6 类型推导

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 了解抽象语法树
- 掌握标识符索引化方法
- 掌握类型规则设计和应用方法

6.1 TeaPL 的类型系统

TeaPL 采用静态类型系统，即所有标识符的类型应在编译时确定。类型系统由类型和规则组成。TeaPL 中的基础类型包括标量类型 int 和 bool、复合类型数组、以及函数类型。另外，用户也可以使用 struct 自定义数据类型。

TeaPL 的类型规则主要包括以下几条：

- 所有函数在声明时必须明确参数和返回值类型，不能缺省；对变量声明则不做要求。
- 相同标识符的作用域不能有交集或存在包含关系，如同一函数内的局部变量重名（如代码 6.1中的变量 x）或局部变量与全局变量重名的情况。
- 对于全局标识符在文件中的声明和引用出现顺序不做要求。

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

代码 6.1: 类型错误举例：x 被重复声明

6.2 类型推导问题定义

由于 TeaPL 中的标识符类型在声明时是可以缺省的，需要确定其具体类型才可以进行后续的编译，该问题称为类型推导。类型推导需要考虑变量使用的上下文限制，不一定有解。如果有解，则说明代码可类型（typable），否则编译器应提示类型错误或进行隐式类型转换。因此，类型推导也可以达到类型检查的效果；类型检查可以认为是类型已知情况下的类型推导特例。

这种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，缩写为 AST）相较于语法解析树 (Parse Tree 或 Concrete Syntax Tree) 是一种更精简的树形中间代码。AST 一般去除了语法解析树中的括号等冗余节点，并且对单一展开形式（只有一个孩子节点）的情况进行了塌陷处理，如

将 A->B->C->D 缩短为 A->D。AST 在整个编译过程中可能会被编译器多次编辑，记录更新代码编译过程的中间结果。

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。

6.3 标识符索引化

标识符索引化的目的是对标识符去重，解决代码中存在标识符名字相同，但指代对象不同的问题。这一步的输出结果是缺少类型信息的符号表，以及索引化后的 AST。类型推导本质上是为去重后的标识符确定类型。

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，其中每一行为一条索引。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。创建符号表时无需考虑变量的使用节点。如果某些变量是缺省类型，待后续类型推导时再进行填充。

符号表一般分为全局符号表和局部变量符号表。以代码 6.3为例，其符号表包括一个全局符号表(表 6.1)和两个函数的局部变量符号表（表 6.3和 6.2)。

```
let g: int = 10;
fn fib(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 6.2: TeaPL 代码

表 6.1: 代码 6.3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 6.3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
x	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 6.3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

6.3.2 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，即在遇到标识符声明时创建新索引；遇到标识符引用时关联已创建索引。

假设全局标识符都已经具备索引，下面以函数内部的标识符索引问题为例阐述一种标识符索引化算法。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局部变量，蓝色节点表示引用一个标识符；另外还包括声明 + 引用的情况，即使用其它标识符对新声明的变量进行初始化。算法 1描述了标识符的索引化的过程。其主要思路是为每个函数维护一个标识符字典 `dict`，记录当前节点可用的标识符。在遍历 AST 时为每一个中间节点都维护一个字典 `subdict`，记录当前子树中声明的标识符，跳出该节点作用域时应将其子树中声明的标识符从 `dict` 移出。

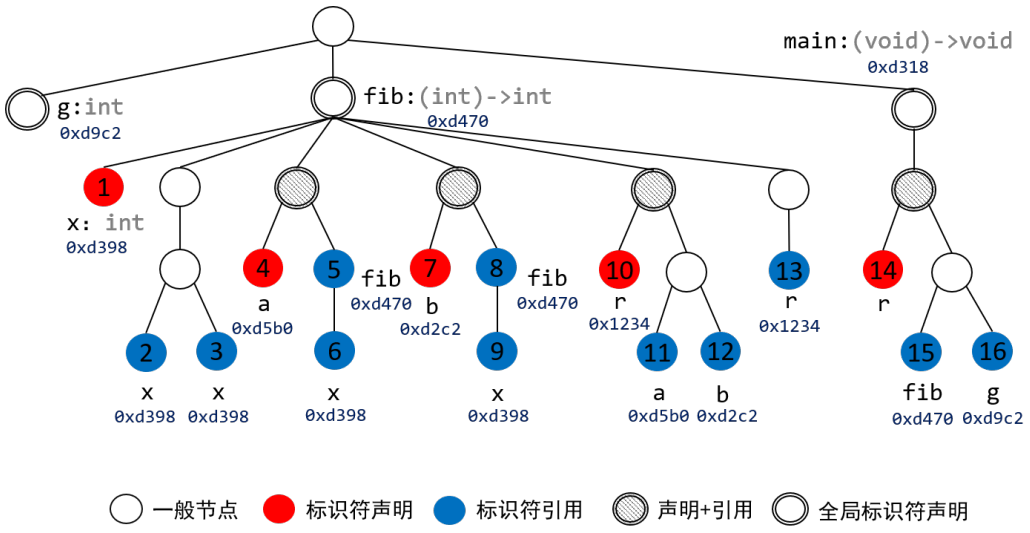

图 6.1: 变量索引问题举例

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

类型推导指的是为缺省类型的标识符分配具体类型。常用的类型推导方法是基于约束求解的 Hindley-Milner 方法 [1, 2]。这种方法首先对不同的标识符使用模式建立不同的约束提取规则，通过对代码提取类型约束并求解来确定标识符类型；如果无解则说明存在类型错误，需要进行隐式类型转换或直接报错。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	类型约束	含义
`X: Ty`	$[\![X]\!] = \text{Ty}$	声明 X 的类型为 Ty
`I`	$[\![I]\!] = \text{int}$	数字类型为 int
`X[I]: Ty`	$[\![X]\!] = \&\text{Ty}$, $[\![I]\!] = \text{int}$	声明 X 数组的类型为 &Ty
`{I1,...,In}`	$[\![I_1, ..., I_n]\!] = \&\text{int}$	数组类型为 &int
`{I; N}`	$[\![I; N]\!] = \&\text{int}$	数组类型为 &int
`X = Y`	$[\![X]\!] = [\![Y]\!]$	等号左右节点类型相同
`X = Y[Z]`	$[\![Z]\!] = \text{int}$, $[\![X]\!]=[\![*Y]\!]$, $[\![Y]\!]=\&[\![*Y]\!]$	数组解引用作为右值
`X[Z] = Y`	$[\![Z]\!] = \text{int}$, $[\![X]\!]= \&[\![Y]\!]$	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类型相同
`X bRelOp Y`	$[\![X]\!] = [\![Y]\!]$, $[\![X\ bRelOp\ Y]\!] = \text{bool}$	二元关系运算操作数类型相同，结果为 bool
`if(X) {...}`	$[\![X]\!] = \text{bool}$	条件语句类型为布尔类型
`while(X) {...}`	$[\![X]\!] = \text{bool}$	条件语句类型为布尔类型
`X bLogOp Y`	$[\![X]\!] = [\![Y]\!] = [\![X\ bLogOp\ Y]\!] = \text{bool}$	二元逻辑运算操作数和结果类型均为布尔类型
`uLogOp X`	$[\![X]\!] = [\![uLogOp\ X]\!] = \text{bool}$	一元逻辑运算操作数和结果类型均为布尔类型
`F(X: Ty1)-> Ty2 {...ret Y;}`	$[\![F]\!] = (\text{Ty1}) \to \text{TY2}$, $[\![Y]\!] = \text{Ty2}$	函数定义和返回语句的类型约束
`F(X)`	$[\![F]\!] = ([\![X]\!]) \to [\![F(X)]\!]$	函数调用的类型约束
`struct ST {A:Ty1, B:Ty2}`	$[\![ST]\!] = (\text{Ty1}, \text{Ty2})$	结构体类型
`X.A = Y`	$[\![X.A]\!] = [\![Y]\!]$, $[\![X]\!] = [\![X.A,\ \_]\!]$	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 6.3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：

$$
\begin{aligned}
&[\![0xd9c2]\!] = int,\ [\![0xd9c2]\!] = [\![10]\!],\ [\![10]\!] = int \\
&[\![0xd470]\!] = (int) -> int,\ [\![0xd398]\!] = int,\ [\![0x1234]\!] = int \\
&[\![0xd398]\!] = [\![1]\!],\ [\![1]\!] = int,\ [\![0xd398 <= 1]\!] = bool \\
&[\![0xd5b0]\!] = [\![0xd470(0xd398 - 1)]\!],\ [\![0xd470]\!] = ([\![0xd398 - 1]\!]) \to [\![0xd470(0xd398 - 1)]\!] \\
&[\![0xd398]\!] = [\![1]\!] = [\![0xd398 - 1]\!] \\
&[\![0xd2c2]\!] = [\![0xd470(0xd398 - 2)]\!],\ [\![0xd470]\!] = ([\![0xd398 - 2]\!]) \to [\![0xd470(0xd398 - 2)]\!] \\
&[\![0xd398]\!] = [\![2]\!] = [\![0xd398 - 2]\!] \\
&[\![0x1234]\!] = [\![0xd5b0]\!] = [\![0xd2c2]\!] = [\![0xd5b0 + 0xd2c2]\!] \\
&[\![0xd318]\!] = (void) -> void \\
&[\![0x1234]\!] = [\![0xd470(10) + 0xd9c2]\!] = [\![0xd470(10)]\!] = [\![0xd9c2]\!]
\end{aligned}
\tag{6.1}
$$

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。

练习

1. 为下列代码进行类型推导。

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

代码 6.3: TeaPL 代码

2. 如果改变一下 TeaPL 的类型系统设计：“允许两个相同的标识符作用域存在包含关系，出现标识符引用时以作用域最小的标识符为准”，应如何修改类型推导方法？

Bibliography

[1] Roger Hindley. “The principal type-scheme of an object in combinatory logic.” Transactions of the american mathematical society 146 (1969): 29-60.

[2] Robin Milner. “A theory of type polymorphism in programming.” Journal of computer and system sciences 17, no. 3 (1978): 348-375.