Introduction

In the past six weeks, I worked on my portfolio project--Object-Oriented construct in NotQuiteJava. The purpose of my project was not only the implementation of an object-oriented constructor in NQJ but also the documentation and explanation of the code I implemented. I started by doing the project in Java which took me about five full working days spread over two weeks, then I wrote the documentation the next two weeks and finally my reflection paper in the days of after. When I was conducting my project, the questions of how to distribute the time to the different parties in order to respect the deadline, how to achieve each part and how to check that my work is on the right path, were always my top concerns.

The most interesting thing that I learned

While I am confident in my programming knowledge and understanding, writing thorough documentation is a skill I hope to develop further. Whiles writing the documentation of my portfolio I was able to use my knowledge in graph theory to explain why the algorithm to detect cyclic inheritance was correct, but was also able to see in more detail why some algorithms worked and thus detect the bugs that I had forgotten during the implementation and fix them. Writing such thorough documentation was definitely the most interesting and challenging thing that to do, not just because I have to learn how express my ideas but also because I have to find right way to organize them. Because I did not have the possibility to write a documentation for such a project before, this portfolio was also an occasion for me to see the points to be aware of when writing documentation. One surprising aspect was seeing how the code implementation affected the way that I organized my ideas in the documentation paper. For example, it took me a long time to know how to start the documentation of the translation phase, because I wanted to ensure that sequence of explanations follows the program order. Another surprising aspect was to see that explaining an algorithm was not a trivial task especially when the algorithm in question is long and complicated. For example, it was particularly difficult to explain what the method `translateFieldsAndVirtualMethodTables()` of Translator does and how it does it this method contains a while-loop which runs through the heritage chains from the subclass to the highest superclass. All of this remind me that while I was able to implement the project well, I need to develop my succinct technical writing skills further. I am very interested in developing my writing style I learned from this experience, and I hope it will help me later when I write new materials.

The part of the portfolio I'm proud of

Although every part of the portfolio was challenging, I must say that the code implementation is what makes me proud, not because the program is perfect, since I can always improve it, but because it required me to apply a concept like graph theory that I learn from other courses. The abstract of the cyclic inheritance problem by reducing it to a "simple" problem of cycle detection in a directed graph, as well as the analysis of algorithm execution time for graph traversal in order to optimize it was quite impressive for me. Moreover, I find my approach of the implementation of the name and type analysis much better than what I had when I made the exercise sheet 3, considering the fact that I better

analyzed the language Java in order to be able to understand how errors in a program were reported. For example, I found that the body of the class was not analyzed when this class was involved in an inheritance cycle. Similar to my approach in implementing name and type analysis, I also adapted the translation phase implementation approach to make it compatible with that of the template of exercise sheet 4. Indeed, I am quite proud not to have had to change the existing code too much, while keeping a structure of the code which allows me to use patterns already defined in the template. For example, I maximized the use of the function already defined like those defined on TypeContext or on the abstract syntax tree nodes to facilitate passing data and avoid rewriting code unnecessarily.

The adjustments to my design and implementation

The name and type analysis was the most difficult phase to implement, because the program had to develop my thoughts of program from the ground up. Indeed, I had to review the design several times, in particular by adjusting the creation of ClassType so that the cycle in the chains of dependencies can be taken into account. I also had to change the **ClassType** so that the `getField()` and `getMethod()` functions could load the fields and methods only when needed, instead of doing it in the constructor and running the risk that the **ClassType** of the superclass had still not been created. The translation phase was easier, because I already had an idea of how I could implement it. However, I had to change my implementation of overriding in the virtual method table twice so that it took into account typecast, because I ignored at the start that even in the case of typecast the methods of the subclass always override those which the superclass. Furthermore, at the end of the implementation I also went through each line of code to find the one that was useless or that could be optimized. This is how I was able to delete iterations that were useless and also the lines of dead code in the **Analysis** and **Translator** classes.

Considering all these adjustments, there are still some points that I would have liked to improve upon if I had had more time, among other things the execution time of the analysis and translation phases, and the code generated in LLVM. Indeed, the current execution time of a file with 60 lines is about 250ms, but this time can be reduced if the **ClassType** objects created during the analysis phase were reused during the translation and if the positions of the fields of structures were and calculated once and saved for reuse later, because field access and method calls on classes are very frequent. In addition, during the translation phase I could just convert the classes and functions that are actually used at runtime to LLVM, instead of translating everything as I do now. This would allow me not only to reduce the generated llvm code but also to reduce the execution time. Another aspect which should be taken into account but which has not been specified in the portfolio is to verify if each function does indeed have at least one return expression.

The most exciting topic and outlook

For me almost all the topics that were approached during this course were very interesting and very enriching, however if I had to choose one the translation to the intermediate representation, is definitely the one which excited me the most. One of the reasons for this choice is that the knowledge gained there, like how implementing object-oriented construct in LLVM with simple things like

structures and functions can be reused to implement the object-oriented construct in another low level language like C. Also it allowed me in depth understanding about how arrays worked, how the context was handled in Java, but also what shadowing, obscuring and hiding was and how they can be implemented. Moreover, in my case this knowledge helps me to solve the problems of code translation in a more efficient way, because it happens to me sometimes for my personal projects when translating the codes which I find on the internet in functional language like JS.

Although the current course content is quite broad and exhaustive, I would have liked it to cover more subjects like scopes and exceptions. Indeed, throughout the course the discussions of scopes have been focussed on the scope of variables, but modern languages like Java also offer to restrict the accessibility of fields, methods and classes using the keywords "public", "private. "and" protected ". So I think an explanation of the possible implementation of these concepts in LLVM would have been very interesting. The exceptions were not also covered in particular how to implement them in LLVM, i.e. how to throw an exception, catch it and process it. As an outlook, I would also have liked to have been presented as additional content at the end of the course the current state of research in compilers including the use of machine learning in the construction of compilers, and the improvement of code analysis and autocompletion of code editors.

Conclusion

Overall my work on this portfolio although being very challenging in particular the drafting of the documentation, allowed me to acquire additional knowledge not only in the construction of compiler but especially in understanding of object-oriented languages like Java. In addition, the implementation of the name and type analysis phase aroused in me a particular interest in syntactic preprocessors and their many advantages, in particular the possibility of creating a strongly typed language from a weakly typed language. This portfolio beyond its academic framework also gave me an excellent opportunity to enrich my curriculum vitae.