DELFT UNIVERSITY OF TECHNOLOGY

IN4313 - Report

Monotone Framework implementation for the *While* language

Rogier Slag - 1507761

November 8, 2014

1 INTRODUCTION

Meta programming is considered to be the science which concerns programming with programs as data. Hence programs are modified and analyzed by other programs. While unknown to many programmers, these tools are used by themselves on a daily basis; examples are compiles and debuggers.

Many forms of analyses are possible on any programming language. However, for most of these special software should be written to perform any analysis. It is therefore greatly advantageous to have a single framework which is capable of performing several analyses, and which can be extended relatively easily to new analyses. Such a system requires more effort to build, but is easier to maintain, enhance, and extend. These systems are called *monotone frameworks* and are the main focus of this assignment.

The assignment itself is done for the TUDelft course IN4313 of Prof. dr. E. Visser [1] and derived from a Utrecht University course of dr. J. Hage [2].

2 THE *While* LANGUAGE

The language under consideration is the While language as introduced in the book [3]. For this assignment there have been some small changes in order to focus more on the framework than on the parsing of the language.

1. Expressions should be enclosed in brackets.
2. Procedures should be defined before they can be used

3. For while loops, the ‘do‘ keyword is dropped to make the language behave more C-like.
4. For if-else constructs, the ‘then‘ keyword is droppen to make the language look more C-like.
5. The ‘not‘-construct requires the subexpression to be wrapped in parentheses.
6. Procedures always have a return value (no ‘void‘s).
7. No overloading or overriding is supported (the behavior is undocumented).
8. Procedures cannot be defined within any other block or statement (the behavior is undefined).

This set of constraints make the language smaller and thereby more useful to develop an actual monotone framework. Any changes only have to be done on the level of the parser, not to the actual monotone framework itself.

3 MONOTONE FRAMEWORK

The monotone framework itself is fairly straightforward. It offers three basic functions:

1. Compute the *Abstract Syntax Tree* (AST) of the program,
2. Compute the *Control Flow Graph* (CFG) of the program,
3. Perform a program analysis.

For the program analysis, three examples have been given: *Reaching definitions*, *Available Expressions*, and *Definite Assignments*. As the definition of the monotone framework suggests, it is straightforward to extend it with other analyses as well, such as *Very busy expressions*, or *Live variables*.

Generally each analysis requires for different steps:

1. Defining which parts should be killed due to statement with label l,
2. Defining which parts should be generated due to statements with label l,
3. Creating an entry set for each statement,
4. Creating an exit set for each statement.

Due to loops in the CFG of the program one cannot determine the last two sets directly, but this has to be done iteratively. ‘kill‘ and ‘gen‘ sets can be computed directly though, these will not change when iterations occur.

4 IMPLEMENTATION

The implementation is divided into three parts. First there is the generation of the actual AST of the input program. Secondly the control flow is discussed. Finally the monotone framework itself is described.

4.1 ABSTRACT SYNTAX TREE

Even before the AST is generated one needs to parse the language under consideration into computer processable code, which in turn can be used to create the AST. For this Antlr [4] was used. Using Antlr one writes a grammar, which can then be used to read and process the input file into Java.

The choice for Antlr was relatively straightforward: it is a generally well-documented piece of software, which has a lot of community support. Additionally the remaining entries of the program can be easily traversed in Java, which was considered an advantage by the author.

The AST was subsequently generated using general Stacks and Maps in Java to create the required form. Once complete, the final program would be saved in the static variable 'FINAL_PROGRAM'. The parser data can be found in the 'parser' package and the class 'MPWhileListener' in the 'framework' package. The AST generation software resides entirely in the 'framework/ast' package.

4.2 CONTROL FLOW GRAPH

Once the AST was generated, one could continue with the control flow. For this, there are two things to consider. First any procedure calls should be taken into account, since the *While* language was extended by these. Secondly one could easily defend that any part of the language has three separate sections for the CFG creation: the entry (one point), the internal flow (unbounded), and exit points (finite set).
For example: an 'if-else' statement has the entry on the 'if', some internal flow from the conditional to the blocks and within the blocks, and finally two exit points (the last statement of the 'if'-block and the last statement of the 'else'-block). A 'while' loop has one entry point (the conditional), internal flow (from the conditional to the block and within the block), and one exit point (the conditional).

Using the above approach, the CFG could be generated (one needs to take the constraints posed by section 2 into account). This is done by calling the method 'internalFlow()' on the earlier generated 'FINAL_PROGRAM' variable. The method than processes the tree recursively. For clarity, additional 'START' and 'END' nodes are added to the CFG. The control flow logic data can be found in the 'framework/ast' package.

4.3 PROGRAM ANALYSIS

Finally one can perform the analysis. For this, one needs to subclass the 'Analysis' class, and when desired the 'AnalysisResult' class. Next step is to define four methods: two for determining the 'gen' and 'kill' sets of every label, and two for defining the analysis results on the entry and exit of every label. If one is familiar with the analysis, this is fairly straightforward (it took the author only 125 lines to define the 'AvailableExpressions' analysis, including all imports and whitespace [5]). Next, one defines which direction should be used to traverse the CFG.

Finally one can create an object from the class and call the 'performAnalysis()' method on it. The results will then be sent back in a Map, consisting of labels and the entry and exit values for the label. The values in the Map will be calculated by reiteration over previous solutions several times, so it is ensures the solution which is returned is stable. This part of the software can be found in the 'framework' package. The map contains 'ExitEntryPair's, which are the type and effect systems of the analysis.

REFERENCES

[1] TUDelft. In4313 - seminar metaprogramming, August 2014. URL `http://www.studiegids.tudelft.nl/a101_displayCourse.do?course_id=31677`.

[2] dr. J. Hage. Master course automatic program analysis, April 2013. URL `http://www.cs.uu.nl/wiki/Apa/DataflowAndAbstractInterpretation`.

[3] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[4] Terence Parr. Antlr 4 documentation, June 2014. URL `https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation`.

[5] Rogier Slag. Availableexpressions.java, November 2014. URL `https://github.com/rogierslag/MetaProgramming/blob/master/src/framework/AvailableExpres`