1）标识符索引化；2）根据类型规则提取 AST 中的所有类型约束并求解。类型检查可以认为是所有类型已知情况下的类型推导特例。

6.2 标识符索引化

标识符索引化的目的是创建符号表，维护所有变量和函数的类型信息；同时将 AST 中的所有的标识符索引化，关联到符号表的项目。

6.2.1 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。通过对 AST 进行扫描，识别其中的变量和函数定义节点扫描即可得到符号表。符号表的创建一般无需考虑变量的使用节点。如果某些变量是缺省类型，则等后续类型推导结束后进行填充。

符号表一般分为全局符号表和局部变量符号表。以代码 6.1为例，其符号表包括一个全局符号表和两个函数的局部变量符号表。

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

```
fn main() { //scope main
    let r = fib(10) + g; // { scope 1
    // }
}
```

代码 6.1: TeaPL 代码

表 6.1: 代码 6.1对应的全局符号表

标识符	作用域 (辅助信息)	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void

表 6.2: 代码 6.1中函数 fib 对应的局部变量符号表

标识符	作用域 (辅助信息)	索引	类型
fib	fib	0xd398	int
a	fib:scope1	0xd5b0	未知
b	fib:scope2	0xd2c2	未知
r	fib:scope3	0x1234	未知

表 6.3: 代码 6.1中函数 main 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	0x82d0	未知

6.2.2 添加标识符索引

该步骤为 AST 上的每个标识符添加索引信息。在实际编译器实现时，该步骤可以和符号表的创建一起进行，即 1）标识符声明时创建新索引；2）标识符引用时关联已创建索引。

下面以变量的标识索引为例分析标识符索引化算法。图 6.1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；另外还包括声明引用节点（使用其它变量对声明变量进行初始化）和其它普通节点。算法 1描述了标识符的索引化的过程。

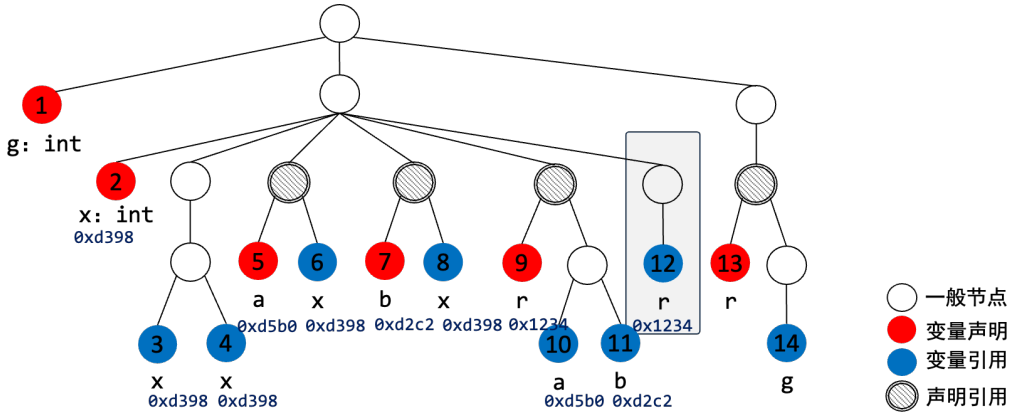

图 6.1: 变量索引问题举例

算法 1 变量索引算法

Input: AST root of a function;

let $dict = \emptyset$ // all variables of the function
procedure VarIndexing($root$)
    let $cur = root$
    while cur **do**
        let $children = cur.children$;
        $subdict = \emptyset$; // variables defined in the current subtree;
        for each $child \in children$ **do** // left to right visit in order;
            match $child.type$:
                case VarDecl $\Rightarrow$ // 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 $\Rightarrow$ // reference node
                    $child.refid.index = dict$.getIndex($child.refid$) //this step may fail; or return none if not existed;
                case VarDeclRef $\Rightarrow$ // declaration and reference that may reference multiple vars, e.g., d = a + b;
                    for $refid \in child.refids$ **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 $\Rightarrow$ // other leaf node that has no idenfier
                    Continue;
                case NonLeafNode $\Rightarrow$ // for intermidiate nodes: recursively indexing the subtree;
                    VarIndexing($child$);
            end match
        end for
        for each $entry \in subdict$ **do** // remove the variables defined in the current subtree;
            $dict$.remove(entry);
        end for
    end while
end procedure

6.3 类型约束和求解

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 6.4定义了 TeaPL 语言的主要类型约束规则。

表 6.4: TeaPL 中的主要类型约束规则

代码模式	代码示例	类型约束	含义
N	0	$[\![N]\!] = \text{int}$	数字类型为 int
{M,...,N}	1,2,3,4,5	$[\![M,...,N]\!] = \&\text{int}$	数组类型为 &int
X = Y	a = b	$[\![X]\!] = [\![Y]\!]$	等号左右节点类型相同
X = Y[Z]	a = b	$[\![Z]\!] = \text{int}$, $[\![X]\!]=[\![*Y]\!]$, $[\![Y]\!]=\&[\![*Y]\!]$	数组解引用作为右值
X[Z] = Y	a = b	$[\![Z]\!] = \text{int}$, $[\![X]\!]= \&[\![Y]\!]$	数组解引用作为左值
X binArithOp Y	a + b	$[\![X]\!] = [\![Y]\!] = [\![X\ binArithOp\ Y]\!]$	二元算数运算操作数和运算结果类型相同
X binRelOp Y	a > b	$[\![X]\!] = [\![Y]\!]$, $[\![X\ binRelOp\ Y]\!] = \text{bool}$	二元关系运算操作数类型相同，结果为布尔类型
if(X)	if(a>b)	$[\![X]\!] = \text{bool}$	条件语句类型为布尔类型
while(X)	while(a>b)	$[\![X]\!] = \text{bool}$	条件语句类型为布尔类型
X binLogOp Y	a && b	$[\![X]\!] = [\![Y]\!] = [\![X\ binLogOp\ Y]\!] = \text{bool}$	二元逻辑运算操作数和结果类型均为布尔类型
uniLogOp X	!b	$[\![X]\!] = [\![uniLogOp\ X]\!] = \text{bool}$	一元逻辑运算操作数和结果类型均为布尔类型
F(X,Y)	foo(a, b);	$[\![F]\!] = ([\![X]\!],[\![Y]\!]) \rightarrow [\![F(X,Y)]\!]$	函数调用的类型约束
F(X)->Y {...; ret Z;}	ret a;	$[\![F]\!] = ([\![X]\!]) \rightarrow [\![Z]\!]$, $[\![Z]\!] = [\![Y]\!]$	函数返回语句的类型约束
struct ST { A:int, B:int }	struct Foo { a:int, b:int };	$[\![ST]\!] = [\![int, int]\!]$	结构体类型
X.A = Y	foo.a = b	$[\![X.A]\!] = [\![Y]\!]$, $[\![X.A,\ \_]\!] = [\![X]\!]$	结构体类型

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。以函数 fib 为例，其类型约束模型如下：

$$
\begin{aligned}
&[\![0xd9c2]\!] = [\![10]\!], [\![10]\!] = int,\\
&[\![0xd5b0]\!] = [\![0xd2c2]\!] = [\![0xd5b0 + 0xd2c2]\!] = [\![0x1234]\!],\\
&[\![0xd398]\!] = [\![1]\!], [\![1]\!] = int, [\![0xd398 <= 1]\!] = bool,\\
&[\![0xd398]\!] = [\![1]\!] = [\![0xd398 - 1]\!],\\
&[\![0xd470]\!] = ([\![0xd398 - 1]\!]) \rightarrow [\![0xd470(0xd398 - 1)]\!], [\![0xd470(0xd398 - 1)]\!] = [\![0xd5b0]\!]\\
&[\![0xd470]\!] = ([\![0xd398 - 2]\!]) \rightarrow [\![0xd470(0xd398 - 2)]\!], [\![0xd470(0xd398 - 1)]\!] = [\![0xd2c2]\!],\\
&[\![0xd470]\!] = ([\![0xd398]\!]) \rightarrow 0x1234
\end{aligned}
\tag{6.1}
$$

将符号表中的类型约束加入到类型约束模型中：

$$
\begin{aligned}
&[\![0xd9c2]\!] = int\\
&[\![0xd470]\!] = (int) \rightarrow int\\
&[\![0xd318]\!] = (int) \rightarrow void\\
&[\![0xd398]\!] = int
\end{aligned}
\tag{6.2}
$$

由于上述类型约束关系都是等价关系，因此可采用并查集方法得到 $[\![0xd5b0]\!] = \text{int}$，$[\![0xd2c2]\!] = \text{int}$，$[\![0x1234]\!] = \text{int}$。如果类型系统中包括子类型或范型，则类型约束关系为包含关系。