Danmarks Tekniske Universitet

02242 Program Analysis

Project

First draft

Ibrahim Nemli
s093477

Kim Rostgaard Christensen
s084283

Peter Gammelgaard Poulsen
s093263

October 9th, 2013

Abstract

This is the report documenting and describing the work done in the course 02242: Program Analysis.

Ibrahim Nemli Kim Rostgaard Christensen
Peter Gammelgaard Poulsen

Contents

1 Introduction **1**

2 Exercise 1 - Data structures **1**
2.1 Abstract syntax tree data structure 1
2.2 Flow graph data structure 1
2.2.1 Creating flow graphs 1
2.3 Program graph data structure 2
2.3.1 Creating program graphs 2

3 Exercise 2 - Program Slicing **2**
3.1 Example program 2
3.2 Reaching Definitions Analysis table 3
3.3 (c) Flow graph 3
3.4 (d) Program slice calculation algorithm 3

4 Exercise 3 **3**

5 Exercise 4 **3**

1 Introduction

Program analysis is the discipline of extracting and deriving information about the structure, and potential behaviour of computer programs. It has many applications, such as; bug finding, optimizations, code reuse, formal validation and model mapping. In this report, we take a closer look at some of the specialized and general theory used for transforming source code text into analysis-friendly structures, and then performing the actual analysis on them.

2 Exercise 1 - Data structures

The first problems we need to solve is which data structures should be used for our abstract syntax tree, which will serve as the first step of our intermediate representation.

2.1 Abstract syntax tree data structure

We have decided that we could use an unbalanced tree for storage of our abstract syntax tree in our target programming language. Constraining the tree construction will give us the necessary check for syntax errors. A simple tree-traversal will give us the appropriate unique label numbers.

Using a parser a program written in the extended WHILE language will be parsed to a abstract syntax tree. This abstract syntax tree needs to be represented using a data structure in our implementation. Figure ?? shows the defined data structure.

2.2 Flow graph data structure

For the flow graph, we have chosen a network graph data structure. This gives an intuitive way of linking statements to potential paths.

It is convenient to transform the abstract syntax tree into a flow graph when performing a program analysis. For that reason a data structure for the flow graph as well as mapping from a abstract syntax tree to a flow graph is needed

Figure 1 shows the classes used to represent a flow graph. The idea is that there exists two kinds of nodes that both inherits from the abstract `Node` class. Each `Node` has a `label` and a `prev`, which is the previous node in the graph. The first node in the program will not have a previous node. A `Statement Node` is a node that contains a `CodeBlock` and it is used for everything that does not use branching. It has a pointer to the next node in the flow graph. A `Branching Node` is used for if statements and while loops where branching is needed. It contains a `condition` and a pointer to two possible next nodes.

2.2.1 Creating flow graphs

The way a flow graph representation will be created is by first going through all declarations in the abstract syntax tree. This will for sure not include branching so they should end up in `Statement Nodes`. One for each declaration. The next thing to do is to go through each statement and check whenever it is a branching statement or not and create the correct nodes. When a statement is met that contains more statements (for instance a while statement), these statements needs to be added to the graph before the iteration of statements can continue.

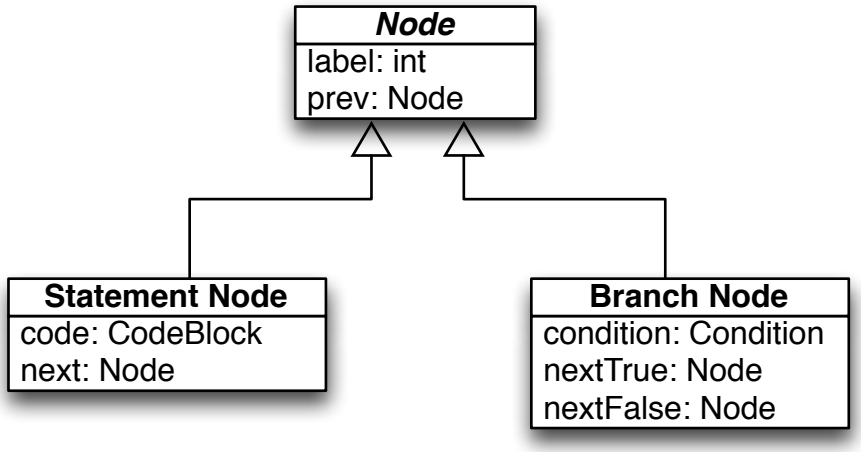

Figure 1: Data structures used to represent a flow graph

2.3 Program graph data structure

For some types of analysis it is more convenient to use a program graph compared to a flow graph. In that case a way of transforming a abstract syntax tree into a program graph is needed. Figure 2 shows a data structure for a program graph. It is a bit more complicated than flow graphs since we need to introduce both nodes and edges. A node has a `label`. If it is a `Branch Node` then it contains two `Branch Edges`. A `Branch Edge` is an edge that contains a `condition`. If it is a `Statement Node` is contains a `Statement Edge`. A `Statement Edge` is an edge that contains a `codeBlock`.

2.3.1 Creating program graphs

A program graph can be created much in the same way as a flow graph by going through all the declarations first and then the statements. Basically the only difference is that the labels are stored in nodes and code blocks and conditions are stored in edges.

3 Exercise 2 - Program Slicing

3.1 Example program

Figure 3.1 shows an example program written in the `WHILE` language that will be used as an example for this exercise. The program consists of 3 declared variables which are assigned and a while loop.

- If the point of interest is label 8. The result of program slice analysis would be; $[y:=x]^4$, $[y:=y-1]^8$, $[\text{int } x]^1$.
- If instead the point of interest is label 7. The result of program slice analysis would be; $[y:=x]^4$, $[z:=1]^5$, $[z:=z*y]^7$, $[\text{int } x]^1$.

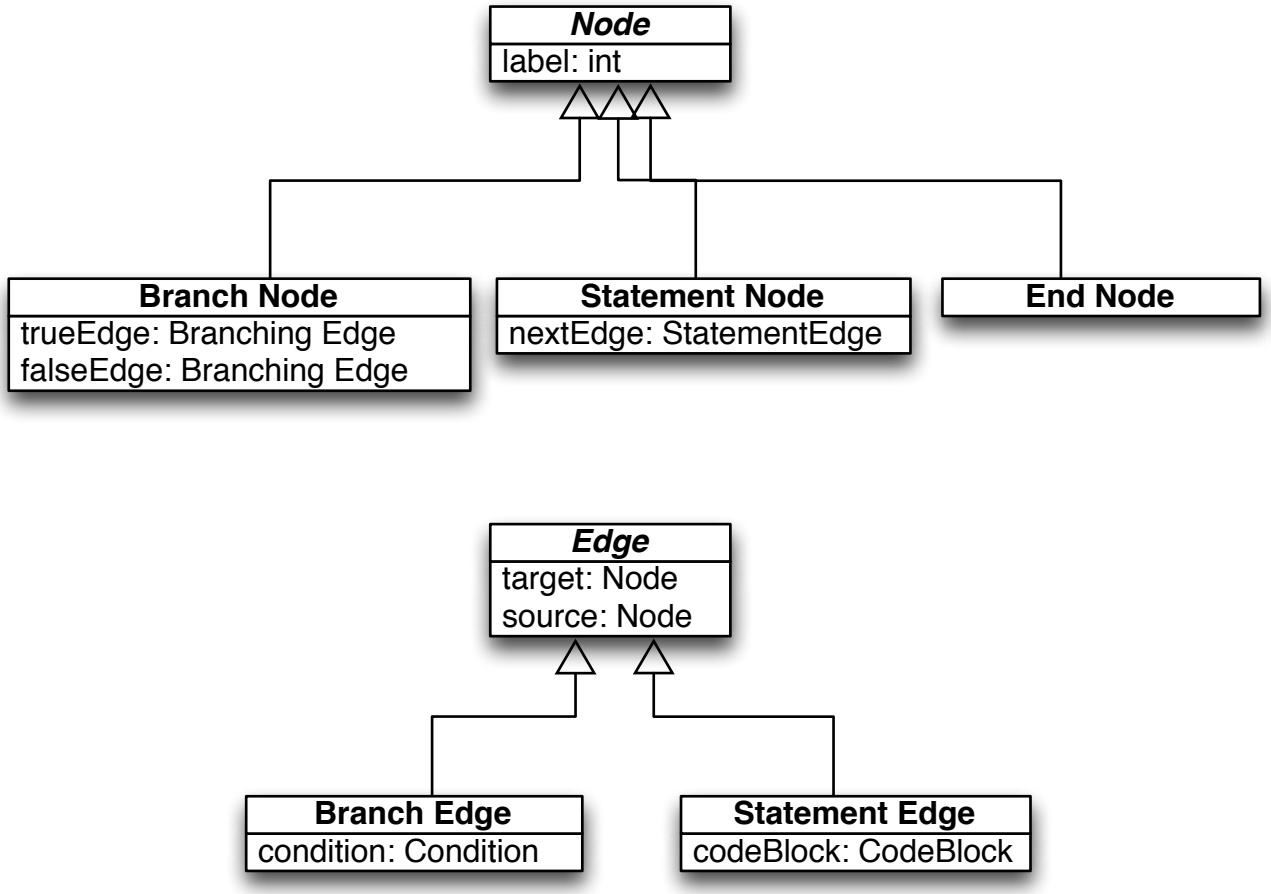

Figure 2: Data structures used to represent a program graph

3.2 Reaching Definitions Analysis table

Table 1 shows the Reaching Definitions Analysis table for the extended `WHILE` language. For each statement it is possible to see what is generated and what is killed.

3.3 (c) Flow graph

3.4 (d) Program slice calculation algorithm

4 Exercise 3

$$(L, \mathcal{F}, F, E, \iota, \mathrm{f}.)\ \frac{1}{x^2} \tag{1}$$

Where

5 Exercise 4

```
program
[int x]^1
[int y]^2
[int z]^3
[y := x]^4
[z := 1]^5
while [y>0]^6 do
    [z:= z*y]^7
    [y:= y-1]^8
od
[y:=0]^9
end
```

Figure 3: Code example used in to calculate program slice.

int x	$\text{kill}_{RD}([\text{int x}]^l) = \emptyset$ $\text{gen}_{RD}([\text{int x}]^l) = \{(x, l)\}$
int A[n]	$\text{kill}_{RD}([\text{A[n]}]^l) = \emptyset$ $\text{gen}_{RD}([\text{A[n]}^l) = \{(A[0], l), ...(A[n-1], l)\}$
A[a_1] = a_2	$\text{kill}_{RD}([\text{A}[a_1]]^l) = \{ (\text{A}[a_1],\text{l'}) \mid B^{l'}$ is a declaration or an assignment to $A[a_1]$ $\}$ $\text{gen}_{RD}([\text{A[n]}^l) = \{(A[0], l), ...(A[i-a], l)\}$
read x	$\text{kill}_{RD}(\text{read x}) = \{ (\text{x,l'}) \mid B^{l'}$ is a declaration or an assignment to x $\}$ $\text{gen}_{RD}(\text{read x}) = \{(x, l)\}$
read A[a]	$\text{kill}_{RD}(\text{read A[a]}) = \{ (\text{A[a],l'}) \mid B^{l'}$ is a declaration or an assignment to A[a] $\}$ $\text{gen}_{RD}(\text{read A[a]}) = \{(A[a], l)\}$
write a	$\text{kill}_{RD}([\text{write a}]^l) = \emptyset$ $\text{gen}_{RD}([[\text{write a}]^l) = \emptyset$
x:=a	$\text{kill}_{RD}(\text{x:=a}) = \{ (\text{x,l'}) \mid B^{l'}$ is a declaration or an assignment to x $\}$ $\text{gen}_{RD}(\text{x:=a}) = \{(x, l)\}$

Table 1: RD Equations