IDFFL: An Intraprocedural Data Flow Framework for LLVM

A project report submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Sunil Pratap Singh

18111076

under the guidance of
Prof. Amey Karkare

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June 25, 2020

Declaration

I hereby declare that

1. The research work presented in the project report titled **IDFFL: An Intraprocedural Data Flow Framework for LLVM** has been conducted by me under the guidance by my supervisor **Prof Amey Karkare**.

2. The project report has been formatted as per Institute guidelines.

3. The content of the project report (text, illustration, data, plots, pictures etc.) is original and is the outcome of my research work. Any relevant material taken from the open literature has been referred and cited, as per established ethical norms and practices.

4. All collaborations and critiques that have contributed to giving the project report its final shape is duly acknowledged and credited.

5. Care has been taken to give due credit to the state-of-the-art in the project report research area.

6. I fully understand that in case the project report is found to be unoriginal or plagiarized, the Institute reserves the right to withdraw the project report from its archive and also revoke the associated degree conferred. Additionally, the Institute also reserves the right to apprise all concerned sections of society of the matter, for their information and necessary action (if any).

[illegible]

Name : Sunil Pratap Singh

Programme: Masters of Technology

Roll No: 18111076

Department of Computer Science & Engineering

Indian Institute of Technology Kanpur Kanpur 208016

Declaration

This is to certify that the project report titled **IDFFL: An Intraprocedural Data Flow Framework for LLVM** has been authored by me. It presents the research conducted by me under the supervision of **Prof Amey Karkare**. To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

[illegible]

Name : Sunil Pratap Singh

Programme: Masters of Technology

Department of Computer Science & Engineering

Indian Institute of Technology Kanpur Kanpur 208016

CERTIFICATE

It is certified that the work contained in the project report titled **IDFFL: An Intraprocedural Data Flow Framework for LLVM**, by **Sunil Pratap Singh**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare
26/06/2020

Prof Amey Karkare
Department of Computer Science & Engineering
IIT Kanpur

June 25, 2020

ABSTRACT

Name of student: **Sunil Pratap Singh** Roll no: **18111076**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Project title: **IDFFL: An Intraprocedural Data Flow Framework for LLVM**

Name of Project Supervisor: **Prof Amey Karkare**

Month and year of project report submission: **June 25, 2020**

LLVM, a compiler infrastructure, provides a lot of analysis and transformation passes for various optimizations but lacks in providing a framework for analyzing data flow. Data flow analysis plays a significant role in compiler optimization and program debugging. The framework is generalized enough that with the same components, various analyses can be described with only small changes in the description in these components. This framework is explained with some mathematical phenomenon, and it is proved to be safe.

This project describes an implementation of a generalized framework for intraprocedural data flow analysis for LLVM. We try to simplify the interface so that the user can create an analysis with minimal effort.

To my father

Acknowledgements

I would like to thank my supervisor, who helped me during my project work. Many thanks go to my family and my friends who supported me for my studies and given me their valuable advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The information derived by the program analysis of a program has many applications such as program transformation, understanding behavioral aspects of programs, validating programs against different desired properties, etc. When the analysis is performed on the source code(not necessary on HLL) is called static analysis. When the analysis is performed while the program is executing, it is called dynamic analysis.

For dynamic analysis, static analysis can find some approximate results, and the dynamic analysis can make those results more precise. The coverage area of dynamic analysis is less. A dynamic analysis needs to be executed a few times with different input values on a code to cover most of the runtime paths. Still, there is no surety that all runtime paths will be covered. Using static analysis, the entire source code can be analyzed, and the results include each potential runtime path. Although the results are imprecise and some set of results may correspond to some path that is not going to be followed at runtime, they are safe. When the concern is runtime overhead, dynamic analysis can be avoided entirely, and the approximate result of some corresponding static analysis should be used.

One of the many ways of doing program analysis is Data Flow Analysis where the flow of some properties or data is analysed by extracting those properties from the source code and finding how the information related to properties is flowing. The information of the flowing properties need to be safe and correct. It starts with the most imprecise values, and as the analysis runs, the flowing information appears to be more precise than before. Eventually, it is the safest approximation of a program's runtime behavior with the described properties. Data flow analysis have some applications such as semantic validity of a program, program transformation, debugging, etc.

Intraprocedural data flow analysis, which applies on a function(procedure), has a well-defined theory behind it, and using that theory, many analyses can be implemented. The theory is so generalized that a framework can be build using the well-denied principles and procedures. The framework can provide simple ways to implement any intraprocedural data flow analysis. A tool with such a framework can be handy for the purpose of analysis.

LLVM being a very useful resource for many kind of compiler related problems such as optimization, analysis, transformation, etc, does not provide any framework for Intraprocedural data flow analysis.

This project has the objective of implementing a framework that will be generalized enough for creating various intraprocedural data flow analyses and provide the user easy interfaces to implementation. For the implementation, some of the features LLVM compiler infrastructure are used.

The outline of this project is as follows: chapter 2 Background, we give a description of the theory behind the data flow analysis. The description of why this is a general framework is also given.

Chapter 3 Implementation of the framework, we describe the details of the implementation that includes architecture of the framework, the data structures used and how user can use this framework to implement an analysis.

Chapter 4 Conclusion, we summaries the project work and give some possible future work that can be done.

Chapter 2

Background

The term *Data Flow Analysis* happens to be a helpful technique for analysing programs for or against different properties. Most of the Intraprocedural Data flow Analyses are developed with the same specifications and features. The result of these intraprocedural analyses having the same features is a *generalized framework*. The benefit of having a generalized framework is that a library can be implemented with the specifications of the framework. This library gives a simple interface for implementing different data flow analyses. This generalized technique was introduded by Gary A. Kildall[1].

Kildall defines the generalization of the framework in terms of the meet semi-lattice. This induced an order among the resulting values and helps in the convergence of the algorithm. His algorithm works on a directed graph structure of the program, and generalized flow functions are defined for each node of the graph. Several analyses that help in various program optimizations can be derived from this.

In this chapter, we describe the generalized intraprocedural data flow framework and later discussion is about the early and recent works of interproceural data flow. Section 2.1 discuss about intraprocedural DFA framework and subsection 2.1.1 gives Liveness analysis as an example of data flow analysis. Section 2.2 discuss about the interprocedural DFA.

2.1 Intraprocedural Data Flow Analysis Framework

We are going to proceed with the discussion as follows: we present a well known analysis based on the framework and then we will give the description of the framework.

Mostly for doing such static analyses the program is represented in a form called *intermediate representation.* Representing the program in a graph that includes and describes the various paths a program can follow surely helps for the data flow analysis. The control flow graph is the representation that has the information which is needed to understand the path of the program control.

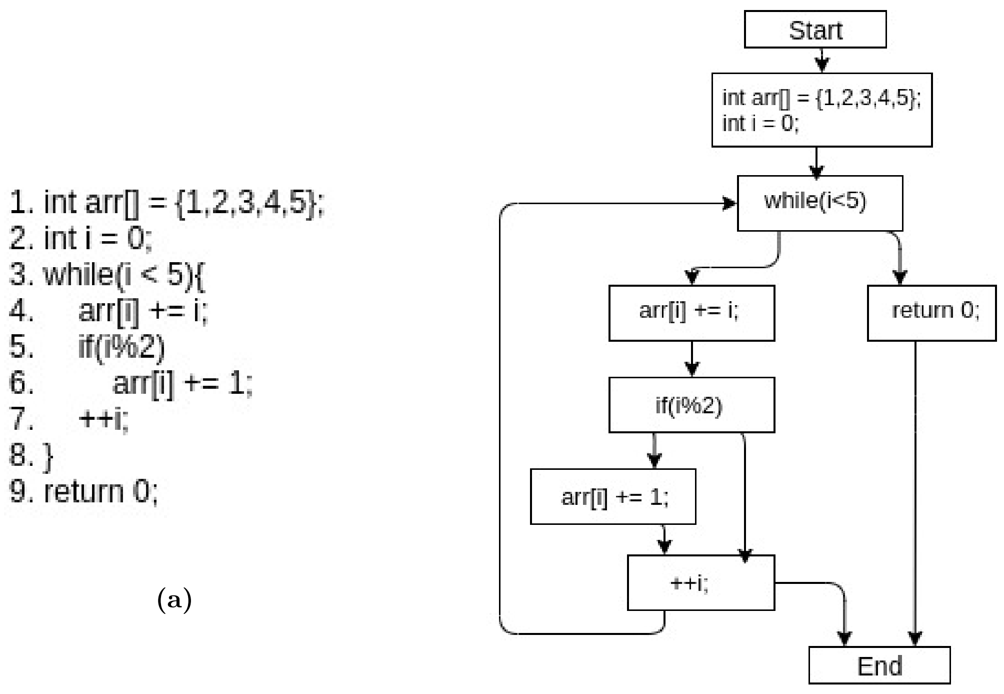
Definition 2.1 *A Control flow graph(CFG) of a program is a directed graph presented as $\{N, E, n_s, n_e\}$. Here N and E are two finite and non-empty sets of nodes and edges respectively. Each program statement is a node and an edge is described as a pair n_i, n_j, where n_i and n_j are two nodes and it shows that the control flows from n_i to n_j. An entry and an exit node, n_s and n_e respectively are added to the graph. For every $n \in N$, Control reaches n from n_s and n_e from n.*

Because CFG is a directed graph, there also exists the notion of predecessor and successor. Consider two nodes A, B in a graph, we call A a predecessor of B and B a successor of A if an edge goes from A to B.

Definition 2.2 *A basic block is a sequence of instructions $\{i_1, i_2, ..., i_j\}$ such that the control only enters the block at the first instruction i_1 and exits the block at the last instruction i_j and there is no instruction $k \in \{i_1, i_2, ..., i_j\}$ such that it is a jump, branch or halt instruction except i_j.*

From a sequence of instructions(a program) we can get a list of basic block by finding the leaders. For the purpose of efficiency we use a basic block as a node of the program. An example for the CFG of a program is given in the figure 2.1.

The information about the flow of the data in the flow analyses is stored in a data structure that consist of either program properties or a mapping from program properties to some other property. The program properties or statements which are generally used in an analysis are statements involving assignment of variable, use of variable, expressions, etc.



```
int arr[] = {1,2,3,4,5};
int i = 0;
while(i < 5){
    arr[i] += i;
    if(i%2)
        arr[i] += 1;
    ++i;
}
return 0;
```

(a)

(b)

Figure 2.1: Program with the corresponding CFG

In the next section we give an example of a data flow analysis called Liveness analysis. Later we use this analysis to define the framework.

2.1.1 Liveness Analysis

This example discuss about an example of data flow analysis called *Liveness Analysis*[2]. For this analysis consider a set V of all the variables appearing in the program. Figure 2.2 will work as an example program.

Definition 2.3 *At a program point* p*, a variable* v ∈ V *is considered to be live when there is at least a path with the use of* v *from* p *to the* End *node, and the use is not preceded by any definition of* v *on that path.*

There is only one start node and only one end node in the CFG in the figure 2.2 so there is no need to add any dummy node. From the definition of the live variable, we can say that at EXIT(bb15), no variable is live. An intuitive observation from the above definition one can derive is that liveness information is obtained from the use of variables and to derive this information, one has to move from use to definition. This provides a direction to the analysis. Because a variable v is considered live at program point p if at least one path has use of v, we combine the liveness information reaching a node from its successor nodes. Statements in each node either change the information or passes it to the predecessor node without any modification. For this computation, a function that uses properties derived from nodes computes the result to move to the predecessor. Following, we define some sets and then give an equation:

Out_n contains the outcoming information from node n.

In_n contains the incoming information reaching node n.

$Kill_n$ contains the information killed in the node n. A definition of a variable kills the liveness of the same. Variable at the left side of an assignment belongs to $Kill_n$.

Gen_n contains the information generated in the node n. Because a use of a variable says a variable will be live, any variable used in n belongs to Gen_n if it is not killed before use.

BI is the boundary information, represents liveness at the EXIT(bb15).

The following is the equation that defines the liveness data flow analysis using the above mentioned variable sets:

$$In_n = (Out_n - Kill_n) \cup Gen_n$$

$$Out_n = \begin{cases} BI & \text{for end block} \\ \bigcup_{s \in succ(n)} In_n & \text{otherwise} \end{cases} \tag{2.1}$$

BI and initial values of each node will be initialized to $\emptyset$ meaning that nothing is live at the start. We apply these equations iteratively on each node till the convergence.

Table 2.1 shows the result of Livenes Analysis on the the CFG in figure 2.2. When two consecutive iterations have the same results, the process converges. The results of the iteration 3 is not begin shown in the table because they are same as that of iteration 2. In_n contains the final result for node n saying what are live variables at the beginning of each node.

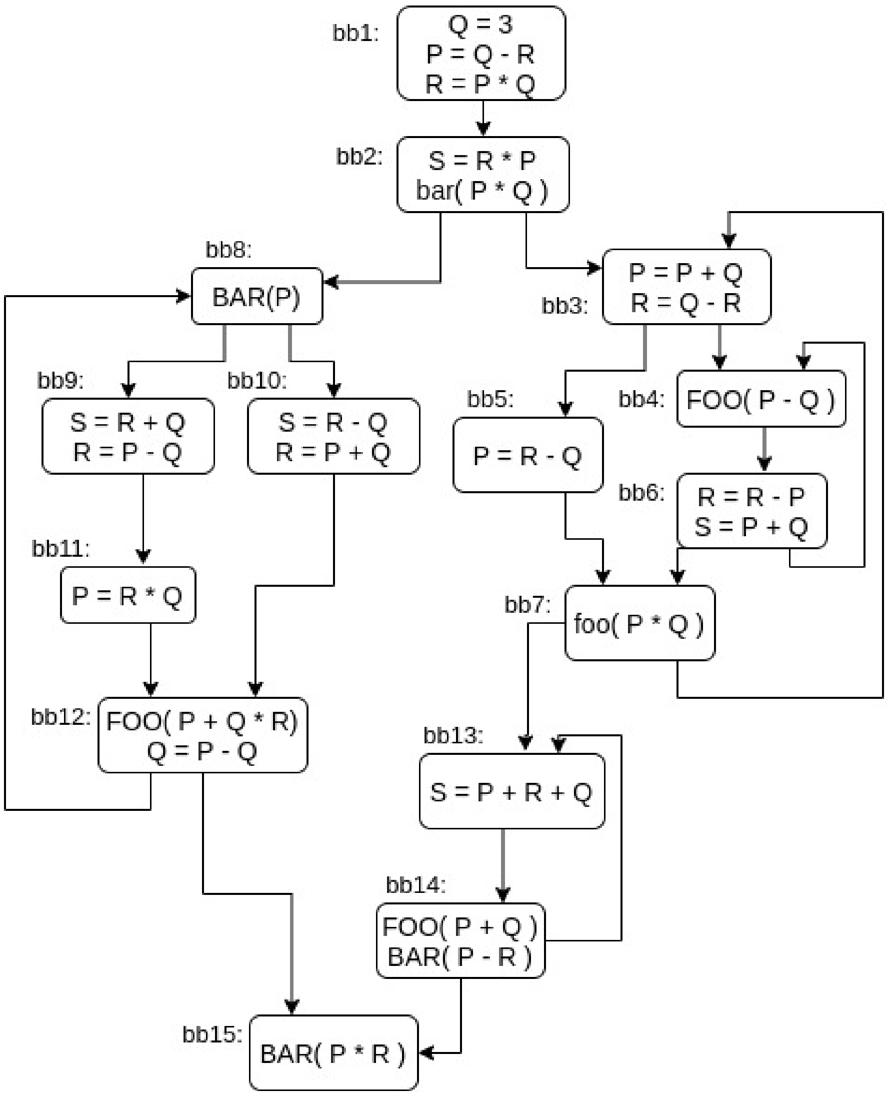


Figure 2.2: An example program CFG being used in Liveness Analysis

Register allocation and the elimination of the dead code are two useful compiler optimizations that use the results of liveness analysis.A live variable can be store in a register for faster access. One can observe that variable S is defined but not used any where is also not live for any basic block. All the definitions of S becomes useless then and can be eliminated.

Basic Block	Gen and Kill		Iter #1		Iter #2	
	gen_b	kill_b	out_b	in_b	out_b	in_b
bb15	P,R	∅	∅	P,R	∅	P,R
bb14	P,R,Q	∅	P,R	P,Q,R	P,Q,R	P,Q,R
bb13	P,Q,R	∅	P,Q,R	P,Q,R	P,Q,R	P,Q,R
bb12	P,Q	Q	P,R	P,Q,R	P,Q,R	P,Q,R
bb11	R,Q	P	P,Q,R	Q,R	P,Q,R	Q,R
bb10	P,Q	R,S	P,Q,R	P,Q	P,Q,R	P,Q
bb9	P,Q	R,S	Q,R	P,Q	Q,R	P,Q
bb8	P	∅	P,Q	P,Q	P,Q	P,Q
bb7	P,Q	∅	P,Q,R	P,Q,R	P,Q,R	P,Q,R
bb6	P,Q,R	R,S	P,Q,R	P,Q,R	P,Q,R	P,Q,R
bb5	Q,R	P	P,Q,R	Q,R	P,Q,R	Q,R
bb4	P,Q	∅	P,Q,R	P,Q,R	P,Q,R	P,Q,R
bb3	P,Q,R	P,R	P,Q,R	P,Q,R	P,Q,R	P,Q,R
bb2	P,Q,R	S	P,Q,R	P,Q,R	P,Q,R	P,Q,R
bb1	R	P,Q,R	P,Q,R	R	P,Q,R	R

Table 2.1: Iterative Result of Liveness Analysis

2.1.2 Theoretical abstraction of the framework

In the liveness analysis we said that the analysis converges after running for few iterations. This gives the notion of some order and approximation among the intermediate and final results. Because partial ordered set represents approximation among the elements, it works as the basis theory for the data flow analysis:

Definition 2.4 *A partial order set* S,$\leq$ *is a set* S *with a binary relation* $\leq$ *defined on* SxS*. For* a,b $\in$ S *and* a $\leq$ b, a *is said to be weaker than* b*. The binary relation* *is:*

Reflexive : *every p is related to itself by the relation, where p is in S*

Antisymmetric : *if (p,q) and (q,p) both are related by the relation then p and q are same, where p,q are in S,*

Transitive : *if (p,q) and (q,r) both are related by the relation then (p,r) is also related, where p,q,r are in S*

We call the binary relation $\leq$, partial order.

Below in Figure 2.3 an example showing the structure of a partial ordered set is given. Say V = {a,b,c}, partial ordered set is the power set of V and the partial order is $\subseteq$. Some other example of poset are real numbers with less than equal to relation, natural numbers with divisibility relation.

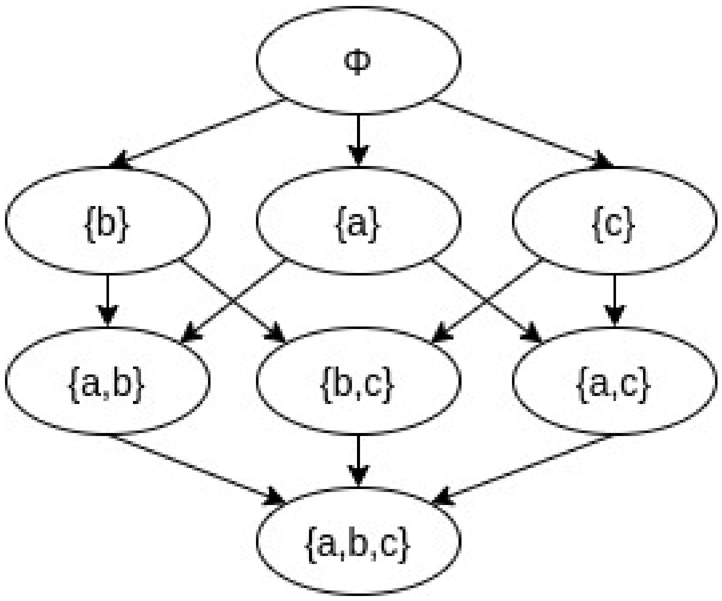


Figure 2.3: Hasse diagram of Partial ordered set with subset and equal to as the binary relation

When every other element *a* is weaker than a single element *g*, *g* is called the greatest element and when an element *l* is weaker than every other element *a*, *l* is called least element. A poset can have at most one greatest and least element.

There is an analogy between the notion weaker than of posets and the notion safe and approximate of the data flow framework. Consider two data flow values *p* and *q*. Suppose p $\leq$ q, then *p* is approximate value of *q* and *p* can be used at the place of q safely. If we consider V as the set of variables in a liveness analysis, {a,b,c} is considered weaker than {a,b} and former can be use in place of later safely in an optimization. If we choose later in place of former, the optimization may not be safe to use and also the behavior of the program may differ from the original. Some more notions of poset are:

When there is no element a such that $g \leq a$ then g is called a maximal element and when that is no element a such that $a \leq l$ then l is called a minimal element. There can be one or more maximal and minimal elements. When only one they are called greatest and least element.

An upper bound u is defined as $a \leq u$ where a represents elements of a subset A of poset P. It is not necessary for u to be in A. When there are more than one upper bounds the least of those is called lowest upper bound. In the same manner a lower bound l is defined as $l \leq a$ where a represents elements of a subset A of poset P. It is not necessary for l to be in A. When there are more than one upper bounds the least of those is called greatest lower bound bound. Greatest lower bound is called meet of A and least upper bound is called join of A.

When meet is applied on two elements of the set the result is weaker than both of those elements. This operation is analogous to the merge of data flow values of two nodes on a single node. Consider the liveness analysis, when a basic block has more than one successors, the values of the IN set of all those successors need to be merged and copied to OUT set of the basic block. Because meet gives weaker result which are interpreted as approximate and safe, meet operator is used instead of join.

When for every pair of elements of a subset of a poset the binary relation is defined i.e P is a poset and S $\subseteq$ P then $\forall$a,b $\in$ S : a $\leq$ b or b $\leq$ a, the subset is called a total order set or chain.

Definition 2.5 *A poset $(L, \leq)$, when for each subset of L a lowest upper bound is defined, the poset is called join-semilattice, and when for each subset of L a greatest lower bound is defined, the poset is called meet-semilattice. When for each subset* lub *and* glb *both are defined, poset is called a lattice. The subset should be non-empty and finite.*

Definition 2.6 *A poset $(L, \leq)$ is called a complete lattice when even for an infinite lattice lub and glb are defined.*

The greatest element in the complete lattice is represented as $\top$ and the least element in the complete lattice is represented as $\perp$. At least one element is needed to be $\top$ and $\perp$ both so $\emptyset$ can not be a complete lattice.

A lattice or a meet-semilattice is the concerned structure our data flow framework. Term which is common in both is that a meet is defined in both. The result of meet of a subset of a poset is the weakest of elements. As we have discussed that weaker in poset is analogous to safe and approximate in our framework, meet-semilattice turns out to be the concerned structure of our framework. Algorithm that we use for the analysis can work fine on both structures having a top element or not. Also a spurious element can be added to a meet-semilattice.

Definition 2.7 *A bounded lattice is a lattice* L *with two elements $\top$ and $\perp$ such that for each element* a $\in$ L *(*a *meet $\perp$) = $\perp$ and (*a *join $\top$) = $\top$.*

Consider 2.1, we can redefine the equation as follows:

$$In_n = f_p(Out_n)$$

$$Out_n = \begin{cases} BI & \text{for end block} \\ \sqcap_{s \in succ(n)} In_n & \text{otherwise} \end{cases} \tag{2.2}$$

Now in equation 2.2, f_p is said flow function and $\sqcap$ is said meet operator. Using this meet operator we merge values of two nodes on one node.

We use lattice because lattice represents the order that is required for our framework. The flow function should also preserve the order. We define two properties for the flow functions that introduces order in the functions. These two properties are monotonicity and distributivity.

Definition 2.8 *A function f_p is called monotonic if and only if* $\forall\ a,\ b \in \mathrm{P} : a \leq b \implies f_p(a) \leq f_p(b)$

A monotonic function provides the order to the framework when the values passes through a node and lattice's meet operator provides the order when the values of some nodes are merged.

Definition 2.9 *We call a function f_p distributive if and only if* $\forall\ p,\ q \in \mathrm{P} : f_p(p * q) = f_p(p) * f_p(q)$ *where $*$ is a binary relation. Every distributive function is monotonic.*

Now we define the framework as follows:

Definition 2.10 *We define the framework with the help of the CFG of the program function. We give a tuple* (S,F,D)*. The elements of the tuple are defined as S a semilattice with a meet beniary relation, F being a set for the flow functions for the nodes of the cfg, and D the direction of the flow.*

The set of flow functions F uses local information of the nodes. This local information is generally the generated information by the node and the destroyed information by the node. These flow functions are monotonic in nature. In the liveness analysis, the local information used by the flow function is Gen, which is a set of generated information and kill, which is a set destroyed information.

An analysis can unidirectional or bidirectional. In an unidirectional analysis, all the flow functions have the same direction. In a bidirectional analysis, the set of flow functions can have some functions of forward direction and some functions can have backward direction.

An analysis derived from the framework consists a control flow graph and a mapping from nodes to flow functions. The CFG is used to define the specifications such as the properties the analysis would work on and lattice elements. For some analysis CFG only provides the properties and the lattice elements are defined explicitly. For example even-odd-sign analysis.

Input: a description of the framework tuple *(S,F,D)*
Output: analysis result corresponding to each node
Algorithm:

```
1.    In{Exit} = BI
2.    Out{Exit} = BI

3.    for (every other block B)
4.        In[B] = $\top$

5.    change = 1
6.    while (change)
7.        change = 0
8.        for (every basic block b)
9.            Out$_b$ = $\sqcap_{s \in succ(n)}$ In$_n$
10.             tmp$_b$ = f$_p$(Out$_n$)
11.                 if tmp$_b$ and In$_b$ not same
12.                     In$_b$ = tmp$_b$
13.                     change = 1
```

Figure 2.4: Iterative algorithm used by the data flow analysis

The safest solution that an analysis could have, is the set of solutions on the all paths with the potential of execution. It is nearly impossible to know the possible execution paths as data flow analysis is a type of a static analysis. An other solution that can be computed statically is Meet Over Path solution. In Meet Over Path(MOP) solution, the information on each node is the result of the merge of the information of each path coming to that node. Thus MOP solution can be given as, $\forall$n $\in$ Nodes : $MOP_n = \sqcap_{p \in paths(n)} f_p$(BI). It is proved that the safest solution is the subset of the MOP solution. It has also been shown that MOP is undecidable[3].

Now we give another solution that is the result of the iterative algorithm used in this framework. The algorithm is as follows:

Algorithm in 2.4 is an iterative algorithm which find the results of data flow analysis by iterating over the nodes of the control flow graph until the result of the current iteration is same as the previous iteration. This solution is called fixed point solution. A fixed point p in a function domain is defined as f(p) = p. It has been shown that a MOP solution is subset of maximal fixed point solution.

In the algorithm in 2.4 we iterate over each basic block. The order of visiting the nodes of CFG affects the speed the convergence. Earlier research [4] has proved that iterating over the nodes in reverse post order for forward problems speeds up the convergence and for backward problems iterating in post order speeds up the convergence. Both reverse post order and post order of the nodes of a graph can be calculated by doing a post order traversal of the depth first search tree.

Chapter 3

Implementation of the framework

First at the beginning of the chapter we give an introduction to LLVM is section 3.1 and then in section 3.2 we discuss the implementation.

3.1 LLVM

LLVM[5] that was previously an acronym of Low Level Virtual Machine, is a compiler infrastructure. LLVM started as a masters thesis at University of Illinois[6] [7] [8] and after that it has grown as an umbrella project that now includes many sub-projects being used academic research as well as commercial products.

LLVM is implemented in C++. LLVM works in a three phase design, see figure 3.1.

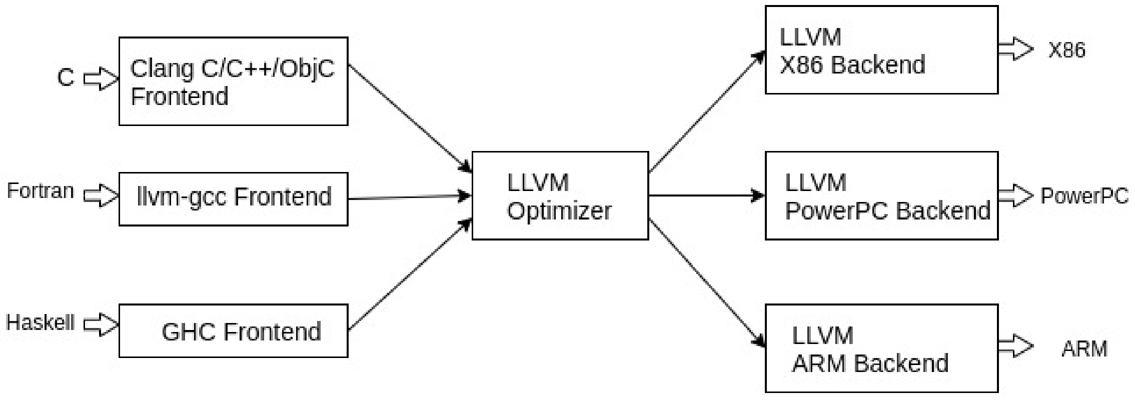


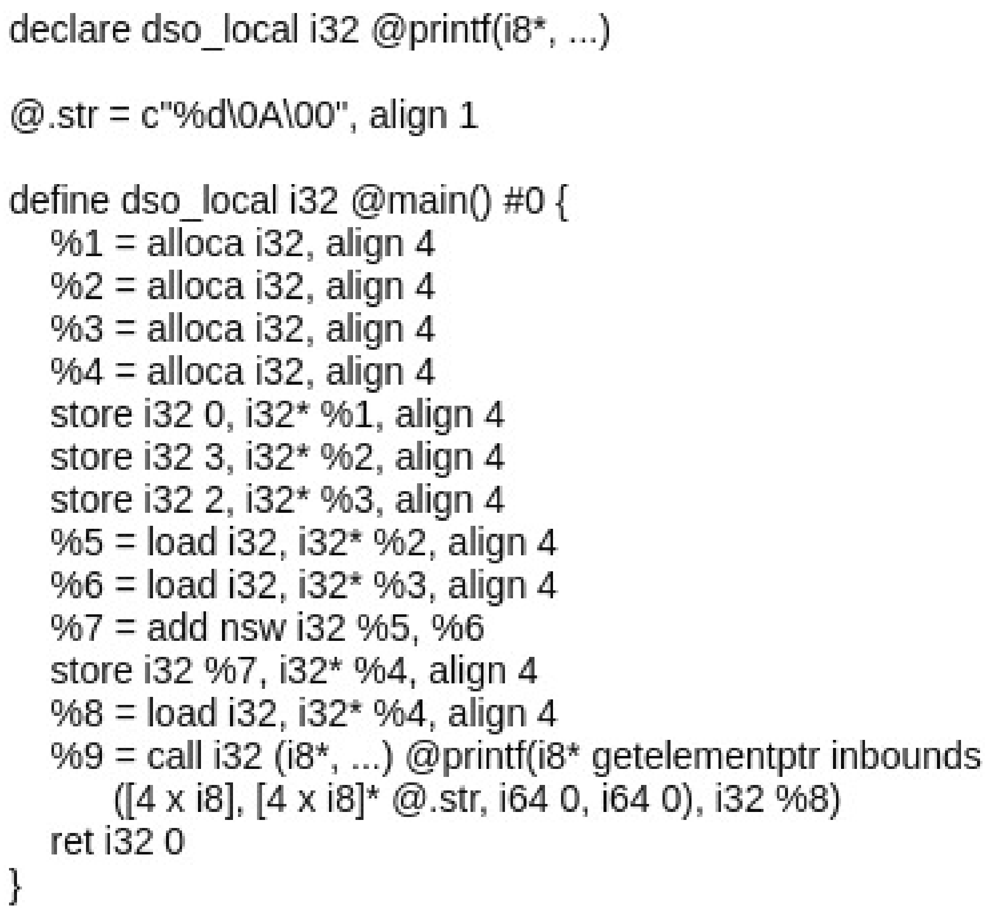
Figure 3.1: General structure of LLVM 3 phase design

Features that make LLVM such useful are a vast range of core libraries and an architecture-independent intermediate representation. Unlike most of the intermediate representations used in other compilers, LLVM's IR is typed and in SSA form. Instructions in IR also contain information about the types of variables and data structure used in the program. LLVM consists of many libraries and tools for analyzing and transformation of the IR. IR has three forms: 1. an in-memory representation, 2. an on-disk bitcode(.bc files), and 3. an human readable text format(.ll files). Each representation is equivalent.

A single file of LLVM IR is called a module that is a transformation of a translation unit of source program or a collection of translation units. A module contains function declarations and definitions, global variables, and a lot of useful information. Instructions in a function are arranged in basic blocks, and these basic blocks create a CFG corresponding to the function. Each basic block starts with a label naming the entry point of the basic block and ends with an instruction either a branch or a return describing the flow of control. Branch instruction has labels of basic blocks where the control can jump. In figure 3.2 a program in LLVM IR is shown. It adds two integers and prints it by printf(...) library function.

A very useful feature of LLVM's three-phase design is that a front-end for a new language can be implemented using the various libraries provided. N source languages can benefit from the LLVM optimizer phase by using LLVM IR, and then LLVM's backend can generate M target machine code for those N languages.

The IR has a well-formed format and the well-formedness is verified by the parser and optimizer libraries. Identifiers in LLVM are of two types: 1. A global identifier is prefixed by a '@'. Global variables and function name comes under global identifier. In figure 3.2 '@*printf*', '@*main*', and '@.*str*' are global identifiers. 2. A local identifier is prefixed by a '%' and virtual registers and functions automatic variables of any type comes under local identifier. Virtual registers are called unnamed temporaries and they are numbered sequentially unless otherwise stated in command line. They are an abstract version of machine registers. In figure 3.2 '%1' to '%9' are unnamed temporaries.



```
declare dso_local i32 @printf(i8*, ...)

@.str = c"%d\0A\00", align 1

define dso_local i32 @main() #0 {
   %1 = alloca i32, align 4
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   %4 = alloca i32, align 4
   store i32 0, i32* %1, align 4
   store i32 3, i32* %2, align 4
   store i32 2, i32* %3, align 4
   %5 = load i32, i32* %2, align 4
   %6 = load i32, i32* %3, align 4
   %7 = add nsw i32 %5, %6
   store i32 %7, i32* %4, align 4
   %8 = load i32, i32* %4, align 4
   %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
        ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %8)
   ret i32 0
}
```

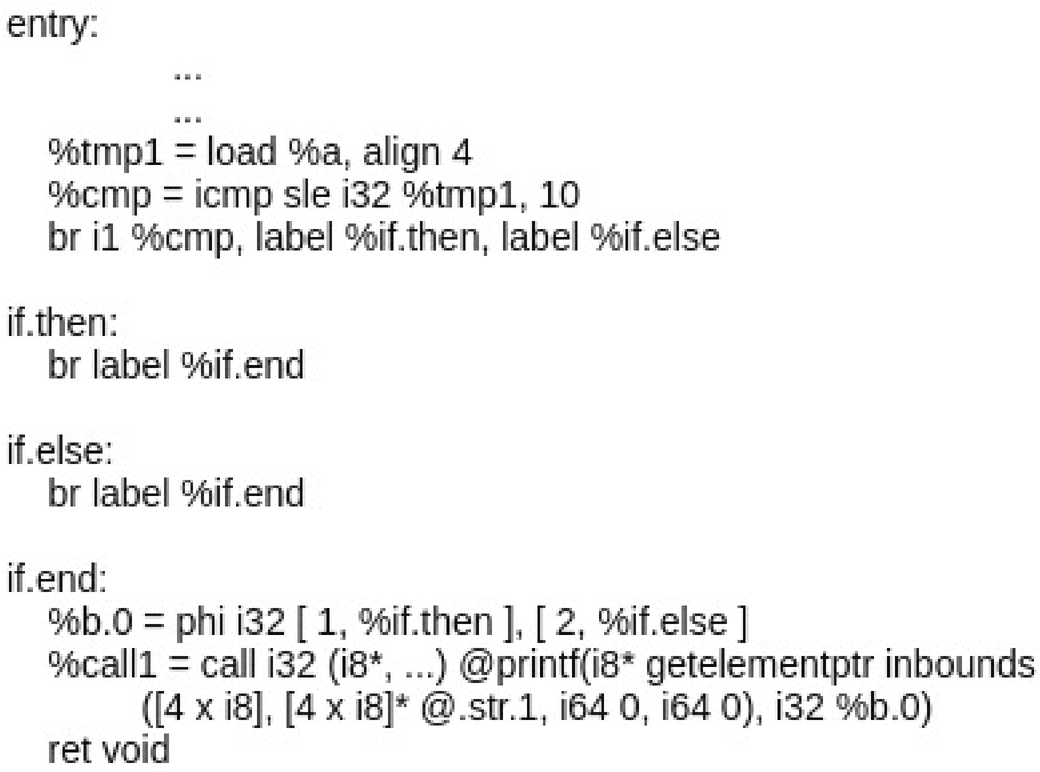
Figure 3.2: An LLVM IR program for adding two numbers and printing it using printf(...)

LLVM IR is in SSA form[9] [10] which provides opportunity for many compiler optimizations. There is also an instruction 'phi' for the ϕ node of SSA form. Each local identifier only appears on left side of the assignment in only one instruction then it appears on right side only. It means each local identifier is assigned only once. Mostly 'phi' instructions are not included in the IR but that introduces redundant load and store. Removing redundant loads or stores sometimes introduces 'phi' instructions. See Figure 3.3 and 3.4 for SSA and 'phi' instruction example.

LLVM has a Type System that is a strong feature of the IR. Being Typed, a number of optimizations can be performed on the IR directly without doing any pre-analysis. This strong type system makes the IR well-formed and increases the readability. We have a few groups of types. First is single value types containing void, integer type(iN) where N is the number of bits ranging from 1 to $2^2$3 - 1, floating-point types(float, double). Second is derived types, that make use of single value types, that contains function type {e.g. float (i16, i32 *) *}, pointer type{e.g i32*, i64**, etc}. Aggregate types include array types{e.g [4 x i8], [5 x [4 x i32]], etc} and structure types{e.g. { float, i32 (i32) * },etc }.

```
int a,b;
scanf("%d",&a);
if(a <= 10){
    b = 1;
}else{
    b = 2;
}
printf("%d\n",b);
```

Figure 3.3: A C program segment



```
entry:
        ...
        ...
  %tmp1 = load %a, align 4
  %cmp = icmp sle i32 %tmp1, 10
  br i1 %cmp, label %if.then, label %if.else

if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %b.0 = phi i32 [ 1, %if.then ], [ 2, %if.else ]
  %call1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
        ([4 x i8], [4 x i8]* @.str.1, i64 0, i64 0), i32 %b.0)
  ret void
```

Figure 3.4: A LLVM IR of figure 3.3 segment to show phi instruction

In the figure 3.2, local identifiers %1,%2,%3, and %4 are pointer types. These pointers points to abstract memory locations allocated to single or aggregated type variables. The abstract memory locations are only accessible using pointer types.

alloca instruction allocates the desired amount of memory and returns a pointer to that memory. To access the contents of a memory location *load* and *store* instructions are used. A global variable prefixed with @ is a pointer to memory always.

To access members of an array or structure, an instruction *getElementPtr* is used. It uses a pointer to the structure and and shifts it by some amount to access the correct element of the data structure. This operation is a ”type indexed” operation and the amount of shift depends on the sizes of the elements of the data structure.

3.2 analyzer library

We implemented a library analyzer. This library implements the generic framework for intraprocedural data flow analysis. This library is implemented in C++ with the use of templates and provides some easy ways to user to define the specification of the analysis. We tried to simplify the interface so that analyses can be defined with minimal efforts. LLVM works as the back-end support as the library is using some of the features of LLVM. In this section we will discuss the implementation details of the library and provide the details on how to use it.

The aim was to implement intraprocedural data flow analysis framework and somehow use the same for the interprocedural data flow analysis. For now the library only implements the intraprocedural data flow analysis framework. Interprocedural part is not implemented yet.

The analyzer library is publicly available at `https://github.com/s-nil/analyzer`. README.md file in the repository provides the installation and usage datails.

3.2.1 Implementation details

We implement the library using C++ templates because we are developing the framework to be general. It gives user freedom of using their own defined properties for the analysis. The entire library is in a namespace ’A’(stands for analyzer).

The implementation is modular and uses multiple classes. Some of these classes are interrelated via inheritance. The architecture is basically divided into two parts, one is storage and other one is analysis.

Figure 3.5 shows the classes in the storage part. *ISet<T>* is an abstract template class that contains only pure virtual functions. This class works as an interface class that groups the related methods and any other class that inherits this *ISet<T>* class has to have all these methods as member functions. Some of those functions are Union, Intersection, Difference, Add, Remove, IsSubSet, etc. This class is inherited by two classes and those are *IBoundedSet<T>* and *AbSet<T>*.

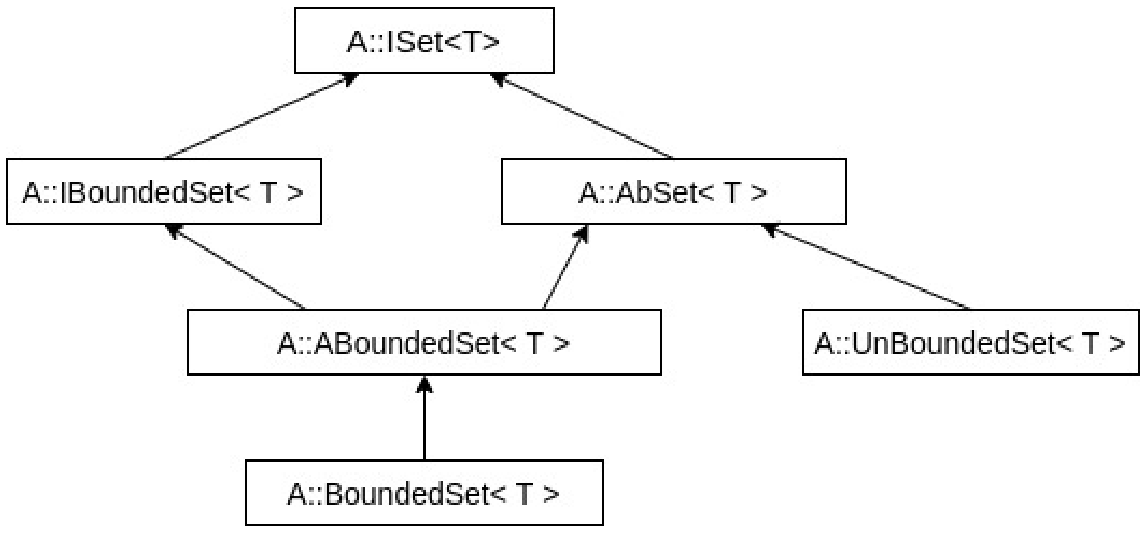


Figure 3.5: Inheritance diagram for storage part of the library

IBoundedSet<T> is also an abstract interface class containing pure virtual functions. Any class inheriting this class provides an implementation of bounded lattice. Functions provided by this class are Complement and TopSet. *ABoundedSet<T>* inherits this class.

AbSet<T> is also an abstract class. Some of the functions in *ISet<T>* can have arguments of other type than the calling object. *AbSet<T>* class gives definitions of these functions in this case. An analysis can be using objects of different types. When an object of some type makes an call to a function with an argument of some other type of objects, the function definition given in class *AbSet<T>* is used.

ABoundedSet<T> inherits from two classes *IBoundedSet<T>* and *AbSet<T>*. This classes purpose is also same as of *AbSet<T>*, but it also includes functions of

IBoundedSet<T> and gives implementation of functions for the same case.

BoundedSet<T> inherits *ABoundedSet<T>* and gives defintions of all the functions. It is an storage for bounded lattice. It has two data members, one is an object of *BiMap<T>* and other is an object of *BitVector*. *BiMap<T>* is a class that implements bidirectional map, uses a vector for index to object mapping and a map for object to index mapping. *BiMap<T>* has member functions such as Add, GetInt, GetObject, Contains, etc. This bidirectional map object with a bitvector work together for the bitvector framework.

UnBoundedSet<T> inherits *AbSet<T>* and it also gives definition of all the functions. This is not a bounded lattice implementation. A user implementing an analysis using this class as storage also has to provide a lattice implementation.

Figure 3.6 shows the classes of the analysis part. *IAnalysis<T>* is an abstract class and have pure virtual functions in it. These functions need to be implemented by the derived class. These functions are NewInitialFlowSet, EntryInitialFlowSet, Merge, Copy, FlowThrough, DoAnalysis, IsForward, and SetFunction. This class has two data members, a pointer to function(an object of LLVM Function class) and a pointer to a *FunctionCFG* class. *FunctionCFG* class stores the CFG of the function. We created this class because LLVM does not provide any class to get the CFG of the function as an object itself.

One useful class for the analysis is *NodeData<T>*. This class stores the intermediate and final results corresponding to a single Node or BasicBlock. This class also stores pointers to *NodeData<T>* objects of the predecessor nodes and successor nodes. *FlowAnalysis* is the class that implements the iterative algorithm of the framework. It use the CFG and NodeData to compute the results. It has two map of BasicBlock and flowtype as key and value.

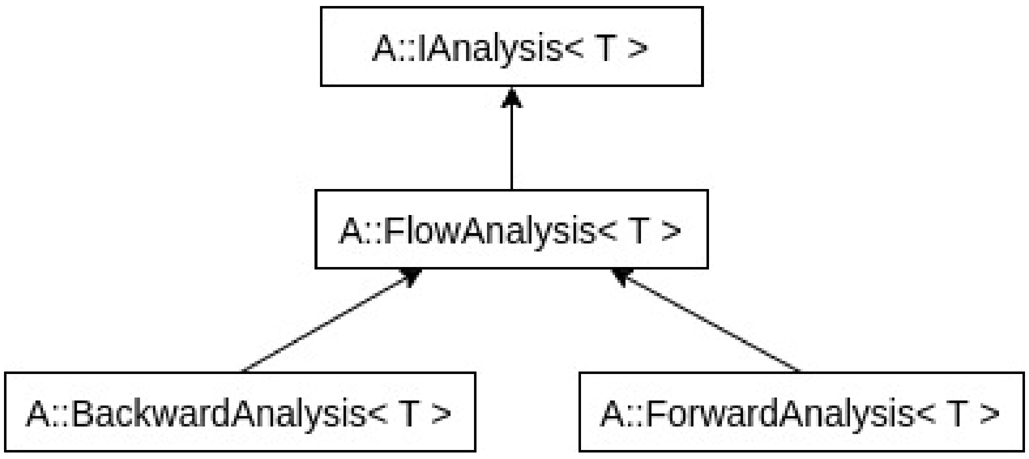


Figure 3.6: Inheritance diagram for analysis part of the library

The functions which are declared in the *IAnalysis<T>* are not defined in this class. User is responsible for giving the definitions of those functions. *FlowAnalysis* has calls to those functions and it uses the user given definition of those functions.

BackwardAnalysis and *ForwardAnalysis* only gives definition of *isForward(). ForwardAnalysis* returns true and *BackwardAnalysis* returns false. *FlowAnalysis* uses isForward for the direction of the flow, adds a dummy node according to the direction and creates NodeData objects for each BasicBlock with pointers to predecessor and successor BasicBlocks NodeData. All these NodeData objects are added in a vector in reverse post order. This vector is used in the iterative algorithm. All the classes shown in figure 3.6 are abstract classes. User according to the desired analysis inherits one of the two *BackwardAnalysis* and *ForwardAnalysis* classes. An other class *ValueUniverse* is used to extract the properties from the intermediate representation. The declaration of this class is given but still user has to give a function definition. We don't give any property class. User has to define the property and because of this reason we ask the user to define the function for extracting the properties also. It turns out to be very easy to implement.

3.2.2 Usage details

The implementation in the above subsection does not give definitions of some function, and the user has to override those functions in an instance analysis of the framework. Because the functions defined by any analysis are overridden, the signature of those function is already known. To make the framework easy to use, these signatures can be replaced with macros. We define some macros that can be used in an analysis instance that help the user to create an analysis with minimal effort.

Below we give the details of the framework:

An analysis can be either forward or backward. Because we have two choices, we give two interfaces for both. For forward analysis, we use FORWARDANALYSIS(...){...}, and for backward analysis, we use BACKWARD(...){...}. Both take three arguments.

- a name for the analysis
- a class name for storing the information. This is one of the two-class BoundedSet or UnBoundedSet
- a user-defined class to give the definition of the property

Apart from defining a class for properties, the user also has to give on more definition. This is for finding the properties from the IR and storing them in a data structure. This definition goes in *EXTRACT(){...}*. In this definition, the user needs to iterate the IR and insert the property class object in a vector when one is found. This is defined as global outside the analysis.

Inside the analysis, three data members

1. a static char for LLVM internal use,
2. a string for storing the function name, and
3. a pointer to storage class object of properties to store the extracted properties(let's say domain).

Add a constructor with a single string argument and initialize function name with this argument and pass the static char to *FunctionPass*(*FunctionPass* is a parent class of the analysis). Now the user needs to do the following things 1. check for the function, 2. assign an object to the pointer data member, and 3. call DoAnalysis()

to start the execution of the analysis. All this goes in *RUN(){...}*.To check and setup the function to be analyzed, call SETUP_FUNCTION(). When an object is assigned to the pointer, it contains all the possible elements of the set. DoAnalysis() initializes the boundary value of the entry or exit node, initial value of every other node, and starts executing the iterative data flow algorithm.

Next, the user provides the definition for the initial value and boundary value. When BoundedSet is being used for storage class, use BS_INITIALVALUE() and BS_ENTRYVALUE() for initial value and boundary value, respectively.

When UnBoundedSet is being used for storage class, use UBS_INITIALVALUE() and UBS_ENTRYVALUE() for initial value and boundary value, respectively. When an analysis's initialized value needs to be empty-set, empty set of the domain can be returned(using EmptySet() function call), and when it needs to be the same as domain, a clone of the domain can be returned(using Clone() function call).

The Other three definitions that the user needs to provide are for flow function, copy, and meet. When BoundedSet is being used for storage class, use BS_MERGE(), BS_COPY() and BS_FLOWTH() for meet, copy and flow function, respectively. When UnBoundedSet is being used for storage class use UBS_MERGE(), UBS_COPY() and UBS_FLOWTH() for meet, copy and flow function, respectively. Every function here takes an argument which is the name of the property class name. In merge function, either of two functions, Union or Intersection, can be used. Arguments for two incoming values are in1 and in2, and for storing result is out. Syntax of the example can be used. Function for copying is not a necessary function to implement. If the analysis does some changes before copying values from objects to others, then only it needs to be implemented; otherwise, it simply copies one object to another. In the flow function, kill and gen of the node are defined, and using various functions provided in the framework, the equations are defined. For gen and kill, first, check if they are already computed for this basic block using function calls GetGen(.) and GetKill(.). Both take node as the parameter. When both returns null, provide the definition for calculating both gen and kill. Add those calculated values for this node using call AddGenKill(.,.,.). It takes three arguments node, gen, and kill. Now, At last, add SETUP(analysis_name) globally.

The result of the analysis and gen and kill for each node are provided in the results. Results can also be shown on a browser by creating an HTML file using a compile-time macro.

Chapter 4

Conclusions

Although this project was going to provide both intraprocedural and interprocedural data flow analyses, but currently, it only contains the implementation of intraprocedural data flow analysis. We have a well-defined theory behind the framework for intraprocedural, so this project turned out to be an implementation based project. In this project, we created a library that implements the various portions of the framework. The user only has very minimum work to do. For this, we provide easy to use interface. We also give examples of some analyses to show how easy it is to create an analysis with our library.

4.1 Scope for further work

This project lacks features such as executing the analysis with a graphical interface. We can look at how to make it even easier for the user to implement an analysis but still providing a general framework so that the user has the freedom to create any type of data flow analysis. Implementing Interprocedural data flow analysis as part of this project would be a real research work.

References

[1] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: Association for Computing Machinery, 1973, pp. 194–206. ISBN: 9781450373494. DOI: `10.1145/512927.512945`. URL: `https://doi.org/10.1145/512927.512945`.

[2] Ken Kennedy. “A global flow analysis algorithm”. In: *International Journal of Computer Mathematics* 3.1-4 (1972), pp. 5–15. DOI: `10.1080/00207167108803048`. eprint: `https://doi.org/10.1080/00207167108803048`. URL: `https://doi.org/10.1080/00207167108803048`.

[3] John B. Kam and Jeffrey D. Ullman. “Monotone Data Flow Analysis Frameworks”. In: *Acta Inf.* 7.3 (Sept. 1977), pp. 305–317. ISSN: 0001-5903. DOI: `10.1007/BF00290339`. URL: `https://doi.org/10.1007/BF00290339`.

[4] John B. Kam and Jeffrey D. Ullman. “Global Data Flow Analysis and Iterative Algorithms”. In: *J. ACM* 23.1 (Jan. 1976), pp. 158–171. ISSN: 0004-5411. DOI: `10.1145/321921.321938`. URL: `https://doi.org/10.1145/321921.321938`.

[5] *The LLVM Compiler Infrastructure Project*. URL: `http://llvm.org/`.

[6] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. *See* `http://llvm.cs.uiuc.edu`. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[7] Vikram Adve et al. “LLVA: A Low-level Virtual Instruction Set Architecture”. In: *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*. San Diego, California, Dec. 2003.

[8] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, Mar. 2004.

[9] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting Equality of Variables in Programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 1–11. ISBN: 0897912527. DOI: `10.1145/73560.73561`. URL: `https://doi.org/10.1145/73560.73561`.

[10] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Global Value Numbers and Redundant Computations”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 12–27. ISBN: 0897912527. DOI: 10.1145/73560.73562. URL: https://doi.org/10.1145/73560.73562.