M. Tech Project Stage-II Report

on

Incremental Data flow Analysis using PRISM

by

Rashmi Rekha Mech
Roll No : 133050089

Under the Guidance of

Prof. Uday Khedker

Abstract

When a program undergoes changes during development, updating the data flow information by doing exhaustive analysis is cost inefficient. In such cases, modifying the data flow information of the changed portion of the program, while avoiding recomputation of the entire data flow can lead to significant savings.

We first describe PRISM, an analyzer generator developed at Tata Research Development and Design Centre (TRDDC). We next present the enhancements done to the PRISM solver to add the capabilities of incremental analysis, elaborating on the challenges faced and the changes made. Such incremental analysis methods are well studied for bit-vector frameworks. However, these methods are not directly applicable to general frameworks. This report also presents some challenges involved and possible solutions for doing incremental analysis for general frameworks.

Contents

List of Tables

List of Figures

Chapter 1

Overview

1.1 Introduction

When a program undergoes changes during development, some or all of data flow information computed earlier becomes invalid. Thus, recomputation of data flow values is required.

Motivating Example

Consider a control flow graph in Fig 1.1 [10] for available expression analysis. Table 1.1(a) shows an initial result which requires three iterations to converge. Suppose, expression $a + b$ in node $n6$ has been removed. To reflect this change, recomputing data flow information from scratch is shown in Table 1.1(b).

The removal of expression $a + b$ does not affect the data flow values and hence no new information is added as shown in Table 1.1(b). For such changes, it is not desirable to recompute the information from scratch. This may unnecessarily analyze unaffected program behaviours which leads to redundant computation of old values which is very inefficient.

For such changes, incremental data flow analysis can be used. It modifies only those data flow information which has been affected by the change rather than recomputing entire data flow information. Clearly, we expect this method to be more cost-effective than exhaustive analysis in general.

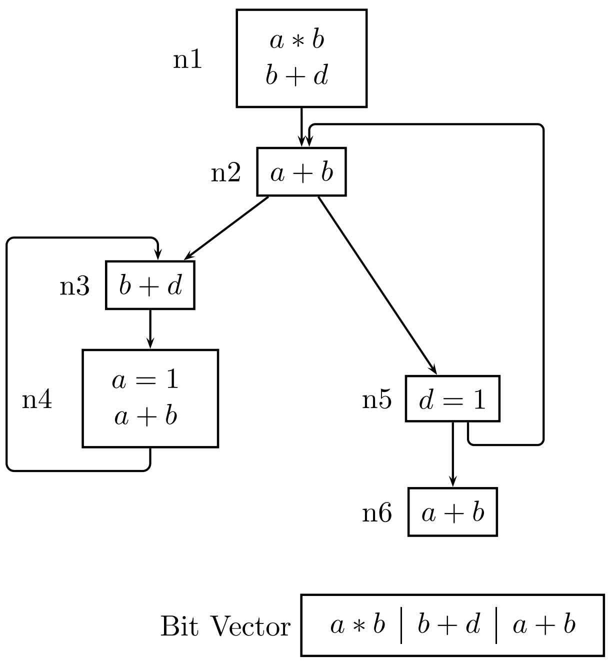


Figure 1.1: Control Flow Graph.

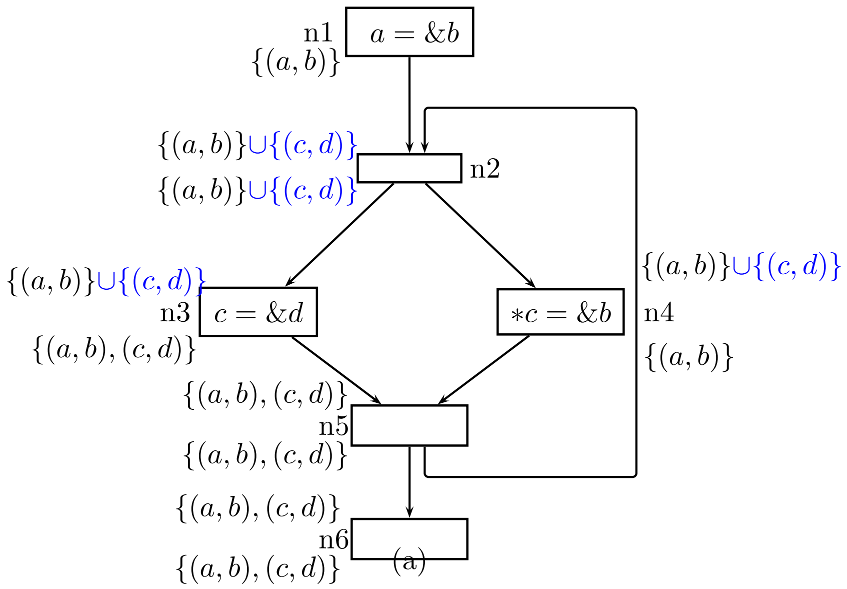
Table 1.1 (a). Initial available expression analysis for Fig. 1.1 (b). Exhaustive analysis to validate the program change.

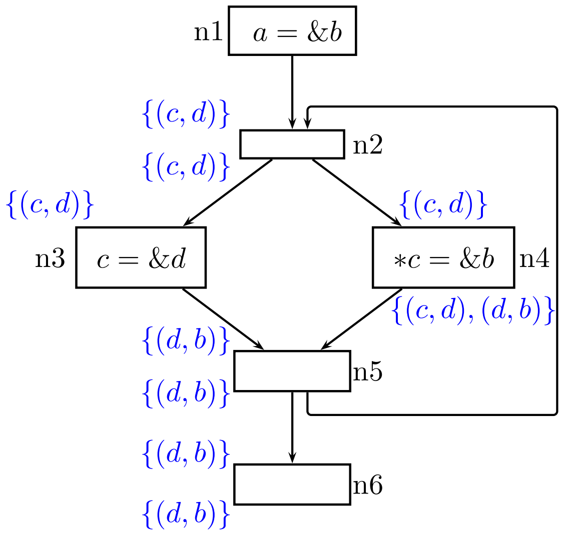
	*Iteration*1		*Iteration*2	
Node	In	Out	In	Out
1.	000	110		
2.	110	111	100	101
3.	111	111	000	010
4.	111	010	010	
5.	111	101	101	
6.	101	101		

(a)

	*Iteration*1		*Iteration*2	
Node	In	Out	In	Out
1.	000	110		
2.	110	111	100	101
3.	111	111	000	010
4.	111	010	010	
5.	111	101	101	
6.	101	101		

(b)





(b)

Figure 1.2: Motivating example for incremental points-to analysis across different iterations. In order to show the increments in the sets, we have used ∪ instead of writing the values after performing set union.

Incremental data flow analysis can be done in the following situations:

- **Across different iterations in a fixed point computation:** Instead of computing all values afresh in each iteration of an analysis, the values can be accumulated across iterations. If each iteration computes only the values not computed before, or modifies the values that needs to change will eliminate redundant computation of old values.

Consider points-to analysis in figure 1.2. After first iteration, at IN of node $n2$ points-to pair (c,d) is added to the pointer set. At node $n4$, we modify only the pointees of c. Hence, pointer information of c is only required to generate new points-to pairs. After first iteration, computation of pair (a, b) at IN of node $n4$ is redundant. The generated new data flow values in an iteration are known as incremental values (figure 1.2(b) shows incremental values). Therefore, propagating the incremental values during the analysis can reduce the redundant computations at a basic block.

- **Across different applications of an analysis:**
  Let us consider the example shown in figure 1.3. After performing available expression analysis, we have a scope of doing constant propagation analysis. After first pass of available expression analysis, expressions $x + 1$ and $a + b$ are available at the OUT of $n2$. At the OUT of $n1$, the value of x is 10. Therefore, after first pass of constant propagation the value of y (at the OUT of $n2$) becomes 11. This can be further optimized by directly propagating the value of variable y in print statement. Now, if we further perform dead code elimination on the modified graph. The dead statements at node $n1$ and $n2$ will be removed as shown in Fig 1.3(c). Hence, information of availability of expressions stored with the modified graph should be updated.

  Therefore, incremental analysis is performed on the modified control flow graph using updated information.

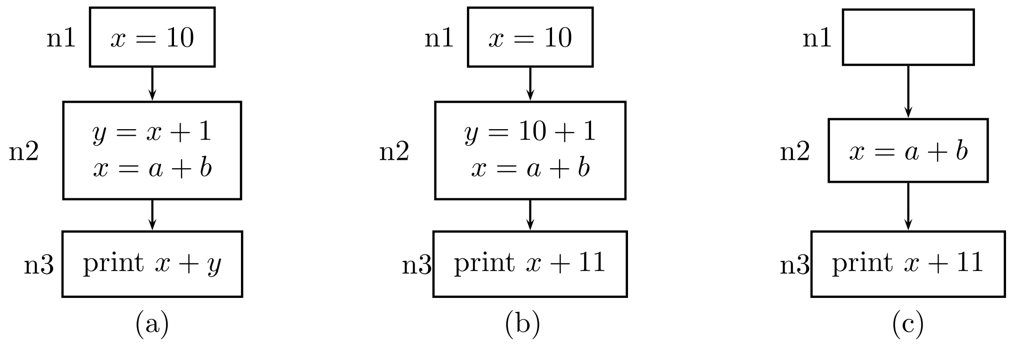


Figure 1.3: Incremental analysis across different applications.

1.2 Scope of the Project

This project aims at providing support for incremental data flow analysis in PRISM. Old version of PRISM [3] performs both unidirectional and bi-directional context sensitive analysis. However, it doesn't perform incremental data flow analysis.

This report explains the implemented incremental solver. It also presents some issues in implementing incremental data flow analysis for general frameworks. It describes the implemented Liveness-based Inter-procedural Reaching Definition Analysis in PRISM.

1.3 Organization of the Report

The report is organized as follows. Chapter 2 describes incremental analysis for bit-vector frameworks. Chapter 3 focuses on an overview of PRISM. Chapter 4 describes our implementation of Liveness-based Reaching Definition analysis in PRISM and performance measurements. Our addition to PRISM and performance measurement is described in Chapter 5. Chapter 6 describes some of issues in doing incremental analysis for general frameworks. Chapter 7 describes some extensions that can be made to the current work.

Chapter 2

Incremental Data Flow Analysis

A Data Flow framework [1] is defined as a triple $D = \langle L, \sqcap, F \rangle$, where L represents information associated with entry/exit of a basic block (L is a partially ordered set), $\sqcap$ represents a binary meet operation (e.g intersection or union) which determines the way the global information is combined when it reaches a basic block, and F represents a flow function. There are two special elements are associated with this framework, *top* denoted by $\top$ and *bot* denoted by $\perp$, which can be defined as follows:

- Top. $\forall x \in L : x \sqcap \top = x$

  (Using $\top$, in place of any data flow value, will never miss out any possible value. Hence, it is an exhaustive approximation of all values.)

- Bottom. $\forall x \in L : x \sqcap \perp = \perp$

  (Using $\perp$, in place of any data flow value, will never be incorrect. Hence it is a safe approximation of all values.)

When a program undergoes changes during development some or all data flow information computed earlier becomes invalid. Updating data flow information to incorporate the effect of changes by repeating an exhaustive analysis can be very cost inefficient since it may compute redundant values. So incremental data flow analysis can be used. It modifies only those data flow information which has been affected by the change. So is more cost-effective than an exhaustive analysis.

When a program gets modified, the changes in the old data flow information may take place either globally or locally. Local changes are those which are associated with the node in which the original change has taken place. Global changes are those which are associated with some other node. Global changes can be found by incorporating the effect of local changes over the rest of the nodes of the graph. Incremental Data flow analysis basically focuses on global changes.

2.1 Incremental Analysis for Bit-vector Frameworks

In this section, first we describe the flow functions associated with bit-vector frameworks. Then we briefly describes the possible changes in bit-vector frameworks and how to handle those changes.

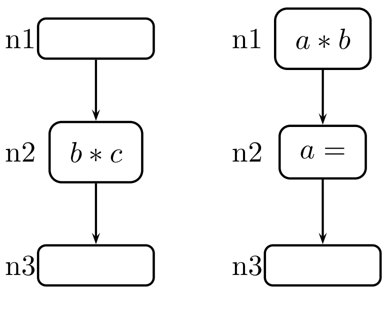
2.1.1 Flow Functions in Bit-vector Frameworks

In bit-vector analysis, following types of functions are possible.

- Raise : Result is always top ($\top$). Consider an available expression analysis in figure 2.1(a). At node n2, *GEN* is 1 and *KILL* is 0. Therefore, $OUT_2 = 1$ or $\top$ ($OUT_n = GEN \cup (IN_n - KILL)$). Flow function at n2 is a Raise function whose result is always $\top$

- Lower : Result is always bottom ($\perp$). Consider an available expression analysis in figure 2.1(b). At node n2, *GEN* is 0 and *KILL* is 1. Therefore, $OUT_2 = 0$ or $\perp$. Flow function at n2 is a Lower function whose result is always $\perp$

- Propagate : It propagates the value at IN to OUT of the node. Consider an available expression analysis in figure 2.1(b). At node n3, *GEN* is 0 and *KILL* is 0. Therefore, $OUT_3 = IN_3$. Flow function at n3 propagates the value at the IN to OUT of the node n3



(a) (b)

Figure 2.1: Flow functions in bit-vector frameworks

2.1.2 Possible Changes in Flow Functions

For a given entity, the following changes could happen as a consequence of some change in a node:

- some data flow values may change from top ($\top$) to bottom ($\perp$). Possible changes in the flow function is shown in fig 2.2(a)

- some data flow values may change from bottom ($\perp$) to top ($\top$). Possible changes in the flow function is shown in fig 2.2(b)
- some data flow values may remain same.

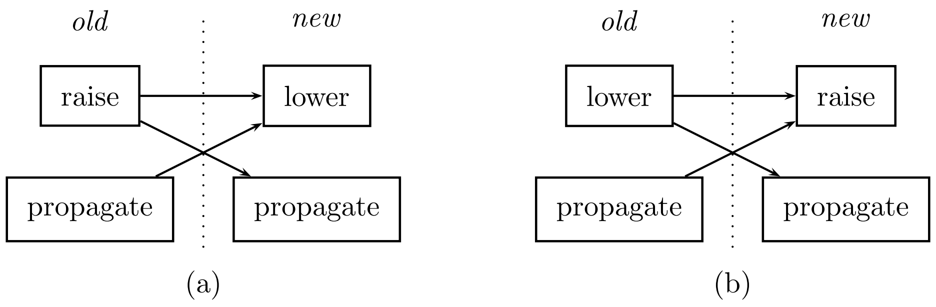


Figure 2.2: Flow functions in Bit-vector framework (a) Possible changes in flow function for top to bottom change. (b) Possible changes in flow functions for bottom to top change.

Handling Top to Bottom Change

Given that $\forall x \in L : x \sqcap \top = x$, a top value for a data flow property is an intermediate value until the analysis is completed. But in case of a bottom value, $\forall x \in L : x \sqcap \perp = \perp$. Thus, a bottom value is a final value even during analysis. So whenever there is a top to bottom change in a program, the changes can be propagated directly to its neighbouring nodes.

Consider an example of available expressions analysis for the control flow graph in figure 2.3. In this case, the expression $b * c$ is available at the OUT of both node n1 and n3 (i.e. the data flow value for $b * c$ is top) and so is available at the IN of $n2$. Let an assignment "$c =$" be inserted after the computation of $b * c$ in node $n3$. After this change, $b * c$ is not available at the OUT of node n3, i.e. top to bottom change (refer to figure 2.2) at the OUT of $n3$ which in turn makes bottom at the IN of $n2$ since $IN_2 = OUT_1 \sqcap OUT_3$, which implies that the value of IN_2 is determined by the value of OUT_3 alone. Thus, the effect of top to bottom change can be incorporated by directly propagating the change to its neighbour.

Handling Bottom to Top Change

Bottom value is a final value even during the analysis. Thus, whenever there is a bottom to top change we cannot directly propagate the changes to its neighbours. We need some more processing to incorporate this change.

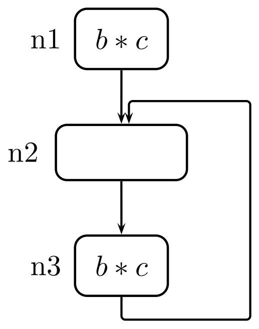


Figure 2.3: Top to Bottom change in control flow graph

Consider control flow graph in figure 2.3, if there is a bottom to top change (figure 2.2) at the OUT of $n3$, the value of IN_2 cannot be directly determined by OUT_3 alone since now it depends on the value of OUT_1 also. Therefore, incorporating the effect of bottom to top change requires some more processing.

The effect of bottom to top change can be incorporated in the following two steps:

- Identify the data flow values which may become top.
- Find out the data flow values, identified in the above step, which must remain bottom due to the effect of some other property.

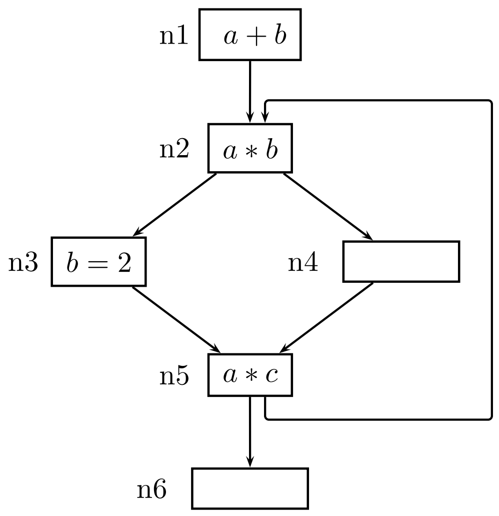


Figure 2.4: Control flow graph for available expression analysis

Motivating Example

Consider a control flow graph as shown in Fig 2.4. Table 2.1 shows the result of available expression analysis for Fig 2.4.

Table 2.1 Available expression analysis for Fig. 2.4

	$a+b$		$a*b$		$a*c$	
Node	In	Out	In	Out	In	Out
1.	0	1	0	0	0	0
2.	0	0	0	1	0	0
3.	0	0	1	0	0	0
4.	0	0	1	1	0	0
5.	0	0	0	0	0	1
6.	0	0	0	0	1	1

Assume that the expression "$b = 2$" in $n3$ has been deleted. Now, we will calculate the updated information in following two steps:

- The data flow values which were 0 and *may* become 1 due to this change (shown in Table 2.2)

  At node $n3$, the expression "$b = 2$" is killing the availability of expressions $a*b$ and $a+b$. So after the removal of $b = 2$, expression $a*b$ and $a+b$ may available at the OUT of $n3$ which in turn affects many other nodes. So, we will construct an **affected region**. The affected region is a set of program points where information may change. The affected region for fig 2.4 includes $\langle$ OUT_3, IN_5,OUT_5, IN_6, OUT_6, IN_2, OUT_2, IN_4, OUT_4, $IN_3\rangle$ program points.

Table 2.2 The data flow values which may become 1.

	$a+b$		$a*b$		$a*c$	
Node	In	Out	In	Out	In	Out
1.						
2.	1	1	1			
3.	1	1		1		
4.	1	1				
5.	1	1	1	1		
6.	1	1	1	1		

- From the data flow values shown in Table 2.2, the data flow values which must remain 0 are shown in Table 2.3

This step is again divided in two following parts:

- **Identifying boundary nodes**: Boundary nodes are those where some information comes from an unaffected part of the program. We need to consider those information to compute the data flow values. In Fig 2.4, $n2$ is a boundary node. To calculate IN_2 we need to consider $OUT_5 \sqcap OUT_1$, where $n1$ is a node in unaffected region.
- **Computing values at boundary nodes and propagating them**.
  At OUT_1, $a * b$ is not available which makes $a * b$ not available at IN_2. But $a + b$ is available at both OUT_1 and OUT_5 and hence is available at IN_2. After computing the information at the IN_2, we will propagate this information throughout the affected region. The resultant values which must remain 0 are shown in Table 2.3

Table 2.3 The data flow values which must remain 0.

	$a+b$		$a*b$		$a*c$	
Node	In	Out	In	Out	In	Out
1.						
2.			0			
3.						
4.						
5.						
6.						

Thus, the final solution is shown in Table 2.4

Table 2.4 Final updated information.

	$a+b$		$a*b$		$a*c$	
Node	In	Out	In	Out	In	Out
1.	0	1	0	0	0	0
2.	1	1	0	1	0	0
3.	1	1	1	1	0	0
4.	1	1	1	1	0	0
5.	1	1	1	1	0	1
6.	1	1	1	1	1	1

We note that although the example above is for a single change, a similar method can be applied for multiple changes in the control flow graph.

2.1.3 Node Listing as Affected Region

A *node listing* [7] for control flow graph *G=(N,E,n_0)*, where N is the set of nodes, E is the set of edges, and n_0 is the program entry nodes, is defined to be a sequence

$$l = (n_1, n_2, ...n_m) \tag{2.1}$$

of nodes from N(nodes may be repeated) such that every single path in G is a subsequence of l.

We can use *node listing* as an affected region without traversing the control flow graph. But *node listing* may include some unaffected nodes which leads to redundant computation.

Consider an example shown in figure 2.5. The node listing for this graph is *l=(1, 2, 3, 4, 5, 4, 5, 6)*. Suppose, expression $b = 1$ is present at node 4. Due to a program change if expression $a * b$ is inserted at node 1 then we can use list '*l*' as an affected region without traversing the control flow graph. Due to the presence of expression $b = 1$ at node 4, the nodes 5 and 6 is not affected by this change. But node listing method will include nodes 5 and 6 also which leads to redundant computation. Our method of creating an affected region will exclude node 5 and 6.

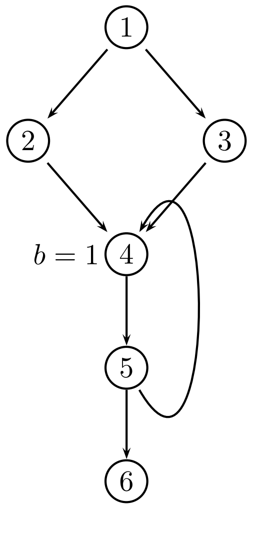


Figure 2.5: Control flow graph for *node listing*

2.1.4 Handling Structural Changes

The following two kinds of changes may take place:

- Changes in the lattice.
  If there is a change in a lattice element, i.e. adding a new expression in the program or removing all computations of an expression from the program. Such a change needs to be processed using an exhaustive analysis.

- Changes in the graph.
  Consider an edge $e = (i, j)$. Let u $\equiv$ OUT$_i$ and v $\equiv$ IN$_j$, two flow functions $h(u, v)$ and $h'(v, u)$ are associated with an edge e.

  - Deletion of an edge:
    Deleting an edge e will delete functions $h(u, v)$ and $h'(v, u)$. We know that a non-existant function is a constant function $\top$. Thus, some functions may change to $\top$ which is a bottom to top change.
  - Insertion of an edge:
    Inserting an edge e will add two new functions $h(u, v)$ and $h'(v, u)$ which is a top to bottom change.
  - Deletion of a node:
    Deletion of a node involves both top to bottom and bottom to top changes as illustrated in figure 2.6. The deletion of the node k starts with the removal of edges $e1 = (i, k)$ and $e2 = (k, j)$, a bottom to top change. This is followed by addition of the edge $e3 = (i, j)$, a top to bottom change.

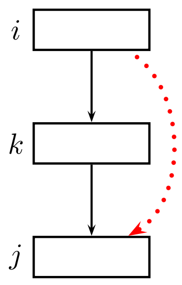


Figure 2.6: Deletion of a Node

  - Insertion of a node:
    The process of insertion of a node is complementary to the deletion process explained above. The insertion process, as illustrated in figure 2.7, starts with deletion of the edge $e1 = (i, j)$, a bottom to top change. This is followed by insertion of edges $e2 = (i, k)$ and $e3 = (k, j)$, a top to bottom change.

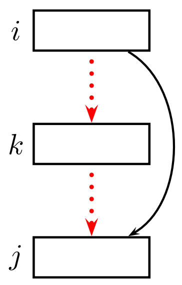


Figure 2.7: Insertion of a Node

2.1.5 Validation

We can validate the results of incremental analysis by comparing it with the result of exhaustive analysis.

2.1.6 Complexity

Let n be the total number of nodes in the graph and m be the total number of nodes in the affected region. The worst case time complexity to identify the boundary nodes is *O(m)* since each node in an affected region needs to be checked. We note that in the extreme case, $m = n$.

2.2 Other Approaches of Incremental Data Flow Analysis

One approach [8] is based on Context Free Language(CFG) reachability for incremental analysis of context-sensitive points-to analysis. It traces the CFL-reachable paths which was traversed during computation of points-to sets to precisely identify and recompute the affected points-to sets when the program changes made.

Another approach [9], generates a structural difference between the old and new code version, called as *diff*. The *diff* provides an information of added and removed nodes/edges from the inter-procedural control-flow graph(ICFG). Added or removed nodes are called as *changed* nodes. The set of *affected* node contains all the nodes which are reachable from the *changed* nodes(which is an over approximation). The set of *affected* nodes are inferred from the set of *changed* node. This is followed by a clear-and-propagate strategy: for each affected node, it clears the computed information and then re-propagates the information from all the nodes predecessors. Basically, it creates an affected region for all kind of changes.

2.3 Summary

When a program undergoes changes, data flow values computed earlier become invalid. Updating the data flow information to incorporate the effect of changes by repeating an exhaustive analysis can be cost inefficient since it may compute redundant values. In such cases, incremental data flow analysis, which modifies only the data flow information of the program that has been affected by the change is more cost effective.

In bit-vector frameworks, due to program change the data flow value may change either from top to bottom or vice versa. Top to bottom change can be incorporated by directly propagating the values to the neighbors of the changed node. For bottom to top change, we need to create an affected region to incorporate the change to the neighbors of the changed node.

Chapter 3

An Overview of PRISM

PRISM is a program analyzer generator developed bt TATA Research Development and Design Center(TRDDC). This chapter contains an overview of the architecture of PRISM analyzer generator.

PRISM has the following two basic components:

- Kulang compiler: Kulang compiler parses analyzer specifications and generates some java classes.
- Solver : These classes are used by solver to solve data flow problems. It also uses many other packages which provide many utility functions. These functions are used my the solver.

3.1 PRISM IR

The program to be analyzed is compiled using a front end IR generator. It generates a language independent intermediate representation of the program. Unlike intermediate language of typical compiler, IR is a high level language and contains complex expressions. Different front end translators are used for different languages which converts the language to the PRISM IR.

Figure 3.1 [3] shows a block diagram of various PRISM components. IR generator generates independent intermediate representation of the program to be analyzed. The generated analyzer takes IR as an input and produces output reports.

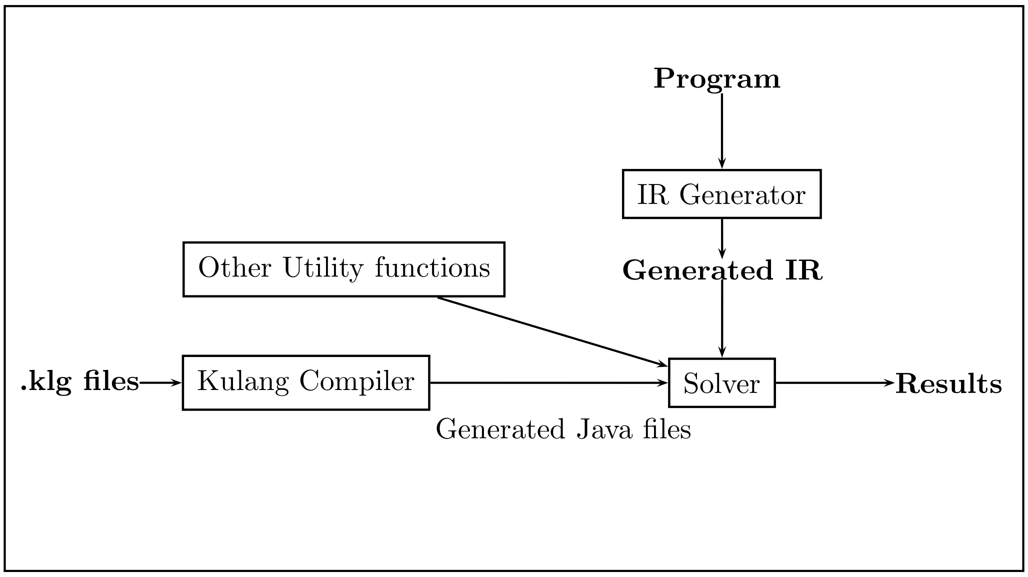


Figure 3.1: Architecture of PRISM

Accessing statements in PRISM IR

Each function in the program is represented by an `IRObject`. Each `IRObject` is associated with an instance of `ICFG` (represents control flow graph of that function). An `ICFG` contains a list of `ICFGnodes` which represents one statement in a program. It also contains a pointer to the entry `ICFGnode` of that function. Each `ICFGnodes` stores a list of its successor and predecessor. Between two `ICFGnodes`, there is an `ICFGEdge` which can be either data flow or control flow edge. An array of `IRObjects`, representing all functions in a program, is provided by PRISM API. If an `ICFGnode` represents a function call, an API over it can be used to get `ICFG` of the function that is called.

Interpreting statements in PRISM IR

Each `ICFGNode` represents a statement in PRISM IR. The `ICFGNode` is associated with an instance of `Expr` which represents the expression of the statement. `Expr` is a base type and can be many of its subtypes. Some possible subtypes are:

- Unary
- Binary
- Call
- Return Statement

An expression is composed of a tree structure of operands and operators. Each expression type has API function to access operators and operands. A variable in an expression is represented using an instance of `NamedEntity`. A `NamedEntity` cantains various properties of the variable, such as name, scope, data type etc. Various API functions are available to determine the type of the `NamedEntity`, such as pointer, static etc. Unlike variables, `NamedEntity` can have expression like structure. Following are the types of `NamedEntities`:

- `STNamedEntity` : ST stands for Symbol Table. It returns variables used in the program.

  e.g. In the expression "a=b+c", a, b and c are of type `STNamedEntity`

- `PSTNamedEntity` : A part of `STNamedEntity` is represented by `PSTNamedEntity`

  e.g. In the expression "`x.y=a[0]`", `a.b` is a `PSTNamedEntity` which is a part of `STNamedEntity` `a`

- `IndirectNamedEntity` : `NamedEntity` to represent dereference.

  e.g. In the expression "`int *p`", `*p` is a `IndirectNamedEntity`.

- `PIndirectNamedEntity`: A part of another `IndirectNamedEntity` is represented by `PIndirectNamedEntity`. In the following example $x \rightarrow y$ is a part of `IndirectNamedEntity` which is represented by `PIndirectNamedEntity`

  e.g. `x → y= a[0]`

- `HeapNamedEntity`: During dynamic allocation `HeapNamedEntity` is created. Each heap location is assigned a unique number according to the line number where `malloc` or `calloc` is used.

  e.g. `p =(float *) malloc(sizeof(float));`

- `PHeapNamedEntity` : A part of `HeapNamedEntity` is represented by `PHeapNamedEntity`. In the following example $p \rightarrow q$ represents a part of `HeapNamedEntity`

  e.g.
  ```
  p = (float *) malloc(sizeof(float));
     p → q = a[0] ;
  ```

- `EnumlitNamedEntity` : `NamedEntity` to represents a value of enum type

- `AddressNamedEntity` : An address data item is represented by `AddressNamedEntity`. In the following example &b represents an `AddressNamedEntity`

  e.g. `a = &b;`

- `SizeofNamedEntity` : It represents use of sizeof operator
- `StructNamedEntity`: String literal is represented by `StructNamedEntity`. In the following example "Hello world" is a `StructNamedEntity`

  e.g. `string c* =''Hello world'';`
- `LabelNamedEntity`: A label in a program is represented by `LabelNamedEntity`. In the following goto statement, a `LabelNamedEntity` is created for the label begin

  e.g. `goto begin;`
- `ArrayindexNamedEntity` : An expression where an array is indexed is represented by `ArrayindexNamedEntity`. In the following example a[0] is a `ArrayindexNamedEntity`

  e.g. `q = a[0];`
- `NullNamedEntity` : If a pointer is made to point to address 0 then it is represented by `NullNamedEntity`
- `FunctionnameNamedEntity` : A function name is represented by `FunctionnameNamedEntity`
- `UndefinedNamedEntity` : If a variable is undefined then the value represented by that variable is `UndefinedNamedEntity`

3.2 Structure of analyser generator

An analyser writer needs to write two following components:

- Kulang specifications [6]
- Data flow analysis API class

3.2.1 Kulang specifications

While analyzing a program, Intraprocedural Control Flow Graph (ICFG) is created for each function. For each function in ICFG, GEN and KILL summary is computed. The specification for computing GEN and KILL summary is specified using Kulang query files. A kulang query file has "**.klg**" extension. The query file consists of a Kulang query to be solved. The file is organized in the following way:

- Package declaration : This is the first line in the kulang file. It tells the Kulang compiler to put the generated Java files into the specified package, for all the queries written in the file.

  Syntax :

package identifier

For e.g. `package darpan.klgLib;`

- Typedef :
  Syntax :

  TypeName::Type

  For e.g. `tup::tuple(int,NamedEntity);`

- Use declaration : The "use" declaration is used when the user wants to use/call other Kulang queries from other packages. It is similar to "`import`" statement in java. There are two following types:

  - `fun_use` : This defines the path from where the query generated Java files should be accessed
    Syntax :

    fn_use identifier;

    For e.g. `fn_use darpan.klgLib;`

  - `java_use` : It specifies the class file where Java files accessed in this query are present. It is followed by `java_decl`, which specifies the Java functions used from the file specified in `java_use`
    Syntax :

    java_use identifier;
    java_decl
    type function ;

    For e.g.
    ```
    java_use darpan.klgLib.Aux;
        java_decl
        int print_set(set NameEntity);
    ```

- Mode : Mode is defined as follow. FLA stands for Function Level Analysis.

  Syntax :

  `[FLA];`

- Lattice Type : Type of the elements in lattice. Two following types are possible:

  - Forward lattice : Used in forward analysis
    Syntax :

    Forwardlattice IDENT::typedef;

    For e.g. `Forwardlattice L::tup;`

  - Backward lattice : Used in backward analysis
    Syntax :

    Backwardlattice IDENT::typedef;

    For e.g. `Backwardlattice L::tup;`

- Top : Top of the lattice for data flow analysis. Type of the Expr should be compatible with the type of lattice element given in the Lattice Specification. Following are the types

  - Forward top : top of forwardlattice
    Syntax :

    ForwardTop::Expr;

    For e.g. `ForwardTop::(tup);`

  - Backward top : top of backwardlattice
    Syntax :

    BackwardTop::Expr;

    For e.g. `BackwardTop::(tup);`

- Meet : specifies the meet operation to be performed at the merge of two paths. Following are the types:

  - Forward meet : meet operation for forward analysis
    Syntax :

    IDENT ForwardMeet IDENT : Expr;

    For e.g. `A ForwardMeet B : A+B;`

  - Backward meet : meet operation for backward analysis
    Syntax :

    IDENT BackwardMeet IDENT : Expr;

    For e.g. `A BackardMeet B : A+B;`

- Boundary values : It gives the starting value to solve the Data Flow problem.The Program Entry (in case of a Forward Problem) / Program Exit (in case of a Backward Problem) node will be initialized with this value.

  - Forward boundary value : boundary value for forward analysis
    Syntax :

    ForwardBoundaryValue : Expr;

    For e.g. `ForwardBoundaryValue :(tup);`

  - Backward boundary value : boundary value for backward analysis
    Syntax :

    BackwardBoundaryValue : Expr;

    For e.g. `BackwardBoundaryValue :(tup);`

- Flow functions : It defines how data flow value changes when it passes through a node or edge. The specific node or edge is available to the function as a parameter. Four types of flow function exits - `ForwardNodeflow`, `BackwardNodeFlow`, `ForwardEdgeFlow` and `BackwardEdgeFlow`
  Syntax for flow functions:
  - *Forward/BackwardNodeflow(IDENT:nodeType, IDENT:LatticeType)*
    let

        Expressions;

    in

        Return variable;

    item
    Forward/BackwardEdgeflow(IDENT:nodeType, IDENT:LatticeType)
    let

        Expressions;

    in

        Return variable;

Kulang Constructs

- Types : It supports standard data types such as int, char, string etc, IR model types and JAVA methods that operate over those types. Type casting is also supported in kulang specifications.
- Constants : PRISM allows usual constants for standard types,such as NULL, true, 1, a etc.
- Expressions : Expressions are bound to identifiers. Contional expressions are also allowed.

  For e.g.

  ```
  x=if(condition)
    then
         expression1
    else
         expression2
    endif;
  ```

- Tuples : It is collection of data items of same or different data types.

  For e.g. `Tuple=[Expr1,Expr2];`

- Set : It is a collection of data items of same data type.

  For e.g. `Set={a,b};`

- Iterations : Consider an example as shown below, x takes each element in S and check for the condition after '|'. If it satisfies the condition then that element is returned and store it in `newSet`.

  For e.g. `newSet = {x←S | condition };`

- Accumulation : Consider an example as shown below, **a** is an accumulator which is initialized to an empty set. **'x'** takes each element in **'S'**. Operation is performed on **'x'** and then the result is added to the accumulator.

  For e.g.

  ```
  newSet= <<a :(set NamedEntity){}; x←A; a+{operation}>>;
  ```

3.2.2 Data flow analysis API class

This class contains functions to invoke the execution of the analysis. It also provides an API to get results of the analysis. It can be accessed by user to get data flow values at some program point. It can also be use while solving a data flow problem dependent on the implemented data flow problem.

Figure 3.2 shows an architecture of generated analyzer. The solver takes data flow analysis API class, functions generated by kulang compiler and some utility functions provided by PRISM as input and generates a result.

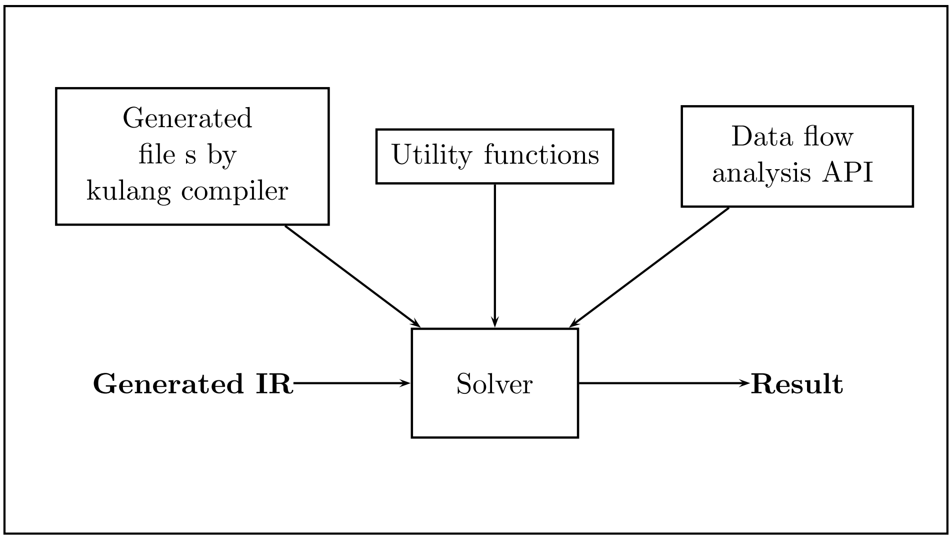


Figure 3.2: Architecture of analyzer generator

3.2.3 Running the generated analyser

The steps for configuring and running the analyzer are given in the Appendix B.

3.3 Bi-directional Solver

In this section, we describe the Bi-directional [3] solver, its modules and its limitations. The Kulang, a specification language, is used to specify the data flow analysis in PRISM. The bi-directional Kulang compiler parses the specifications and generates java classes. The Bi-directional Solver invokes the generated java classes to solve the data flow problem.

Following are generated .java files:

- `kulang<queryname>FG_Flow.java`: It contains *forward and backward* flow and inter-procedural edge functions.
- `kulang<queryname>FG_Meet.java`: It contains *forward and backward* meet functions.
- `kulang<queryname>FG_Adapter.java`: It provides an interface to create an instance of the query.
- `kulang<queryname>FG_ModelPopulator.java`: It populates the model.

3.3.1 Modules of Bi-directional Solver

Following modules are used in the bi-directional Solver.

- **DFStore:** It stores context sensitive data flow information. It also provides an API to access the data.
- **ValueContexts:** It stores information about contexts formed during analysis.
- **ContextTransition:** It stores context transition graph which is a method of representing call strings.
- **DFSolverLogger:** This module provides utilities for logging, measuring performance parameters of the core solver.
- **InterproceduralWorklist:** This module manages work list and applies a priority scheme to the work list.

More details about these modules are presented in the Appendix A

3.3.2 Limitations of Bi-directional Solver

Following are some of the limitations of the Bi-directional solver:

- It does not support Incremental data flow analysis.
- It outputs the results in a text file.

- Meet function needs to be explicitly specified in kulang specifications. The meet function can be inferred from the lattice of the data flow problem.
- There is no proper way to debug the kulang specifications.
- The specification language is not very intuitive and has a steep learning curve.

Chapter 4

Liveness Based Reaching Definition Analysis using PRISM

In order to understand PRISM, we have implemented a query for Liveness-based reaching definition analysis using PRISM. Other queries which are implemented using PRISM are Liveness analysis, Strongly Liveness analysis, Liveness analysis with aliasing and Strong liveness analysis with aliasing. Full specifications of Reaching definition with and without liveness are given in Appendix D. This section describes flow functions for reaching definition analysis with and without liveness.

4.1 Reaching definition analysis

This section describes the flow functions for Intra-procedural reaching definition analysis. The Data flow equations for Reaching Definition analysis is shown below:

$$\begin{aligned} In_n &= \begin{cases} BI & \text{n is Start block} \\ \bigcup\limits_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \\ Out_n &= Gen_n \cup (In_n - Kill_n) \\ BI &= \{d_x : x = undef \mid x \in Var\} \end{aligned} \tag{4.1}$$

Flow function for binary statements is shown below. Here, we are checking statements having '=' as an operator, otherwise we directly propagates the values at the IN of that node. First we find the NE present in the expression (line no.4-6). '`line`' function at line 7, returns the line number of the statement. '`kill`' set at line 8 contains the set of elements that need to be removed from the set of data flow values reaching at that node. '`HasElement`' is a function(as shown below) defined in '**Aux.java**' file, it takes `NamedEntity` and set of tuples as a parameter, and returns a set of tuples whose `NamedEntity` is same as that of the `NamedEntity` present in the `lhs` of the statement. '`def`' set contains the definition defined at that node.

```
ForwardNodeflow( n: Binary, S: L )
    if(operator(n) == '=') then
       let
               rt_expr = rhs(n);
               l_exp = lhs(n);
               ne = NE(n);
               line_no = line(l_exp);
               kill = HasElement(ne,S);
               def = {[line_no,ne]};
       in
               (S-kill)+def
       else
               S
       endif;
```

Following is a code for `unary` statements. Similarly, it will generate a '`kill`' set by using a function '`HasElement`' . `def` contains a definition defined at that node. Statement `'(S-kill)+def'`, will calculate the values at the OUT of that node.

```
ForwardNodeflow( n: Unary, S: L )
    let
            ne = NE(n);
            line_no = line(n);
            kill = HasElement(ne,S);
            def = {[line_no,ne]};
    in
            (S-kill)+def;
```

For other types of statements, it will just propagates the values reaching at the IN of that node to its OUT. Following in the code.

```
ForwardNodeflow( n: _, S: L )
    S;
```

Java code for function '`HasElement`' is shown below :

```
static Set HasElement(NamedEntity ne , Set info_in)
{
    int no_ele;
    Set ret= new HashSet();
    no_ele=info_in.nElems();
    Enumeration en=info_in.Enumerate();
    while(en.hasMoreElements())
    {
            Tuple t=(Tuple)en.nextElement();
            if(t.AtIndex(1).equals(ne))
```

```
                        ret.insert(t);
        }

        return ret;
}
```

4.2 Liveness-Based Reaching Definition Analysis

This section describes the flow functions for Liveness-Based Inter-procedural reaching definition analysis. This is a bi-directional analysis and hence flow function will take two lattices `R` and `L` as a parameter. Lattice `R` is for reaching definition analysis and `L` is for strongly liveness analysis. In first phase, it will do strongly liveness analysis which is a backward flow analysis. Data flow equations for liveness-Based reaching definition analysis in shown below. The reaching definition sets (RIn_n and $ROut_n$) are restricted to liveness set (LIn_n and $LOut_n$)

$$
\begin{aligned}
LIn_n &= f_n(Out_n) \\
LOut_n &= \begin{cases} BI & \text{n is End} \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \\
&\text{where,} \\
f_n(X) &= \begin{cases} (X - \{y\}) \cup (\mathit{Opd(e)} \cap \mathit{Var}) & \text{n is y = e, e} \in \text{Expr, y} \in \text{X} \\ X - y & \text{n is input(y)} \\ X \cup y & \text{n is use(y)} \\ X & \text{otherwise} \end{cases} \\
RIn_n &= \begin{cases} RBI & \text{n is Start block} \\ \bigcup\limits_{p \in pred(n)} Out_p \,|_{LIn_n} & \text{otherwise} \end{cases} \\
ROut_n &= Gen_n \cup (In_n - Kill_n) \,|_{LOut_n} \\
RBI &= \{d_x : x = undef \mid x \in Var\}
\end{aligned}
\tag{4.2}
$$

Flow function for `binary` statements is shown below. The set '`rNE`' at line 6, contains `NamedEntity` in rhs of the statement and '`lNE`' at line 7, contains `NamedEntity` in lhs of the statement. At line 8, if the statement contains "=" operator and variable in lhs is present at the OUT of node, then we will add all the variables used and remove the defined variable in the statement from the set 'x'. Otherwise, we will add all the used variables in the set 'x'.

```
1. BackwardNodeflow( n: Binary, R: rec, L: Liv )
2.        let
3.
4.                rt_expr=rhs(n);
5.                lt_expr=lhs(n);
6.                rNE=getNEs(rt_expr);
7.                lNE=getNEs(lt_expr);
8.                x = if(operator(n)=='=')
9.                        then
10.                                if(isLive(lt_expr,L)==true)
11.                                        then
12.                                                (L - lNE) + rNE
13.                                        else
14.                                                L
15.                                endif
16.                        else
17.                                ( L + lNE ) + rNE
18.                        endif;
19.       in
20.               x;
```

Flow function for `unary` statements is given below. At line 23, `operands_ne` contains the `NamedEntities` of all the variables used in the statement. At line 25, variables which are used in the statement are added to the Liveness set 'L'.

```
21. BackwardNodeflow( n: Unary, R: rec, L: Liv )
22.       let
23.               operands_ne=getNEs(n);
24.       in
25.               L+operands_ne;
```

Flow function for `Call` statements is given below. At line 29, `UseInCall` contains `NamedEntities` of all the variables present in call statement. At line 31, the used variables are added to the Liveness set.

```
26. BackwardNodeflow( n: Call, R: rec, L: Liv )
27.       let
28.               d = emptySet();
29.               UseInCall = getNEsFromCall(n);
30.       in
31.               d + L + UseInCall
```

Flow function for other types of nodes is shown below.

```
32. BackwardNodeflow( n: _, R: rec, L: Liv)
33.       L;
```

After computing liveness set as discussed above, based on that set reaching definition is performed. Here we will compute reaching definition of those variables which are live at the IN of that node. This is a forward flow analysis.

Flow function for `binary` statements is shown below. At line 39, `line_no` will contain the line number of the statement. At line 40, `kill` set will contain `NamedEntity` of all the values at the IN of node whose `NamedEntity` is same as that of the variable defined at the statement. At line 41, `sur` will contain the computed data flow value at the node. From line 43 to 49, if statement contains "=" operator then it will check whether the value which is defined is live or not, otherwise it will propagate only those values which is live at the IN of that node. Function `removeNonLive` checks the liveness of the variable defined at the node. If the variable is live then add the definition to the reaching set.

```
34. ForwardNodeflow( n: Binary, R: rec , L:Live )
35.        let
36.                rt_expr=rhs(n);
37.                l_exp=lhs(n);
38.                ne=NE(n);
39.                line_no=line(l_exp);
40.                kill=HasElement(ne,R);
41.                def={[line_no,ne]};
42.                sur=(R-kill)+def;
43.               survive=if(operator(n)=='=')
44.                        then
45.                                removeNonLive(sur,L)
46.                        else
47.                                removeNonLive(R,L)
48.
49.                        endif;
50.        in
51.                survive;
```

Following is the flow function for **unary** statements. At line 60, after computing the values at the node (55-59), if the variable is live at the IN of the node then we will add the definition of the variable defined at that not.

```
52. ForwardNodeflow( n: Unary, R: rec , L:Liv )
53.        let
54.
55.                ne=NE(n);
56.                line_no=line(n);
57.                kill=HasElement(ne,R);
58.                def={[line_no,ne]};
59.                sur=(R-kill)+def;
60.               survive=removeNonLive(sur,L);
61.        in
62.               survive;
```

Flow function for other types of statements is given below.

```
63. ForwardNodeflow( n: _, R: rec , L:Liv)
64.        let
65.                survive=removeNonLive(R,L);
```

```
        in
                survive;
```

Following is the Java code for function ‘`isLive`’

```
static boolean isLive(Expr ex,Set s)
{
      try
      {
        Ident_AST iast = (Ident_AST) ex.operands().getAt(0);
        NamedEntity ne = iast.NE();
        if(s == null)
              return false;
        if(s.belongs(ne))
              return true;
        else
              return false;
       }
       catch(Exception e)
      {
        return true;
        }
}
```

4.3 SPEC Benchmark Evaluation

In order to compare the performance between two analysis i.e. reaching definition analysis with and without liveness, we tested both for SPEC Benchmarks. For each query, we measured the average size of the set of data flow values computed at each program point. In general, average size of the set at each program point in liveness-based analysis is much smaller then that of the normal reaching definition analysis. However, PRISM solver is not including local and global declarations as a statement and thus there is a possibility that the results from the benchmark may not concur with the actual numbers.

Performance measurement of reaching definition without liveness is shown in table 4.1.

Table 4.1 Benchmark results for normal reaching definition analysis.

Name	No. of Basic Blocks	Avg. values at Entry (in %)	Avg. values at Exit (in %)	Total values (in %)
bzip2	8004	26.878	26.826	26.852
mcf	1066	3.991	4.001	3.996
hmmer	24996	2.574	2.566	2.570
sjeng	10892	39.219	39.245	39.232
h264ref	42559	6.370	6.225	6.297
lbm	603	6.515	6.537	6.526

The performance measurement of reaching definition with liveness is shown in table 4.2.

Table 4.2 Benchmark results for liveness-based reaching definition analysis.

Name	No. of Basic Blocks	Avg. values at Entry (in %)	Avg. values at Exit (in %)	Total values (in %)
bzip2	8004	4.852	4.570	4.711
mcf	1066	0.724	0.666	0.695
hmmer	24996	0.672	0.636	0.654
sjeng	10892	3.685	3.466	3.576
h264ref	42559	5.321	5.197	5.259
lbm	603	1.633	1.600	1.616

Figure 4.1 shows a comparison between two analysis.

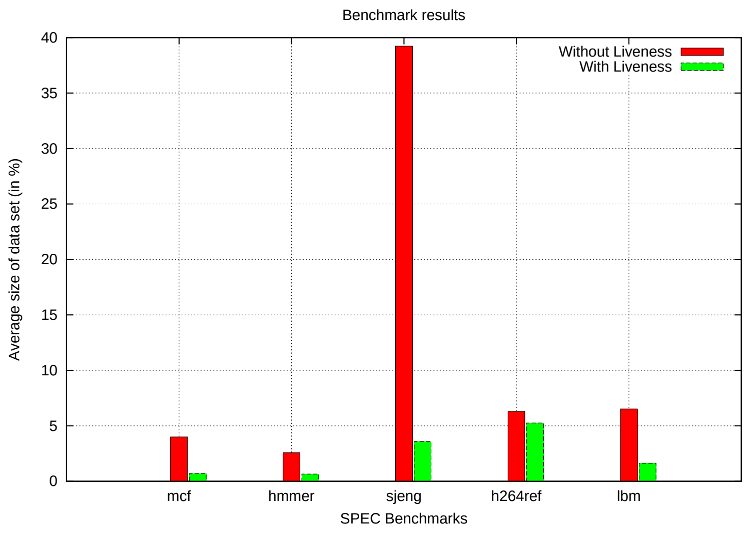


Figure 4.1: Percentage reduction in size of data set for Reaching definition analysis with and without liveness

4.4 Summary

We have implemented a query for reaching definition analysis with and without liveness using PRISM. The reaching definition analysis described in this chapter is implemented at intra-procedural level, improved implementation in given in Appendix D. Other queries which are implemented using PRISM are Liveness analysis, Strongly Liveness analysis, Liveness analysis with aliasing and Strong liveness analysis with aliasing.

In order to compare the performance between reaching definition analysis with and without liveness, we tested both for SPEC Benchmarks. For each query we measured the average size of the set at each program point. It is observed that the average size of the set at each program point in liveness-based analysis is much smaller than that of the normal reaching definition analysis with a possibility of the empirical values deviating from the actual values owing to non inclusion of the local and global statements.

Chapter 5

An Incremental Solver for PRISM

In this section, we describe implementation of the Incremental Solver. It is divided into the following parts:

- **Preprocessing** It reads the two IRs and finds the difference between them.
- **Data flow computation** It consist of following two parts:
  - *Re-initialization* which constructs the affected region (set of affected node).
  - *Re-computation* which identifies boundary nodes and computes the information only for the affected region.
- **Update** Merges the computed information for the affected region with the old information.

5.1 Architecture of Incremental Solver

Figure 5.1 shows the architecture of the Incremental Solver. The *Preprocessor* takes *Old IR* and *New IR* as inputs. It compares the two IRs and detects the changed nodes. Output of the *Preprocessor* is then passed to the *Re-initializer*. The *Re-initializer* computes the affected region, a set of affected nodes, and a set of boundary nodes. The *Solver* computes the data flow information for the affected region and passes the results to the *Merger*. *Merger* merges the solution and produces the updated results. Note that recreation of specifications for the given analysis is not required in Incremental analysis. Therefore, solver will use the old generated specifications.

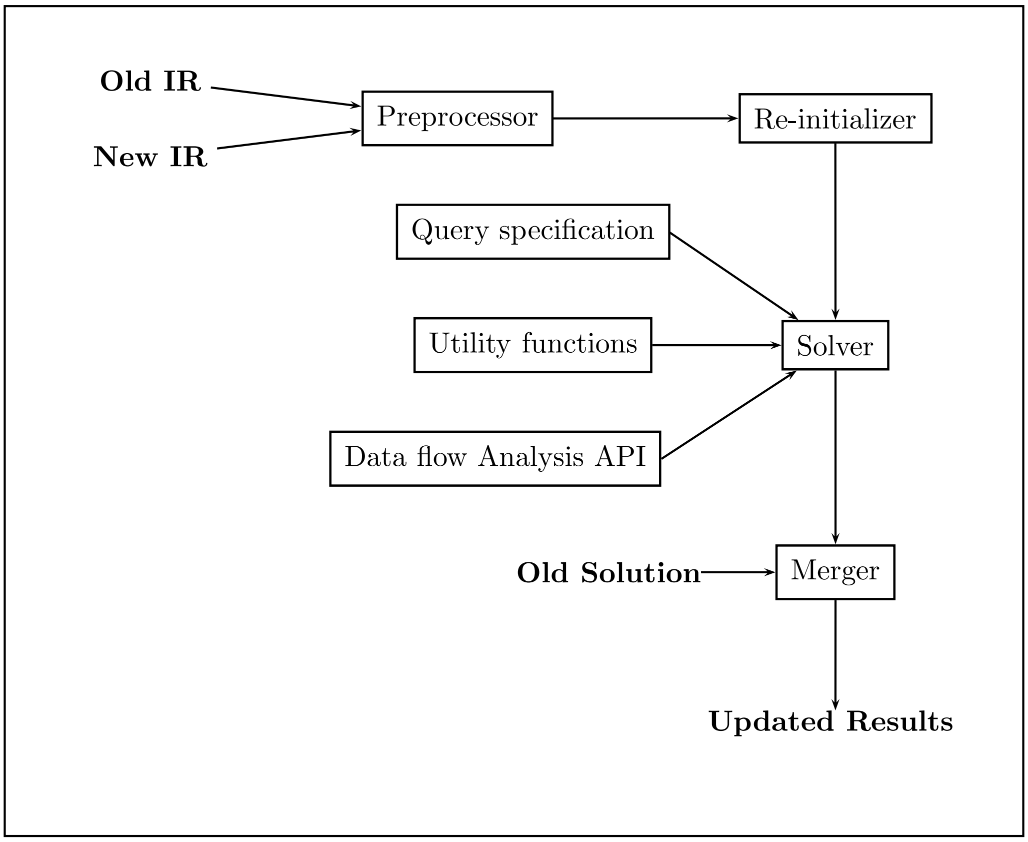


Figure 5.1: Architecture of Incremental PRISM

5.2 Modules of Incremental Solver

The core PRISM solver was extended to support Incremental data flow analysis. Following modules are the extensions to the PRISM solver.

- `IncrementalDriver`: This module takes two IRs as an input and calls `FLA<queryname>Client` to initiate the analysis.
- `FLA<queryname>Client`: This module discards the old IR after storing the old IR node information into `OldIRNode` and initiate incremental analysis of new IR.
- `CreateAffectedRegion`: This module detects the change in the control flow graph. It also detects affected region and a set of boundary nodes. The API functions which detects affected region and boundary nodes are as follows:
  - `detectChangedNode(Application app)`: It compares old and new IRs and detect the changed nodes(or program points).

  - `RDef_affectednodes()`: It detects affected region for reaching definition analysis. Different analysis will have different method of computing affected region.
  - `findBoundaryNodesForwardAnalysis`: It detects boundary nodes for forward analysis.

- `OldIRWorklist`: This modules stores a list of old IR nodes.

5.3 Incremental Reaching Definition Analysis

Using Incremental Solver we have performed reaching definition analysis. It consists of two following parts:

- Specification for reaching definition analysis
  - **Node flow functions** Node flow functions consists of flow functions for each statement. It is specified in the Kulang specification.
  - **Meet function** This function merges information coming from two different nodes. It is specified in the Kulang specification.
  - **Inter-procedural flow functions** These functions enable transferring information from a caller context to callee and vice versa. Since Kulang does not currently support specification for inter-procedural analysis, they are implemented in java.
- Java code for creating affected region:
  - **Creation of Affected Region** The function `RDef_affectednodes()` creates affected region for reaching definition analysis. The affected region creation algorithm, and hence the implementation depend on the kind of analysis.

5.4 Assumptions and Limitations

The assumptions that we make are listed with the corresponding justifications as follows:

- **Pointer information will remain same.** At present we have not implemented incremental analysis for general frameworks. So, it is not possible to incrementally update the pointer information.
- **There is no change in the context information.** Otherwise, it is not possible to map old information to the new information.

- **Declarations of variables haven't change.** Change of declaration implies that there is a change in the lattice for that particular variable. For handling such changes, we need to run exhaustive analysis, incremental analysis won't suffice.

- **There is no structural change in the graph.** The reason for making this simplifying assumptions is that the output from the old solver is read from a text file. If the output was serialized in form of PRISM objects, we could have handled structural changes as well.

The results from the old solver are stored in a text file, and not serialized into PRISM objects. This renders it impossible to deserialize the results from the old solver and to serialize our results back. Late discovery of this issue has prevented any corrective measures. Therefore, we take the following two assumptions.

- A name can refer to a single variable in a program at any given program point.

- The past information is stored flow sensitively.

The following are the main limitations of the Solver:

- Affected region which is analysis specific, the code for this has to be written manually.

- The result is stored in a non-standard format by the Solver.

5.5 Testing

In the absence of serialization of IR, it was not possible to test the code for real world applications. Hence, we had to artificially add the changes to check the performance of the Incremental Solver. Some of the test cases that we used to check the Incremental Solver are as follows:

- **Changed statement contains a definition of global variable** Consider the control flow graph in figure 5.2. Let a be a global variable and $n7$ be the changed node. The Affected region created by the solver is $\mathbf{AR} = < n7, n9, n11, n15, n16 >$ and $n7$ is the boundary node. The nodes traversed by the solver are $< n7, n9, n11, n15, n16 >$.

- **Changed statement contains a definition of local variable** Let a be a local variable and $n7$ be the changed node in figure 5.2. The affected region created by the solver is $\mathbf{AR} = < n7, n9, n11, n12 >$ and boundary nodes are $n7$ and $n12$. The nodes traversed by the solver are $< n7, n9, n11, n12 >$.

- Let the changed node be $n8$. Variable z is passed as a parameter in function *func1(z)* at $n10$. The affected region created by the solver is **AR** $= < n8, n10, n21, n22, n23, n24, n25, n12, n3, n4 >$ and boundary nodes are $n8$ and $n3$. The node traversed by the solver are $< n8, n10, n21, n22, n23, n24, n25, n12, n3, n4 >$.

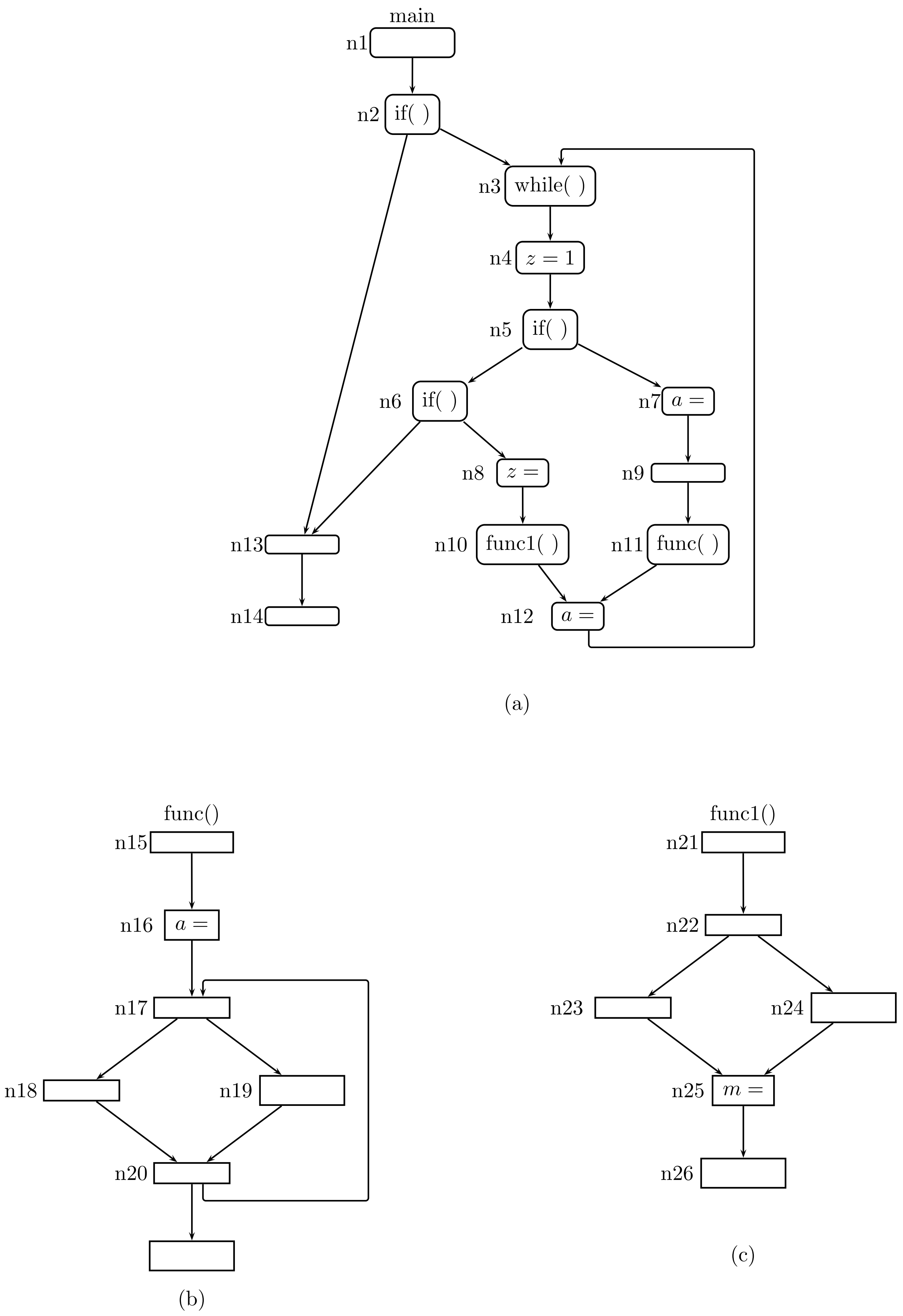


Figure 5.2: Test case

Chapter 6

Issues in Incremental Analysis for General Frameworks

The incremental analysis discussed in chapter 2 is restricted to bit-vector frameworks only. Bit-vector frameworks cover a relatively simpler case where the data flow information of different entities is independent of each other. In situations where the data flow information of entities is interdependent, a more general kind of flow functions are needed. The corresponding general frameworks are called *non-separable*.

In this chapter, we will discuss some of the issues in extending the existing incremental analysis to a general frameworks by means of constant propagation analysis.

6.1 Issues in Incremental Analysis for Constant Propagation

Consider a component lattice for Constant Propagation analysis in fig 6.1. Given a variable x and a program point u, apart from associating integer constants with x at u, this analysis associates two additional values: *undef* to indicate that no definition of x has been seen along any path reaching u, and *nonconst* to indicate that x can have different values at u along different paths reaching u.

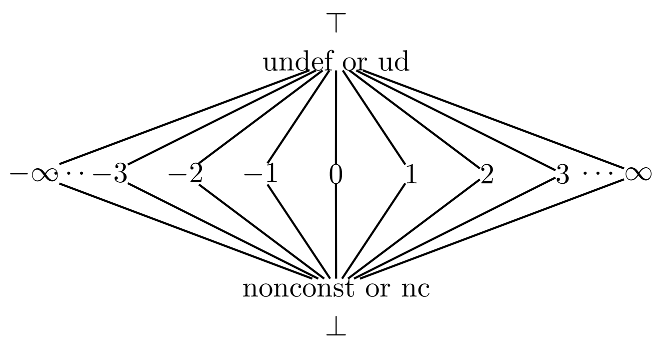


Figure 6.1: Component Lattice for Integer Constant Propagation

6.1.1 Flow Functions

Following are the possible flow functions in Constant Propagation analysis:

- Top : It is similar to the raise function in bit-vector frameworks. It always results in Top value.
- Bottom : It is similar to the lower function in bit-vector frameworks. It always results in Bottom value.
- Constant : Function whose result is always constant. Consider a constant propagation analysis in fig 6.2(a), the flow function at node n2 will always produce a constant value 1 at the OUT of n2. There is one function per value.
- Side level : Consider a constant propagation analysis in fig 6.2(b), the value of a at node n3 depends on the value of b and c. If b and c is constant, as in the figure, then a will also become constant. If any of them is top/bottom then a will become top/bottom. Flow functions for these type of statements whose value depends on the operands of the statement is known as side level function.

6.1.2 Possible Changes in Flow Functions

In addition to the changes given in section 2.1.1, a change from intermediate level to other levels is also possible.

- a change to top,
- a change to bottom, and
- a side level change

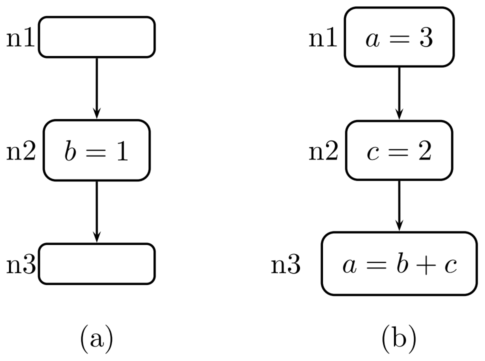


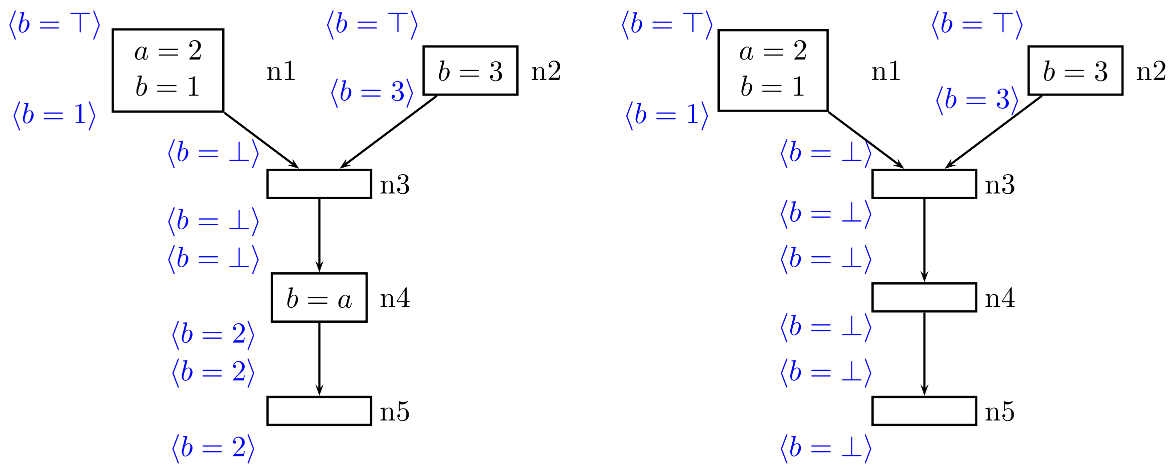
Figure 6.2: Flow functions in Constant Propagation analysis

Change from intermediate level to top

Consider a Constant propagation analysis in control flow graph shown in figure 6.3. In this case, if expression $b = a$ is removed from the node then b will become $\top$ at the OUT of node $n3$, a change from intermediate level to top($\bot$).

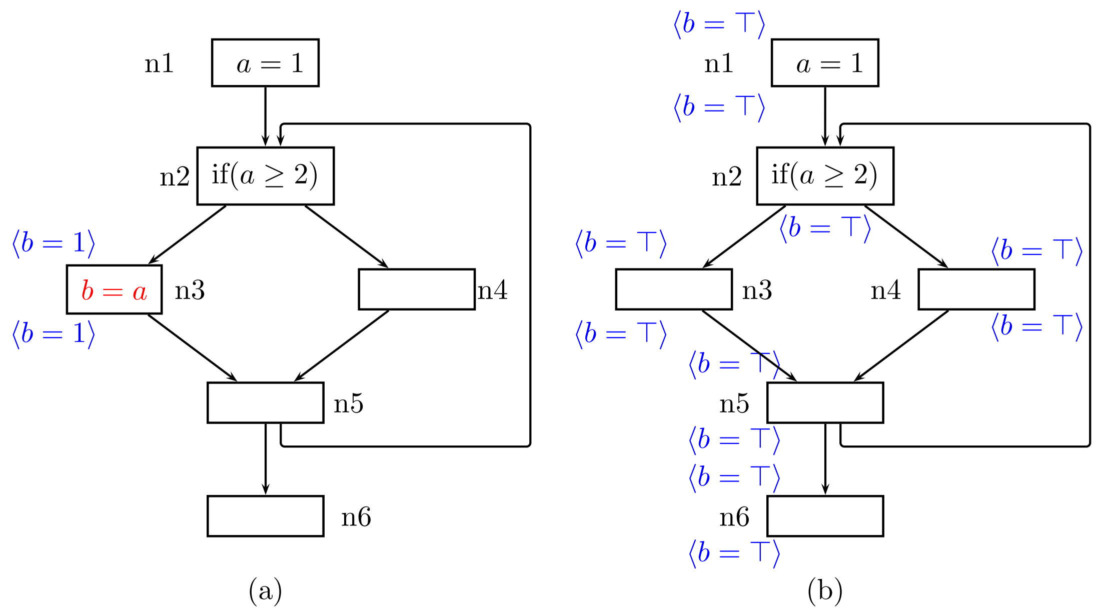
Change from intermediate level to bottom

Consider a control flow graph as shown in figure 6.4. After a removal of expression $b = a$ at node $n4$, b becomes bottom($\bot$) at the OUT of node $n4$ (as shown in figure 6.4(b)), a change from intermediate level to bottom($\bot$).



$\langle b = \top\rangle$
$a = 2$
$b = 1$
$\langle b = 1\rangle$
n1
$\langle b = \top\rangle$
$\langle b = 3\rangle$
$b = 3$
n2
$\langle b = \bot\rangle$
n3
$\langle b = \bot\rangle$
$\langle b = \bot\rangle$
$b = a$
n4
$\langle b = 2\rangle$
$\langle b = 2\rangle$
n5
$\langle b = 2\rangle$
$\langle b = \top\rangle$
$a = 2$
$b = 1$
$\langle b = 1\rangle$
n1
$\langle b = \top\rangle$
$\langle b = 3\rangle$
$b = 3$
n2
$\langle b = \bot\rangle$
n3
$\langle b = \bot\rangle$
$\langle b = \bot\rangle$
n4
$\langle b = \bot\rangle$
$\langle b = \bot\rangle$
n5
$\langle b = \bot\rangle$

Figure 6.4: Change from intermediate level to bottom



n1
$a = 1$
n2
if($a \geq 2$)
$\langle b = 1 \rangle$
$b = a$
n3
$\langle b = 1 \rangle$
n4
n5
n6
(a)
$\langle b = \top \rangle$
n1
$a = 1$
$\langle b = \top \rangle$
n2
if($a \geq 2$)
$\langle b = \top \rangle$
$\langle b = \top \rangle$
n3
n4
$\langle b = \top \rangle$
$\langle b = \top \rangle$
$\langle b = \top \rangle$
$\langle b = \top \rangle$
n5
$\langle b = \top \rangle$
$\langle b = \top \rangle$
n6
$\langle b = \top \rangle$
(b)

Figure 6.3: Change from intermediate level to top

Side level change

Consider a control flow graph as shown in figure 6.5. Suppose, expression $b = a$ at node $n3$ has been removed as shown in figure 6.5(b). Due to this change b becomes 2 at the OUT of node $n3$, a side level change in the lattice.

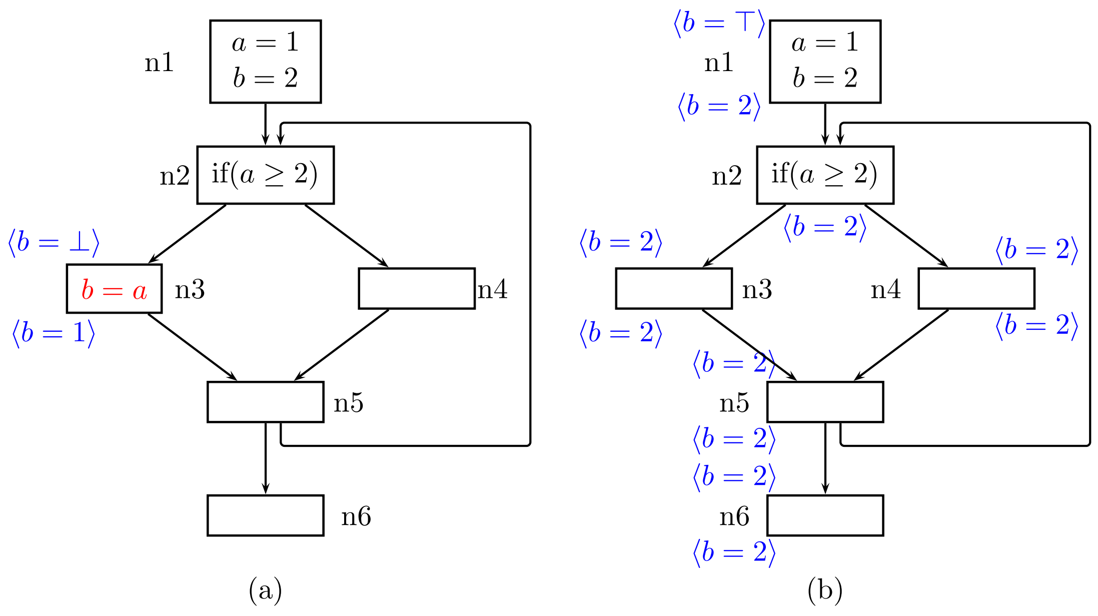


Figure 6.5: Side level change

6.1.3 Need for Affected Region for Every Change

Unlike bit-vector frameworks when there is a change to bottom, we cannot directly propagate the change to its neighboring nodes since it may depend on the OUT information of the node. We illustrate this via an example in figure 6.6. If expression $b = 2$ is removed, then according to existing incremental analysis, this is a change to bottom and we can directly propagate the change to its neighboring nodes. However, this is incorrect. The information at the IN of node 4 is bottom because of the expression $b = 2$ in node 4. If we will propagate $\perp$ to neighboring nodes, it will give incorrect results. Unlike bit-vector frameworks, we may need to create an affected region even if there is a change to bottom. Thus, the solution is to create an affected region for all kind of changes.

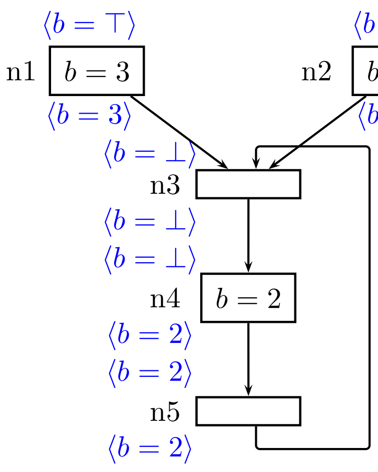


Figure 6.6: Constant propagation analysis

6.2 General Optimization of the size of Affected Region

We outline an optimization idea to reduce the size of the affected region which can eliminate certain boundary nodes from being included in the boundary region. This optimization is based on the observation that some boundary nodes can be characterized by the concept of *dominance frontier*. We define the concept of *Dominance Frontier* before elaborating on the optimization idea.

Dominance Frontier

Let n and m be nodes in the CFG. The node n is said to *dominate* m ($n \geq m$), if every path from **Start** to m passes through n. If $n \neq m$, then n *strictly dominates* m and is denotes as $n > m$.

The *dominance frontier* [4] of a node n, denoted by **df(n)** is given as:

$$df(n) = \{m \mid \exists p \in pred(m), (n \geq p \text{ and } nm)\} \tag{6.1}$$

Consider a graph in figure 6.7. Node $n1$ contains a definition of x. Node $n4$ is *dominated* by node $n1$. Consider a node $n5$, which is a immediate successor of $n4$, is node dominated by $n1$. Node such as $n5$, is said to be in the *dominance frontier* of $n1$.

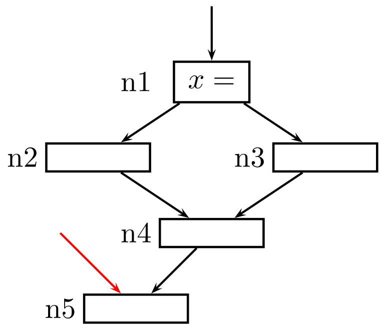


Figure 6.7: Figure shows a dominance frontier of $n1$

We first illustrate via an example that dominance frontier nodes are a subset of the boundary nodes. We then establish that all the possible candidates of removal form a dominance frontier. Given these two facts, it follows that the nodes that can be deleted can be identified during the AR construction itself, without further delays.

All Dominance Frontier nodes are Boundary nodes (but not vice versa) Consider a node n, in an affected region (AR) and c be the changed node. If n is a dominance frontier of c, then n must be a boundary node. This follows from the fact that there is some path that reaches m without passing through the changed node c. But if some node m is a boundary node of AR(C), then m need not be a dominance frontier. We explain this via an example in figure 6.8. If i is a changed node, then $AR(i) =< i, j, l, m, n >$ Here m is a boundary node but it is dominated by block i and hence not its dominance frontier.

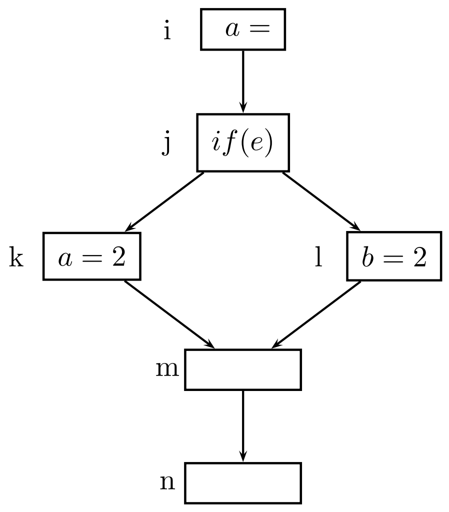


Figure 6.8: Control flow graph for constant propagation analysis

Possible Removal Candidates Form a Dominance Frontier Let us consider the example in figure 6.9. Let the changed node be $n5$. The affected region (AR) will contain the nodes $n5, n6, n7, n3$ and $n4$ with the node $n3$ as the boundary node. The information coming from nodes $n1$ and $n2$ will keep the value at the IN of $n3 = \perp$. The information coming from OUT of $n7$ will not affect the IN information of $n3$, and thus the information of the successors of $n3$. This implies that we can exclude the node $n3$, and thus its successor $n4$ from the affected region, reducing the size of the latter. We further note that the node $n3$ must be a dominance frontier of the change node ($n4$).

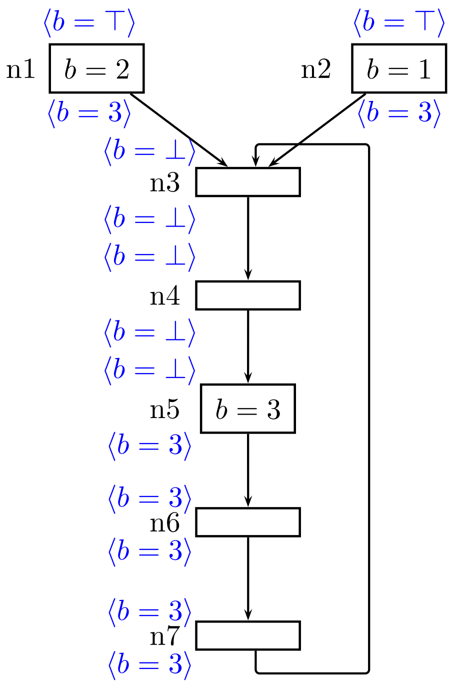


Figure 6.9: Constant propagation analysis

6.3 Reducing the size of the AR region

It follows from the above discussion that it is possible to identify such boundary nodes (dominance frontier of changed node) during AR construction. All such nodes have an outside influence. If the outside influence keeps the value to $\perp$ (or does not allow it to change), then there is no need to include this node in AR. This can help in reducing the size of AR. In general frameworks, we need to create AR for almost all the cases. However, we can still reduce the size of the affected region. This optimization is applicable for bit-vector vector frameworks also.

This optimization can work only on single change in a program. Handling for multiple changes need to be studied.

Chapter 7

Conclusions and Future Work

Incremental analysis is a method of avoiding redundant analysis in the case when only a part of the program has changed. In this report, we discussed the theory and implementation of incremental analysis for bit-vector frameworks, and proposed methods of extending it to general frameworks. The experiments on our test set have yielded motivating results. We also sketch a number of possible improvements in the implementation that may add to the performance and scalability of the incremental solver.

The following three limitations prevent us from testing our solver on real life, industry scale code bases:

- **Non inclusion of global and local declarations** PRISM solver is not including local and global declarations as a statement which may result in a loss of information.
- **Persistence of Results** The results of PRISM solver are currently written to a text file, whereas they should ideally be packed in a *result* object provided by the PRISM Solver and persisted.
- **Creation of Affected Region** From the fact that creation of affected region is a analysis specific, currently the code for this has to be written manually. Future work is to change the Kulang compiler to generate the affected region from the flow function.

Resolving these issues will help us in exposing our solver to some new challenges that might lead to new insights and avenues for improvements.

We have proposed a new idea for the optimization of affected region (AR). The idea is framework agnostic (can be fitted to both bit vector and general frameworks), and may lead to significant time savings when incorporated in the solver.

References

[1] Uday P. Khedkar. *A Generalised Theory of Bit Vector Data Flow Analysis.* PhD thesis, Computer Science and Engineering, IIT Bombay, 1995.

[2] Mukta Joglekar. *Liveness based pointer analysis in prism.* Mtech dissertation, Computer Science and Engineering, IIT Bombay, 2013.

[3] Vinit Deodhar. *Bidirectional Context sensitive analysis framework in PRISM.* Mtech dissertation, Computer Science and Engineering, IIT Bombay, 2014.

[4] Uday P.Khedker, Amitabha Sanyal and Bageshri Karkare *Data Flow Analysis: Theory and Practice.*

[5] Pritam Gharat. *Interprocedural analysis in PRISM.* Mtech report (Stage 1), Computer Science and Engineering, IIT Bombay, 2012.

[6] TATA Research Development and Design Center. *User Manual of Kulang.*

[7] Steven S. Muchnick and Neil D. Jones *Program Flow analysis: Theory and Applications.*

[8] Yi Lu, Lei Shang, Xinwei Xie and Jingling Xue. *An incremental Points-to Analysis with CFL-Reachability. Programming Languages and Compilers Group School of Computer Science and Engineering University of New South Wales, Sydney, NSW 2052, Australia. ylu,shangl,xinweix,jingling@cse.unsw.edu.au, School of Computer Science National University of Defence Technology, Changsha 410073, China.*

[9] Steven Arzt and Eric Bodden, *Efficiently updating IDE-based data-flow analyses in response to incremental program changes. Secure Software Engineering Group, European Center for Security and Privacy by Design (EC SPRIDE), Technische Universitt Darmstadt, 2 Fraunhofer SIT Darmstadt, Germany.*

[10] Rashmi Rekha Mech. *Incremental Data Flow analysis.* Seminar Report, Computer Science and Engineering, IIT Bombay, 2014.

Appendix A

Bi-directional Solver

In this section, we describe the implementation of core Bi-directional *PRISM* solver. The core bi-directional solver integrates with the generated bi-directional solver from kulang compiler and solves the given data flow problem in a context sensitive manner. The solver is the main driver of the data flow analysis. Its role in the data flow analysis is shown in figure A.1. In this chapter we describe the modules of bidirectional solver and data structures used by core solver.

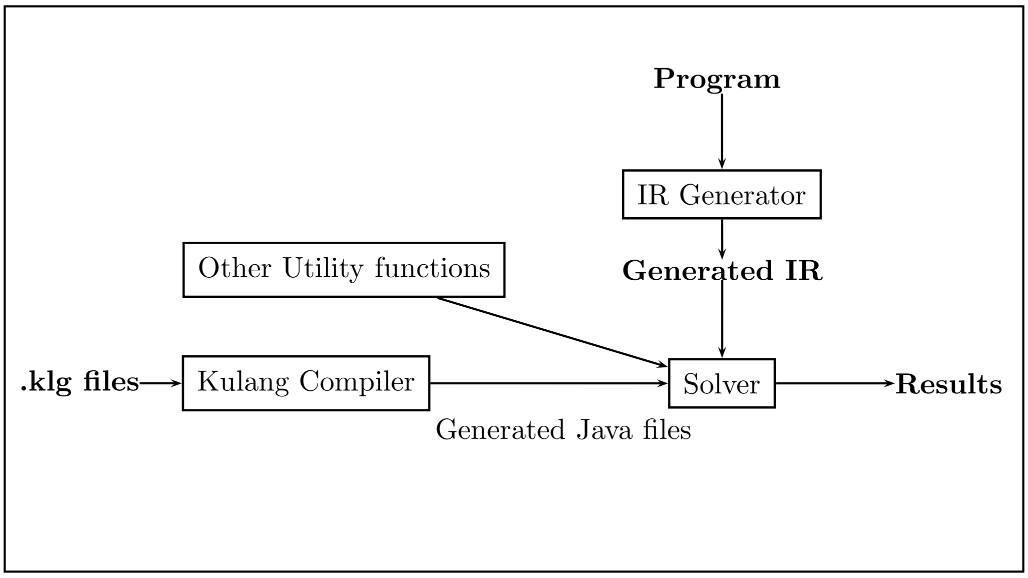


Figure A.1: Role of PRISM solver

Modules of Bidirectional solver

The core *PRISM* solver was extended to support bi-directional data flow problems. The following modules are the extensions to the *PRISM* solver:

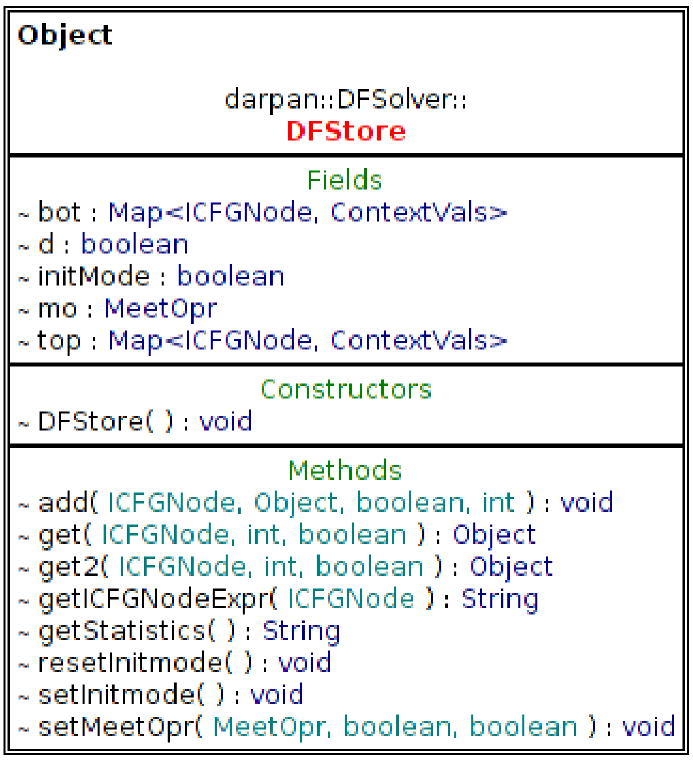

Object
darpan::DFSolver::
DFStore
Fields
~ bot : Map<ICFGNode, ContextVals>
~ d : boolean
~ initMode : boolean
~ mo : MeetOpr
~ top : Map<ICFGNode, ContextVals>
Constructors
~ DFStore( ) : void
Methods
~ add(ICFGNode, Object, boolean, int) : void
~ get(ICFGNode, int, boolean) : Object
~ get2(ICFGNode, int, boolean) : Object
~ getICFGNodeExpr(ICFGNode) : String
~ getStatistics( ) : String
~ resetInitmode( ) : void
~ setInitmode( ) : void
~ setMeetOpr(MeetOpr, boolean, boolean) : void

Figure A.2: Class Diagram for DFStore

1. *DFStore*: This module stores context sensitive data flow information and provides an API to access it. The class diagram for this module is shown in figure A.2.

   It stores data flow values at IN/OUT of every node. The information at IN/OUT consist of *(contextId, dataflow information)* pairs. A Hashmap is used to store this information. The key of the Hashmap consists of program statement or *ICFGNode* and the value consists of another HashMap containing *(contextId, data flow value)* pairs.

2. *ValueContexts*: This module holds information about contexts formed during the analysis. The class diagram for this module is shown in figure A.3.

   ValueContexts stores information of value contexts and maps each value context to a unique context id. It is implemented using a Vector which provides fast random access to query information about a value context. Each cell of the vector corresponds to a context id and stores information about context of the corresponding context id.

   An API is defined to store and retrieve values from the data structure. It is implemented in *ValueContexts.java* in the *DFSolver* package. The data structure is shown in figure A.3.

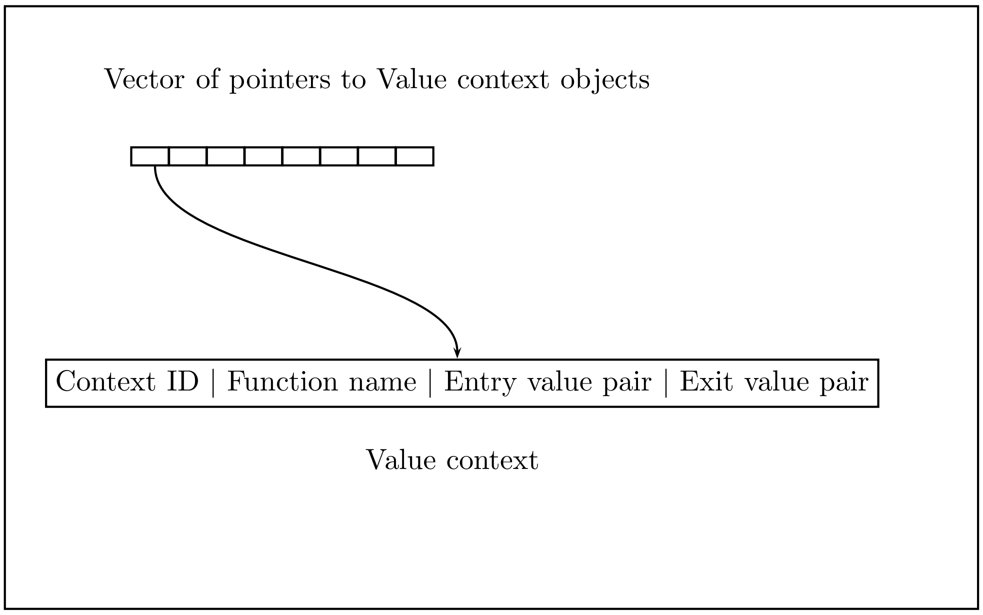

Vector of pointers to Value context objects
Context ID | Function name | Entry value pair | Exit value pair
Value context

Figure A.3: Storing value contexts

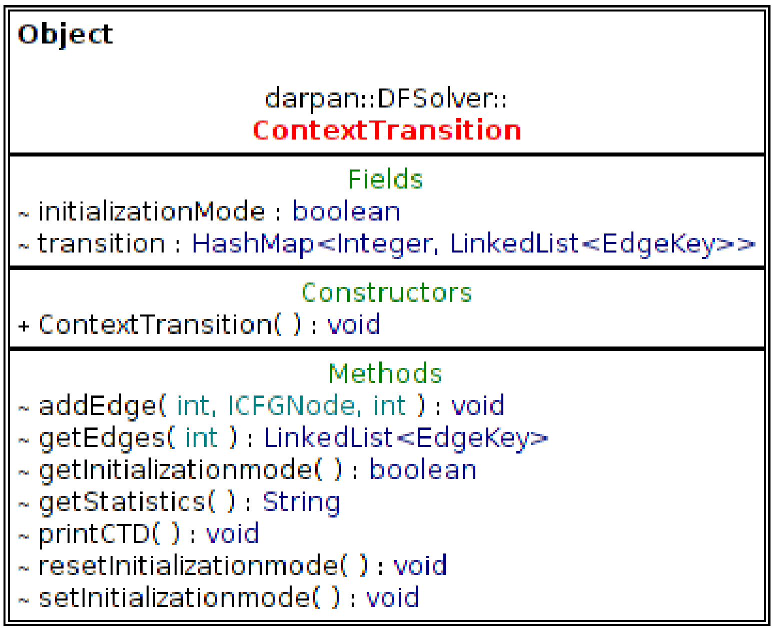
3. *ContextTransition*: This module stores context transition graph which is a method of representing call strings and provides an API to access it. The class diagram for this module in shown in figure A.4.

   Each context is represented as a node in a graph. Each function call in the program adds an edge from a caller context to callee context. It is implemented in *ContextTransition.java*. A Hashmap is used to store this information. The key consists of a combination of caller context id and *ICFGnode* of the function call. The Value consist of the callee context id. Bi-directional mappings are maintained using two hash maps which stores mapping in both the directions.

   The key value mapping for the data structure is,
   (caller context id, Call site) $\leftrightarrow$ *callee context id*

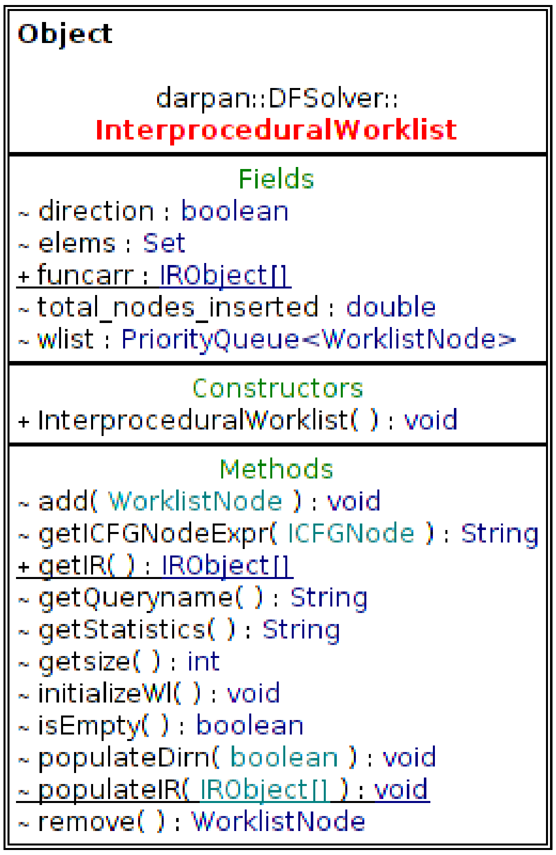
4. *DFSolverLogger*: This module provides utilities for logging, measuring performance parameters of the core solver.

5. *InterproceduralWorklist*: This module manages work list and applies a priority scheme to the work list. The class diagram for this module is shown in figure A.5.



```
Object

darpan::DFSolver::
ContextTransition

Fields
~ initializationMode : boolean
~ transition : HashMap<Integer, LinkedList<EdgeKey>>

Constructors
+ ContextTransition( ) : void

Methods
~ addEdge( int, ICFGNode, int ) : void
~ getEdges( int ) : LinkedList<EdgeKey>
~ getInitializationmode( ) : boolean
~ getStatistics( ) : String
~ printCTD( ) : void
~ resetInitializationmode( ) : void
~ setInitializationmode( ) : void
```

Figure A.4: Class Diagram for ContextTransition



```
Object

darpan::DFSolver::
InterproceduralWorklist

Fields
~ direction : boolean
~ elems : Set
+ funcarr : IRObject[]
~ total_nodes_inserted : double
~ wlist : PriorityQueue<WorklistNode>

Constructors
+ InterproceduralWorklist( ) : void

Methods
~ add( WorklistNode ) : void
~ getICFGNodeExpr( ICFGNode ) : String
+ getIR( ) : IRObject[]
~ getQueryname( ) : String
~ getStatistics( ) : String
~ getsize( ) : int
~ initializeWl( ) : void
~ isEmpty( ) : boolean
~ populateDirn( boolean ) : void
~ populateIR( IRObject[] ) : void
~ remove( ) : WorklistNode
```

Figure A.5: Class Diagram for Interprocedural work list

Appendix B

User Manual of PRISM

This section describes setting up PRISM on Ubuntu system. First section describes steps for setting up PRISM and how to generate an analyzer. Second section describes how to analyze a program using generated analyzer.

B.1 Steps for setting up PRISM on Ubuntu and generating the analyzer

The following are the steps :

- Add all the locations of jar files to classpath by setting the environment variable **CLASSPATH**.
- Create an environment variable PRISMROOT and store it in the url of PRISM root directory
- Create a directory inside $PRISMROOT/darpan directory and name the directory as the name of the package defined in kulang files. Copy all the specification files.
- Run the script **populatemodel.sh**. This will create a signature of the analysis.
- Update '**.ini**' file.
  - $**PRISMROOT** : location of PRISM root directory.
  - $**PRISMMODEL** : location of the signature file i.e.**Lpum.cdf**
  - $**REPOSDIR** : location where test result should be dumped
  - $**RREPOSDIR** : location from where IR should be read
- Update **.prj** file and set the path of a program to be analyzed

- Compile all the kulang files using command '**runKulangC filename**'. After successfull compilation, '**.java**' files will be created and, the generated analyzer would be created and compiled in the same directory.

B.2 Running an Analysis in PRISM

Following are the steps to run an analysis in PRISM.

- Write a program to be analyzed.
- Compile the program by using following command:

  ./cppfe –edg–gcc -O 'location_of_IR' 'filename'

  location of IR should be same as the location present in $**IRREPOSDIR**. For more options of cppfe, use the argument '**–help**'.
- Go to the analysis directory and update '**runPrism.sh**' file.
- Run the script 'runPrism.sh' script. The results will be dumped into the director $**REPOSDIR** as specified in '.ini' file

B.3 Steps for Setting up PRISM in Eclipse

- Open eclipse, create a new project and import the Driver and Client files of the analysis in the project.
- Go to project propertiesbuild pathadd external jar. Add all the jar files present in **$PRISMROOT/lib**.
- Go to run configurations and create a new run configuration. In the VM arguments section, create an argument -DENVFILE=path of .ini file In the program arguments give first argument the path of .prj file and second argument the location where test results are to be dumped.
- Write a c program to be analyzer. The name of the front end compiler to be used is cppfe. Compile the file using the command cppfe filename. Also give the argument edggcc and -O followed by location where IR should be generated. It should be the same as location of **$IRREPOSDIR** given in previous section.
- Go to eclipse and run the Driver file using the run configuration created in the previous section.
- The results will be dumped into the director $**REPOSDIR** as specified in '.ini' file

B.4 Steps for Running Incremental Driver

Following are the steps to run a Incremental driver:

- Compile two programs (old and new programs).
- Go to eclipse > Run Congigurations > Java Application > Arguments and add paths of both old and new IR as an argument. First argument should contain the path of old IR which is followed by the path of new IR.
- Upon execution, command line will prompt asking for an user input to run incremental driver or not. Enter "1" to run incremental driver

Appendix C

PRISM APIs

This section describes the APIs that we have used in our implementation. An exhaustive list appears in PRISM documentation.

C.1 API to Access IR

`Expr` is the base class that represents the expression in IR. This class has sub classes to represent the type of expressions, such as Unary, Binary etc. Following are some of the API to excess the expression in IR :

- `getCorrNE(Set st)`: This function takes the pointer information at the program point as a parameter and returns a set of variables that an expression represents.

  For e.g. If `*b` is an expression and {`b`→ `a`, `b`→ c} is a pointer information, then function will return {a,c}

- `RvalNE(boolean lr, Set aliasSet, Set copySet)` : This function takes pointer information as a parameter and returns a set of variables used in an expression

  For e.g. If `*b` is an expression and {`b`→ `a`, `b`→`c`} is a pointer information, then function will return {a,b,c}

- `pointsTo(Set st)` : This function takes a set of pointer information as a parameter and returns pointees of that expression

- `operands()` : returns the operands used in the expression

- `operator()` : returns operators used in the expression

- `lineNum()` : returns line number of the expression

- `NE()` : returns the NamedEntities of the operands in the expression

C.2 APIs to find properties of NamedEntity

- `coveredBy(NamedEntity ne, boolean must)` : it returns whether a given namedEntity is may or must covered by the passed namedEntity.

  For e.g. a.b is covered by a

- `getDataType()` : returns the data type of the operand
- `isGlobal()` : returns true if the given namedEntity is global
- `getNEsFromReturn(Expr ex)` : returns the namedEntity of the variables used in the return statement
- `getNEsFromCall(Expr ex)` : returns the namedEntity of the variables used in the call statements

Appendix D

Specifications of developed kulang queries

D.1 Reaching definition analysis

Following is the specification for Intra-procedural reaching definition analysis.

```
//Package declaration
package darpan.klgLib;

//Typedef
tup :: tuple(int,NamedEntity),
res :: set tup;

//Use declaration
fn_use darpan.klgLib;
fn_decl
res Meet (res, res),
res cartProduct ( set NamedEntity, set NamedEntity);

java_use darpan.klgLib.Aux;
java_decl
int line(Expr),
set tup HasElement(NamedEntity,res),
set NamedEntity IRefValNE@darpan.solverlib.FLAAnalysis(Expr,
                                          boolean,IRObject),
set NamedEntity getNEs(Expr),
set NamedEntity getNEsFromCall(Expr),
set NamedEntity getNEsFromReturn(ASTnode),
boolean isLive(Expr,set NamedEntity),
set NamedEntity aliasClosure ( Expr, IRObject),
boolean isDref(Expr),
boolean isMust(set NamedEntity),
set NamedEntity getPtsto(res,Expr),
res getPointesOf(res,set NamedEntity),
set NamedEntity dref(res,Expr),
set NamedEntity getLhsSet(res,Expr),
NamedEntity getKillCandidate(set NamedEntity),
NamedEntity getIllegalNE();

//Mode
[FLA];

//Declaration of lattice type
lattice L ::  res;

lCreaches_FG implements lCreaches_FG:

//Declaration of top value
top : (res){};

//Declaration of meet function
A meet B : A+B;

//Declaration of boundary value
BoundaryValue :(res){} ;

//Specifications of flow function for various statements
ForwardNodeflow( n: Symbol_AST, S: L )
        let
                ne=NE(n);
                exp=initExpr(n);
                line_no=line(exp);
                kill=HasElement(ne,S);
                def={[line_no,ne]};
        in
                (S-kill)+def;

ForwardNodeflow( n: Binary, S: L )
        if(operator(n)=='=')
        then
                let
                        ne=NE(n);
                        l_exp=lhs(n);
                        l_var=getNEs(l_exp);
                        line_no=line(l_exp);
                        kill=HasElement(ne,S);
                        def={[line_no,ne]};
                in
                        (S-kill)+def
        else
                S
        endif;


ForwardNodeflow( n: Unary, S: L )
        let
                ne=NE(n);
                line_no=line(n);
                kill=HasElement(ne,S);
                def={[line_no,ne]};
        in
                (S-kill)+def;

ForwardNodeflow( n: Call, S: L )
        S;

ForwardNodeflow( n: _, S: L )
        S;

ForwardEdgeflow(E: _, S: L)
        S;
```

D.2 Liveness-based Intra-procedural Reaching Definition Analysis

Following is the specification for Liveness-based Intra-procedural Reaching definition analysis.

```
//Package declaration
package darpan.klgLib;

//Typedef
tup :: tuple(int,NamedEntity),
livenesslattice :: set NamedEntity,
res :: set tup;

//Use declaration
fn_use darpan.klgLib;
fn_decl
res Meet (res, res),
res cartProduct ( set NamedEntity, set NamedEntity);

java_use darpan.klgLib.Aux;

java_decl
int print_set(set NamedEntity),
res removeNonLive(res,livenesslattice),
int line(Expr),
set tup HasElement(NamedEntity,res),
set NamedEntity IRefValNE@darpan.solverlib.FLAAnalysis(Expr,
                                          boolean,IRObject),
livenesslattice getNE(Expr),
livenesslattice getNEs(Expr),
set NamedEntity getNEsFromCall(Expr),
set NamedEntity getNEsFromReturn(ASTnode),
boolean isLive(Expr,set NamedEntity),
boolean Check_datatype_call(Expr),
set NamedEntity aliasClosure ( Expr, IRObject),
livenesslattice useInRhs(res,Expr),
boolean check_datatype(Expr),
boolean check_datatype_lhs(Expr),
boolean isDref(Expr),
boolean isMust(set NamedEntity),
set NamedEntity getPtsto(res,Expr),
res getPointesOf(res,set NamedEntity),
set NamedEntity dref(res,Expr),
set NamedEntity getLhsSet(res,Expr),
NamedEntity getKillCandidate(set NamedEntity),
set NamedEntity emptySet(),
NamedEntity getIllegalNE();

//Mode
[FLA];

//Declaration of lattice types
Forwardlattice Rec ::  res;
Backwardlattice Liv :: livenesslattice;

lCreaches_FG implements lCreaches_FG:

//Declaration of types of forward and backward lattices
ForwardTop : (res){};
BackwardTop : (livenesslattice){};

//Declaration of meet functions
A ForwardMeet B : Meet(A,B);
A BackwardMeet B : A+B;

// Specification of boundry values
ForwardBoundaryValue : (res){};
BackwardBoundaryValue :(livenesslattice){} ;

// Specification of backward flow functions for various statement types
BackwardNodeflow( n: Binary, R: Rec, L:Liv )
if(check_datatype_lhs(lhs(n))==true &&
                      check_datatype(rhs(n))==true)
then
        let
                rt_expr=rhs(n);
                check =CHECK(n);
                lt_expr=lhs(n);
                rone=getNEs(rt_expr);
                lone=getNEs(lt_expr);
                x = if(operator(n)=='=')
                    then
                            if(isLive(lt_expr,L)==true)
                            then
                                 (L - lone) + rone
                            else
                                  L
                            endif
                   else
                                (L + lone) + rone
                   endif;
        in
```

```
                x
else
        L
endif;

BackwardNodeflow( n: Unary,R: Rec, L:Liv )
if(check_datatype(n)==true)
then
        let
             operands_ne=getNEs(n);
        in
             L+operands_ne
else
        L
endif;

BackwardNodeflow( n: Call, R: Rec, L:Liv )
if(Check_datatype_call(n)==true)
then
        let
             d = emptySet();
             UseInCall = getNEsFromCall(n);

        in
             d + L + UseInCall

else
        L
endif;

BackwardNodeflow( n: _,R: Rec, L:Liv)
        L;


BackwardEdgeflow(E: _,R: Rec, L:Liv)
        L;



ForwardNodeflow( n: Symbol_AST,R: Rec,L:Liv)
if(check_datatype(initExpr(n))==true)
then
        let
                ne=NE(n);
                exp=initExpr(n);
                check =CHECK(exp);
                line_no=line(exp);
                kill=HasElement(ne,R);
                def={[line_no,ne]};
                sur=(R-kill)+def;
                survive=removeNonLive(sur,L);
        in
                survive
else
```

```
        R
endif;



ForwardNodeflow(n: Binary,R: Rec,L:Liv)
if(check_datatype_lhs(lhs(n))==true &&
                      check_datatype(rhs(n))==true)
then
        let
                check =CHECK(n);
                rt_expr=rhs(n);
                l_exp=lhs(n);
                ne=NE(n);
                line_no=line(l_exp);
                kill=HasElement(ne,R);
                def={[line_no,ne]};
                sur=(R-kill)+def;
                survive=if(operator(n)=='=')
                        then
                               removeNonLive(sur,L)
                        else
                               removeNonLive(R,L)

                        endif;
        in
                survive
else
        S
endif;




ForwardNodeflow( n: Unary,R: Rec,L:Liv)
if(check_datatype(n)==true)
then
        let
                check =CHECK(n);
                ne=NE(n);
                line_no=line(n);
                kill=HasElement(ne,R);
                def={[line_no,ne]};
                sur=(R-kill)+def;
                survive=removeNonLive(sur,L);
        in
                survive
else
        R
endif;

ForwardNodeflow( n: _,R: Rec,L:Liv)
        let
```

```
                survive=removeNonLive(R,L);
        in
                survive;

ForwardEdgeflow(E: _,R: Rec,L:Liv)
        let
                survive=removeNonLive(R,L);
        in
                survive;

Function res Meet ( A: res, B: res)
let
        diff=A+B;

in
        diff;
```

D.3 Liveness-based Inter-procedural Reaching definition analysis

Following is the specification for Liveness-based Inter-procedural Reaching definition analysis.

```
package darpan.klgLib;

res :: set NamedEntity,
tup :: tuple(int,NamedEntity,NamedEntity),
tup1 :: tuple(NamedEntity,NamedEntity),
pntr_info :: set tup1,
pointstolattice :: set tup;


fn_use darpan.klgLib;
fn_decl
pointstolattice Meet (pointstolattice, pointstolattice),
pointstolattice cartProduct ( set NamedEntity, set NamedEntity);

java_use darpan.klgLib.Aux;
java_decl
set NamedEntity getNEs(Expr),
set NamedEntity getNEsFromCall(pntr_info,Expr),
set NamedEntity getNEsFromCall(Expr),
set NamedEntity getNEsFromReturn(ASTnode),
boolean isLive(Expr,set NamedEntity),
set NamedEntity aliasClosure ( Expr, IRObject),
boolean isDref(Expr),
pointstolattice HasElement(NamedEntity,pointstolattice),
set NamedEntity removeIndirection@darpan.klgLib.AnalysisAPI(Expr, pointstolattice) ,
boolean isMust(set NamedEntity),
set NamedEntity getPtsto(pntr_info,Expr),
pntr_info getPointesOf(pntr_info,set NamedEntity),
set NamedEntity dref(pointstolattice,Expr),
set NamedEntity getLhsSet(pntr_info,Expr),
set NamedEntity getKillCandidate(pntr_info,res,Expr),
NamedEntity getIllegalNE(),
pntr_info removeNonLive(set NamedEntity, pntr_info),
set tup cst(set tup),
set NamedEntity emptySet(),
set NamedEntity useInLhsDref(pntr_info,Expr),
set NamedEntity useInRhs(pntr_info,Expr,boolean),
pointstolattice getSurvive(pntr_info,pointstolattice),
boolean check_datatype_lhs(Expr),
boolean check_datatype_lne(NamedEntity),
boolean check_datatype(Expr),
boolean check_datatype_new(NamedEntity),
NamedEntity getN(Expr),
int Line(Expr),
int Line(ASTnode),
NamedEntity getRvalue(Expr),
NamedEntity pointsToInfo(NamedEntity,pntr_info),
boolean isEqualNe(NamedEntity, NamedEntity),
boolean isStrongUpdate(Expr);

[FLA];

Forwardlattice L1 ::  pointstolattice;
Backwardlattice L2 ::  res;

sLiveness_FG implements sLiveness_FG:

ForwardTop : (pointstolattice){};
BackwardTop : (res){};

A ForwardMeet B : A+B;
A BackwardMeet B : A+B;

ForwardBoundaryValue: (pointstolattice){};
BackwardBoundaryValue : (res){};

BackwardNodeflow( n: Unary, S: L1, S1: L2 )
        let
                pntr=<<a :(pntr_info) {} ; x <- S;
                             a + {[atIndex(x,1), atIndex(x,2)]}>>; //extracting
                                                                  pointer information

                useinrhs = if(check_datatype(n) == false) // if operand
                                                         is a pointer
                           then
                               useInRhs(pntr,n,true) // get the pointees
                           else
$$                             getNEs(n)
                           endif;

        in
                useinrhs + S1;
```

```
BackwardNodeflow( n: Binary, S: L1, S1: L2 )
let
        opr = operator(n);
        d = emptySet();
        lt_expr = lhs(n);
        rt_expr = rhs(n);
        lNE = getNEs(lt_expr);
        rNE = getNEs(rt_expr);
        ss = d + S1;
        pntr=<<a :(pntr_info) {} ; x <- S;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        chck_livness = isLive(lt_expr,S1); // checking the liveness
                                              of lhs operands
        useinlhs = if(!check_datatype_lhs(lt_expr))
                   then
                        if ( ( opr == '=' ))
                        then
                                useInLhsDref(pntr,lt_expr)
                        else
                                useInRhs(pntr,n,chck_livness)
                        endif
                   else
                        if ( ( opr == '=' ))
                        then
                                (set NamedEntity){}
                        else
                                lNE
                        endif
                   endif;
        useinrhs = if(!check_datatype(rt_expr))
                   then
                        useInRhs(pntr,rt_expr,chck_livness)
                   else
                        if(operator(n) == '=')
                        then
                                if(isLive(lt_expr,S1) == true)
                                then
                                        rNE
                                else
                                        (set NamedEntity){}
                                endif
                        else
                                rNE
                        endif
                   endif;

        killcandidate = if(!check_datatype_lhs(lt_expr))
                       then
                                if ( ( opr == '=' ))
                                then
                                        getKillCandidate(pntr,S1,lt_expr)
                                else
                                        (set NamedEntity){}
                                endif
                        else
                                if(operator(n) == '=')
                                then
                                        lNE
                                else
                                        (set NamedEntity){}
                                endif
                        endif;
        gen = useinlhs + useinrhs + ss;
        sur = gen - killcandidate;
in
        sur;

BackwardNodeflow( n: Returnstmt, S: L1, S1: L2 )
let
        re = retExpr(n);
        pntr = <<a :(pntr_info) {} ; x <- S;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        s = useInRhs(pntr,re,true);
in
        S1 + s;

BackwardNodeflow( n: Call, S: L1, S1: L2 )
let
        d = emptySet();
        pntr = <<a :(pntr_info) {} ; x <- S;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        nes = if(check_datatype(n) == false)
              then
                getNEsFromCall(pntr,n)
              else
                getNEsFromCall(n)
              endif;
in
        d + nes + S1;


BackwardNodeflow( n: _, S: L1 ,S1: L2)
        S1;

BackwardEdgeflow(E: _, S: L1 ,S1: L2)
        S1;

ForwardNodeflow( n: Unary, S: L1, S1: L2 )
let
        line = Line(n);
        pntr = <<a :(pntr_info) {} ; x <- S;
                       a + {[atIndex(x,1), atIndex(x,2)]}>>;

        useinrhs = if(check_datatype(n) == false)
                   then
                         useInRhs(pntr,n,true)
                   else
                         getNEs(n)
                   endif;
        def = <<a :(pointstolattice) {} ; x <- useinrhs;
                a+
                    if(!isIllegalNE(x))
                    then
                         {[line,x,pointsToInfo(x,pntr)]}
                    else
                         {}
                    endif
                        >>;
        ne = NE(n);
        kill = HasElement(ne,S);
in
        (S - kill) + def;

ForwardNodeflow( n: Function_AST, S: L1, S1: L2 )
let
        para = parameterAST(n);
        line = Line(n);
        total_def = << a: (pointstolattice) {} ; x <- para;
                    a+
                        if(!isIllegalNE(x))
                        then
                            {[line,x,getIllegalNE()]}
                        else
                            {}
                        endif >>;
        def = <<a :(pntr_info) {} ; x <- total_def;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        kill = << a : (pointstolattice)  {} ; x <- S;
                 a+
                        << b : (pointstolattice) {} ; y <- total_def;
                                if(isEqualNe(atIndex(x,1),atIndex(y,1)))
                                then
                                        {x}
                                else
                                        {}
                                endif
                        >>

                >>;
        surv = removeNonLive(S1,def);
        survive = getSurvive(surv,total_def);
in
        survive;

ForwardNodeflow( n: Symbol_AST, S: L1, S1: L2 )
let
        exp = initExpr(n);
        ne = NE(n);
        line = Line(exp);
        kill = HasElement(ne,S);
        def = if(!check_datatype_new(ne))
             then
```

```
                {[line,ne,getRvalue(exp)]}
             else
                {[line,ne,getIllegalNE()]}
             endif;
        su = (S - kill) + def;
        pntr = <<a :(pntr_info) {} ; x <- su;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        surv = removeNonLive(S1,pntr);
        survive = getSurvive(surv,su);
in
        survive;

ForwardNodeflow( n: Binary, S: L1, S1: L2 )
let
        lt_expr = lhs(n);
        rt_expr = rhs(n);
        line = Line(lt_expr);
        ne = NE(n);
        pntr_info1 = <<a :(pntr_info) {} ; x <- S;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        rne = if(!check_datatype_lhs(rhs(n)))
              then
                getPtsto(pntr_info1,rt_expr)
              else
                getNEs(rt_expr)
              endif;
        lne = if(!check_datatype_lhs(lhs(n)))
             then
                 getLhsSet(pntr_info1,lt_expr)
             else
                getNEs(lt_expr)
             endif;
        rne_new = if(rne == emptySet())
                then
                        {getIllegalNE()}
                else
                        rne
                endif;
        gn = if(operator(n) == ’=’)
            then
                << a: (pointstolattice) {} ; x <- lne;
                    a+
                        if(check_datatype_lne(x))
                        then
                                << b: (pointstolattice) {} ; t <- rne_new;
                                   b+
                                        if(!isIllegalNE(x))
                                        then
                                                {[line,x,t]}
                                        else
                                                {}
                                        endif
                                >>
                         else
```

```
                                {[line,x,getIllegalNE()]}
                        endif
                >>
            else
                (pointstolattice) {}
            endif;
        kl = if(!check_datatype_lhs(lhs(n)))
             then
                if((isMust(lne)))
                then
                        if(isStrongUpdate(lt_expr))
                        then
                                getPointesOf(pntr_info1,lne)
                        else
                                (pntr_info){}
                        endif

                else
                        (pntr_info){}
                endif
              else
                        (pntr_info){}
              endif;
        kill = if(!check_datatype_lhs(lhs(n)))
             then
                 << a: (pointstolattice) {} ; x <- kl;
                    a+
                           << b: (pointstolattice) {} ; t <- S;
                                   b+
                                        if(atIndex(x,0)==atIndex(t,1))
                                        then
                                                {t}
                                        else
                                               (pointstolattice){}
                                        endif
                           >>
                >>
            else
                HasElement(ne,S)
            endif;
        su = (S - kill) + gn;
        temp = emptySet();
        liveset = temp + S1;
        pntr = <<a :(pntr_info) {} ; x <- su;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;

        surv1 = removeNonLive(liveset,pntr);
        surv = removeNonLive(liveset,surv1);
        survive=getSurvive(surv,su);

in
        survive;

ForwardNodeflow( n: _, S: L1 ,S1: L2)
```

```
let
        temp = emptySet();
        liveset = temp + S1;
        pntr = <<a :(pntr_info) {} ; x <- S;
                     a + {[atIndex(x,1), atIndex(x,2)]}>>;
        surv = removeNonLive(liveset,pntr);
        survive = << a: (pointstolattice) {} ; x <- surv;
                    a+
                          << b: (pointstolattice) {} ; t <- S;
                                  b+
                                       if(atIndex(x,0)==atIndex(t,1))
                                       then
                                               {t}
                                       else
                                              (pointstolattice){}
                                       endif
                          >>
                >>;

in
        survive;

ForwardEdgeflow(E: _, S: L1 ,S1: L2)
        S;
```