

02242 Program Analysis

Group 07

Amandeep Dhir(s094744)

Sameer KC(s094746)

Fariha Nazmul(s094747)

Report Sections	Author(s)
Report Layout	Sameer
Introduction	Aman
Project Design	Fariha, Aman, Sameer
Program Slicing	Fariha
Dead Code Elimination	Sameer
Constant Folding	Aman
ALFP	Fariha, Aman, Sameer
Implementation	Fariha, Aman, Sameer
Testing	Fariha
Conclusion	Fariha, Aman, Sameer
Future Work	Fariha, Aman, Sameer

29, November, 2010

1 Report Layout

The aim of the report is to outline the work carried out during the course "02242 Program Analysis". We focus mainly on the Data Flow Analysis techniques like - Reaching Definition Analysis, Live Variable Analysis and Constant Propagation. All the three analysis techniques are actually the instances of the Monotone Framework and solution to the Data Flow equations is produced by these techniques. The solution is computed using Maximal Fixed Point (MFP) algorithm. Optimization procedures such as Program Slicing, Dead Code Elimination and Constant Folding are carried on the Guarded Command language. In our report, we have constructed a tool that will perform static program analysis for programs written in the Guarded Command language. The syntax for the Guarded Command language is provided here 3.1 . The tool is developed using F# that performs applications⁵ based on Data Flow Analysis for programs in Guarded Command Language.

2 Introduction

Program analysis is an approach to construct robust programs. It is a method in which the behaviour of a program is analyzed automatically. Program analysis is a wide subject and generally studied as Static Analysis and Dynamic Analysis. Our project will only use Static Analysis.

Static program analysis is a technique that helps in evaluating and finding the properties that influence the behaviour of a program without actually executing it for example compilers rely on this for producing high quality code. Dynamic Analysis is used for predicting the behaviour or approximations of the values dynamically at run-time making sure that it is efficient, safe and computable. It is efficient due to low space consumption and good time performance. It can be under or over approximate depending upon the result of the program execution.

There are four different approaches for doing Program analysis namely - Data flow analysis, Abstract interpretation, Constraint based analysis, and Type and effect systems. Our implementation is based mainly on the Data flow analysis. It is a lattice based technique for collecting information about some set of values calculated at different points of the computer program. We use Control Flow Graph(CFG) that will help in determining those parts of a program to which a particular value is assigned to a block might propagate. Overall, the information collected through this will help in various optimization processes.

The command line interface of our project acts as frontend where the user can give any input program written in Guarded Command Language to perform different data flow applications. The description of our program analysis tool follows these steps:

- Step 1 – Input program is converted into system of equations over a partial order of program properties. This is important because the correctness of the output depends upon the correctness of the particular property to be analyzed.
- Step 2 – Actual solution will be calculated based on the system of equations generated at the step1.
- Step 3 – Suitable optimization procedures are carried out on the information received from the step2.

3 Project Design

This section will present an overview of the guarded command language, Data Flow analysis operations, internal representations and data structures used in our project.

3.1 Guarded Command Language

The grammar of Guarded Command Language in BNF is given as:

Expressions: $e ::= n \mid \text{true} \mid \text{false} \mid x$
 $\mid e_1 \text{ opb } e_2 \mid \text{opm } e \mid (e)$

Commands: $C ::= x := e \mid \text{skip} \mid \text{abort}$
 $\mid \text{read } x \mid \text{write } e \mid C_1; C_2$
 $\mid \{C\} \mid \text{if } gC \text{ fi} \mid \text{do } gC \text{ od}$

Guarded commands $gC ::= e \rightarrow C \mid gC_1 [] gC_2$

Programs: $P ::= \text{module name} : C \text{ end}$

opb $\{+, -, *, /, \text{int int int}$
 $<, >, <=, >=, =, !=, \text{int int bool}$
 $\&, \mid\}$ bool bool bool

opm $\{!, \text{bool bool} -\} \text{int int}$

Some key features of the language are listed below -

- The precedence rules are as usual like in C,Java
- The read/write works on integers only
- The skip command does nothing
- The abort command stops the execution
- In $e \rightarrow C$, the guard e must evaluate to a boolean.
- The commands like if and do contains a set of guards. Upon execution of a selection all guards are evaluated. If none of the guards evaluates to true then execution of the selection aborts, otherwise one of the guards that has the value true is chosen non-deterministically and the corresponding statement is executed

An instance of a program written in Guarded Command Language looks like the following example:

```
modulegcd
readx;
ready;
dox < y → y := y - x
```

```

[] y < x → x := x - y
od;
writex
end

```

3.2 Data Flow Analysis Operations

Data-flow analysis is used to gather information about the possible set of values calculated at various points in a program. The information gathered is often used by compilers when optimizing a program. The techniques such as Reaching Definitions Analysis, Live Variables Analysis, Available Expressions, Very Busy Expressions, Constant Propagation etc. are examples of Data Flow Analysis. We will be going in detail into Reaching Definition Analysis 5.1.3, Live Variables 5.2.3 and Constant Propagation 5.3.3 in later sections but first we present the semantics of number of operations on programs and labels that will be used in these analysis.

3.2.1 Initial labels and Final labels

$init : Cmd \cup Gcmd \cup Prog \rightarrow P(Lab)$: returns the initial label of a command:

```

init([x := a]l) = {l}
init([skip]l) = {l}
init([abort]l) = {l}
init([read x]l) = {l}
init([write x]l) = {l}
init([C1; C2]) = init(C1)
init([e]l → C) = {l}
init([gC1 [] gC2]) = init(gC1) ∪ init(gC2)
init(if gC fi) = init(gC)
init(do gC od) = init(gC)
init({C}) = init(C)
init(module name : C end) = init(C)

```

$final : Cmd \cup Gcmd \cup Prog \rightarrow P(Lab)$: returns the final labels of a command:

```

final([x := a]l) = {l}
final([skip]l) = {l}
final([abort]l) = ∅
final([read x]l) = {l}
final([write x]l) = {l}
final([C1; C2]) = final(C2)
final([e]l → C) = {l}
final([gC1 [] gC2]) = final(gC1) ∪ final(gC2)
final(if gC fi) = final(gC)
final(do gC od) = init(gC)

```

$$\begin{aligned} final(\{C\}) &= final(C) \\ final(module\ name : C\ end) &= final(C) \end{aligned}$$

3.2.2. Blocks

$blocks : Cmd \cup Gcmd \cup Prog \rightarrow P(Lab)$: returns the set of commands or elementary blocks of a command

$$\begin{aligned} blocks([x := a]^l) &= \{[x := a]^l\} \\ blocks([skip]^l) &= \{[skip]^l\} \\ blocks([abort]^l) &= \{[abort]^l\} \\ blocks([read\ x]^l) &= \{[read]^l\} \\ blocks([write\ x]^l) &= \{[write]^l\} \\ blocks([C_1; C_2]) &= blocks(C_1) \cup blocks(C_2) \\ blocks([e]^l \rightarrow C) &= \{[e]^l\} \cup blocks(C) \\ blocks([gC_1 \parallel gC_2]) &= blocks(gC_1) \cup blocks(gC_2) \\ blocks(if\ gC\ fi) &= blocks(gC) \\ blocks(do\ gC\ od) &= blocks(gC) \\ blocks(\{C\}) &= blocks(C) \\ blocks(module\ name : C\ end) &= blocks(C) \end{aligned}$$

3.2.3 Labels

$labels : Cmd \cup Gcmd \cup Prog \rightarrow P(Lab)$: returns the label of a command

$$\begin{aligned} labels(C) &= \{l \mid [B]^l \in blocks(C)\} \\ init(C) \in labels(C) \text{ and } final(C) \in labels(C) \end{aligned}$$

3.2.4 Forward and Reverse flow

$flow : Cmd \cup Gcmd \cup Prog \rightarrow P(Lab \times Lab)$: returns the forward flow of the program

$$\begin{aligned} flow([x := a]^l) &= \emptyset \\ flow([skip]^l) &= \emptyset \\ flow([abort]^l) &= \emptyset \\ flow([read\ x]^l) &= \emptyset \\ flow([write\ x]^l) &= \emptyset \\ flow([C_1; C_2]) &= flow(C_1) \cup flow(gC_2) \cup \{(l, l') \mid l \in final(C_1), l' \in final(C_2)\} \\ flow([e]^l \rightarrow C) &= flow(C) \cup \{(l, l') \mid l \in init(C)\} \\ flow([gC_1 \parallel gC_2]) &= flow(gC_1) \cup flow(gC_2) \\ flow(if\ gC\ fi) &= flow(gC) \\ flow(do\ gC\ od) &= flow(gC) \cup \{(l, l') \mid l \in final(gC), l' \in init(gC)\} \\ flow(\{C\}) &= final(C) \\ flow(module\ name : C\ end) &= flow(C) \end{aligned}$$

$$\begin{aligned} flow^R : Cmd \cup Gcmd \cup Prog &\rightarrow P(Lab \times Lab) : \text{returns the reverse flow of the program} \\ flow^R(C) &= flow(l, l') \cup \{(l, l') \mid (l, l') \in flow(C)\} \end{aligned}$$

3.2.5 Notation of input program

Here we provide the meanings of the notations we are using in the analysis.

- Cmd – Commands,
- Gcmd – Guarded commands,
- Prog – Programs
- C* – commands to be analyzed
- Lab – labels
- Lab* – labels occurring in C*
- Blocks – assignments, skip, abort, read, write commands
- Blocks* – blocks in C*
- Var* – variables in FV(C*)
- FV (l) – the set of free variables occurring at label l

3.3 Lexer and Parser

Parsing - It refers to the process of converting a well defined input into an internal representation that can be represented by Abstract Syntax Tree (AST) using the grammar rules. AST represents abstract syntactic view of the input program. Every node of the AST represent the construct occurring in the input program i.e Guarded Command language in our case. We perform parsing in two steps -

- Step 1 refers to Lexical Analysis where the input program is parsed into the internal representation by breaking the program into sequence of tokens. It is carried out in the lexer unit of the parser. We call it tokenizing the program.
- Step 2 refers to Syntactic Analysis where the internal representation is constructed based on the grammar rules of the parser. A parser takes a stream of tokens generated by the lexer as input and tries to map them to a set of rules where the end result maps the token streams to the AST.

3.3.1 FsLex and FsYacc

We have received two plugins for F# namely FsLex and FsYacc which are used for generating lexical analyser and parser. We can define both of them as below-

- FsLex - It will generate a code which will spilt the program string into tokens as per the guidelines for the regular expressions.
- FsYacc - It will generate a code that recognises token sequences and build an expression tree.

3.4 Abstract Syntax Tree

The abstract syntax tree for our Guarded Command Language is shown in Figure 1 below1. The root node is Program which takes a program name and list of commands as parameters.

These parameters are shown in the tuple form in the figure. Command can be of many types such as skip, write, read etc. as shown in the figure 2. These commands in turn can lead to Guarded Command or Expression. Guarded Command can be of single guard or complex guard. Single guard means that there is only one guard/test condition and complex guard means that it consists of multiple guards/test conditions. Expression can evaluate to arithmetic and boolean operations such as add, subtract, greater/smaller than etc. We use only two system data types in our language namely int and string. Labels are of type int and variables are of type string.

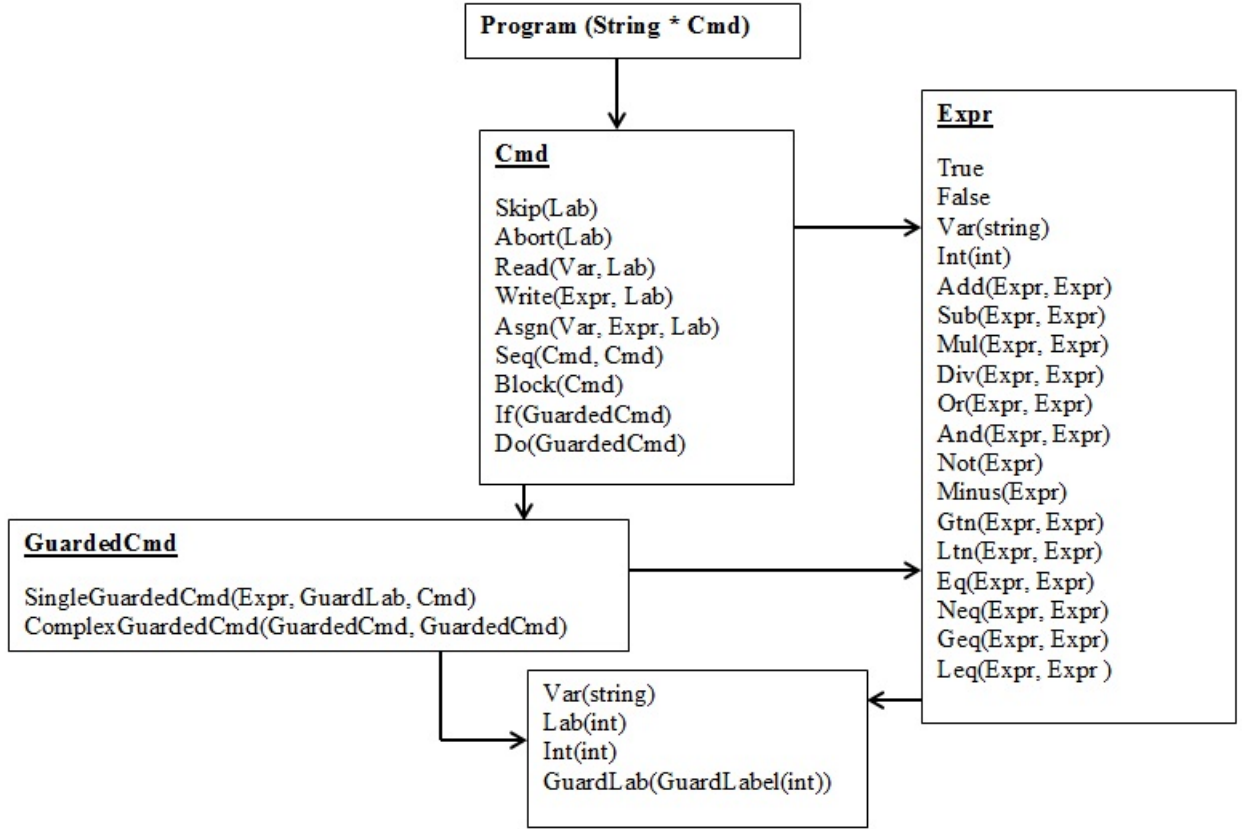


Figure 1: AST of Guarded Command Language

3.5 Data Structure

In this section we are going to present the data structures of our program. Basically, our program consists of six kinds of blocks: read, write, skip, abort, assignment and guard constructed on the basis of data flow analysis operation as mentioned in 3.2. Figure2 gives a diagrammatic representation of our block structure. All different types of blocks are inherited from a single block entity called Block. All the blocks have following common properties-

1. EntrySet: set of labels of blocks from which the current block can be reached

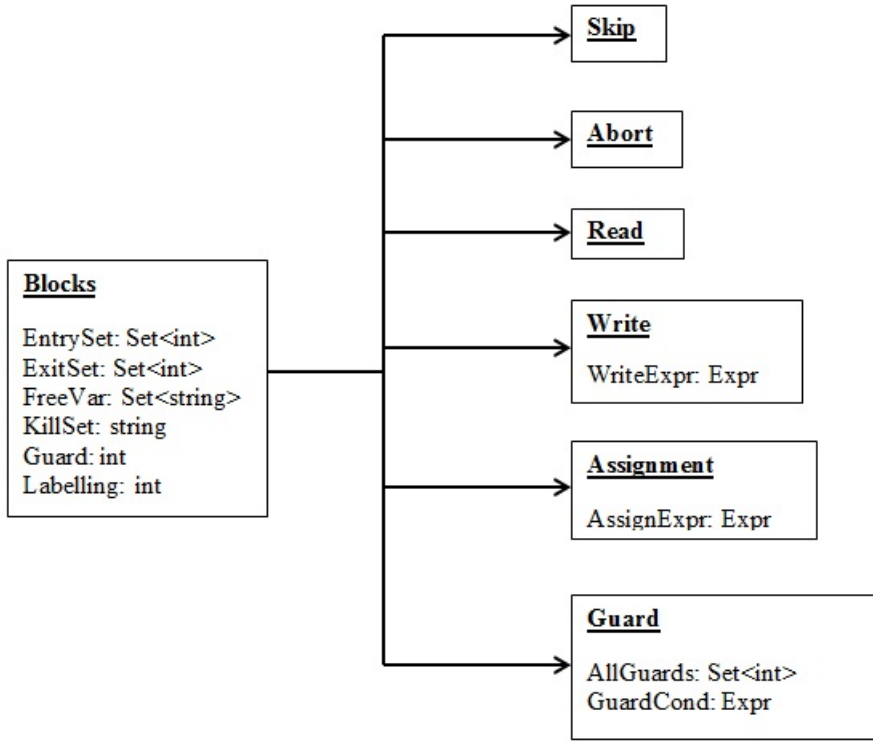


Figure 2: Block Structure

2. ExitSet: set of labels of blocks to which this blocks leads
 3. FreeVar: set of variables generated at the current block
 4. KillSet: set of variables that are killed at the current block. Variables are killed when a new value to the variable is assigned
 5. Guard: the label of the guard under which the current block resides
 6. Labelling: used to label a block
- Beside these common properties, Assignment block has an extra property called AssignExpr which is the expression of the current assignment block. For example. in ($x := a+b$), assign expression is (a+b). Similarly, Guard block takes two additional properties namely AllGuards and GuardCon. AllGuards is the set of labels of all the sister guards and the guard itself and GuardCon is the guard expression. For instance, in the following code snippet if label 1 is the current block of interest then label 3 is the sister guard of the block 1. So, AllGuards will be (1,3). The GuardCon of block 1 will be the expression of that block i.e. (x>y)

```

if [x>y] -> [x:=1]
[] [x<=y] -> [x:=2]
fi
  
```

3.6 Example

In this section, we present a sample program and describe that at each labels what kind of values we generate. Below is the sample program in Guarded Command Language:

```
module prime:
  [read x;] 1
  [prime:=true;] 2
  [m:=2;] 3
  [do m<x] 4 ->
  {
    [t:=x;] 5
    {
      do [t>m] 6 -> [t:= t-m] 7
      od;
      {
        if [t=0] 8 -> [prime:=false] 9
        [] [t!=0] 10 -> [skip] 11
        fi;
        [m:=m+1] 12
      }
    }
  } od;
  [write prime] 13
end
```

Figure 3: Program Block example

Figure4 below shows different kinds of values generated at each label. To briefly explain this, let's look at the block ($t := t-m$) at label 7. The block is an instance of assignment and thus is a type of Assign block and has assign expression (t-m). This block can be only reached via label 6 and thus entry set is {6}. Since it's a do loop the current statement will evaluate and return to do guard and if guard is not true then it will exit from there. So the exit set is {6}. It is under guard ($t>m$) which is label 6 and so guard is {6}. The free variables in this block are t and m. Since new value is assigned to t, so killset is {t}. In label 8, since it is a complex guard so it's sister guard is 10 and thus AllGuards is (8,10) and GuardExpr is the test condition i.e. (t=0).

read x;	1	Read	entry set: {} guard : 0 freeVariables: {}	exit set: {2 } killSet : {x}
prime:=true;	2	Assign	entry set: {1 } guard: 0 freeVariables: {}	exit set : {3 } AssignExpr : true killSet: {prime}
m:=2;	3	Assign	entry set: {2 } guard: 0 freeVariables: {}	exit set: {4 } AssignExpr: 2 killSet: {m}
m<x	4	Guard	entry set : {3, 12 } guard: 0 GuardExpr: m<x freeVariables: {m, x}	exit set : {5, 13 } Allguards: {} killSet: {}
t:=x;	5	Assign	entry set: {4 } guard: 4 freeVariables: {x}	exit set: {6 } AssignExpr: x killSet: {t}
t>m	6	Guard	entry set : {5, 7 } guard: 4 GuardExpr: t>m freeVariables: {m, t}	exit set : {7, 8, 10 } Allguards: {} killSet: {}
t:= t-m	7	Assign	entry set: {6 } guard: 6 freeVariables: {m,t}	exit set : {6 } AssignExpr: t-m killSet: {t}
t=0	8	Guard	entry set: {6 } guard: 4 GuardExpr: t=0 freeVariables: {t}	exit set: {9 } Allguard: {8,10} killSet: {}
prime:=false	9	Assign	entry set: {8 } guard: 8 freeVariables: {}	exit set: {12 } AssignExpr: false killSet: {prime}
t!=0	10	Guard	entry set : {6 } guard: 4 GuardExpr: t!=0 freeVariables: {t}	exit set: {11 } Allguard: {8,10} killSet: {}
skip	11	Skip	entry set : {10 } guard: 10 freeVariables: {}	exit set : {12 } killSet: {}
m:=m+1	12	Assign	entry set: {9, 11 } guard: 4 freeVariables: {m}	exit set: {4 } AssignExpr: m+1 killSet: {m}
write prime	13	Write	entry set: {4 } guard: 0 freeVariables: {prime}	exit set: {} WriteExpr: prime killSet: {}

Figure 4: Block Structure Example

4 Monotone Frameworks

There are some of the definitions and properties which are important to discuss here before proceeding further with the Data Flow Analysis and specifically Monotone Framework.

Let us define L as complete lattice whenever it is partially ordered set (L, \sqsubseteq) where each of its subset has a least upper bound. On contrary to above if we have L as a complete lattice then it satisfies the ascending chain condition only if each of the ascending chain eventually stabilizes. This means we have $(l_n)_n$ such that $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \dots$ then there exists n such that $l_n = l_{n+1} = l_{n+2} \dots$.

If f is any function which is monotone over partially ordered set (L, \sqsubseteq) if

$$\forall (l, l') \in L: l \sqsubseteq l' \Rightarrow f(l) \sqsubseteq f(l')$$

The complete lattice L satisfies the Ascending Chain Condition and set F of the monotone function with $f \in F : f \rightarrow L \rightarrow L$. This is identity function that is closed in the function composition.

The Worklist Algorithm of Monotone Framework is given below-

Input: An instance (L, F, E, i, f) of a Monotone framework Where

L : complete lattice

F - $flow(S^*)$ or $flow^R(S^*)$

E - $\{init(S^*)\} / final(S^*)$

i - initial or final analysis information

f - transfer function associated with block

Output: The MFP Solution : $MFP_{\circ}, MFP_{\bullet}$

Data Structures:

- *Analysis* : the current analysis result for block entries
- *workList* : a list of pairs (l, l') indicating that the current analysis result has changed at the entry to the block l and hence the entry information must be recomputed for l'

algorithm*MFPusingWorklist*

```
1. workList  $\leftarrow \emptyset$ ;  
2. for all  $(l, l')$  in  $F$  do  
3.   workList  $\leftarrow (l, l') :: \textit{workList}$ ;  
4. endfor  
5. for all  $l$  in  $E$  do  
6.   if  $(l \in E)$  then  
7.     Analysis $[l] = i$ ;  
8.   else  
9.     Analysis $[l] = \textit{nil}$ ;  
10.  endifor  
11. while (workList  $\neq \textit{nil}$ ) do  
12.    $(l, l') \leftarrow \textit{head}(\textit{workList})$ ;  
13.   workList  $\leftarrow \textit{tail}(\textit{workList})$ ;  
14.   if  $f(\textit{Analysis}[l]) \not\sqsubseteq \textit{Analysis}[l']$  then  
15.     Analysis $[l'] = \textit{Analysis}[l'] \sqcup f(\textit{Analysis}[l])$ ;  
16.     for all  $l''$  with  $(l', l'')$  in  $F$  do  
17.       workList  $\leftarrow (l', l'') :: \textit{workList}$ ;  
18.     endfor  
19.   endif  
20. for all  $l$  in  $E$  do  
21.    $\textit{MFP}_o(l) = \textit{Analysis}[l]$ ;  
22.    $\textit{MFP}_\bullet(l) = f(\textit{Analysis}[l])$ ;  
23. endfor
```

Figure 5: Worklist Algorithm for calculating Monotone Framework

5 Applications

5.1 Program Slicing

In this section, we will discuss technique called Program Slicing where it uses Reaching definition analysis to compute the program slice at a point of interest for a given input program.

5.1.1 Definition

Program slicing can be defined as a technique used to determine the part of a program that may influence the values computed at a given point of interest. The program part computed as the result is often referred to as the program slice.

So from the above definition it is clear that program slicing only considers about those parts of the program that may affect the values computed at the point of interest i.e slicing criterion. Program slicing ignores those entities or parts of the program which do not affect the slicing criterion. Due to this the program slice generally has less number of lines compared to the original input program. There are several benefits of using Program Slicing such as in testing, de-bugging, integration, maintainance and safety. It is quite useful in error like situations.

Given an input program and a Point of Interest (POI), the corresponding program slice can be calculated using the analysis information provided by Reaching Definitions Analysis defined in the section 5.1.3. The aim of the Reaching Definitions Analysis is to determine for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path. This analysis gives us a set of RD_{\circ} and RD_{\bullet} as result that are used in the Program Slicing algorithm. We have introduced an example in the next section for better understanding of this concept.

5.1.2 Example

In order to have more precise idea of program slicing the following example is presented. Here in the example we use an example program written in guarded command language and a POI is given as an input.

The input program has total 6 labels and 2 variables. Here the POI is at label 5. Now we can see that at label 5 we have $x = x - 1$ hence only variable x is the free variable in this expression. The value of variable x at POI depends on the commands at labels 1, 3 and 5. As the free variable at POI is x , we are not interested in the value of y . This is a simple example but important to understand the basic concept of the Program Slicing. In the code snippet above we have shown that only labels 1, 3, and 5 can influence the POI i.e. label 5. Hence the output of program slice will only contain these labels.

5.1.3 Reaching Definition Analysis

Reaching Definition Analysis is Data Flow Analysis Technique that is used to determine statistically which definitions may reach a given point in a program

```

//Program slicing having a program in Gaureded
//command language as input

Module PSexample
[x:=5] 1;      I
[y:=1] 2;
do [x>1] 3->
    [y:=x*y] 4;
    [x:=x-1] 5; //point of interest
od;
[write x] 6
end

//Output of Program Slicing
[x:=5] 1;
[x>1] 3
[x:=x-1] 5;

```

Figure 6: Example of Program Slicing

The goal of this analysis is to find out that for each program statement which assignments may has been made and not overwritten when the program execution reaches each statement along some path. Once the analysis is done then we can know for each program statement which other statement(s) may have directly influenced the computed values of a particular statement. Reaching Definition Analysis is used to find the program slice as it helps in getting neccessary information for finding the Program Slicing. It is also commonly known as an instance of the MFP framework. The data flow analysis operations for reaching definations is given below-

$$\begin{aligned}
&kill_{RD} : Blocks_* \rightarrow P(Var_*x(Lab?_*)) \\
&kill_{RD}([x := a]^l) \rightarrow \{(x, ?)\} \cup \{x, l'\} \mid B^l \text{ is a read or an assignment to } x \text{ in } C_*\} \\
&kill_{RD}([skip]^l) \rightarrow \phi \\
&kill_{RD}([abort]^l) \rightarrow \phi \\
&kill_{RD}([read \ x]^l) \rightarrow \{(x, ?)\} \cup \{x, l'\} \mid B^l \text{ is a read or an assignment to } x \text{ in } C_*\} \\
&kill_{RD}([write \ e]) \rightarrow \phi \\
&kill_{RD}([e]^l) \rightarrow \phi \\
&gen_{RD} : Blocks_* \rightarrow P(Var_*x(Lab?_*)) \\
&gen_{RD}([x := a]^l) \rightarrow \{(x, l)\} \\
&gen_{RD}([skip]^l) \rightarrow \phi \\
&gen_{RD}([abort]^l) \rightarrow \phi \\
&gen_{RD}([read \ x]^l) \rightarrow \{(x, l)\} \\
&gen_{RD}([write \ e]^l) \rightarrow \phi \\
&gen_{RD}([e]^l) \rightarrow \phi
\end{aligned}$$

The data flow equations are -

$$RD_{\circ}(l) = \{(x, ?) \mid x \in FV(C_*)\} \text{ if } l \in \text{init}(C_*)$$

$$RD_{\circ}(l) = \bigcup \{RD_{\bullet}(l') \mid (l', l) \in \text{flow}(C_*)\} \text{ otherwise}$$

$$RD_{\bullet}(l) = (RD_{\circ}(l) \setminus \text{kill}_{RD}(B^l)) \cup \text{gen}_{RD}(B^l) \text{ where } (B^l) \in \text{blocks}(C_*)$$

5.1.4 Program Slicing Algorithm

Here we present the development of the algorithm for Program Slicing. For a given input program in Guarded command language, the programSlicing algorithm returns the set of labels of the commands in the program that might influence the values computed at that point. For the blocks that do not compute anything, the idea of program slicing will be to determine the part of the program that may have influence on reaching that command. For example, if the command at the point of interest is a skip command, we would like to find which blocks had impact to reach that command. For any guarded command, the guard itself is also in the program slice of the command. It is because the guard determines when the command can be executed. If more then one guard is true, then the guarded command to be executed is chosen non-deterministically. For this reason, a command may not be executed when two of the guards were satisfied, and the other command has been chosen over this guard. Taking this into consideration a program slice of a guarded command, includes its guard and all the “sister” guards.

Given an input program, the point of interest for the program and the sets of RD_{\circ} , the following algorithm in the Figure 77 can be used to get the set of labels that makes the program slice for the given point of interest.

The two important data structures used for collecting the information that is required in the execution of the algorithm. These data structures can be explained as follows-

- programSlice- It refers to the set of labels included in the program slice.
- workList- It denotes the list of labels that are going to be analyzed in the algorithm.

The algorithm is divided into two phases namely -

- Initialization Phase - It refers to the lines 1-11 of our algorithm where two data structures i.e programSlice and workList are defined and initialized. First of all we compute the RD_{\circ} and RD_{\bullet} for the input program block. After that the programSlice is initialized as empty since there are no labels present in the slice in the beginning. At the end of the program this data structure will contain the label(s) of the program slice. Now we need to initialize the workList. At first is assigned as empty: After that we need to initialize it with the label of POI so that this label will be visited first when creating the slice. But before that we need to check if the POI itself is a guard or not. If the POI itself is a guard then we extract the labels of all the sister guards of POI also and add them in the workList along with the POI. Otherwise we add only the label of POI to the workList.

algorithmProgramSlicing (programBlocks, labelPOI)

```

1. (RDcircle, RDbullet)  $\leftarrow$  ReachingDefinitions(prgoramBlocks);
2. programSlice  $\leftarrow \emptyset$ ;
3. workList  $\leftarrow \emptyset$ ;
4. allGuards  $\leftarrow$  getAllGuards(labelPOI);
5. if (allGuards  $\neq$  nil) then
6.   for each ( $g \in$  allGuards) do
7.     workList  $\leftarrow g ::$  workList;
8.   endfor
9. else
10.  workList  $\leftarrow$  labelPOI :: [];
11. endif
12. while workList  $\neq$  nil do
13.  currentLabel  $\leftarrow$  head(workList);
14.  workList  $\leftarrow$  tail(workList);
15.  programSlice  $\leftarrow$  programSlice  $\cup$  {currentLabel};
16.  guardLabel  $\leftarrow$  ifUnderGuard(currentLabel);
17.  if (guardLabel  $\neq$  0) then
18.    allGuards  $\leftarrow$  getAllGuards(guardLabel);
19.    for each ( $g \in$  allGuards) do
20.      if (( $g \neq 0$ ) and ( $g \notin$  workList) and ( $g \notin$  programSlice)) then
21.        workList  $\leftarrow g ::$  workList;
22.      endif
23.    endfor
24.  endif
25.  for each ( $x \in$  freeVariables(currentLabel)) do
26.    for each ( $(x, lPrime) \in$  RDcircle(currentLabel)) do
27.      if (( $lprime \neq 0$ ) and ( $lprime \notin$  workList)
28.        and ( $lprime \notin$  programSlice)) then
29.        workList  $\leftarrow lPrime ::$  workList;
30.      endif
31.    endfor
32.  endfor
33. endwhile
34. return programSlice

```

Figure 7: Algorithm for Program Slicing

- **Iteration Phase-** It denotes the lines 12-33 of our algorithm. We will perform iterations from the line 12 until we have analyzed all the desired elements in the workList. This section is responsible for creating the program slice. Initially, the workList includes the label of the given point of interest and/or the sister guards of POI as this label has to be analyzed first to construct the program slice. In each iteration of the loop, the label at the head of the workList is removed from the list and analyzed. First of all, it is added to the set programSlice. Now, it is checked if the label taken into consideration at this iteration is guarded or not i.e. if the label in consideration resides within a DO loop or an IF branch. This check is done by retrieving the label of the guard using the function `ifUnderGuard()`. If the label is guarded then we need to consider all the sister guards of the guard also. For this reason we use the function `getAllGuards()`. After that these found labels are also added to the workList. Once all the guards are included, then the list of free variables involved in the arithmetic or boolean expression of this label is retrieved by using the function `freeVariables()`. Now the free variables of the label are analyzed as these set of variables have the influence on the value of the POI label block. For each of the free variables, the labels of their assignments reaching to the current label are retrieved from the RD_{\circ} of current label. After that, we add all the labels found to the workList because they influence the variables present in the current label. To handle the case, $(x, ?)$ where the variable x is uninitialized, we have used label 0 instead of ? as the label of the program starts from 1. To check that a label is a member of all labels used in the program, we check if the label is not 0. The algorithm terminates when there are no more labels in workList to be analyzed and returns the set programSlice as its return value.

5.1.5 Example

To demonstrate our Program Slicing algorithm we will use the following example program.

The Figure 9 presents a table that shows the kill set and gen set of each label of the input program:

The solution of the Reaching Definitions analysis for the given input program is presented in Figure 10.

The following table (Figure 11) lists the state of the algorithm after each iteration:

After the last iteration of the algorithm, the set programSlice $\{1, 3, 4, 12\}$ contains the labels of the program slice when the point of interest is at 12.

```

//Program Slicing using a program in Gaurded
//Command Language as inputProgram to find prime number

module prime:
  [read x] 1;
  [prime:=true] 2;
  [m:=2] 3;
  do [m<x] 4 ->
    [t:=x] 5;
    {
      do [t>m] 6 ->
        [t:= t-m] 7
      od;
      {
        if [t=0] 8 ->
          [prime:=false] 9
        [] [t!=0] 10 ->
          [skip] 11
        fi;
        [m:=m+1] 12 //POI
      }
    }
  od;
  [write prime] 13
end

//Point of Interest: 12 (m:=m+1)
Output : [1;3;4;12]
[read x] 1;
[m:=2] 3;
[m<x] 4
[m:=m+1] 12

```

Figure 8: Example of Program Slicing Algorithm

label	genRD	killRD
1	[x, 1]	[x]
2	[prime, 2]	[prime]
3	[m, 3]	[m]
4	[]	[]
5	[t, 5]	[t]
6	[]	[]
7	[t, 7]	[t]
8	[]	[]
9	[prime, 9]	[prime]
10	[]	[]
11	[]	[]
12	[m, 12]	[m]
13	[]	[]

Figure 9: kill and gen set of RD

Label	RD _*	RD [*]
1	[]	[(x, 1)]
	[(x, 1)]	[(prime, 2) (x, 1)]
3	[(prime, 2) (x, 1)]	[(m, 3) (prime, 2) (x, 1)]
4	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]
5	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (x, 1)]
6	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]
7	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 7) (x, 1)]
8	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]
9	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 9) (t, 5) (t, 7) (x, 1)]
10	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]
11	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]
12	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]
13	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]

Figure 10: Solution of the Reaching Definitions analysis

Label	RD:	WorkList	Program Slice	Free Var	Current Guard	All Guards
Initial	[]	[12]	[]	[]	[]	[]
12	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[3 4]	[12]	[]	4	[]
3	[(prime, 2) (x, 1)]	[4]	[3 12]	[]	0	[]
4	[(m, 3) (m, 12) (prime, 2) (prime, 9) (t, 5) (t, 7) (x, 1)]	[1]	[3 4 12]	[]	0	[]
1	[]	[]	[1 3 4 12]	[]	0	[]

Figure 11: State of the Program Slicing Algorithm

5.2 Dead Code Elimination

This section will present the introduction of Dead Code Elimination along with its example.

5.2.1 Definition

Dead Code Elimination is defined as an optimization technique which removes code that does not affect the overall program outcome. The code part eliminated by this technique is often referred to as dead code.

Dead code elimination also takes into account those operations who are not directly or indirectly affecting the output of the given program. It is used to remove redundant code, i.e. the part of the program which is dead, which is never executed and containing variables which are never used or irrelevant to the program execution. It uses the result of a Live Variables Analysis in order to determine the live variables in each block. A variable is live in a block if there exists some path from this block to a block downwards the program flow in which the variable is used without being re-assigned before.

5.2.2 Example

We can understand the concept of Dead Code Elimination through the below example where we took Guarded Command Language as an input program.

Now as an input program we have total of 9 labels and 3 variables as x, y and z where we can see that at label 1 and 2, variables x and y are assigned values but at label 3, variable x is again assigned a new value. Hence the previous value of 2 is overwritten at label 3 with new value equivalent to 3. In consequence the assignment in label 1 has become dead. This kind of situations will be checked by the Dead Code Elimination algorithm while evaluating the dead code. The output of the program after this analysis will be $[x:=2]^1$ as this is the dead code for the above example.

5.2.3 Live Variables Analysis

Live Variable Analysis refers to the process of determining set of live variables present at each label of any input program. Any label is marked as live if there exist some path from that label to any other label in which variable is used but it is not reassigned in that path. This type of analysis is done backwards where it makes use of the reversed program flow so as

```

// Dead Code Elimination using a program in Gaurded
// Command Language as input
Module DCexample:
[x:=2] 1; I
[y:=4] 2;
[x:=1] 3;
if [y>x] 4->
    [z:=y] 5
[] [y<=x] 6->
    [z:=y*y] 7
fi;
[x:=z] 8;
[write x] 9
end

//Output after Dead Code
Output (Dead Block): [x:=2] 1;

```

Figure 12: Example of Dead Code

to determine where the potential variable usage has propagated to all labels. It is referred as classic data flow analysis which is used by compilers at real time to determine if a variable is read by some operation before its next write.

Live Variables Analysis is used to produce the necessary analysis information required for Dead Code Elimination. Now for generating the analysis information, we use the MFP algorithm. The data flow analysis operations for live variable analysis are given below-

$kill_{LV} : Blocks_* \rightarrow P(Var_*)$

$kill_{LV}([x := a]^l) \rightarrow \{x\}$

$kill_{LV}([skip]^l) \rightarrow \phi$

$kill_{LV}([abort]^l) \rightarrow \phi$

$kill_{LV}([read\ x]^l) \rightarrow \{x\}$

$kill_{LV}([write\ x]^l) \rightarrow \phi$

$kill_{LV}([e]^l) \rightarrow \phi$

$gen_{LV} : Blocks_* \rightarrow P(Var_*)$

$gen_{LV}([x := a]^l) \rightarrow FV(a)$

$gen_{LV}([skip]^l) \rightarrow \phi$

$gen_{LV}([abort]^l) \rightarrow \phi$

$gen_{LV}([read\ x]^l) \rightarrow \phi$

$gen_{LV}([write\ e]^l) \rightarrow FV(e)$

$gen_{LV}([e]^l) \rightarrow FV(e)$

The equations are :


$LV_o(l) = \phi \text{ if } l \in \text{init}(C_*)$

$$\begin{aligned}
LV_{\circ}(l) &= \bigcup \{LV_{\bullet}(l') \mid (l', l) \in flow^R(C_*)\} \text{ otherwise} \\
LV_{\bullet}(l) &= (LV_{\circ}(l) \setminus kill_{LV}(B^l)) \cup gen_{LV}(B^l) \text{ where } (B^l) \in blocks(C_*)
\end{aligned}$$

5.2.4 Dead Code Elimination Algorithm

In this section, we discuss about the algorithm for the optimization technique Dead Code Elimination. Dead Code Elimination is a technique that removes code that does not affect the overall result computed by the program. The part of the program eliminated as the result is denoted as the dead code of the program. The algorithm of Dead Code Elimination uses the analysis information resulted from a Live Variables Analysis. The aim of the Live Variable Analysis is to determine the live variables in each program block. A variable is live in a block, if there exists some path from the residing block to another block afterwards in the program flow, where this variable is used without being modified in between. This analysis gives us a set of killLV, genLV, LV_o and LV_• as result that are used in the Dead Code Elimination algorithm.

Given an input program, the following algorithm (Figure 13) can be used to determine the set of labels that contain the dead codes that can be removed from the input program without affecting the actual flow of the program.

 **algorithmDeadCodeElimination(programBlocks, deadCodeLabels)**

1. *programCopy* \leftarrow *programBlocks*;
2. (*LVcircle*, *LVbullet*) \leftarrow *LiveVariables(programCopy)*;
3. *tempDeadCodes* \leftarrow \emptyset ;
4. *for each label* \in *allLabels(programCopy)* *do*
5. *if* $((killLV(label) \notin LVcircle(label)) \text{ and } (label \notin deadCodeLabels))$ *then*
6. *tempDeadCodes* \leftarrow *tempDeadCodes* \cup *killLV(label)*;
7. *endif*
8. *endfor*
9. *deadCodeLabels* \leftarrow *deadCodeLabels* \cup *tempDeadCodes*;
10. *if* (*tempDeadCodes* \neq \emptyset) *then*
11. *for each label* \in *tempDeadCodes* *do*
12. *programCopy[label]* \leftarrow *skip*;
13. *endfor*
14. *return algorithmDeadCodeElimination(programCopy, deadCodeLabels)*;
15. *endif*
16. *return deadCodeLabels*;

Figure 13: Algorithm for Dead Code Elimination

The phases of the algorithm is explained below:

- **Initialization Phase:** First of all, the algorithm runs the Live Variables Analysis on the input program and generates the set of `lvCircle [LVo]` which includes the variables that are live on entry to each label, in reverse program flow direction. Then it initializes the set `tempDeadCodes` as an empty set. We also initialize the `programCopy` as the input program and modify it later.
- **Iteration Phase:** Then in each iteration of the algorithm, we take each label present in the program one by one and check if the `killLV` of the label being analyzed is a subset of the `lvCircle` of that particular label. If that is the case, then it denotes that the variable assigned in this label is live and used afterwards in the program. So removing this program block will result in an uninitialized/wrong value of the variable. But if the `killLV` of the label being analyzed is not a subset of the `lvCircle` of that particular label, then it is clear that the variable assigned in this program block is no longer live. Either the variable is never used or it might be reassigned afterwards. Thus the block represented by the label being analyzed can be considered as a dead code and it can be added to the set `tempDeadCodes`. Simply removing a command from the program may result in a violation of the syntax of the language i.e. when a single guarded command has to be eliminated. To avoid this problem we have replaced the dead blocks with skip statements. This is done to compute the new value of Live Variables Analysis that helps to find the dead codes in the next iteration.
- **Recursion Phase:** The algorithm presented above is recursive in nature. The recursion of the algorithm is used to remove those variables that seem to be live in the first iteration of the algorithm but actually these variables are used in a dead statement. Thus after removing the dead codes, these variables will also become dead. To remove these newly introduced dead variables/codes, we have to re-run the whole algorithm and this is done by recursion. But before recursive call of the algorithm, we check if the set of `tempDeadCodes` is empty or not. In a way, this check helps us to determine if any changes have been made to the input program or not. We only call it recursively if any changes have been made already to the input program in this iteration. The final output of this algorithm is the set labels of the dead codes that can be eliminated from the given program.

The algorithm also works with conditional branching, due to the fact that the live variables analysis also works on multiple paths induced by condition branching constructs, e.g. IF command. A similar reasoning can be applied to DO commands. Since the analysis cannot detect if the DO is ever entered or not, assignments preceding the DO cannot be killed by the DO body, but assignments inside the body, that do not influence the DO guards and that are unused inside the body, can be killed by re-assignments after the DO. But the algorithm cannot detect the dead guards as we cannot detect using the live variables analysis result if the guard itself is true or false.

5.2.5 Example

We will use the following example given in Figure 14 to demonstrate the Dead Code Elimination algorithm-


```

//Guarded Command input program for
//Dead Code Elimination
module : add_modulo

[ read a ] 1 ;
[ read b ] 2 ;
[ read m ] 3 ;

[t:=0] 4 ;
[r:=0] 5 ;
[t:=a+b] 6 ;
[r:=a+b] 7 ;

do[ t>m ] 8 ->
    [t:=t-m] 9 ;
    [r:=t] 10 ;
od ;
[ write t ] 11
//Output (Dead Block):
[t:=0] 4
[r:=0] 5
[r:=a+b] 7
[r:=t] 10

```

Figure 14: Dead Code Algorithm Example

The following table as depicted in Figure 15 shows the kill set and gen set of each label of the input program:

The solution of the Live Variables analysis for the given input program is presented in the Figure 16-

The following table as shown in Figure 17 lists the state of the algorithm after each iteration:

After the last iteration of the algorithm, the set deadCodes {4,5,7,10} contains the labels of the program which are dead.

<i>label</i>	genLV	killLV
1	[]	a
2	[]	b
3	[]	m
4	[]	t
5	[]	v
6	[a, b]	t
7	[a, b]	r
8	[m, t]	[]
9	[m, t]	t
10	[t]	r
11	[t]	[]

Figure 15: Kill and gen set of LV

<i>label</i>	LV _o	LV _•
1	[a]	[]
2	[a, b]	[a]
3	[a, b, m]	[a, b]
4	[a, b, m]	[a, b, m]
5	[a, b, m]	[a, b, m]
6	[a, b, m, t]	[a, b, m]
7	[m, t]	[a, b, m, t]
8	[m, t]	[m, t]
9	[m, t]	[m, t]
10	[]	[m, t]
11	[]	[t]

Figure 16: Live Variable Analysis

5.3 Constant Folding

In this section, we discuss about the optimization technique Constant Folding.

5.3.1 Introduction

Constant Folding is a technique that transforms the program so that sub-expressions are replaced by their values whenever they can be determined at compile time. In other words, sub-expressions of the given program that evaluate to the same constant will be replaced by their result in this technique. The algorithm of Constant Folding uses the analysis information resulted from the Constant Propagation Analysis. Constant Propagation Analysis is the process of determining for each program point, whether or not a variable has a constant value whenever execution reaches that point. This analysis gives us a set of CPo as result giving information about the states of the variables and this information is used in the Constant Folding algorithm.

label	deadCodes	killLV	Lvo	kill _{LV} \notin LV _o
1	{}	a	[a]	FALSE
2	{}	b	[a, b]	FALSE
3	{}	m	[a, b, m]	FALSE
4	{4}	t	[a, b, m]	TRUE
5	{4,5}	v	[a, b, m]	TRUE
6	{4,5}	t	[a, b, m, t]	FALSE
7	{4,5,7}	r	[m, t]	TRUE
8	{4,5,7}	[]	[m, t]	FALSE
9	{4,5,7}	t	[m, t]	FALSE
10	{4,5,7,10}	r	[]	TRUE
11	{4,5,7,10}	[]	[]	FALSE

Figure 17: Algorithm iteration of Dead Code Elimination

5.3.2 Example

We can understand the concept of Constant Folding through the below example where we took Gaurded Command Language as a input program.

The input program has 6 labels and 3 variables as x,y and z. Now in the label 1 and 2 we are assigning values to variable x and y and at label 5 we are computing product of variable y with itself. We know that before reaching the label 5, variable y contains value 3 so at label 5 we can directly assign value 9 instead of writing $y*y$ as this will optimize our whole program. The final output of the constand folding is also shown in the above program listing where at label 6, variable z contains value 9.

5.3.3 Constant Propagation

For each program point, Constant Propagation Analysis determines whether or not a variable is constant value whenever execution reaches that point. This can be used as the basis for an optimization technique namely Constant Folding. While all the aforementioned Monotone Frameworks were distributive, Constant Propagation is not distributive.

Constant Propagation Framework consists of following definitions:

1. $Statecp = (Var_* \rightarrow ZT)$ This is the property space or the complete lattice mapping set of variables to their respective state value. A state can be top i.e. not constant, bottom i.e. undefined and CONST i.e. constant.
2. $F_{CP} = \{f \mid f \text{ is a monotone function of StateCP}\}$: the space of functions/transfer functions
3. $F = flow(S_*)$: forward flow of the set of statements S_*
4. $E = init(S_*)$: returns the initial labels of the statement in S
5. l_{CP} = variables live at the end of the program
6. f = maps labels to transfer functions F_{CP}

5.3.4 Constant Folding Algorithm

The algorithm of Constant Folding uses the analysis information resulted from the Constant Propagation Analysis. Constant Propagation Analysis gives us a set of CPo as result giving

```

// Constant Folding using a program in
//Guarded Command Language as input
Module CFexample:
  [x:=6] 1;
  [y:=3] 2;
  do [x > y] 3 ->
    [x:=x - 1] 4;
    [z:= y * y] 5
  od
  [write z] 6
end

//Output after Constant Folding
[x:=6] 1;
[y:=3] 2;
  do [x > 3] 3 ->
    [x:=x - 1] 4;
    [z:= 9] 5
  od
  [write z] 6
end

```

Figure 18: Constant Folding Example

information about the states of the variables. Given an input program, the following algorithm as shown in Figure 19 can be used to replace the sub-expressions with their constant results.

In Constant Folding, assignments in the form of assigning one variable to another one can be folded if the variable assigned is a constant. Assignments in form of an arithmetic expression can be folded partially or totally. If any one of the variables in the expression is constant then only that variable is replaced by its constant result and if all the variables involved in the expression are constant then the expression can be replaced by the evaluated result of that expression. Program blocks in the form of write will have the similar folding technique as the expression in the write block can be handled as any other expression in the right side of an assignment. The expression used in the guard construct can also be folded if any/all of the variables in that expression are constant.

The flow of the algorithm can be described in the following two phases:

- **Initialiation:** In the algorithm Constant Folding, first of all the Constant Propagation Analysis is run for the program and we get the sets of CPo and then all the labels of the program are checked. Then we make a copy of the current program blocks that will hold the folded program.
- **Iteration:** For each label of the program, we get the listOfConstants in that label from

algorithmConstantFolding (programBlocks)

```

1. (CPcircle, CPbullet) ← ConstantPropagation(programBlocks);
2. foldedProgram ← programBlocks;
3. for each label ∈ allLabels(foldedProgram) do
4.     currentCPCircle = CPcircle(label);
5.     listOfConstants ← getConstants(currentCPCircle);
6.     currentBlock ← foldedProgram(label);
7.     if ((currentBlock = Assign) or (currentBlock = Guard) or
8.        (currentBlock = Write)) then
9.         currentExp ← getExpression(currentBlock);
10.        modifiedExp ← substituteConstants(currentExp, listOfConstants);
11.        if (freeVariables(modifiedExp) = ∅) then
12.            currentExp ← evaluateExp(modifiedExp);
13.        else
14.            currentExp ← modifiedExp;
15.        endif
16.        setExpression(foldedProgram(label), currentExp);
17.    endif
18. endfor
19. return foldedProgram;

```

Figure 19: Algorithm for Constant Folding

CPo using the function `getConstants()`. This function takes the current CPo as input and returns the set of variables that are constant at that point along with their values. For each Assign//Write/Guard program block, the arithmetic/boolean expression in that block can be retrieved using the function `getExpression()`. After that all the constant variables involved in that expression are replaced by the function `replaceConstants()`. Then if there is no free variables left in the analyzed expression then it can be evaluated to a constant result using the function `evaluateExp()`. After that the folded expression is placed in the folded program blocks using `setExpression()`. After the last iteration of the algorithm, the folded program blocks will have the input program with constant folding done on the commands.

5.3.5 Example

The following example program as shown in Figure 20 is used to demonstrate the Constant Folding algorithm:

The solution of the Constant Propagation analysis for the given input program is presented in the Figure 21-

The following table as shown in Figure 22 lists the state of the algorithm after each iteration:

```

//Guarded Command input program for Constant Folding
module : checkCP
[ read x ]1 ;           I
[ read y ]2 ;
[ read h ]3 ;
[x:=1]4 ;
[y:=1]5 ;
[h:=5]6 ;
[h:=x-h]7 ;
[y:=x+h]8 ;
do[ y>0 ]9 - >
    [y:=y-1]10 ;
    [h:=x]11
od ;
[ write x ]12

//Output after Constant Folding
[ read x ]1 ;
[ read y ]2 ;           I
[ read h ]3 ;
[x:=1]4 ;
[y:=1]5 ;
[h:=5]6 ;
[h:=-4]7 ;
[y:=-3]8 ;
do[ y>0 ]9 - >
    [y:=y-1]10 ;
    [h:=1]11
od ;
[ write 1 ]12

```

Figure 20: Constant Folding Example

<i>label</i>	<i>CP₀</i>	<i>CP_•</i>
1	[("h", B); ("x", B); ("y", B)]	[("h", B); ("x", T); ("y", B)]
2	[("h", B); ("x", T); ("y", B)]	[("h", B); ("x", T); ("y", T)]
3	[("h", B); ("x", T); ("y", T)]	[("h", T); ("x", T); ("y", T)]
4	[("h", T); ("x", T); ("y", T)]	[("h", T); ("x", 1); ("y", T)]
5	[("h", T); ("x", 1); ("y", T)]	[("h", T); ("x", 1); ("y", 1)]
6	[("h", T); ("x", 1); ("y", 1)]	[("h", 5); ("x", 1); ("y", 1)]
7	[("h", 5); ("x", 1); ("y", 1)]	[("h", -4); ("x", 1); ("y", 1)]
8	[("h", -4); ("x", 1); ("y", 1)]	[("h", -4); ("x", 1); ("y", -3)]
9	[("h", -4); ("h", 1); ("x", 1); ("y", T); ("y", -4); ("y", CONST -3)]	[("h", T); ("x", 1); ("y", T)]
10	[("h", T); ("h", CONST -4); ("x", 1); ("y", T); ("y", -3)]	[("h", T); ("x", 1); ("y", T)]
11	[("h", T); ("h", -4); ("x", 1); ("y", T); ("y", -4)]	[("h", 1); ("x", 1); ("y", T)]
12	[("h", T); ("h", -4); ("x", 1); ("y", T); ("y", -3)]	[("h", T); ("x", 1); ("y", T)]

Figure 21: Constant Propagation

<i>label</i>	<i>CP₀</i>	<i>listOfConstants</i>	<i>current Expr</i>	<i>replace Constants</i>	<i>Folded Expr</i>
1	[("h", B); ("x", B); ("y", B)]	[]	read x		read x
2	[("h", B); ("x", T); ("y", B)]	[]	read y		read y
3	[("h", B); ("x", T); ("y", T)]	[]	read h		read h
4	[("h", T); ("x", TOP); ("y", T)]	[]	x:=1		x:=1
5	[("h", T); ("x", 1); ("y", T)]	[(x, 1)]	y:=1		y:=1
6	[("h", T); ("x", 1); ("y", 1)]	[(x, 1) (y, 1)]	h:=5		h:=5
7	[("h", 5); ("x", 1); ("y", 1)]	[(h, 5) (x, 1) (y, 1)]	h:=x-h	h:=1-5	h:=-4
8	[("h", -4); ("x", 1); ("y", 1)]	[(h, -4) (x, 1) (y, 1)]	y:=x+h	y:=1-4	y:=-3
9	[("h", -4); ("h", 1); ("x", 1); ("y", T); ("y", -4); ("y", -3)]	[(x, 1)]	y>0		y>0
10	[("h", T); ("h", -4); ("x", 1); ("y", T); ("y", -3)]	[(x, 1)]	y:=y-1		y:=y-1
11	[("h", T); ("h", -4); ("x", 1); ("y", T); ("y", -4)]	[(x, 1)]	h:=x	h:=1	h:=1
12	[("h", T); ("h", -4); ("x", 1); ("y", T); ("y", -3)]	[(x, 1)]	write x	write 1	write 1

Figure 22: Constant Folding Algorithm Iteration

5.4 ALFP Analysis

In this project we are performing static analysis on an input program by data flow approach. There is an alternative method namely ALFP (Alternation-Free Least Fixed Point Logic) to perform static analysis on the program. In this section we produce ALFP clauses for the two sample programmes and use web interface at <http://succinctsolver.imm.dtu.dk/> to obtain the analysis solution for Reaching Definition which returns the time to solve the problem along with the solution . We will also obtain the Reaching Definition of the two samples with our tool and measure the time that our tool takes. Then we provide some comment on timing differences. Below is the figure of two approaches to obtain a least solution of a give problem.

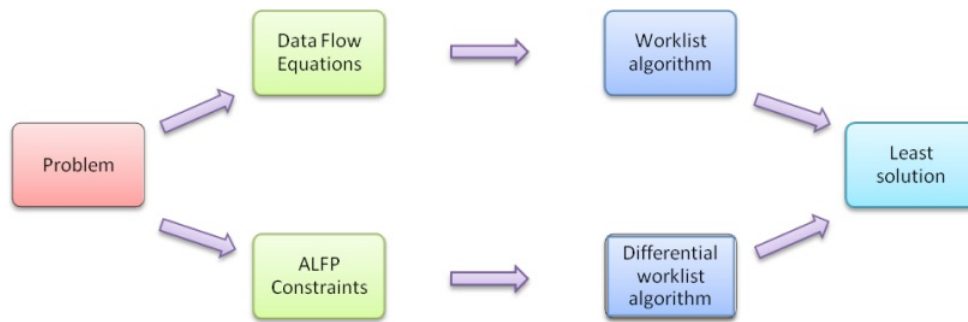


Figure 23: Static Analysis

ALFP clauses:

```

init(1) &
(flow(1, 2) & flow(2, 3) & flow(2, 5) & flow(3, 4) & flow(5, 6)) &
( kill (1, x, 0) & kill (1, x, 1) & kill (1, x, 4) & kill (2, y, 0) & kill (2, y, 2) & kill (4, x, 0)
& kill (4, x, 1) & kill (4, x, 4)) &
(gen (1, x, 1) & gen(2, y, 2) & gen (4, x, 4)) &
(extremal (x, 0) & extremal (y, 0)) &
(A l. A x. A p. init(l) & extremal(x,p) => C(l,x,p)) &

```

```

Program 1:      I
module test:
[read x;] 1
[read y;] 2
if [true] 3 -> [x:=x+2] 4
[] [x>2] 5 -> [write y] 6
fi
end

```

Figure 24: Program 1 for ALFP Testing

(A l. A p. A x. A q. flow(l,p) & B(l,x,q) => C(p,x,q)) &
 (A l. A x. A p. C(l,x,p) & !kill(l,x,p) | gen(l,x,p) => B(l,x,p))

Solving Time:

4 milliseconds (ALFP solver)

around 9.2578 milliseconds (ours)

```

Program 2
module : gcd
[ read x ]1 ;
[ read y ]2 ;
do [ x<y ]3 -> [y:=y-x]4
[] [ y<x ]5 -> [x:=x-y]6
od ;
[ write x ]7
end

```

Figure 25: Program 2 for ALFP Testing

ALFP clauses:

(init (1)) &
 (flow (6, 5) & flow (6, 3) & flow (5, 7) & flow (5, 6) & flow (4, 5) & flow (4, 3) & flow
 (3, 7) & flow (3, 4) & flow (2, 5) & flow (2, 3) & flow (1, 2)) &
 (kill(1, x, 0) & kill(1, x, 1) & kill(1, x, 6) & kill(2, y, 0) & kill(2, y, 2) & kill(2, y, 4) &
 kill(6, x, 0) & kill(6, x, 1) & kill(6, x, 6)) & (gen(1,x,1) & gen(6,x,6) &
 gen (2, y, 2) & gen (4, y, 4)) &
 (extremal (x, 0) & extremal (y, 0)) &
 (A l. A x. A p. init(l) & extremal(x,p) => C(l,x,p)) &
 (A l. A p. A x. A q. flow(l,p) & B(l,x,q) => C(p,x,q)) &
 (A l. A x. A p. C(l,x,p) & !kill(l,x,p) | gen(l,x,p) => B(l,x,p))

Solving Time:

6-7 milliseconds (ALFP solver)

around 19.7406 milliseconds (ours)

While running examples for instance program 1 in our tool the execution time was fluctuating. Under repeated execution of same program, the execution time came as close to 7 ms but the consistent time was 9 ms. However, ALFP solved the same program in 4 ms. The reason behind this time difference can be the different type of algorithms used in data flow approach and ALFP solver. ALFP as mentioned in the slide uses continuation and memoization. Continuations can eliminate list traversals and pair disassembly which can lead to reduction of heap requirements. Memoization is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed inputs. Furthermore, ALFP is using runtime stack as worklist which will also result in time reduction.

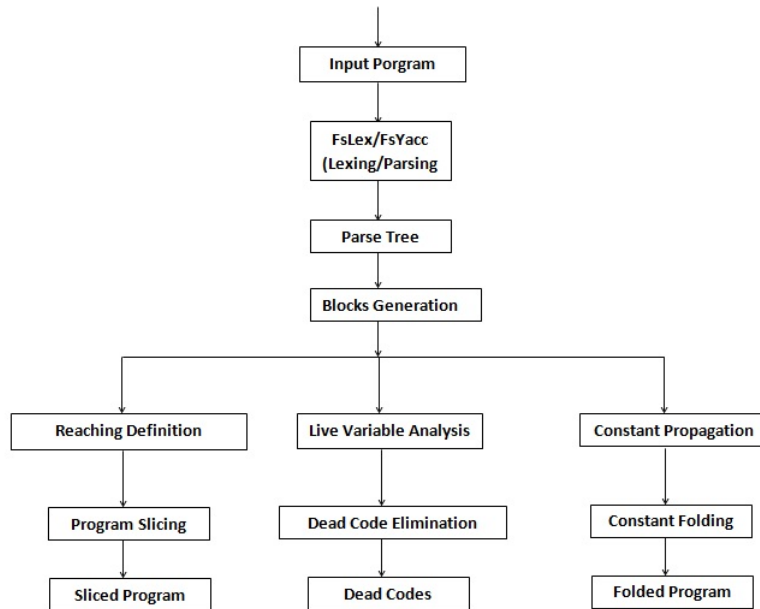


Figure 26: Implementation Structure

6 Implementation

In this section, we'll present the details about the implementation part of the project.

6.1 Language

The project is developed in a functional language. We choose F# as the language of implementation. We choose this language over other functional languages like Haskell, ML etc. because of the familiarity with .NET framework.

6.2 Overall Design

In Figure 26, we present a diagram that explains the overall design of our tool and then explain each step of implementation.

Step 1: A program written in Guarded Command Language is given as input

Step 2: FsLex and FsYacc is run on the input to produce an Abstract Syntax Tree

Step 3: After we get a parse tree we then generate blocks (skip, abort, read, write, assignment and guard) for the program at each label

Step 4: User can then choose which analysis to run on the parsed program. Choices are Program Slicing, Dead Code Elimination and Constant Folding

Step 5: In program slicing, we use the algorithm of Reaching Definition Analysis to produce program slice for a given point of interest. In dead code elimination, we run the Live Variable Analysis to produce a set of dead codes. And finally for the constant folding we use the algorithm of Constant Propagation to output a folded program. No matter which analysis

we are performing we use the MFP algorithm to generate the respective circles and bullets i.e. RDCircle and RDBullet in case of Reaching Definition Analysis.

6.3 File Details

After we have given an overall structure of our program, we would like to give some details of how our code is structured that is to give details about different files.

6.3.1 Program/Main

This is the main file or entry point to the program. This file contains code for creating command line menu.

6.3.2 BlockDefinition

Codes here are used to define various types of blocks as described in Figure2. The base class is Block and all other types of Blocks are derived from this base class. Only,

Assignment and Guard classes have extra members as described in section 3.53.5

6.3.3 BlockGenerator

This file is used to create different types of blocks defined in BlockDefinition. Following are the main functions used in this file:

generateProgramBlocks (program:Program):Cmd :: This takes a parsed program as input and returns the list of commands in the program. This function calls another function to analyze what kind of commands are in the list. Also it generates init and exitset.

analyzeCommand (cmd:Cmd) (previousExitLabel:Set<int>): This takes a list of command and last label created before the current label. It then tries to match the types of command from the command list and calls the respective classes to generate the respective block. For example, if a command is write then it will match with Write command and then it will call “generateWriteBlock” function to generate a write block. It works similarly for other blocks.

analyzeGC (gc:GuardedCmd) (allGuards:Set<int>) (previousExitLabel:Set<int>): This takes a guarded command and all its sister guards as well as last label before the current label as input. It then tries to analyze if the gC is complex or single. If complex it will call “analyzeComplexGC” and if gC is single it will call “analyzeSingleGC”. “analyzeSingleGC” will eventually create a guard block. Similarly, for do command first “analyzeDO” is called which will eventually call this “analyzeGC”.

forwardFlow\reverseFlow (currentBlocks:Dictionary<int,Block>): These functions are used to compute forward and reverse flow the passed block of the program.

6.3.4 MFP

MFP performs the worklist algorithm to calculate the respective circles and bullets of the program i.e. `RDcircle\RDbullet` in case of Reaching Definition and `LVcircle\LVbullet` in case of Live Variable Analysis. MFP takes following parameters:

- L : complete lattice
- F - `flow(Cmd*)` or `flowR(Cmd*)`
- E - `{init(Cmd*)}/final(Cmd*)`
- i - initial or final analysis information
- f - transfer function associated with `B[l]` in `currentBlocks(Cmd*)`

6.3.5 ProgramSlice

This is used for calculating the program slice. Input parameter to this is point of interest. It then calls MFP to get `RDcircle` and `RDbullet` and then it runs the reaching definition algorithm to get the program slice.

RDtransfer (RDcircle:Set<(string*int)>) (label:int) (currentBlocks:Dictionary<int,Block>)::

This is the transfer function defined in this file which takes `RDcircle`, current label and program block as input. This transfer function is then passed to MFP along with other parameters as described in the section 6.3.4

6.3.6 DeadCodeElimination

This is used to calculate dead codes for a program.

LVtransfer (LVcircle:Set<string>) (label:int) (currentBlocks:Dictionary<int,Block>):Set<string>

:: This is the transfer function for live variable analysis which is passed to MFP. After getting `LVcircle` and `LVbullet` from MFP this file then runs the live variable analysis algorithm to find the dead codes of the program. In dead code elimination, we recursively call the dead code algorithm until the program has no more dead codes.

6.3.7 ConstantFolding

This is used to produce a folded program i.e. replacing the constant values in the program. We use the constant propagation algorithm to achieve a constant folded program. Following things are done in this:

1. State: this defines the state of a variable i.e. whether it is constant, undefined or not a constant.

2. functions are used to fold the respective blocks i.e. `foldAssignment` to fold the assignment, `foldWrite` to fold a write etc.

3. functions mentioned in step 2 call “`foldExpression`”

4. Then “`replaceConstants`” is called to replace variable with constant and after that “`evaluateExpr`” is called to evaluate the expressions i.e. add, multiply, comparisons etc.

5. **CPtransfer (CPcircle:Set<(string*State)>) (label:int) (currentBlocks:Dictionary<int,Block>):Set<(string*State)>::** is the transfer function for this.

6.3.8 DebugPrinting

This file contains the code for debugging and printing purposes.

6.4 Running Instructions

Following steps are required to run the project:

1. A command box appears with the list of available commands.
2. First you need to parse the file. To parse a file use command: `parse filepath` where filepath is the absolute path of the file. To parse a string, use command: `parse "string"` where string is the program.
3. To slice a program, use command: `PS poi [detail]` where poi is number of label for which you want to slice the program. `detail` is optional and if used will print the extra information such as killset, genset, RDCircles, RDbullets etc.
4. For dead code elimination, use command: `DC [detail]`
5. For constant propagation, use command: `CP [detail]`
6. `help` prints the list of commands
7. `exit` will exit the program
8. commands are not case sensitive

7 Evaluation

In this section, we will evaluate our whole implementation on the basis of the program analysis techniques we practiced and different set of exhaustive test cases.

7.1 Testing

It is important part of any development project as it helps to ensure the correctness of any implementation using different list of test cases. We have manually carried the whole testing without using any tool or scripting mechanisms. The results of the manual testing was verified twice to avoid any human error in the same. We will discuss brief testing performed, results obtained and their analysis to understand the behaviour of our framework.

7.1.1 Testing (Program Slicing)

We use our own benchmark program and other benchmark programs to test the program slicing. Here we provide the test output for the benchmark program provide by group 3 which evaluated to the results mentioned by them.

```
Input program:
module : Fermat
[ read n ]1 ;
[ total:=3]2 ;
[x:=1]3 ;
do[ x<=total-2 ]4
[y:=1]5 ;
do[ y<=total-x-1 ]6
[z:=total-x-y]7 ;
[xpn:=1]8 ;
[i:=1]9 ;
do[ i<n ]10 [xpn:=xpn*i]11 ;
[i:=i+1]12 od ; [ypn:=1]13 ;
[i:=1]14 ;
do[ i<n ]15
[ypn:=ypn*i]16 ;
[i:=i+1]17 od ;
[zpn:=1]18 ;
[i:=1]19 ;
do[ i<n ]20
[zpn:=zpn*i]21 ;
[i:=i+1]22 od ;
if [ exp+ypn=zpn ]23
[ write n ]24 fi ;
[y:=y+1]25 od ;
[x:=x+1]26
od
```

POI: 13

Slice Program :: set [2; 3; 4; 5; 6; 13; 25; 26]

7.1.2 Testing (Dead Code Elimination)

For this also we tested a bunch of programs including our own benchmark programs. Here we provide a test output of group 16:

```
Input program:
module :
prime [ read n ]1 ;
[prime:=1]2 ;
if
[ n<2 ]3 [prime:=0]4
[] [ n=2 ]5 [prime:=1]6
[][ n>2 ]7
[i:=2]8 ;
[prime:=1]9 ;
do[ i<=n/2&prime=1 ]10
if
[ n/i*i=n ]11
[prime:=0]12
[][ n/i*i!=n ]13
[ skip ]14 fi ;
[i:=i+1]15 od
fi ;
[ write prime ]16
```

Dead Codes "[2]"

7.1.3 Testing (Constant Folding)

As all other tests we tested with a number of programs and here we present the test output of our own benchmark program.

```
Input program:
module : checkCP
[ read x ]1 ;
[ read y ]2 ;
[ read h ]3 ;
[x:=1]4 ;
[y:=1]5 ;
[h:=5]6 ;
[h:=x-h]7 ;
[y:=x+h]8 ;
```

```
do[ y>0 ]9
[y:=y-1]10 ;
[h:=x]11 od ;
[ write h ]12
```

Folded Program

```
"[ read x ]1"
"[ read y ]2"
"[ read h ]3"
"[ x:=1]4"
"[ y:=1]5"
"[ h:=5]6"
"[ h:=-4]7"
"[ y:=-3]8"
"[y>0]9"
"[ y:=y-1]10"
"[ h:=1]11"
"[ write h]12"
```


8 Conclusion

The project experience was really beneficial for all three of us as we were able to apply the theoretical knowledge gained during the course lectures. Before attending this course - “Program Analysis” was a new concept for all of us but now at the completion of project report we have gained sufficient knowledge about different aspects of the statical analysis namely the “Data Flow Analysis”. We learnt and implemented program analysis techniques such as “Reaching Definitions Analysis”, “Live Variable Analysis” and “Constant Propagation Analysis” which are implemented as instances of an MFP framework on a Guarded Command Language. We used the applications such as “Program Slicing”, “Dead Code Elimination” and “Constant Folding ” for gaining information so as to perform above mentioned program analysis techniques. These applications are used for optimizing the program codes. All the three team members had experience in using different programming languages but none of us had any experience of using “Functional Language” but we have taken this opportunity as a challenge. We have successfully implemented project problem in F# using Visual studio 2010. We have explained the particular modules of the implementation in detail in terms of their functionality. We tested the correctness of our application based on several input programs written in Guarded Command Languages. The results of the output we get from our application were compared with the manually calculated results to ensure the correctness of the results. Furthermore, we also tested the code examples provided by other groups as uploaded on the Campusnet. At present our application is performed one type of analysis on the user input which means on the input program, the tool will ask for type of analysis the user want like - Program slicing, Dead Code Elimination or Constant Folding. This shows that our implementation offers some features and we also foresee possibilities for improving it such that on input program, our tool will perform all three types of analysis to produce a optimized result. The reason behind the time difference while running the analysis on our system and the provided ALP system can be explained as - Different type of algorithms used in data flow approach and ALFP solver. ALFP as mentioned in the slide uses continuation and memoization. Continuations can eliminate list traversals and pair disassembly which can lead to reduction of heap requirements. Memoization is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed inputs. Furthermore, ALFP is using runtime stack as worklist which will also result in time reduction.

9 Future Work

As we explained in the Conclusion section, we intend to device a framework which can perform different optimizations that we studied during the course at the same time. It means that we give input program to our tool then it will apply Program Slicing then Dead Code Elimination and Constant Folding. We would also like to reduce the computation time our algorithms are taking through different mechanisms introduced in the course. Furthermore, Guarded command language can be extended to include more constructs and types so as to test our framework on it.

10 Reference

1. F. Nielson, H. Nielson, C. Hankin, Book on Principles of Program Analysis, Springer-Verlag New York Inc, 1999
2. Lecture Notes by Hanne R. Nielson, 02242 Program Analysis, Denmark Technical University, DTU, 2010
3. Visual Studio 2010, Microsoft Corporation <http://www.microsoft.com/visualstudio/en-us/products/2010-editions>
4. Tutorial on F#, http://en.wikibooks.org/wiki/Category:F_Sharp_Programming
5. F# Language, Microsoft Research, <http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/>

11 Source Code

The source code is attached on the next page and the same is also uploaded on the Campusnet.

Lang.fs

```
module Lang

open System

type Var = string
type Int = int
type Lab = int

type Program =
    | Program of string * Cmd

and Cmd =
    | Skip of Lab
    | Abort of Lab
    | Read of Var * Lab
    | Write of Expr * Lab
    | Asgn of Var * Expr * Lab
    | Seq of Cmd * Cmd
    | Block of Cmd
    | If of GuardedCmd
    | Do of GuardedCmd

and GuardedCmd =
    | SingleGuard of Expr * GuardLab * Cmd // GuardLab was added to make the labelling in sequence
    | ComplexGuard of GuardedCmd * GuardedCmd
    // member generating all the sister guards including its own
    member this.allGuards =
        let rec findGuards gc : Set<int> =
            match gc with
            | SingleGuard (expr, lab, cmd) -> Set.empty.Add(lab.value)
            | ComplexGuard(gc1, gc2) -> findGuards gc1 + findGuards gc2
        findGuards this

    // to make the labels in sequence
and GuardLab =
    | GuardLabel of int
    // member holding the int value
    member this.value =
        let findValue gl:int =
            match gl with
            | GuardLabel(lab) -> lab
        findValue this

and Expr =
    | True
    | False
    | Var of string
    | Int of int
    | Add of Expr * Expr
    | Sub of Expr * Expr
    | Mul of Expr * Expr
    | Div of Expr * Expr
    | Or of Expr * Expr
    | And of Expr * Expr
    | Not of Expr
    | Minus of Expr
    | Gtn of Expr * Expr
    | Ltn of Expr * Expr
    | Eq of Expr * Expr
    | Neq of Expr * Expr
    | Geq of Expr * Expr
    | Leq of Expr * Expr
```

```

// member holding the free variables in the expression
member this.FreeVariables =
    let rec getFreeVariables exp =
        match exp with
        | True -> Set.empty
        | False -> Set.empty
        | Var(x) -> Set[x]
        | Int(c) -> Set.empty
        | Add(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Sub(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Mul(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Div(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Or(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | And(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Not(exp1) -> getFreeVariables exp1
        | Minus(exp1) -> getFreeVariables exp1
        | Gtn(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Ltn(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Eq(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Neq(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Geq(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
        | Leq(exp1,exp2) -> getFreeVariables exp1 + getFreeVariables exp2
    getFreeVariables this

// Override funtion for string representation of expression
override this.ToString() =
    let rec exprToString exp =
        match exp with
        | True -> "TRUE"
        | False -> "FALSE"
        | Var(x) -> string(x)
        | Int(c) -> string(c)
        | Add(exp1,exp2) -> exprToString exp1 + "+" + exprToString exp2
        | Mul(exp1,exp2) -> exprToString exp1 + "*" + exprToString exp2
        | Div(exp1,exp2) -> exprToString exp1 + "/" + exprToString exp2
        | Sub(exp1,exp2) -> exprToString exp1 + "-" + exprToString exp2
        | Or(exp1,exp2) -> exprToString exp1 + "|" + exprToString exp2
        | And(exp1,exp2) -> exprToString exp1 + "&" + exprToString exp2
        | Not(exp1) -> "!" + exprToString exp1
        | Minus(exp1) -> "-" + exprToString exp1
        | Gtn(exp1,exp2) -> exprToString exp1 + ">" + exprToString exp2
        | Ltn(exp1,exp2) -> exprToString exp1 + "<" + exprToString exp2
        | Eq(exp1,exp2) -> exprToString exp1 + "=" + exprToString exp2
        | Neq(exp1,exp2) -> exprToString exp1 + "!=" + exprToString exp2
        | Geq(exp1,exp2) -> exprToString exp1 + ">=" + exprToString exp2
        | Leq(exp1,exp2) -> exprToString exp1 + "<=" + exprToString exp2
    exprToString this

```

LangParser.fs

```
%{
open System
open Lang
let currentLabel = ref 1
let labelCmd() =
    let tmp = !currentLabel
    incr currentLabel
    tmp
let getLabel():int =
    let l = !currentLabel
    l-1
}%

%token <string> ID
%token <int> NUMBER
%token OR AND ADD DIV MUL NOT MINUS EQ LTN GTN NEQ LEQ GEQ DO IF ASGN COLON
%token OD FI LPAREN RPAREN LBRACK RBRACK GUARD MODULE END SEMICOLON ARROW SKIP
%token ABORT READ WRITE TRUE FALSE
%token EOF

// start
%start start
%type <Program> start
%type <Cmd> Command
%%

start: LangProgram { $1 }

LangProgram: MODULE ID COLON Command END { Program($2, $4) }

Command:
| ID ASGN Expression { Asgn($1, $3, labelCmd()) }
| SKIP { Skip(labelCmd()) }
| ABORT { Abort(labelCmd()) }
| READ ID { Read($2, labelCmd()) }
| WRITE Expression { Write($2, labelCmd()) }
| Command SEMICOLON Command { Seq($1, $3) }
| LBRACK Command RBRACK { $2 }
| DO GCmd OD { Do($2) }
| IF GCmd FI { If($2) }

GCmd:
| Expression Arrow Command { SingleGuard($1,$2,$3) }
| GCmd GUARD GCmd { ComplexGuard($1,$3) }

Arrow:
| ARROW {labelCmd(); GuardLabel(getLabel())}

Expression:
| ID { Var($1) }
| NUMBER { Int($1) }
| TRUE { True }
| FALSE { False }
| LPAREN Expression RPAREN { $2 }
| Expression ADD Expression { Add($1, $3) }
| Expression MINUS Expression { Sub($1, $3) }
| Expression MUL Expression { Mul($1, $3) }
| Expression DIV Expression { Div($1, $3) }
| Expression OR Expression { Or($1, $3) }
| Expression AND Expression { And($1, $3) }
| NOT Expression { Not($2) }
| MINUS Expression { Minus($2) }
| Expression GTN Expression { Gtn($1, $3) }
```

```
| Expression LTN Expression { Ltn($1, $3) }  
| Expression GEQ Expression { Geq($1, $3) }  
| Expression LEQ Expression { Leq($1, $3) }  
| Expression EQ Expression { Eq($1, $3) }  
| Expression NEQ Expression { Neq($1, $3) }
```

%%

Langlexer.fsl

```
{
module Langlexer
open System
open LangParser
open Microsoft.FSharp.Text.Lexing
}

let char      = ['a'-'z' 'A'-'Z']
let digit     = ['0'-'9']
let identifier = char(char|digit|['_' ''])*
let number    = digit(digit)*
let whitespace = ' ' | '\t'
let newline   = '\r' '\n' | '\n' | '\r'

rule tokenize = parse
| whitespace { tokenize lexbuf }
| newline { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf; }
| "read" { READ }
| "end" { END }
| "module" { MODULE }
| "write" { WRITE }
| "true" { TRUE }
| "false" { FALSE }
| "skip" { SKIP }
| "abort" { ABORT }
| "(" { LPAREN }
| ")" { RPAREN }
| "{" { LBRACK }
| "}" { RBRACK }
| "do" { DO }
| "od" { OD }
| "if" { IF }
| "fi" { FI }
| "!=" { ASGN }
| "+" { ADD }
| "-" { MINUS }
| "*" { MUL }
| "/" { DIV }
| "|" { OR }
| "&" { AND }
| "!" { NOT }
| ">" { GTN }
| "<" { LTN }
| ">=" { GEQ }
| "<=" { LEQ }
| "=" { EQ }
| "!=" { NEQ }
| ":" { COLON }
| ";" { SEMICOLON }
| "[" { GUARD }
| "->" { ARROW }
| identifier { ID (String(System.Text.Encoding.ASCII.GetChars(lexbuf.Lexeme))) }
| number { NUMBER (Int32.Parse(String(System.Text.Encoding.ASCII.GetChars(lexbuf.Lexeme)))) }
| eof { EOF }
```

BlockDefinition.fs

```
module BlockDefinition

open Lang

/// type for a basic block
type Block(entry_set:int Set, free_var:string Set, kill:string option,guard:int,scopenumber:int)=
    let mutable entrySet = entry_set
    let mutable exitSet:int Set = Set.empty
    let mutable freeVar = free_var
    let mutable killSet = kill
    let mutable scopeNumber = scopenumber
    let mutable guardNo = guard
    member self.EntrySet
        with get() = entrySet
        and set(in_set)= entrySet <- in_set
    member self.ExitSet
        with get() = exitSet
        and set(out_set) =exitSet <- out_set
    member self.FreeVar
        with get() =freeVar
        and set(free_V) = freeVar <- free_V
    member self.KillSet
        with get() = killSet
        and set(kill_s) = killSet <- kill_s
    member self.Guard
        with get() = guardNo
        and set(gd_no) = guardNo <- gd_no
    member self.ScopeNumber
        with get() = scopeNumber
        and set(nsn) = scopeNumber <- nsn
    abstract member Labelling: int -> string
    default self.Labelling (label:int) = "[" + self.ToString() + "]" + string(label)
    abstract member Clone: unit -> Block
    default self.Clone() =
        let tempBlock = new Block(entrySet, Set.empty, None,scopenumber, guard)
        tempBlock.ExitSet <- self.ExitSet
        tempBlock

/// Type for skip block
type Skip(entry_set:int Set,guard:int, scopenumber:int) =
    inherit Block(entry_set, Set.empty,None,guard,scopenumber)
    override self.ToString() = "skip"
    override self.Clone() =
        let tempSkip = new Skip(self.EntrySet,self.Guard,self.ScopeNumber)
        tempSkip.ExitSet <- self.ExitSet
        tempSkip :> Block

/// Type for abort block
type Abort(entry_set:int Set, guard:int,scopenumber:int) =
    inherit Block(entry_set, Set.empty, None,guard,scopenumber)
    override self.ToString() ="abort"
    override self.Clone() =
        let tempAbort = new Abort(self.EntrySet, self.Guard, self.ScopeNumber)
        tempAbort.ExitSet <- self.ExitSet
        tempAbort :> Block

/// Type for read block
type Read(entry_set:int Set, kill:string option, guard:int, scopenumber:int) =
    inherit Block(entry_set, Set.empty, kill, guard, scopenumber)
    override self.ToString() =
        "read " + self.KillSet.Value
    override self.Clone() =
        let tempRead = new Read(self.EntrySet, self.KillSet,self.ScopeNumber, self.Guard)
```

```
tempRead.ExitSet <- self.ExitSet
tempRead :> Block
```

```
/// Type for write block
```

```
type Write(entry_set:int Set, free_var:string Set, expr: Expr, guard:int,scopenumber:int) =
  inherit Block(entry_set, free_var, None, guard, scopenumber)
  let mutable writeExpr = expr
  member self.WriteExpr
    with get() = writeExpr
    and set(exp) = writeExpr <- exp
  override self.ToString() = "write " + self.WriteExpr.ToString()
  override self.Clone() =
    let tempWrite = new Write(self.EntrySet, self.FreeVar, self.WriteExpr, self.Guard, self.
ScopeNumber)
    tempWrite.ExitSet <- self.ExitSet
    tempWrite :> Block
```

```
/// Type for assignment block
```

```
type Assignment(entry_set:int Set,free_var:string Set,kill:string option,expr:Expr,guard:int,scopenumber:
int) =
  inherit Block(entry_set,free_var,kill,guard, scopenumber)
  let mutable assignExpr = expr
  member self.AssignExpr
    with get():Expr =assignExpr
    and set(exp) = assignExpr <- exp
  override self.ToString() =
    self.KillSet.Value + ":@" + (self.AssignExpr.ToString())
  override self.Clone() =
    let tempAssign = new Assignment(self.EntrySet,self.FreeVar, self.KillSet, self.AssignExpr,self.
Guard, self.ScopeNumber)
    tempAssign.ExitSet <- self.ExitSet
    tempAssign :> Block
```

```
/// Type for guard block
```

```
type Guard(entry_set:int Set, free_var:string Set, guard_exp: Expr,all_guards:int Set, guard:int,
scopenumber:int) =
  inherit Block(entry_set, free_var, None, guard, scopenumber)
  let mutable guardExp = guard_exp
  let mutable allGuards = all_guards
  member self.AllGuards
    with get() = allGuards
    and set(gSet) = allGuards <- gSet
  member self.GuardCon
    with get() = guardExp
    and set(exp) = guardExp <- exp
  override self.ToString() =self.GuardCon.ToString()
  override self.Clone() =
    let tempGuard = new Guard(self.EntrySet, self.FreeVar, self.GuardCon,self.AllGuards, self.Guard,
self.ScopeNumber)
    tempGuard.ExitSet <- self.ExitSet
    tempGuard :> Block
```

DebugPrinting.fs

```
module DebugPrinting

open Lang
open BlockDefinition
open System.Collections.Generic

///function to print the contents of blocks
let printBlocks(analysedProgramBlocks:Dictionary<int,Block>) =

    let mutable nextBlocks =List.empty
    nextBlocks <- [1]

    let printed=ref Set.empty

    while not nextBlocks.IsEmpty do
        let label = nextBlocks.Head
        if Set.contains label (Set(analysedProgramBlocks.Keys)) then

            printf "label %d : " label

            printf "\n"
            printf "entry set {"
            for entry in analysedProgramBlocks.[label].EntrySet do
                printf "%d " entry
            printf "}"

            printf "\n"
            printf "exit set {"
            for exitL in analysedProgramBlocks.[label].ExitSet do
                printf "%d " exitL
            printf "}"

            printf "\n"
            printf "guard number: %d " analysedProgramBlocks.[label].Guard

            printf "\n"
            printf "freeVariables: %A" analysedProgramBlocks.[label].FreeVar

            printf "\n"
            printf "killSet : %A\n" analysedProgramBlocks.[label].KillSet

            let mutable temp = List.empty
            for exitLab in analysedProgramBlocks.[label].ExitSet do
                if not (Set.contains exitLab !printed) then
                    temp <- temp @ [exitLab]
            nextBlocks<-(nextBlocks.Tail) @ temp

            printed:=!printed + Set.filter(fun elem->not(Set.contains elem !printed))((
                analysedProgramBlocks.[label]):Block).ExitSet)
            printf "\n"

// function generating string from command given as parameter
let rec commandToString(cmd: Cmd):string=
    match cmd with
    |Asgn(var,exp,lab)-> "[" + var+ " :=" + exp.ToString() + "]" + (lab:int).ToString()
    |Skip(lab)-> "[ skip ]" + (lab:int).ToString()
    |Abort(lab)-> "[ abort ]" + (lab:int).ToString()
    |Read(var, lab)-> "[ read " + var + " ]" + (lab:int).ToString()
    |Write(exp, lab)-> "[ write " + exp.ToString() + " ]" + (lab:int).ToString()
    |Do(gC)-> "do" + (gcToString gC) + "\nod"
```



```

|If(gC)-> "if " + "\n" + (gcToString gc) + "fi "
|Seq(cmd1, cmd2) -> (commandToString cmd1) + " ;\n" + (commandToString) cmd2
|Block(cmd1)-> "{" + (commandToString cmd1) + "}"

and gcToString (gc: GuardedCmd): string=
match gc with
|SingleGuard(exp, glab, cmd)-> "[ " + exp.ToString() + " ]" + (glab.value).ToString() + "\n" +
(commandToString cmd)
|ComplexGuard(gc1, gc2)-> "\n" + (gcToString gc1) + "\n[]" + (gcToString gc2)

let programToString (prg: Program) =
match prg with
|Program(name, cmd)-> "module : " + name + "\n" + commandToString(cmd) + "\n"

/// Function printing a single block into a string
let blockToString (currentBlocks:Dictionary<int,Block>)(label:int)=

match (currentBlocks.[label]):Block with
| :? Skip-> "[skip]" + (label.ToString())
| :? Abort-> "[abort]" + (label.ToString())
| :? Read as read when read.KillSet.IsSome-> "[read " + read.KillSet.Value + "]" + (label.ToString())
| :? Write as write-> "[write " + (write.WriteExpr.ToString()) + "]" + (label.ToString())
| :? Assignment as assign when assign.KillSet.IsSome-> "[ " + (assign.KillSet.Value) + " :=" + (assign.
AssignExpr.ToString())+ "]" + (label.ToString())
| :? Guard as guard-> ("[" + (guard.GuardCon.ToString()) + "]" + (label.ToString()))
| _ -> ""

let rec allBlocksToString (cmd:Cmd) (currentBlocks:Dictionary<int,Block>) =

match cmd with
| Skip (label) -> "\n" + (blockToString (currentBlocks) (label))
| Abort(label) -> "\n" + (blockToString (currentBlocks) (label))
| Read(var,label) -> "\n" + (blockToString (currentBlocks) (label))
| Write(exp,label) -> "\n" + (blockToString (currentBlocks) (label))
| Asgn(var,exp,label) -> "\n" + (blockToString (currentBlocks) (label))
| Seq(cmd1,cmd2)-> "\n" + ((allBlocksToString (cmd1) (currentBlocks) ) + " ;" + (allBlocksToString
(cmd2) (currentBlocks)))
| Block(cmd) -> "\n" + ( "{" + (allBlocksToString (cmd) (currentBlocks) ) + "\n }")
| If(gc) -> "\n" + ("If " + (GCToString gc currentBlocks) + " fi")
| Do(gc) -> "\n" + ("Do " + (GCToString gc currentBlocks)+ " od")

and GCToString (gc:GuardedCmd) (currentBlocks:Dictionary<int,Block>) =
match gc with
|SingleGuard(exp,label,cmd)->
[" " + (exp.ToString()) + "]" + ((label.value).ToString()) + "->\n" + (allBlocksToString (cmd)
(currentBlocks))
|ComplexGuard(gc1,gc2)->
(GCToString (gc1) (currentBlocks) )+ "\n[]" + (GCToString (gc2) (currentBlocks))

```

BlockGenerator.fs

```
module BlockGenerator
open System
open System.IO
open System.Collections.Generic
open Lang
open BlockDefinition
open DebugPrinting

/// holds the program blocks and corresponding start label
let blocks = new Dictionary<int,Block>()

/// list of guards the command is nested under
let mutable underGuards:int list = [0]
/// scope entered
let mutable scope:int list = [0]

let mutable programFinal:Set<int> = Set.empty
let mutable programInit:Set<int> = Set.empty
let mutable allVariables:Set<string> = Set.empty

/// function to get a clone of blocks
let CloneBlocks(currentBlocks:Dictionary<int,Block>) =
    let clonedBlocks = new Dictionary<int,Block>()
    for key in currentBlocks.Keys do
        clonedBlocks.[key] <- currentBlocks.[key].Clone()
    clonedBlocks

/// type of exception thrown if parse error
exception ParseError of string

/// function for parsing the program
let parseFormString (guardProg:string) =
    let lexbuf = Lexing.LexBuffer<byte>.FromBytes (System.Text.Encoding.ASCII.GetBytes(guardProg))
    try
        LangParser.start LangLexer.tokenize lexbuf
    with ex ->
        raise (ParseError("Parse Error. Incorrect syntax"))

/// function for parsing file
let parseFromFile inputFilePath =
    use stream = System.IO.File.OpenRead(inputFilePath)
    use reader = new System.IO.BinaryReader(stream)
    let lexbuf = Lexing.LexBuffer<byte>.FromBinaryReader reader
    LangParser.start LangLexer.tokenize lexbuf

/// function extracting information from Assign statement and generating block
let generateAssignBlock (var:string) (exp:Expr) (currentLabel:Lab) (previousExitLabel:Set<int>) =
    let assignBlock =
        new Assignment(previousExitLabel,exp.FreeVariables,Some(var), exp, underGuards.Head, List.head scope)
    blocks.[currentLabel] <- assignBlock

    allVariables <- allVariables.Add(var) + Set.ofSeq(exp.FreeVariables);

    // returning current label of the assignment
    Set[currentLabel]

/// function extracting information from skip statement and generating block
let generateSkipBlock (currentLabel:Lab) (previousExitLabel:Set<int>) =
    let skipBlock = new Skip(previousExitLabel,underGuards.Head, List.head scope)
    blocks.[currentLabel] <- skipBlock
    // returning current label of the skip
    Set[currentLabel]
```

```

/// function extracting information from abort statement and generating block
let generateAbortBlock (currentLabel:Lab) (previousExitLabel:Set<int>) =
    let abortBlock = new Abort(previousExitLabel,underGuards.Head, List.head scope)
    blocks.[currentLabel] <- abortBlock
    // returning empty set
    Set.empty

/// function extracting information from read statement and generating block
let generateReadBlock (var:string) (currentLabel:Lab) (previousExitLabel:Set<int>) =
    let readBlock =
        new Read(previousExitLabel, Some(var), underGuards.Head, List.head scope)
    blocks.[currentLabel] <- readBlock

    allVariables <- allVariables.Add(var);

    // returning current label of the Read
    Set[currentLabel]

/// Function extracting information from write statement and generating block
let generateWriteBlock (exp:Expr) (currentLabel:Lab) (previousExitLabel:Set<int>) =
    //let currentLabel = label//labels.nextLabel()
    let writeBlock =
        new Write(previousExitLabel, exp.FreeVariables, exp, underGuards.Head, List.head scope)
    blocks.[currentLabel] <- writeBlock

    allVariables <- allVariables + Set.ofSeq(exp.FreeVariables);

    // returning current label of the write
    Set[currentLabel]

/// function extracting information from sequence of commands
let rec analyzeSequence (cmd1:Cmd) (cmd2:Cmd) (previousExitLabel:Set<int>) =
    let exitCmd1 = analyzeCommand cmd1 previousExitLabel
    let exitCmd2 = analyzeCommand cmd2 exitCmd1
    exitCmd2

/// function extracting information from single GC command
and analyzeSingleGC (exp:Expr) (cmd:Cmd) (currentLabel:Lab) (allGuards:Set<int>) (previousExitLabel:Set
<int>) =
    let guardBlock =
        new Guard(previousExitLabel, exp.FreeVariables,exp,allGuards, underGuards.Head, List.head scope)
    blocks.[currentLabel]<-guardBlock

    allVariables <- allVariables + Set.ofSeq(exp.FreeVariables);

    // adding the label of the current guard to underGuard
    underGuards <- currentLabel :: underGuards
    let tempSet: Set<int> = Set.empty.Add(currentLabel)
    let exitSet =
        analyzeCommand cmd tempSet
    //removing the current guard from underGuard
    underGuards <- underGuards.Tail
    exitSet

/// function extracting information from complex GC command
and analyzeComplexGC (gc1:GuardedCmd) (gc2:GuardedCmd) (allGuards:Set<int>) (previousExitLabel:Set<int>) =
    let exitSet1 =analyzeGC gc1 allGuards previousExitLabel
    let exitSet2 =analyzeGC gc2 allGuards previousExitLabel
    exitSet1 + exitSet2

/// function extracting information from GC command
and analyzeGC (gc:GuardedCmd) (allGuards:Set<int>) (previousExitLabel:Set<int>) =
    match gc with
    |SingleGuard(exp,label,cmd)->

```

```

        analyzeSingleGC exp cmd label.value allGuards previousExitLabel
|ComplexGuard(gc1,gc2)->
        analyzeComplexGC gc1 gc2 allGuards previousExitLabel

/// function extracting information from Do command
and analyzeDO (gc:GuardedCmd) (previousExitLabel:Set<int>) =
    let exitGC = analyzeGC gc gc.allGuards previousExitLabel
    //exit from GC command is appended to the entry of Do
    for i in gc.allGuards do
        blocks.[i].EntrySet <- blocks.[i].EntrySet + exitGC
    gc.allGuards

/// function extracting information from block of command
and analyzeBlockCommand (cmd:Cmd) (previousExitLabel:Set<int>) =
    analyzeCommand cmd previousExitLabel

/// function to find out the type of the command
and analyzeCommand (cmd:Cmd) (previousExitLabel:Set<int>) =
    match cmd with
    | Skip (label) -> generateSkipBlock label previousExitLabel
    | Abort(label) -> generateAbortBlock label previousExitLabel
    | Read(var,label) -> generateReadBlock var label previousExitLabel
    | Write(exp,label) -> generateWriteBlock exp label previousExitLabel
    | Asgn(var,exp,label) -> generateAssignBlock var exp label previousExitLabel
    | Seq(cmd1,cmd2)-> analyzeSequence cmd1 cmd2 previousExitLabel
    | Block(cmd) -> analyzeCommand cmd previousExitLabel
    | If(gc) -> analyzeGC gc gc.allGuards previousExitLabel
    | Do(gc) -> analyzeDO gc previousExitLabel

/// function to generate exit set
let generateExitSet() =
    for label in blocks.Keys do
        for entryLabel in blocks.[label].EntrySet do
            blocks.[entryLabel].ExitSet <- blocks.[entryLabel].ExitSet.Add(label)

/// function to generate forward flow sequence of int*int
let forwardFlow(currentBlocks:Dictionary<int,Block>)=
    seq<(int*int)>{ for label in currentBlocks.Keys do
        for finalLabel in currentBlocks.[label].ExitSet do
            //printfn "forwardFlow %A" (label, finalLabel)
            yield(label,finalLabel)} |> List.ofSeq |> List.rev |> Seq.ofList

/// function to generate reverse flow sequence of int*int
let reverseFlow(currentBlocks:Dictionary<int,Block>)=
    seq<(int*int)>{ for label in currentBlocks.Keys do
        for initLabel in currentBlocks.[label].EntrySet do
            //printfn "reverse flow %A" (label,initLabel)
            yield(label,initLabel)}

/// function to parse the program and generate blocks by calling analyzeCommand
let generateProgramBlocks (program:Program):Cmd =

    // parsing input file using generated parser
    printfn "\nInput program:\n"
    printfn "%s" (programToString(program))
    let command (prg: Program) =
        match prg with
        |Program(name, cmd)-> cmd

    let (programCmd:Cmd) = command(program)

    programFinal <- analyzeCommand programCmd Set.empty
    programInit <- blocks.[1].EntrySet
    generateExitSet ()
    (programCmd)

```

```
/// function for clearing the data structures
let clearDataHandlers()=
  blocks.Clear()
  underGuards <- []
  scope <-[]
```

MFP.fs

```
module MFP
open System
open BlockDefinition
//open BlockGenerator
open System.Collections.Generic
open System.Diagnostics

// for execution time
let sw = new Stopwatch()
// for printing purposes
/// string to store the details of program execution for printing purpose
let mutable printProgramDetails:string= ""
/// fun to print all the set contents
let printFull param:string =
    let mutable result = "["
    for elem in param do
        result <- result + " " + elem.ToString()
    result <- result + " ]"
    result
let mutable circle_bullet=""
let mutable printDetail:bool = false
let mutable printKillGenStat:bool = false

/// MFP algorithm for solving data flow equations
// L : complete lattice
// F - flow(Cmd*) or flowR(Cmd*)
// E - {init(Cmd*)}/final(Cmd*)
// i - initial or final analysis information
// f - transfer function associated with B[l] in currentBlocks(Cmd*)
// currentBlocks - map of blocks created for the program to be analysed

let MFP (L:Set<int>) (F:seq<(int*int)>) (E:Set<int>) (i:Set<'a>) (f:Set<'a->int->Dictionary<int,Block>->
Set<'a>)(currentBlocks:Dictionary<int,Block>)=
    sw.Start()
    let mutable workList = List.empty
    for tuple in F do
        workList <- tuple::workList

    let Analysis:Set<'a>[] = [|for label in currentBlocks.Keys -> Set.empty|]// to change
    for label in currentBlocks.Keys do
        if (E.Contains label) then
            Analysis.[label-1] <- i
    while(workList.Length > 0) do
        let (l,lprime) = workList.Head
        workList<-(workList.Tail)
        let mutable fAnalysis = Set.empty
        fAnalysis <- f (Analysis.[l-1]) l currentBlocks
        if (not (fAnalysis.IsSubsetOf(Analysis.[lprime-1])) ) then
            Analysis.[lprime-1] <- Set.union Analysis.[lprime-1] fAnalysis
            let flowlPrime =
                seq<(int*int)>{for tuple in F do
                    if(fst(tuple) = lprime) then
                        yield tuple}
            workList<-((List.ofSeq flowlPrime) @ workList)
    let mutable MFPCircle:Set<'a>[] = [| for l in currentBlocks.Keys->Set.empty|]
    let mutable MFPbullet:Set<'a>[] = [| for l in currentBlocks.Keys->Set.empty|]
    for label in currentBlocks.Keys do
        printKillGenStat <- true;
        MFPCircle.[label-1]<-Analysis.[label-1]
        MFPbullet.[label-1]<-f (Analysis.[label-1]) label currentBlocks
        circle_bullet <- circle_bullet + "\n\n Label " + label.ToString() + "\n Circle::: " + (printFull
MFPCircle.[label-1])
```

```
    circle_bullet <- circle_bullet + "\n Bullet::: " + (printFull MFPbullet.[label-1])
sw.Stop()
printfn "Elapsed time %A" (String.Format("Minutes :{0} Seconds :{1} Mili seconds :{2}", sw.Elapsed.
Minutes, sw.Elapsed.Seconds, sw.Elapsed.TotalMilliseconds))
sw.Reset()
(MFPcircle, MFPbullet)
```

ProgramSlice.fs

```
// Module for Program Slice using Reaching Definitions analysis
module ProgramSlice
open System.Collections.Generic
open BlockDefinition
open BlockGenerator
open MFP
open DebugPrinting

/// Initializing RD circle with {(x,?)|x in FV(Cmd*)}

let externalValue() : Set<(string*int)> =
    Set.map(fun(var)-> (var,0))(allVariables)

/// Reaching Definitions analysis Transfer Function
let RDtransfer (RDcircle:Set<(string*int)>) (label:int) (currentBlocks:Dictionary<int,Block>)=
    // if any var killed at label
    if printKillGenStat then
        printProgramDetails <- printProgramDetails + " At Label:: " + label.ToString()
    if currentBlocks.[label].KillSet.IsSome then
        let var = currentBlocks.[label].KillSet.Value

        // include every other tuple except (var,1) i.e. RDcircle(label)\killRD
        let rdTransfer =
            seq{for tuple in RDcircle do
                if fst(tuple) <> var then
                    yield tuple}
        // Add (var,label) i.e. genRD
        let currentTuple = Seq.ofList [(currentBlocks.[label].KillSet.Value, label)]
        if printKillGenStat then
            printProgramDetails <- printProgramDetails + "\n      Kill set:: " + var
            printProgramDetails <- printProgramDetails + "\n      Gen set :: " + (printFull currentTuple)
        + "\n\n"
            printKillGenStat<-false
            let RDtransfer = List.ofSeq(rdTransfer) @ List.ofSeq(currentTuple)
            Set.ofList RDtransfer
        else
            if printKillGenStat then
                printProgramDetails <- printProgramDetails + "\n      kill set:: [] \n      Gen set:: []\n\n"
                printKillGenStat<-false
            RDcircle

/// Defining MFP call to calculate Reaching Definitions Analysis
let ReachingDefinition(currentBlocks:Dictionary<int,Block>) =
    MFP Set.empty (forwardFlow(currentBlocks)) programInit (externalValue()) RDtransfer currentBlocks

/// function to return all the sister guards of a guard block
let getAllGuards (block : Block) : Set<int> =
    match block with
    | :? Guard as guard -> guard.AllGuards
    | _ -> Set.empty

/// function to check if a list contains an element
let containsElem number list = List.exists (fun elem -> elem = number) list

/// Program Slicing iterative algorithm.
let programSlice (currentBlocks:Dictionary<int,Block>) (poiLabel:int) =

    // calculate RDcircle and RDbullet
    let (RDcircle:Set<(string*int)>[], RDbullet:Set<(string*int)>[]) = ReachingDefinition(currentBlocks)
    // storing the detail of Program Slice for detail printing purpose
    let mutable PS = Set.empty
```

```

let mutable workList = []
let mutable allGuards = Set.empty

//if POI is a guard itself then get all the sister guards including its own guard
if (poiLabel < currentBlocks.Count && poiLabel>0) then
    allGuards <- getAllGuards (currentBlocks.[poiLabel])
    // if it is a guard then add all sister guards including its own guard
    if (not(allGuards.IsEmpty)) then
        for g in allGuards do
            if ((g <> 0) && (not (containsElem g workList)) && (not (PS.Contains g))) then
                workList <- g :: workList
            else // if its not a guard then just add its label
                workList <- poiLabel::[]
// storing details of the process
printProgramDetails <- printProgramDetails + " Algorithm:: \n At initial :: \n\t WorkList:: " +
(printFull (Set.ofList workList)) + "\n\t Program Slice:: " + (printFull (PS))
    + "\n\t Free Var:: ...." + "\n\t RDCircle:: ...." + "\n\t All Guards:: ....\n"
\n"

// loop until workList is null
while not(workList.IsEmpty) do
    let currentLabel = workList.Head
    workList <- workList.Tail
    PS <- Set.union PS (Set.empty.Add(currentLabel))

// get the guard of currentLabel
let mutable guardLabel:int = currentBlocks.[currentLabel].Guard

//if it is guarded under a do of if
let mutable printAllGuards = Set.empty
if (guardLabel > 0) then
    // get all the sister guards including its own guard
    let mutable allGuards = Set.empty

    allGuards <- getAllGuards (currentBlocks.[guardLabel])

// add all sister guards including its own guard
if (not(allGuards.IsEmpty)) then
    for g in allGuards do
        if ((g <> 0) && (not (containsElem g workList)) && (not (PS.Contains g))) then
            workList <- g :: workList

// for each var in freeVar(currentLabel)
let mutable printFreeVar:Set<string> = Set.empty
for var in currentBlocks.[currentLabel].FreeVar do
    printFreeVar.Add(var) |> ignore
    for tuple in RDCircle.[currentLabel-1] do
        if(fst(tuple)=var) then
            let lprime = snd(tuple)
            // if not 0 and was not already in PS or workList
            if (lprime <> 0
                && (not (containsElem lprime workList))
                && (not (Set.contains lprime PS))) then
                //add to worklist
                workList <- lprime::workList

    printProgramDetails <- printProgramDetails + "\n Label :: " + currentLabel.ToString() + "\n\t
WorkList:: "
    + (printFull (Set.ofList workList)) + "\n\t Program Slice:: " +
(printFull PS)
    + "\n\t Free Var:: " + (printFull printFreeVar) + "\n\t RDCircle::
"+ (printFull RDCircle.[currentLabel-1])
    + "\n\t Current Guard:: " + guardLabel.ToString() +
    "\n\t All Guards:: " + (printFull allGuards)

if printDetail then

```

```
        printfn "\n\n Program Slice Details :::::> \n\n"
        printBlocks(currentBlocks)
        printfn "\n\n RDcircle And RDbullet::\n %A" circle_bullet
        printfn "\n\n %A" printProgramDetails
        printProgramDetails <- ""
    printfn "\n\n Sliced Program :: %A" (printFull PS)
    PS
else
    printfn "\n Sorry the menthion POI doesn't exist in the Program"
    printDetail <- false
    PS
```

DeadCodeElimination.fs

```
// Module for Dead Code Elimination
module DeadCodeElimination
open System.Collections.Generic
open BlockDefinition
open BlockGenerator
open MFP
open DebugPrinting

// storing the detail of Dead Code Elimination for detail printing purpose
let mutable deadCodeStr:string=""
let mutable fCallcount = 0

/// Live Variables analysis Transfer Function
let LVtransfer (LVcircle:Set<string>) (label:int) (currentBlocks:Dictionary<int,Block>):Set<string> =
    let mutable lvTransfer = Set.empty
    let genLV = currentBlocks.[label].FreeVar

    if printKillGenStat then
        printProgramDetails <- printProgramDetails + "\n\n Label " + label.ToString()
        printProgramDetails <- printProgramDetails + "\n          genLV :: " + (printFull genLV)

    if (currentBlocks.[label].KillSet.IsNone) then
        lvTransfer <- LVcircle
        if printKillGenStat then
            printProgramDetails <- printProgramDetails + "\n          KillLV :: []"
            printKillGenStat <- false
        else
            for var in LVcircle do
                if (not(var = (currentBlocks.[label].KillSet.Value))) then
                    lvTransfer <- lvTransfer.Add(var)
            if printKillGenStat then
                printProgramDetails <- printProgramDetails + "\n          KillLV :: " + currentBlocks.[label].KillSet.Value
                printKillGenStat <- false

    lvTransfer <- lvTransfer + genLV
    lvTransfer

/// Calculating live variable analysis
let LiveVariables(currentBlocks:Dictionary<int,Block>)
    = MFP Set.empty (reverseFlow(currentBlocks)) programFinal (Set.empty) LVtransfer currentBlocks

let mutable programCopy = new Dictionary<int,Block>()
let mutable deadCodes = Set.empty

/// Algorithm for Dead Code Elimination
let deadCodeElimination(currentBlocks:Dictionary<int,Block>) =

    // make a copy of the program blocks
    programCopy <- CloneBlocks(currentBlocks)

    let recCount = ref 0;
    let rec deadCodeDetection ():Set<int>=

        let (LVcircle:Set<(string)>[], LVbullet:Set<(string)>[]) = LiveVariables(programCopy)

        let mutable tempDeadCodes = Set.empty
        let forCount = ref 0;
        deadCodeStr <- " "
        deadCodeStr <- deadCodeStr + "\n Dead Codes at :: "
        tempDeadCodes <- Set [for label in programCopy.Keys do
            // label not in deadcode already and there is any kill
            incr forCount
            deadCodeStr <- deadCodeStr + "\n          Iteration " + (!forCount).ToString() + " "
        ]
```

```

:: " + "current label " + label.ToString() + ": "
    if (not(deadCodes.Contains(label)) && (programCopy.[label].KillSet.IsSome)) ✓
then
    let var = programCopy.[label].KillSet.Value
    if not(LVcircle.[label-1].Contains(var)) then
        deadCodeStr <- deadCodeStr + label.ToString()
        yield label]

    deadCodes <- deadCodes + tempDeadCodes
    incr recCount
    printProgramDetails <- printProgramDetails + "\n" + deadCodeStr + "\nToal DeadCodes at recursive ✓
iteration " + (!recCount).ToString()
    + ":: " + (printFull deadCodes)
    if not(tempDeadCodes.IsEmpty) then
        //replace all dead blocks with skip
        for l in tempDeadCodes do
            let skipBlock = new Skip(programCopy.[l].EntrySet,programCopy.[l].Guard, programCopy.[l]. ✓
ScopeNumber)
            skipBlock.ExitSet <- programCopy.[l].ExitSet
            programCopy.[l] <- skipBlock
            // run dead code detection on the updated blocks
            deadCodeDetection() |> ignore
    if printDetail then
        printfn ">>\n"
        printBlocks(currentBlocks)
        printfn "\n\n%A" circle_bullet
        printfn "\n\n DeadCode Elimination Details ::::>> \n\n %A" printProgramDetails
        deadCodeStr<-" "
        circle_bullet<-" "
        printDetail <- false
    deadCodes
deadCodeDetection()

```

ConstantFolding.fs

```
module ConstantFolding

open MFP
open Lang
open BlockDefinition
open BlockGenerator
open System.Collections.Generic

/// Type definition for the State of a variable.
/// If it is Constant->CONST, Not a constant->TOP or Undefined ->BOTTOM
type State =
    | TOP
    | CONST of int
    | BOTTOM
    static member (+) (x:State, y:State) =
        match (x,y) with
        | (CONST(c1), CONST(c2)) -> CONST(c1+c2)
        | (TOP, _) -> TOP
        | (_, TOP) -> TOP
        | (_, _) -> BOTTOM
    static member (-) (x:State, y:State) =
        match (x,y) with
        | (CONST(c1), CONST(c2)) -> CONST(c1-c2)
        | (TOP, _) -> TOP
        | (_, TOP) -> TOP
        | (_, _) -> BOTTOM
    static member ( *) (x:State, y:State) =
        match (x,y) with
        | (CONST(c1), CONST(c2)) -> CONST(c1*c2)
        | (TOP, _) -> TOP
        | (_, TOP) -> TOP
        | (_, _) -> BOTTOM
    static member (/) (x:State, y:State) =
        match (x,y) with
        | (CONST(c1), CONST(c2)) -> CONST(c1/c2)
        | (TOP, _) -> TOP
        | (_, TOP) -> TOP
        | (_, _) -> BOTTOM

/// Function to check if a variable is a constant or not
let isConstant (state:State) =
    match state with
    | CONST(c) -> true
    | _ -> false

/// get the integer value of CONST state.
let valueOfState (state:State) = match state with
    | CONST(c) -> c
    | _ -> 0

/// Function to perform arithmetic operations on variables of type State.
/// Returns of the State type of the given expression
let rec findStateOfExpression (currentMap:Map<string,State>) (exp:Expr) =
    match exp with
    | Int(c) -> CONST(c)
    | Var(x) -> currentMap.[x]
    | Add(exp1,exp2) -> (findStateOfExpression currentMap exp1) + (findStateOfExpression currentMap exp2)
    | Sub(exp1,exp2) -> (findStateOfExpression currentMap exp1) - (findStateOfExpression currentMap exp2)
    | Mul(exp1,exp2) -> (findStateOfExpression currentMap exp1) * (findStateOfExpression currentMap exp2)
    | Div(exp1,exp2) -> (findStateOfExpression currentMap exp1) / (findStateOfExpression currentMap exp2)
    | _ -> BOTTOM

/// initial set of variables with their initial state for CP Analysis.
/// initially all variables have the state BOTTOM
let initCP() : Set<(string*State)> =
```

```

Set.map(fun(var)-> (var,BOTTOM))(allVariables)

/// Function to determine the current State of each variable in the given CPcircle
let mapOfState (currentCP:Set<string*State>) =

    let mutable tempMap:Map<string,State> = Map.empty

    let mutable count = 0
    let mutable state = BOTTOM
    for var in allVariables do
        count <- 0 //state <- BOTTOM
        for tuple in currentCP do
            if var = fst(tuple) then
                count <- count + 1 //if (isConstant (snd(tuple))) then
                state <- (snd(tuple))
            if count = 1 then //tempMap <- tempMap.Add(var,BOTTOM) //else if count = 2 ✓
            then
                tempMap <- tempMap.Add(var, state)
            else
                tempMap <- tempMap.Add(var,TOP)
        tempMap

/// find the constant variables in the given cp_bullet
let getMapOfConstants (CPbullet:Map<string,State>) =
    let mapOfConstants = CPbullet |> Map.toList |> Set.ofList |> Set.filter (fun (x,y) -> (isConstant y))
    mapOfConstants

/// Transfer function for Constant Propagation Analysis
let CPtransfer (CPcircle:Set<(string*State)>) (label:int) (currentBlocks:Dictionary<int,Block>) :Set<
(string*State)> = ✓

    let mutable tempMap = mapOfState CPcircle
    let state =
        match currentBlocks.[label] with
        | :? Assignment as assignment -> Some(findStateOfExpression tempMap assignment.AssignExpr)
        | :? Read as read -> Some(TOP)
        | _ -> None
    if state.IsSome then
        let key = currentBlocks.[label].KillSet.Value
        let newstate = //match tempMap.[key] with //| CONST(c) when tempMap.[key] <> state.Value -> TOP
        //| CONST(c) when tempMap.[key] = state.Value -> tempMap.[key] //| CONST(c) -> state.Value
        //| _ when tempMap.[key] = TOP -> TOP //| _ ->
        state.Value
        tempMap <- tempMap.Add(key,newstate)
    Set.ofList(Map.toList(tempMap))

/// Calculating the Constant Propagation
let ConstantPropagation (currentBlocks:Dictionary<int,Block>) =
    MFP Set.empty (forwardFlow(currentBlocks)) (Set[1]) (initCP()) CPtransfer currentBlocks

/// Function to substitute the variables with constant in an expression
let rec replaceConstants (exp:Expr) (mapWithConstants:Map<string,State>) =
    match exp with
    | Int(c) -> Int(c)
    | Var(x) when mapWithConstants.ContainsKey(x) -> Int(valueOfState mapWithConstants.[x])
    | Var(x) -> Var(x)
    | Add(exp1,exp2) -> Add((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants)) ✓
    | Sub(exp1,exp2) -> Sub((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants)) ✓
    | Mul(exp1,exp2) -> Mul((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants)) ✓

```



```

mapWithConstants))
| Div(exp1,exp2) -> Div((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| And(exp1,exp2) -> And((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Or(exp1,exp2) -> Or((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Not(exp1) -> Not(replaceConstants exp1 mapWithConstants)
| Minus(exp1) -> Minus(replaceConstants exp1 mapWithConstants)
| Gtn(exp1,exp2) -> Gtn((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Ltn(exp1,exp2) -> Ltn((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Eq(exp1,exp2) -> Eq((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Neq(exp1,exp2) -> Neq((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Geq(exp1,exp2) -> Geq((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| Leq(exp1,exp2) -> Leq((replaceConstants exp1 mapWithConstants),(replaceConstants exp2
mapWithConstants))
| True -> True
| False -> False
// ConstantFolding.fs

```

```

/// Function to evaluate expression
let rec evaluateExpression (exp:Expr) =
  match exp with
  | Var(x) -> Var(x)
  | Int(c) -> Int(c)
  | Add(Int(c1),Int(c2)) -> Int(c1+c2)
  | Add(Var(x),_) | Add(_, Var(x)) -> exp
  | Sub(Int(c1),Int(c2)) -> Int(c1-c2)
  | Sub(Var(x),_) | Sub(_, Var(x)) -> exp
  | Mul(Int(c1),Int(c2)) -> Int(c1*c2)
  | Mul(Var(x),_) | Mul(_, Var(x)) -> exp
  | Div(Int(c1),Int(c2)) -> Int(c1/c2)
  | Div(Var(x),_) | Div(_, Var(x)) -> exp
  | Add(exp1, exp2) -> evaluateExpression (Add(evaluateExpression exp1, evaluateExpression exp2))
  | Sub(exp1, exp2) -> evaluateExpression (Sub( evaluateExpression exp1, evaluateExpression exp2))
  | Mul(exp1, exp2) -> evaluateExpression (Mul( evaluateExpression exp1, evaluateExpression exp2))
  | Div(exp1, exp2) -> evaluateExpression (Div(evaluateExpression exp1, evaluateExpression exp2))
  | Gtn(exp1, exp2) -> (Gtn(evaluateExpression exp1, evaluateExpression exp2))
  | Ltn(exp1, exp2) -> (Ltn(evaluateExpression exp1, evaluateExpression exp2))
  | Geq(exp1, exp2) -> (Geq(evaluateExpression exp1, evaluateExpression exp2))
  | Leq(exp1, exp2) -> (Leq(evaluateExpression exp1, evaluateExpression exp2))
  | Eq(exp1, exp2) -> (Eq(evaluateExpression exp1, evaluateExpression exp2))
  | Neq(exp1, exp2) -> (Neq(evaluateExpression exp1, evaluateExpression exp2))
  | And(exp1, exp2) -> evaluateExpression (And(evaluateExpression exp1, evaluateExpression exp2))
  | Or(exp1, exp2) -> evaluateExpression (Or(evaluateExpression exp1, evaluateExpression exp2))
  | Not(exp1) -> evaluateExpression (Not(evaluateExpression exp1))
  | Minus(exp1) -> evaluateExpression (Minus(evaluateExpression exp1))
  | _ -> evaluateExpression exp

```

```

/// Constant folding of Expression
let foldExpression (exp:Expr) (mapWithConstants:Map<string,State>) =
  let exprWithConstant = replaceConstants exp mapWithConstants
  // printfn "\nReplace Constant %A" exprWithConstant
  evaluateExpression exprWithConstant

```

```

/// Constant Folding of Assignment
let foldAssignment (assignment:Assignment) (mapWithConstants:Map<string,State>) =
  let foldedExpression = foldExpression assignment.AssignExpr mapWithConstants
  assignment.AssignExpr <- foldedExpression
  assignment.FreeVar <- foldedExpression.FreeVariables

```

```

/// Constant Folding of Write statements.
let foldWrite (write:Write) (mapWithConstants:Map<string,State>) =
    let foldedExpression = foldExpression write.WriteExpr mapWithConstants
    write.WriteExpr <- foldedExpression
    write.FreeVar <- foldedExpression.FreeVariables

/// Constant Folding of Guard
let foldGuard (guard:Guard) (mapWithConstants:Map<string,State>) =
    let foldedExpression = foldExpression guard.GuardCon mapWithConstants
    guard.GuardCon <- foldedExpression
    guard.FreeVar <- foldedExpression.FreeVariables

/// function to perform Constant Folding and returns the Folded program
let constantFolding (currentBlocks:Dictionary<int,Block>):(Dictionary<int,Block>)=

    // Calculate CPcircle and CPbullet
    let (CPcircle:Set<(string*State)>[], CPbullet:Set<(string*State)>[]) = ConstantPropagation
    (currentBlocks)

    let foldedBlocks = CloneBlocks(currentBlocks)
    for label in foldedBlocks.Keys do
        let currentCPcircle = CPcircle.[label - 1]
        let setOfConstants = getMapOfConstants (mapOfState currentCPcircle)
        let mapOfConstants = setOfConstants |> Set.toList|> Map.ofList
        match (foldedBlocks.[label]):Block with
        | :? Assignment as assignment -> foldAssignment assignment mapOfConstants
        | :? Guard as guard -> foldGuard guard mapOfConstants
        | :? Write as write -> foldWrite write mapOfConstants
        | _ -> ()

    // printing purpose
    printProgramDetails <- printProgramDetails + "\n\n Label: " + label.ToString()
    printProgramDetails <- printProgramDetails + "\n  CPcircle: " + (printFull CPcircle.[label-1])
    printProgramDetails <- printProgramDetails + "\n  CPbullet: " + (printFull CPbullet.[label-1])
    printProgramDetails <- printProgramDetails + "\n  Current Exp: " + (foldedBlocks.[label]).ToString()
    (
        printProgramDetails <- printProgramDetails + "\n  listofConstant " + (printFull setOfConstants)
        printProgramDetails <- printProgramDetails + "\n  Map of Constants: " + (printFull mapOfConstants)
        printProgramDetails <- printProgramDetails + "\n  Folded Exp: " + (foldedBlocks.[label]).ToString()
    )

    foldedBlocks

```

Program.fs

```
// Program.fs

module Main
open System
open BlockDefinition
open BlockGenerator
open DebugPrinting
open ProgramSlice
open DeadCodeElimination
open ConstantFolding
open MFP

// to tokenize the user's command line input

let tokenize (value:System.String) =
    if (value.Contains "detail") then
        printDetail <- true
    else
        printDetail <- false

    let mutable args = value.Split([|' '|])
    if value.Contains("\") then
        let mutable progStr = value.Replace("\", "")
        progStr <- progStr.Substring(args.[1].Length+1)
        args.[1] <- progStr
    if (args.Length>1) then
        args.[1]
    else
        args.[0]

// the main program
let Program()=
    printfn "***** Welcome to Program Analysis ***** \n\n"
    printfn "Commands : \n\n1. Parse File : parsefile FilePath\n\n2. Parse string : parsestr \"string\"\n\n3. Program Slicing : PS pointofInterest [detail]\n\n4. Dead Code Elimination: DC [detail]\n\n5. Constant Folding: CF [detail]\n\n6. Help : help\n\n7. Exit : exit\n\n []:: optional argument \n "

    let rec ProcessCommand() =
        printfn "\n*****\n"
        let mutable command = Console.ReadLine()
        match command with
        | _ when command.StartsWith("PARSEFILE",true,null) // to parse file
        -> clearDataHandlers()
            let filepath = tokenize(command)
            let program = parseFromFile filepath
            let programCmd = generateProgramBlocks program
            ProcessCommand()

        | _ when command.StartsWith("PARSESTR",true,null) // to parse program given as input
        -> clearDataHandlers()
            let programString = tokenize(command)
            let program = parseFormString programString
            let programCmd = generateProgramBlocks program
            ProcessCommand()

        | _ when command.StartsWith("PS",true,null) // Program Slice
        -> let poi = Int32.Parse(tokenize(command))
            let sliceProgram = programSlice(blocks)(poi)
            printDetail <- false
            ProcessCommand()
```

```

| _ when command.StartsWith("DC",true,null) // Dead Code
-> let deatail = tokenize(command)
    let deadCodeEliminate = deadCodeElimination(blocks)
    printfn "Final Dead Codes %A" (printFull deadCodeEliminate)
    printProgramDetails <- ""
    printDetail <- false
    ProcessCommand()

| _ when command.StartsWith("CF",true,null) // Constant Folding
-> let detail = tokenize(command)
    let constantFold = constantFolding(blocks)
    if printDetail then
        printfn "\n\n Constant Propagation Deatails :::::>> \n\n %A" printProgramDetails
        printProgramDetails <- ""
    printfn "\n\n Folded Program \n"
    for i in constantFold.Keys do
        printfn "\n%A" (blockToString (constantFold) (i))
    printDetail <- false
    ProcessCommand()

| _ when command.StartsWith("HELP",true,null) // the help command
-> printfn "Commands : \n\n1. Parse File : parse FilePath
        \n2. Parse string : parse string
        \n3. Program Slicing : PS pointofInterest [detail]
        \n4. Dead Code Elimination: DC [detail]
        \n5. Constant Folding: CF [detail]
        \n6. Help : help
        \n7. Exit : exit
        \n []:: optional argument \n "
    ProcessCommand()

| _ when command.StartsWith("EXIT",true,null) // exits the program
-> printfn "Program terminates..."
    ()

| _
-> printfn "Sorry Invalid Command" // in case of invalid command
    ProcessCommand()
ProcessCommand()
Program()

```