INF564 PROJECT

Mini Java Compiler

March 13, 2025

Gabriel Pereira de Carvalho

CONTENTS

1 TYPE CHECKING

1.1 Typing.java

1.1.1 • Step 1: Declare all classes and check for uniqueness

I defined a class `ClassesTable` as a wrapper to a `HashMap<String,Class_>`. In one linear scan of the parsed AST, I used this hash map to quickly look up class names and throw a typing error if uniqueness was not verified.

```
1     if(ClassesTable.lookup(className.id) != null){
2       error(className.loc, "Class name " + className.id + " is used for more
      than one class");
3       return null;
4     }
5
```

I created the `TDClass` objects

1.1.2 • Step 2: Declare inheritance relations, attributes, constructors and methods

In the file `InheritanceDAG.java` I define a directed graph data structure to model inheritance relations. In Step 1, I created a node for each class in this graph. In Step 2, I perform another linear scan and now that all classes/nodes are declared we can add directed edges (super class → inherited class) to our graph.

```
1     fatherClass_ = ClassesTable.lookup(fatherClassName.id);
2     if(fatherClass_ == null){
3       //class is inheriting from a non-existing class!
4       error(className.loc, "Class " + className.id + " inherits from non-
    existing class " + fatherClassName.id);
5       return null;
6     }
7     class_.extends_ = fatherClass_; //declare inheritance relation (TDclass
    in linked list should point to this updated object)
8     inheritanceDAG.addEdge(fatherClass_, class_);
9
```

When we process an inherited class, it is important that all its super classes have already been type checked because it is possible that we need to reuse attributes and methods. To do this, I implemented Kahn's topological sorting algorithm and all linear scans we do from now on are on this sorted list of classes. Kahn's algorithm also allows us to detected unwanted cycles in our graph, in this case we throw a typing exception.

Also in Step 2, we perform a linear scan of classes to declare attributes, constructors and methods. The first thing we do when we start processing a class here is build a stack with all the classes it inherits from. This way we can pop super classes from the stack one by one and add all attributes and methods to the current class's corresponding hash maps. Here it is important to notice that the hash maps will handle method overriding naturally, and then we add inherited methods to typed AST after all super classes are processed.

To declare attributes, constructors and methods without entering the body of constructors and methods; I added a boolean flag `goIntoBody` to the Visitor class. This way, the visit functions to attributes, constructors and methods can behave differently in this step only declaring and initializing top level objects without visting any statements.

```
1     MyVisitor.goIntoBodyFALSE(); //visitor will NOT enter body of
    constructors and methos
2     ListIterator<PDecl> it2 = classDecl.listIterator();
3     while(it2.hasNext()){
4       PDecl pdecl = it2.next();
5       pdecl.accept(myVisitor);
6     }
7
```

1.1.3 • Step 3: Type check the body of constructors and methods

In this last linear scan of the parsed AST, I set the flag flag `goIntoBody` to true and call the accept function again on all objects of the parsed AST.

1.2 MyVisitor.java

Because the visit functions are all void, I followed the following golden rules in all visit functions:

- after visiting a `PType`, a new `TType` object needs to be created and a static variable `ttype` must reference it.
- after visiting a `PExpr`, a new `TExpr` object needs to be created and a static variable `currentExpr` must reference it.
- after visiting a `PStmt`, a new `TStmt` object needs to be created and a static variable `ttype` must reference it.

Following these rules, after a call to the `accept` function I know where to retrieve the corresponding typed object to continue type checking.

1.2.1 • Auxiliary function: compatibilityTest

I realized that in my many parts of the code I needed to check that a certain `TExpr` was compatible with a `TType` so I created this boolean function. Unfortunately it makes heavy use of `instanceof` which I know is a bad practice, but at least this way it is localized to a specific portion of the code.

1.2.2 • Type checking Binop

I found a similar problem when type checking binops, I needed to check if two `TExpr` were compatible with a certain operation type but there were different operation types and many corner cases.

In the end, I defined 3 different types of operations: comparisons, logical and arithmetic. I created boolean flags with the compatibility conditions for each operations and used `instanceof` only to define these boolean flags.

1.2.3 • Auxiliary function: findParam

There is another auxiliary function, called `findParam` I used to search for an identifier inside a method or constructor. It was a function I need for visiting both `PDmethod` and `PDconstructor` so abstracting it was a way to simplify the code.

1.2.4 • Typing identifiers

To type check identifiers, I used the `java.util.regex` package. And also a `Set` data structure to verify that no keyword was used incorrectly.

```
1     static String identifierRegex = "[a-zA-Z_][a-zA-Z_0-9]*";
2     static Pattern identifierPattern = Pattern.compile(identifierRegex);
3     static boolean isIdentifierOk(String id){
4       Matcher mat = identifierPattern.matcher(id);
5       return mat.matches();
6     }
7
```

When typing `PEident` or `PEassignIdent` objects there are many possible corresponding `TExpr`, so these functions make many searches in hash maps and the order in which we search defines the priority of certain symbols over other. Here, I search for local variables before class attributes.

1.2.5 • Typing call expressions

First I want to mention that I treated the print instructions separately. In all functions were a print call visited, I used if/else statements to treat it as a special case so everything is hardcoded.

Another difficulty I faced was handling differents types of caller. For example, if the caller was an attribute, a variable or the class itself (this) the procedure to retrieve the `Method` object

was different so I used `instanceof` to differentiate between these cases. To check the parameters of a call, I checked the linked list sizes and used the compatibilityTest auxiliary function.

1.2.6 • Typing If and For statements

These statements are special because they can define a new scope of variables and sometimes they do not used the `PSblock` object. So before calling the `accept` function for statement bodies here, I created deep copies of the `variables` hash map and of the `hasReturnStatement` boolean flag.

This way, we can continue visiting inner objects modifying the deep copy associated to that specific scope and after, we can restore a backup stored in a local variable and move on.

In the for statement there is also the case where the loop condition is always false, which can make the code in the body unreachable. In one of the tests provided, the body would normally trigger a typing error but did not because it was unreachable. So treating this special case was important.

```
    variables = variablesIf; //we swap before entering each scope
    hasReturnStatement = false;
    s1.accept(this);
    ifHasReturnStatement = hasReturnStatement;
    TStmt ts1 = currentStmt;

    variables = variablesElse;
    hasReturnStatement = false;
    s2.accept(this);
    elseHasReturnStatement = hasReturnStatement;
    TStmt ts2 = currentStmt;

    variables = temp;//and unswap to move on at the end!
    hasReturnStatement = hasReturnStatementBackup || (ifHasReturnStatement
   && elseHasReturnStatement); //both paths must have returns!!
    currentStmt = new TSif(ifCondition, ts1, ts2);

```

2
CODE GENERATION

2.1 Compile.java

2.1.1 • Step 1: Build the class descriptors

Because the class names are unique by design, and method names are unique inside the class, I used this to build the label strings in order to avoid collisions. In Windows, I had a

naming conflict with WinGet for the name of my main class descriptor, so I ended removing the descriptor for main. I mention this in case you find it strange in the generated assembly code. Because main class has no attributes, I found it didn't affect the rest of the code.

```
1     ret.dlabel("descriptor_" + c.name); // add a labeled block in the data
      section
2     if(c.extends_ != null){// add ref to super class
3       ret.quad("descriptor_" + c.extends_.name);
4     }else{
5       ret.quad(0);
6     }
7     // now add label refs to each method !!
8     for(String methodName : c.methods.keySet()){
9       ret.quad(c.name + "_" + methodName);
10    }
11
```

2.1.2 • Step 2: Set the offsets of attributes and local variables

As mentioned in the course, the offsets for attributes (in the objects) are relative to the address of the descriptor stored in the `%rdi` register when the function is called. Once again we place all superclasses in a stack and pop them one by one to reuse the offsets from their attributes when possible. This way we follow the **simple inheritance** organisation, an inherited attribute will have the same offset in all classes.

```
1     int cummulative_offset = 8; //offset relative to rdi, assume descriptor
      is at offset 0
2     HashMap<String, Integer> offsets = new HashMap<String, Integer>();
3     while(superClasses.isEmpty() == false){
4       c = superClasses.pollLast();
5       for(Attribute attribute : c.attributes.values()){
6         if(offsets.containsKey(attribute.name)){// reuse super class offset
      (it must be the same!!!)
7           attribute.ofs = offsets.get(attribute.name);
8         }else{// we need to set this offset!!
9           attribute.ofs = cummulative_offset;
10          offsets.put(attribute.name, attribute.ofs);
11        }
12        cummulative_offset += 8; // add 1 word to offset
13      }
14    }
15
```

We do a scan of all the variables in the hash map we built during typing to define the stack offsets for local variables. After this loop, we perform a `subq` instruction on `%rsp` to allocate the memory on the stack.

2.1.3 • Step 3: Compile the body of methods and constructors

Finally, we call the visitor on all objects of the typed AST. After all these calls, there is a region in the code to include the wrappers to `libc` functions.

2.2 MyTVisitor.java

Similar to the `MyVisitor` class, I followed a rule that after visiting any `TExpr` I put the result in the `%rax` register. I started compiling the `arith-bool` programs in the tests, so I started working on the binop visit function.

Here I had two difficulties. First, how to skip the second expression of the binop when we are able to solve the binop using only the first expression (in logical operations Or and And). I would like to explain the solution I used with an example, for the statement below

```
if (true || false) System.out.print("ok\n");
```

There are 4 regions of code in the generated assembly

- the first one computes $e1$, pushes the result on the stack and jumps to the third block;
- the second one is the $e2$ block but we visit it later only if necessary. It pushes the result on the stack and jumps to the fourth block;
- the third block verifies if $e1$ is enough to solve the binop, if not it jumps to the second block else it pushes a dummy value on the stack and moves on to the fourth block
- the fourth block pops the two expression results from the stack, puts the result on `%rax` and finishes.

Another difficulty I had was dealing with nested binops without using these push and pop instructions that are not good because they modify the stack pointer registers and make it difficult to work with the offsets we defined in Step 2. Unfortunately, I did not find a solution to this problem, and did not manage to implement a visit function for `PSvar`.

Unfortunately, I spent a lot of time doing type checking and I am sorry that the code generation part of the project is not as complete. I only managed to compile hello world and the `arith-bool` programs.

3 GITHUB REPOSITORY

I would like to mention that I put al my code for the course (initial TDs and project) on Github, you can access al my code at github.com/ArkhamKnightGPC/INF564.

Finally, I just want to say thanks for the course!