CS842 Type Systems Final Project Write Up:
Delayed Type Based for Java on Edge Counting

Zijin Li / Student No.: 20605944 / z542li@uwaterloo.ca

Introduction

For the final project I propose a "delayed type" for java based on the idea of "edge counting". This brief write up will explain the idea and provide description for the implementation.

The Idea

The idea is that objects can have zero or more **delayed fields**, and we add the type checking such that delayed fields can only be accessed until all delayed references of same type in a connected group of objects have been properly initialized ("same type" means that they are fields from the same class and pointing to same type of target). If all delayed references of same type in a connected group of objects are initialized, we mark the delayed fields of that type as **completed**, otherwise we mark them as not completed. Whether delayed fields are completed or not is determined by a technique called "edge counting", which perform the following three steps:

- Identify object partitions: for all objects alive, partition them such that there is no delayed reference connecting any two objects in different partitions.
- Count the numbers of different types of delayed references (edge) that is necessary to make them complete.
- Count the actual numbers of different types of delayed references that exist and compare them with the expected number. For each type of delayed reference, if the actual number is equal to expected number, mark all delayed fields of that type as completed; if the actual number is less than expected number, mark all delayed fields of that type as not completed.

Below are two concrete examples of edge counting. First one is circular linked node:

Object Node {
  Number val;
  @Delayed Node next;
}
 	
Main() {
  Node n1;
  Node n2;
  n1 = new Node();
  n2 = new Node();
 	
  n1.next = n2;
	//Here n1 and n2 form a connected group, with one edge pointing from n1 to n2. Because there are two objects in the group with type Node, and class Node has one delayed fields with type Node, meaning that for the delayed references with type "from Node to Node" to complete, there need to be two delayed references from Node to Node. Because at this point only one such reference exists, delayed type "from Node to Node" is not completed. And try to directly access a delayed field with such type, for example, accessing "n2.next" should cause an error. (See Figure 1)

n2.next = n1;
 	
Node n3 = new Node();
 	
n3.next = n2;
}
	//After this assignment statement another edge from type “Node” to “Node” is added to the graph and the expected number and actual number of delayed references with type “from Node to Node” are the same, so we know that delayed fields with such type are completed (in this case “next”). (See Figure 2)
	
	//At this point n3 is not in the partition where n1 and n2 belongs to, so it is still okay to access the “next” field of n1 and n2, but not n3.
	//Now n3 is added to the partition of n1 and n2. Because the expected and actual number of delayed references with type “from Node to Node” are the same, all delayed references with such type are marked as completed (See Figure 3).

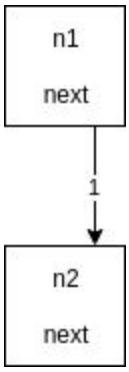

Figure 1

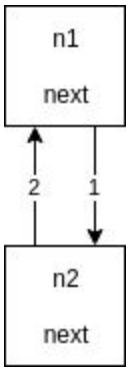

Figure 2

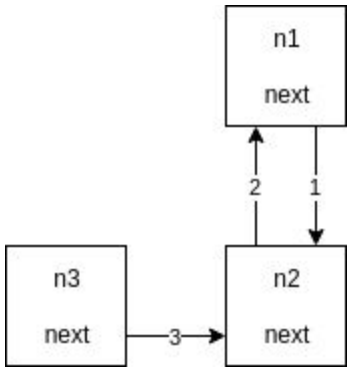

Figure 3

A slightly more complicated example:

Object A {
@Delayed A next_A;
@Delayed B next_B;
}
 	
Object B {
@Delayed B next_B;
}	

In the state shown as Figure 4, delayed references with type “from A to A” are completed, because there are three objects of type “A”, class “A” have one delayed fields of type “A”, and there are three actual delayed references from “A” to “A” exists. So it is okay to access delayed fields with type “from A to A”, in this case “next_A”. However, it is not okay to access delayed fields with type “from A to B” or “from B to B” because the actual reference number is less than expected. In Figure 5, an assignment statement may add a new edge from A3 to B2, now delayed fields with type “from A to B” are completed and okay to access. In Figure 6, an edge from B to B is added, and now delayed fields with type “from B to B” are completed.

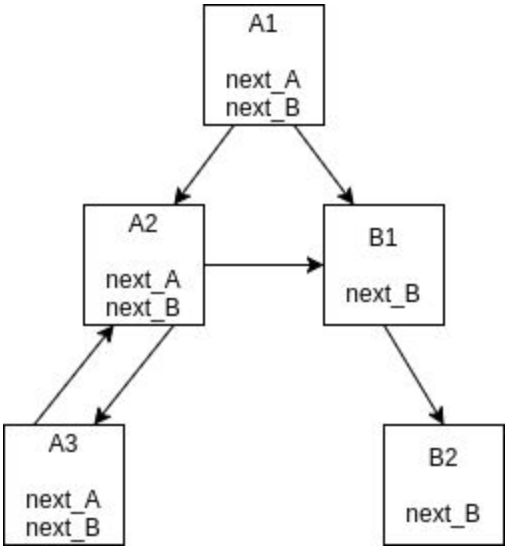

Figure 4

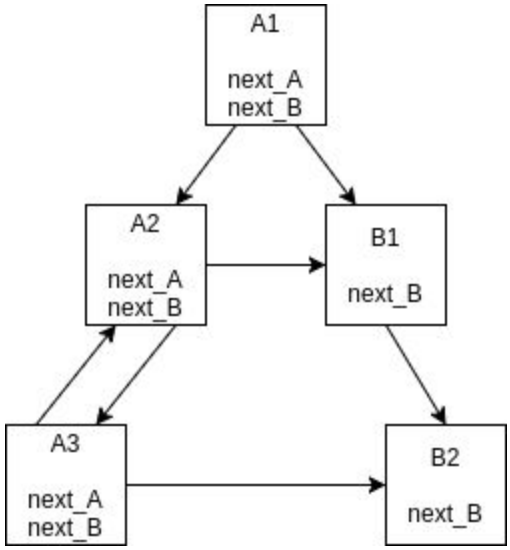

Figure 5

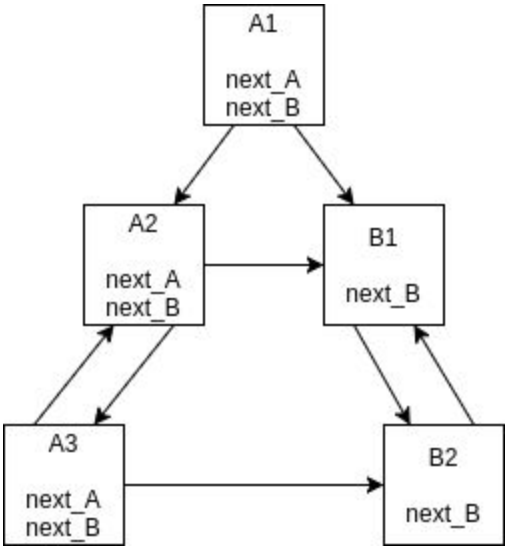

Figure 6

Implementation

A checker tool to check the the validity of accessing of delayed fields for source code written in Java is implemented. It is a stand-alone java maven project, with the help of “JavaParser”. The major classes and their functions are:

- Analyzer: the entrance class. It uses a private visitor class to visit the AST of source code and call appropriate API provided by Tracker in order for Tracker to perform checking.
- Tracker: A wrapper of checking tool. Provide API for Analyzer to handle different AST nodes. Maintains a state tree where each state represents a possible state that the program may be in. It calls on the corresponding method from State to analyze each possible state.

- State: contains the core logic of delayed type checking. Perform checking on a single state that the program may be in.
- Reporter: a tool to report errors or warnings depending on the checking results from all states. For example, if for all possible states, there are no errors, then the reporter considers there is no error in general. If some states are errored but some not, then the reporter generates a warning in general. If all states are errored, then the reporter generates an error in general.

Using the Checker

To use the tool, in the java source code that you wish to be checked, change the name of all delayed fields with prefix "delayed_" (can be changed to using annotation in the future work), please refer to the examples provided in the "examples" folder. Import the project as a java maven project using any IDE. In the main method in Main class, change the value of "srcPath" to the path of your java source code path, and run the program. Sample source codes are provided in the "examples" folder, note that the tool will not print anything if there is no error or warning in the source code. Also it will not check for things not related to delayed type so make sure that your source code complies first.

Complexities

Some major complexities are:

- In order for the checker to perform edge counting, it needs to know exactly how many objects are alive so I need to implement a simple version of reference counting garbage collector on "fake" objects.
- Partitioning the objects is quite complex, because objects may contain delayed fields and non-delayed fields, but partitioning is based on connectivity on delayed fields only.
- Handling method calls while maintaining the current state is quite hard.
- Handling uncertain points of the source program is very hard. My original idea was to introduce edges that "maybe exist" or "maybe not exist", but this makes it impossible to perform garbage collection and edge counting. So my solution is to create a state for each possibility of the program, so that in each state everything is certain, and the overall checking results depend on the result from each state. Below is an example for handling "if" statement:

```
…  // program point 1
if () {
    … // program point 2
}
… // program point 3
```

At program point 1, everything is certain. The checker only has one state which is the current state (shown in Figure 7-left). At program point 2, a new state 2 is derived from state 1 as the program enters "if" statement, and state 2 is now the current state (shown in Figure 7-middle). At program point 3, the program exits the "if" statement, and the current state points to the parent state of state 2, so now current state is state 1 again (shown in Figure 7-right). But at this point, any checking will be performed on both state 1 and state 2 (current state and all its child states) and the results from each state being checked are combined to determine the overall result of the program.

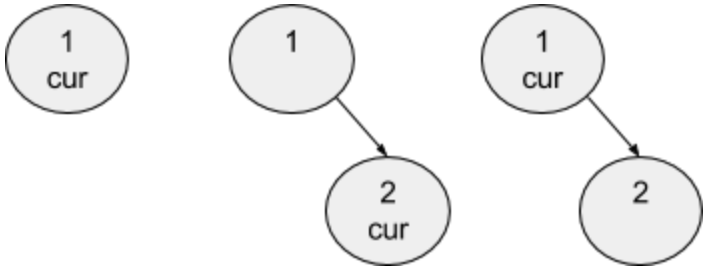

Figure 7

Example for “if-else” statement, state status shown in Figure 8, checker tool will skip invalid states:

```
…  // program point 1
if () {
    … // program point 2
}
else {
   … // program point 3
}
… // program point 4
```

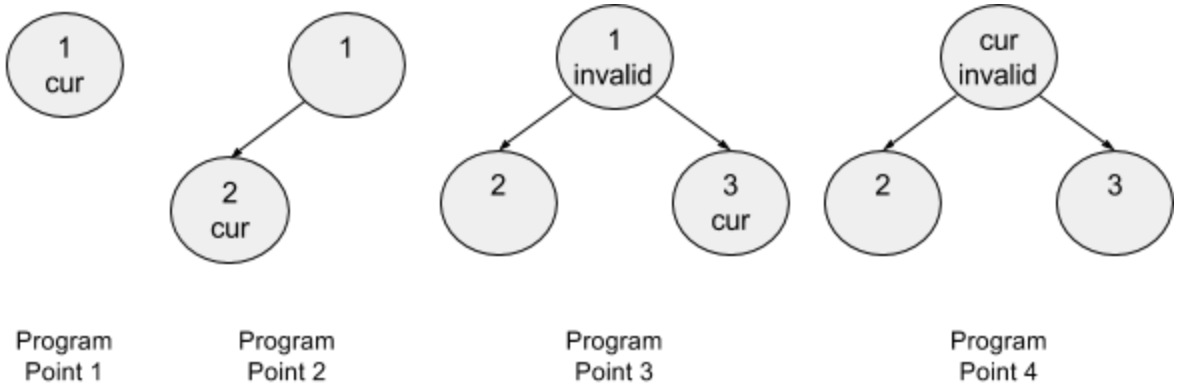

Figure 8

Discussion

I would like to mention two points:

- Java is a pretty big language, due to time limitation not all of the AST nodes have an implementation. But with the current abilities that this tool has, implementing other AST nodes will only take trivial efforts. (if they are possible to implement...)
- The idea is flawed. The fact that it requires to know the exact status of objects is the reason that I didn’t use the Checker Framework. The tool is rather “fake-running” the program and recording each possibility rather than performing a normal static analyze. However, I did my best to try to implement this tool, put great effort on coding and learned a lot about compiler and run-time theory.