PA3报告

韩晓宇18300750006

刘明君18307130064

1 代码核心逻辑

1.1 任务内容

这次PA中，我们需要实现下面三个内容：

- 建立所有类型的继承图；
- 判断继承图是否合法；
- 对于每个类型，遍历抽象语法树，将定义的对象加入到符号表中，进行类型检查并将类型记录到抽象语法树中。

1.2 辅助工具

这次PA提供的代码中自带了一些辅助工具供我们调用：

1. 从之前的PA中的抽象语法树输出及语法树的类；
2. 这次PA中新提供了一个符号表的实现，可以用来维护变量环境。

1.3 代码结构

```
void program_class::semant()
{
    initialize_constants();

    /* ClassTable constructor may do some semantic analysis */
    ClassTable *classtable = new ClassTable(classes);
    classtable_g = classtable;
    if (classtable->errors()) bad ();

    /* some semantic analysis code may go here */

    //check inheritance cycle
    classtable -> checkcycle ();
    if (classtable->errors()) bad ();

    classtable -> install_features ();
    classtable -> checktype ();

    if (classtable->errors()) bad ();
}
```

在`semant.cc`中，在进行静态语法分析时，首先会进入`program_class::semant`函数，我们需要实现的就是对`classtable`进行初始化，记录下所有的类型。之后判断继承图是否构成一个以`Object`类为根的树。最后处理所有类型的属性和方法，判断这些表达式的类型和声明类型是否兼容。

1.4 核心思路

在本次PA中，如果语法树是没有错误的，则工作非常简单，只需要遍历语法树，将每个类型进行记录即可。而复杂的部分在于有类型错误的情况，例如安装类型时需要注意类型名不能重复、不能和基本类型相同、不能继承部分基本类型，在安装属性和方法时，需要注意每个对象的类型是否是合法类型、是否是`SELF_TYPE`，每个方法的形式参数数量是否和调用时给出的参数数量对应、形式参数类型与实际推断类型是否对应，这些类型是否合法等。

1.4.1 `ClassTable`构造函数

在类`ClassTable`中，我们额外定义了一个`std::map < Symbol , Class_ > classmap`，将类型的名字映射到这个类型的`CLASS_`对象上，方便用名字查找类型。

这部分中首先需要将基本类型`Object`，`IO`，`Int`，`String`，`Bool`进行安装，可以直接利用自带的演示函数`ClassTable::install_basic_classes`。

之后遍历初始化时给出的`classes`，判断每个类型的名字是否是基本类型，是否已经定义过了，继承的父类型是否是基本类型，是否是未定义的类型。如果有问题，则报错 。

如果没有问题，则将类型插入`classmap`中，并在父类型中维护`std::vector<Class_> child`来记录这个类型的所有子类型，方便从根类型`Object`进行便利所有的类型。

1.4.2 `checkcycle`函数

这个函数用来判断继承图是否合法，这里为了方便我们只需要枚举所有的类型，每次访问父类型，直到访问到`Object`，则合法；如果访问的次数和所有类型的数量相同，则说明继承图出现了环，对这个类型进行报错。

这个做法的时间复杂度为$O(n^2 \log n)$，其中n为类型的数量，注意到代码的类型不会很多，并且为了实现方便，选择这种简易但较慢的方法可以接受。

1.4.3 `install_feature`函数

这个函数首先预处理所有的属性和方法，判断基础的格式问题，并将这些属性和方法插入到类型的`std::map<Symbol,Feature_class*> methodmap,attrmap`中，来方便下一步类型检查时维护变量环境。

为了处理继承的类型，我们需要先处理父类型，再处理子类型，这样才能进行属性和方法的继承。我们使用了广度优先搜索的方法，从`Object`开始处理所有的类型。

对于每个类型，首先将父类型的`methodmap,attrmap`继承过来，然后枚举每个属性和方法，维护当前类型的`methodmap,attrmap`。对于属性和方法，它们分别不能有重名属性或方法，对于属性来说，`self`是不合法的属性名。

- 如果是一个方法，先遍历它的形式参数表，要求形式参数的类型不能是`SELF_TYPE`，形式参数的名字也不能重复，`self`也不能是形式参数的名字。之后判断返回类型和形式参数类型，不能是未定义的类型。
  如果是继承的方法，需要有相同的形式参数数量，并且对应的形式参数类型和返回类型需要相同。
- 如果是一个属性，它的类型不能是未定义的类型。
  属性不能被重载。

在这时还会判断`Main`类型和`main`方法，要求`Main`类型存在，并且存在一个`main`方法，这个方法不能仅从父类型继承而来，且应该没有形式参数。

1.4.4 `checktype`函数

这个函数对每个类型进行类型检查。

我们维护了一个全局的符号表`SymbolTable<Symbol,Symbol> *stable`，在进入和退出环境是更新符号表。对于每个类型，首先将属性加入到符号表中，并将`self`加入。

这里需要递归处理每个语法树的节点，我们使用了多态的方式进行遍历。对于每个树节点，都有一个`checktype`方法，但是不同类型的节点会各自重载自己的方法，进行相应的处理，并返回此节点的类型。

在属性和方法中，检查是否和声明类型相同。检查方法的类型之前，需要将所有的形式参数加入到环境中。

每类节点的具体处理方式参考`manual`的第12节。这里只讨论其中共性的问题。

如果所有的类型都符合要求，则需要返回正确的类型即可。而不符合类型时，关键有如下几种情况：

- 调用了未定义的对象、类型、方法等；
- 调用方法的参数数量不对；
- 在需要满足$T \leq T'$的继承判断中出现问题；
- 逻辑运算、算术运算是需要满足特定的类型`Bool`，`Int`等；
- 引入新对象时，`self`是非法的变量名；
- 使用类型时，有时`SELF_TYPE`不能使用；
- 在`Case`中，不能出现重复的类型。

这里需要特别说明的是判断两个类型的继承关系、求两个类型的最近公共祖先的问题。

1.4.4.1 继承关系

假设需要判断$T \leq T'$是否成立。

如果某个类型未定义，则返回`true`。

如果都是`SELF_TYPE`，返回`true`。

如果期望的子类型T是`SELF_TYPE`，这等价于将T赋值为当前的类型。

如果期望的父类型T'是`SELF_TYPE`，返回`false`。

在实现中，不断访问T的父类型，直到访问到`Object`或者T'进行判断。

和继承图判断类似，这种方法的时间复杂度足够应对需求。

1.4.4.2 求两个类型的最近公共祖先

在`let`和`case`语句中需要求类型的公共祖先。

如果某个类型未定义，返回另一个类型。

如果两个都是`SELF_TYPE`，返回`SELF_TYPE`。

如果某个是`SELF_TYPE`，则令这个类型为当前类型。

计算时枚举两个类型到`Obejct`的继承链，找到最深的那一个。

同理，时间复杂度可以接受。

2 测试编写思路

实现了一个`checker.py`可以快速测试所有代码：

```
import os
import subprocess
path = os.getcwd()

files = subprocess.run ( ['ls' , path + '/tests/bad' ] , capture_output =
True).stdout.decode().split()
print ( "cases: " , files )
for case in files:
    out = subprocess.getoutput ( './mysemant ' + path + '/tests/bad/' + case )
    ans = subprocess.getoutput ( './stdsemant ' + path + '/tests/bad/' + case )
```

```
    dout = set(out.split('\n'))
    dans = set(ans.split('\n'))
    print ( case + ":" )
    print ( "errors only in your output:" )
    for item in dout:
        if item not in dans:
            print ( item )
    print ( "errors only in answer:" )
    for item in dans:
        if item not in dout:
            print ( item )
    print ( "" )

files = subprocess.run ( ['ls' , path + '/tests/good' ] , capture_output =
True).stdout.decode().split()
print ( "cases: " , files )
for case in files:
    out = subprocess.getoutput ( './mysemant ' + path + '/tests/good/' + case )
    ans = subprocess.getoutput ( './stdsemant ' + path + '/tests/good/' + case
)
    if out != ans :
        print ( case , "wrong:" )
        print ( "your output:" )
        print ( out )
        print ( "answer:" )
        print ( ans )
    else:
        print ( case , " passed" )
```

运行结果如下：

```
^[[A(base) meow@meow:~/compile/student-dist/assignments/PA4$ python checker.py
cases:  ['bad10.cl', 'bad11.cl', 'bad12.cl', 'bad13.cl', 'bad14.cl', 'bad15.cl', 'bad16.cl', 'bad
17.cl', 'bad18.cl', 'bad19.cl', 'bad1.cl', 'bad20.cl', 'bad21.cl', 'bad22.cl', 'bad2.cl', 'bad3.c
l', 'bad4.cl', 'bad5.cl', 'bad6.cl', 'bad7.cl', 'bad8.cl', 'bad9.cl']
bad10.cl:
errors only in your output:
errors only in answer:

bad11.cl:
errors only in your output:
errors only in answer:

bad12.cl:
errors only in your output:
errors only in answer:

bad13.cl:
errors only in your output:
errors only in answer:

bad14.cl:
errors only in your output:
errors only in answer:

bad15.cl:
errors only in your output:
errors only in answer:

bad16.cl:
errors only in your output:
errors only in answer:

bad17.cl:
errors only in your output:
errors only in answer:

bad18.cl:
errors only in your output:
errors only in answer:

bad19.cl:
errors only in your output:
errors only in answer:

bad1.cl:
errors only in your output:
errors only in answer:
```

2.1 good.cl

包括了所有`example`中给出的样例代码。分析器应该输出语法树，并且将每个表达式的类型进行标注。

2.2 bad.cl

包含了各种不能通过类型检查的测试。分析器应输出所有的类型错误，但是在某些时候（例如类型名重复定义等）可以提前停止编译。

`bad1.cl`为网上找到的测试文件。

`bad2.cl`测试了类型的名字不能是基本类型名，类型名字不能重复，部分基本类型不能被继承，不存在的类型不能被继承。

`bad3.cl`测试了类型的循环继承。

`bad4.cl`赋值时引用了未定义的属性和方法，赋值时的推断类型和声明类型不匹配。

`bad5.cl`测试了`self`作为属性名和方法名，属性和方法的重复定义。

`bad6.cl`测试了继承时类型、形式参数表的匹配问题。

`bad7.cl`测试了形式参数表的重复命名。

`bad8.cl`测试了声明类型为未定义类型的情况。

`bad9.cl`测试了未定义类型。

`bad10.cl`测试了未定义类型、`SELF_TYPE`在判断继承关系的表现。

`bad11.cl`测试了所有类型推断中可能出现的问题，特别是在表达式中，类型未定义、类型是`SELF_TYPE`的表现。

`bad12.cl`-`bad22.cl`测试了`Main`类型和`main`方法。