Data Flow Analyzer

M.Tech. R and D Report

by
Harshada A. Gune

07305904

under the guidance of

Prof. Uday Khedker

Dept. of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Contents

Abstract

This document describes a data flow analyzer for intraprocedural bit vector data flow analysis built using Java as programming language. This can be used to implement various bit vector data flow analyses.

1 Motivation

The design and implementation of data flow analyzer is motivated by the following objectives:

- To provide an easy to use and easy to extend data flow analysis infrastructure.
- To facilitate experimentation in terms of studying existing analyses, defining new analyses, and exploring different analysis algorithms.

Theory of data flow analysis can be found at [2]. The description of generic data flow analyzer in gcc can be found at [1]

2 Specifying a Data Flow Analysis

Specifying a data flow analysis requires writing a specification file and providing input file in three address format. Specification file requires to specify following parameters with following possible values.

Parameter	Possible Values
Entity	Variable, Expression, Definition
Direction	Forward, Backward
Confluence	Union, Intersect
GenEffect	Use, Modify
KillEffect	Use, Modify
GenExposition	Upexp, Downexp, Anywhere
KillExposition	Upexp, Downexp, Anywhere
TOP	Ones, Zeros
EntryInfo	Ones, Zeros
ExitInfo	Ones, Zeros

2.1 Specifying Available Expression Analysis

We present the DFA specifications required for available expression analysis. Following is the specification file written for available expression analysis.

```
-----------------------------------------------------------------
1 Analysis:
2 Entity = Expression
3 Direction = Forward
4 GenExposition = Downexp
5 KillExposition = Anywhere
6 GenEffect = Use
7 KillEffect = Modify
8 TOP = Ones
```

```
9 Confluence = Intersect
10 EntryInfo = Zeros
11 ExitInfo = Ones
12 END
```

The semantics described by above specification input is as follows: Line 2 declares that the relevant entities for this analysis are expressions. Line 3 declares the direction of traversal to be FORWARD. Line 8 specifies that $\top$ is the universal set of expressions. Line 9 declares the $\sqcap$ to be $\cap$. Lines 4 to 7 define local data flow properties. Lines 4 and 6 specify that Gen set of the block contains those expressions which are used within the basic block and are downward exposed. While lines 5 and 7 specify that expressions which are modified anywhere in the block are in the Kill set. Lines 10 and 11 give the BI at entry and exit respectively.

2.2 Specifying Other Bit Vector Data Flow Analyses

Now it is easy to give specifications for other bit vector frameworks.

- *Live Variable Analysis:*
  Entity is Variable, dirction is Backward, gen exposition is Upexp, TOP and ExitInfo is ZEROS and confluence is Union. Rest of the things remain same.

- *Reaching Definition Analysis:*
  Entity is Definition, dirction is Forward, gen exposition is Downexp, TOP is ZEROS and confluence is Union. Rest of the things remain same.

3 Implementing Data Flow Analyzer

We have chosen Java as an implementation language. Implementation is carried out in broadly three steps: building the GUI, parsing of DFA specification and input files and analyzing input file according to the specifications provided. We present the overall structure of analyzer.

3.1 Building GUI

At present we have a simple GUI which asks to specify DFA specification and input file paths. On clicking the button "Run Analyzer", analyzer is executed and data flow analysis is carried out. For time being, We have displayed the output on terminal only. It can be redirected to GUI as explained in possible extensions to analyzer. We have built this GUI using swing components of Java. This is done by using the NetBeans which is IDE

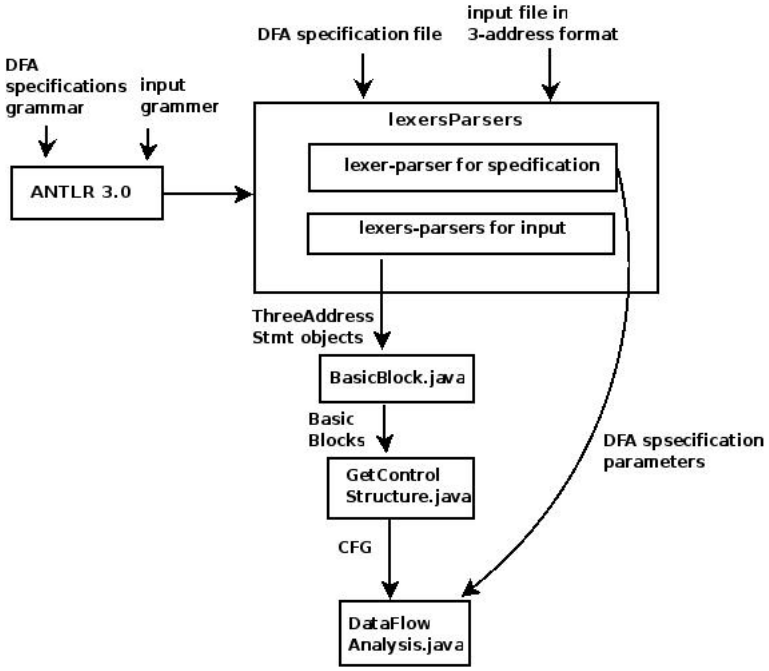

Figure 1: Structure of analyzer

that can be used for the development of Java desktop applications. Netbeans allow to design swing GUIs by dragging and positioning GUI components. The source code generated is copied into file GUI.java. In the click event of button "Run Analyzer", we call the function from BeginAnalyzer class and starts executing the analyzer.

3.2 Parsing

Parsing of input file and DFA specification file is implemented using ANTLR 3.1. ANTLR (ANother Tool for Language Recognition) is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. A brief description about ANTLR is given in following lines:

3.2.1 More about ANTLR

ANTLR reads a language description file called a grammar (with extension .g) and generates a number of source code files and other auxiliary files. The implementation language of these source code files can be chosen optionally (such as C, Python etc). We have chosen Java as target language. Most uses of ANTLR generates at least one (and quite often both) of these tools:

- *Lexer:* This reads an input character or byte stream (i.e. characters, binary data, etc.), divides it into tokens using patterns we specify, and generates a token stream as output.
- *Parser:* This reads a token stream (normally generated by a lexer), and matches phrases in our language via the rules (patterns) we specify, and typically performs some semantic action for each phrase (or sub-phrase) matched. Each match could invoke a custom action that we supply along with rules.

Lexer and parser files are generated by executing following command.

```
-------------------------------------
  java org.antlr.Tool Specification.g
-------------------------------------
```

We copied these source files into the appropriate places for our development environment and compiled them. All such lexer and parser files generated by ANTLR are put in package lexersParsers. Note that users are not required to insatll ANTLR 3 unless they want to change the grammar files.

3.2.2 Parsing of Specification File

Grammar for DFA specifications is given below:

```
-------------------------------------------------------------------------
start                 : statements* EOF ;
statements            : ANALYSIS (stmt)+ END ;
stmt                  : direction_stmt | entity_stmt | TOP_stmt |
                        gen_exposition_stmt | kill_exposition_stmt |
                        gen_effect_stmt | kill_effect_stmt| confluence_stmt|
                        entry_info_stmt | exit_info_stmt;
direction_stmt        : DIRECTION EQUAL (BACKWARD | FORWARD);
entity_stmt           : ENTITY EQUAL (VARIABLE|EXPRESSION|DEFINITION);
TOP_stmt              : TOP EQUAL (ONES|ZEROS);
gen_exposition_stmt   : GENEXPOSITION EQUAL (DOWNEXP|UPEXP|ANYWHERE);
gen_effect_stmt       : GENEFFECT EQUAL (USE|MODIFY);
kill_effect_stmt      : KILLEFFECT EQUAL (USE|MODIFY);
kill_exposition_stmt  : KILLEXPOSITION EQUAL (DOWNEXP|UPEXP|ANYWHERE);
confluence_stmt       : CONFLUENCE EQUAL (UNION|INTERSECT);
entry_info_stmt       : ENTRYINFO EQUAL (ZEROS|ONES);
exit_info_stmt        : EXITINFO EQUAL (ZEROS|ONES);
-------------------------------------------------------------------------
```

This is provided in Specification.g file. The generated source code files are SpecificationLexer.java and SpecificationParser.java. These are used to parse specfication file. Apart from these source code files, ANTLR also generates Specification.tokens file. The parser is called in a following way:

```
-------------------------------------------------
SpecificationLexer specLex;
specLex = new SpecificationLexer(filename);
CommonTokenStream specTokens;
specTokens = new CommonTokenStream(specLex);
SpecificationParser specParser;
specParser = new SpecificationParser(specTokens);
-------------------------------------------------
```

3.2.3 Parsing of Input File

Input file is required to be in three address code format. Parsing of input file is done in two passes. Grammar for input file is written in ThreeAddressCodePass1.g and ThreeAddressCodePass2.g. Grammar rules are same in both files, but the work carried out by them is different. During pass2, each input statement is transformed into ThreeAddressStatement object. Note that we start initializing these objects only if input is syntactically correct. Hence pass1 is used to check syntax errors as well as to count the number of statements in input. This number is then used by pass2 to create array of ThreeAddressStatement objects. The grammar rules are listed below:

```
-------------------------------------------------------------------------
start               : declaration_stmt* begin_stmt statements* EOF;
declaration_stmt    : local_declaration|global_declaration;
local_declaration   : LOCALS '#' VARIABLE ( COMMA VARIABLE)*;
global_declaration  : GLOBALS '#' VARIABLE ( COMMA VARIABLE)*;
statements          : stmt|LABEL stmt;
stmt                : if_stmt|
                      VARIABLE EQUAL(NUMBER|VARIABLE)OP(NUMBER|VARIABLE)|
                      VARIABLE EQUAL (VARIABLE|NUMBER)|
                      VARIABLE EQUAL VARIABLE '(' ')'|
                      VARIABLE '(' ')'|
                      goto_stmt|
                      end_stmt;
begin_stmt          : BEGIN;
end_stmt            : END ;
if_stmt             : IF NEGATE VARIABLE GOTO LABEL|
                      IF VARIABLE GOTO LABEL;
goto_stmt           : GOTO LABEL;
-------------------------------------------------------------------------
```

The corresponding lexer and parser files are ThreeAddressCodePass1Lexer.java, ThreeAddressCodePass2Parser.java, ThreeAddressCodePass2Lexer.java and ThreeAddressCodePass2Parser.java. All these files along with corresponding token files are put in package lexersParsers. We present a sample input

program as given below:

```
------------------------------------------------------------
Globals# b
Locals# a,e,f,d
Begin
l: a = b + 9
 d = e - f
d = d+1
goto end1:
l1:d = d-1
if !d goto l1:
goto end1:
end1: End
------------------------------------------------------------
```

3.3 Analyzing Input File

In this section, we describe how the actual analysis is carried out. The execution begins with the main method in RunAnalyzer class. It calls the RunAnalyzer method in class BeginAnalyzer which begins with calling the parsers as described in section 3.2.2. At the end of pass2 of input, input statements are converted into objects of ThreeAddressStatements. Following are the memeber variables of class ThreeAddressStatement.

```
------------------------------------------------------------
String label;
String operator;
String operand1;
String operand2;
String result;
String definition;
int indexInInput;
------------------------------------------------------------
```

After successful completion of parsers, following steps are executed.

3.3.1 Doing some preparatory work

Vectors vGlobals and vLocals are formed to store global and local variables respectively. Definition names (variable definition in ThreeAddressStatement) are also assigned to assignments statements in input program, as they are needed if reaching definition analysis is to be performed.

3.3.2 Setting DFA Specification Parameters

Having created CFG, following varibales are initialized with corresponding values from the parsed specification file.

- iDirection
- sTopValue
- sEntity
- sOperator
- sGenEffect
- sGenExposition
- sKillEffect
- sKillExposition

3.3.3 Creating Basic Blocks and CFG

Basic blocks are formed from objects of three address input statements using the functions getBlockLeaders and formBasicBlocks. Control flow information is added to them by calling getSuccessors and getPredecessors methods from class GetControlStructure. Structure of a basic block is defined in class BasicBlock and has following members:

```
-------------------------------------------------------------
int blockNumber;
ThreeAddressStatements [] statements;
TreeSet inSet = new TreeSet();
TreeSet outSet = new TreeSet();
TreeSet oldInSet = new TreeSet();
TreeSet oldOutSet = new TreeSet();
TreeSet genSet = new TreeSet();
TreeSet killSet = new TreeSet();
TreeSet defSet = new TreeSet();
int successors[] = {-1,-1};
int predecessors[] ;
-------------------------------------------------------------
```

Note that, no of successors for any block can be maximum two, whereas no of predecessors cannot be statically determined. Value -1 indicates no successor or no predecessor.
Now we are ready to perform data flow analysis of given input program

according to specifications provided. Function doDataFlowAnalysis from package utilities is called by passing specification parameters from section 3.2.2.

3.3.4 Doing Data Flow Analysis

Here we give a brief description of how data flow equations are implemented.

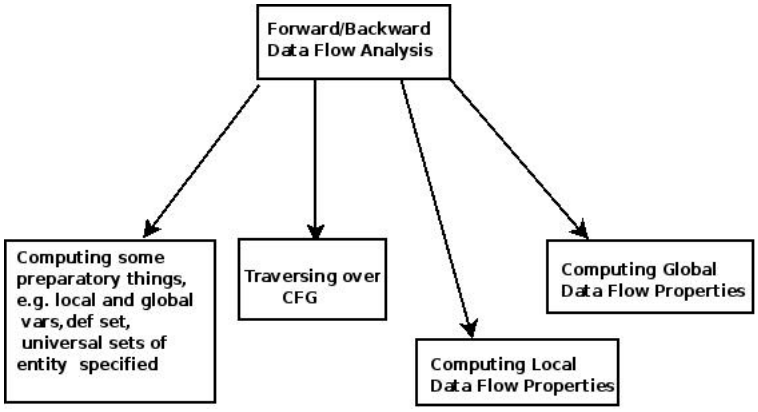

Figure 2: Modules involved in data flow analysis

1. *Initialization:* Some preprocessing like initializing the set called InitialValue with value of $\top$ and then initializing in sets and out sets of every basic block with this value. Also sets Locals and Globals which contain local and global variables from program are formed. Note that variables which are declared neither global nor local are considered as local. Also the sets tEntryBI and tExitBI are initialized with BI at entry and at exit respectively.

2. *Traversal over CFG and BasicBlocks:*
   In a round robin iterative traversal, the basic blocks in a CFG are usually visited in the order of along control flow or against the order of control flow. Depending upon this, it calls either doForwardDataFlowAnalysis or doBackwardDataFlowAnalysis functions.

```
-------------------------------------------------------------------
doForwardDataFlowAnalysis(no_of_basic_blocks,sTopValue,sEntity,
  sGenExposition,sGenEffect,sKillExposition,
  sKillEffect,sOperator)

doBackwardDataFlowAnalysis(no_of_basic_blocks,sTopValue,sEntity,
   sGenExposition,sGenEffect,sKillExposition,
           sKillEffect,sOperator)
-------------------------------------------------------------------
```

Predecessors and successors of a basic block are traversed using its predecessors and successors arrays. After completing an iteration, the check is made if fixed point is reached using function isFixed-PointReached. If not, they continue with the next iteration, otherwise stop the execution.The function isFixedPointReached does the comparison between old and new sets at in and out of every basic block.

3. *Computing Global Data Flow Properties:*
   Global data flow analysis involves to compute following data flow equations using computeOut and computeIn functions.

Forward data flow analysis

$$in = \begin{cases} \top & \text{if } b \text{ is start node} \\ \left(\sqcap_{s \in pred(b)}\right) out_s & \text{otherwise} \end{cases}$$

$$out = (in - kill) \cup gen$$

Backward data flow analysis

$$out = \begin{cases} \top & \text{if } b \text{ is exit node} \\ \left(\sqcap_{s \in succ(b)}\right) in_s & \text{otherwise} \end{cases}$$

$$in = (out - kill) \cup gen$$

The second equations in each of above set are implemented using computeOut and computeIn functions. Following is the code snippet from computeIn function. Function computeOut is defined on similar basis.

```
-------------------------------------------------------
TreeSet computeIn(TreeSet out,TreeSet kill,TreeSet gen)
{
TreeSet answer = new TreeSet();
TreeSet tout = new TreeSet();
```

```
TreeSet tkill = new TreeSet();
TreeSet tgen = new TreeSet();
tout.removeAll(tkill);
tout.addAll(tgen);
answer.addAll(tout);
return answer;
}
-----------------------------------------------------------
```

Remaining two equations are implemented using computeInOut function. Following is code snippet from function computeInOut.

```
-------------------------------------------------------------
TreeSet computeInOut(int blockNo,String operator,int direction)
{
   if(direction==0)//Forward data flow problem
   {
      for(i=0;i<basicBlocks[blockNo].predecessors.length;i++)
      {
         if(operator.equals("union"))
           answer.addAll(....);   //out of predecessors
         else if(operator.equals("intersect"))
           answer.retainAll(....);  // out of predecessors
      }
    }
    else //Backward data flow
    {
       // Do similarly in opposite direction.
    }
}
-----------------------------------------------------------
```

4. *Computing Local Data Flow Properties:*
   This involves computing gen and kill for a given basic block. computeGen and computeKill functions described below are implemented for this purpose.

```
-------------------------------------------------------------
computeGen(blockNo,sEntity,sGenExp,sGenEffect)
{
    TreeSet answer = new TreeSet();
    if(sEntity.equals("var"))
    {
       if(sGenEffect.equals("entity_use"))
```

```
            getLocalPropertiesForVarEffectUse(blockNo,sGenExp)
        if(sGenEffect.equals("entity_mod"))
            getLocalPropertiesForVarEffectMod(blockNo,sGenExp)
    }
    if(sEntity.equals("expr"))
    {
        //Do similarly for expressions
    }
    if(sEntity.equals("def"))
    {
         //Do similarly for definitions
    }
}
------------------------------------------------------------
```

This function is parameterized with gen effect, gen exposition and entity(varibale, expression or definition). Depending upon the entity and gen effect, this gives rise to further function calls(parametarized with gen exposition). Similar to computeGen, computeKill is also parametarized with kill effect, kill exposition and entity and further calls same functions (now parametarized with kill exposition.). Now the effect of exposition is captured by following functions: getLocalPropertiesForVarEffectUse, getLocalPropertiesForVarEffectMod and similarly for expressions and definitions. Following is the code snippet from getLocalPropertiesForVarEffectUse.

```
------------------------------------------------------------
getLocalPropertiesForVarEffectUse(blockNo,exposedTag)
{
   TreeSet answer = new TreeSet();
   for(i=0;i<basicBlocks[blockNo].statements.length;i++)
   {
        //UPWARD EXPOSED
        if(exposedTag.equals("upexp"))
        {
            //from stmt 0 to i-1
        }
         //DOWNWARD EXPOSED
        else if(exposedTag.equals("downexp"))
        {
            //from stmt i to last stmt in block
        }
   }
```

```
}
-----------------------------------------------------
```

Given exposition within block, above function computes the local data flow properties for a given block for entity variable and as an effect of use of a variable. Similarly other functions such as getLocalPropertiesForVarEffectMod,getLocalPropertiesForExprEffectUse, getLocalPropertiesForExprEffectMod etc are defined.

There are some other function defined which help in doing some computations. They are listed as follows:
getExpressionsWithGlobalVar:
This function returns the set of all those expressions whose atleast one operand is global variable.
getExpressionsWithGivenVar:
Given a variable, this returns the set of all expressions which have this variable as their operand.
getDefsOfVar:
Given a variable, it returns set of definitions of this variable.
computeDefinitionSetForEachVariable:
This function computes the vector called vDefSetForEachVar. This is the vector defined to store definitions of each variable. It itself contains as many vectors (let's say, member vectors) as no of variables in the program. The 0th element of each of these member vectors contain the variable name and rest of their elements give definitions of that element.

4 Extending Data Flow Analyzer

We suggest some possible extensions to this data flow analyzer.

- *Extensions to GUI*
  We can think of GUI window to be composed of some components as described below:
  - *Menu bar:*
    The functionality of standard menus such as File, Edit, Help, Exit etc. along with their respective submenus can be implemented.
  - *Tool bar:*
    Shortcuts to some menus can placed on this tool bar.
  - *Explorer Window:*
    It may be used to explore the hierarchy of folders which contain the input and DFA specification files. Although the output

is displayed on terminal in the current architecture, it may be redirected to some file and this file could be created in the same hierarchy of input and specification files.

  - *Editor For Writing Input and Specification Files:*
    It could be implemented that a small editor pops up to assist the users in writing the files. As the DFA specification file has some fixed format and some fixed keywords, the editor further can be improved so that it pops up the possible options while writing the DFA specifications.
  - *Console Window:*
    A separate console window can be created and the output that is being displayed in present architecture can be shown here.
  - *Displaying CFG:*
    After creating the basic blocks and adding control flow information to them, CFG can be shown graphically.
  - *Displaying Data Flow Information For Each Iteration:*
    Data flow information can be shown at each basic block for every iteration, until a fixed point reached. This can be shown in the graph displayed above.

- *Extensions that do not require changing the architecture of analyzer*
  - Implementing work list algorithm instead of round robin iteration algorithm.

- *Extensions that may require minor changes to architecture*
  - Explore the possibility of extending to the data flow frameworks where data flow information can be represented using bit vectors but the frameworks are not bit vector frameworks because they are non-separable e.g.,faint variables analysis, possibly undefined variables, analysis, strongly live variables analysis. This would require changing the local data flow analysis.

- *Extensions that may require major changes to the architecture of analyzer*
  - Extending to non-separable frameworks in which data flow information cannot be represented by bit vectors e.g., constant propagation, points-to analysis, alias analysis, heap reference analysis etc. This would require making fundamental changes to the architecture.
  - Extending to support some variant of context and flow sensitive interprocedural data flow analysis.

References

[1] Uday Khedker. gdfa: A generic data flow analyzer for gcc. Technical report, Department of Computer Science and Engineeing, Indian Institute of Technology Bombay, 2008.

[2] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group), 2009.