M. Tech Project Stage-I 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 very cost inefficient. So incremental data flow analysis is used which modifies only those data flow information of a program that has been changed, rather than recomputing entire data flow information of a program. The incremental analysis methods are known for bit-vector framework. These methods are not directly applicable to constant propagation. This report presents some issues in doing incremental analysis in Constant Propagation.

This report also discribes PRISM, an analyzer generator developed at Tata Research Development and Design Centre (TRDDC). We plan to enhance PRISM to support incremental data flow analysis.

Contents

List of Tables

List of Figures

Chapter 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 [9] 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 both the tables (Table 1.1(a) and (b)) are identical. Therefore, it is not desirable to recompute the information from the scratch, since it may unnecessarily analyze unaffected program behaviours, leads to redundant computation of old values which is very inefficient. Therefore, the data flow information to incorporate the effect of changes by repeating an exhaustive analysis can be very cost inefficient. So incremental data flow analysis is used, which modifies only the data flow information of a program that has been affected by the change rather than recomputing entire data flow information. It is more cost-effective than an exhaustive analysis.

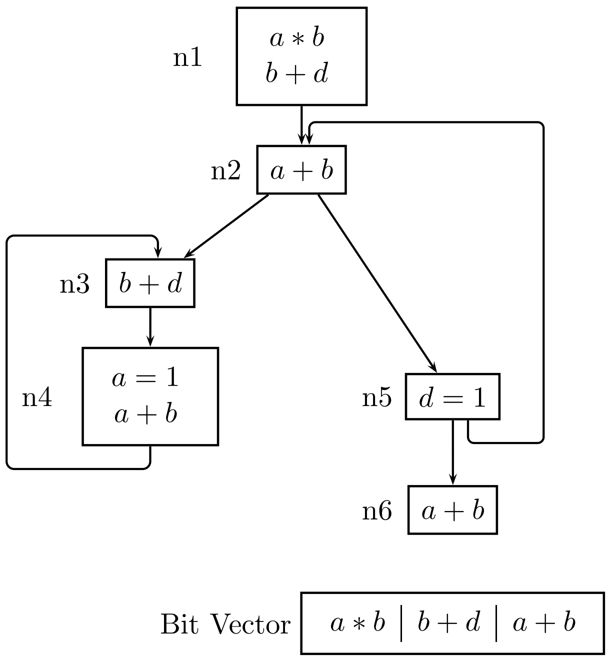


Figure 1.1: Control Flow Graph.

	*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)

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

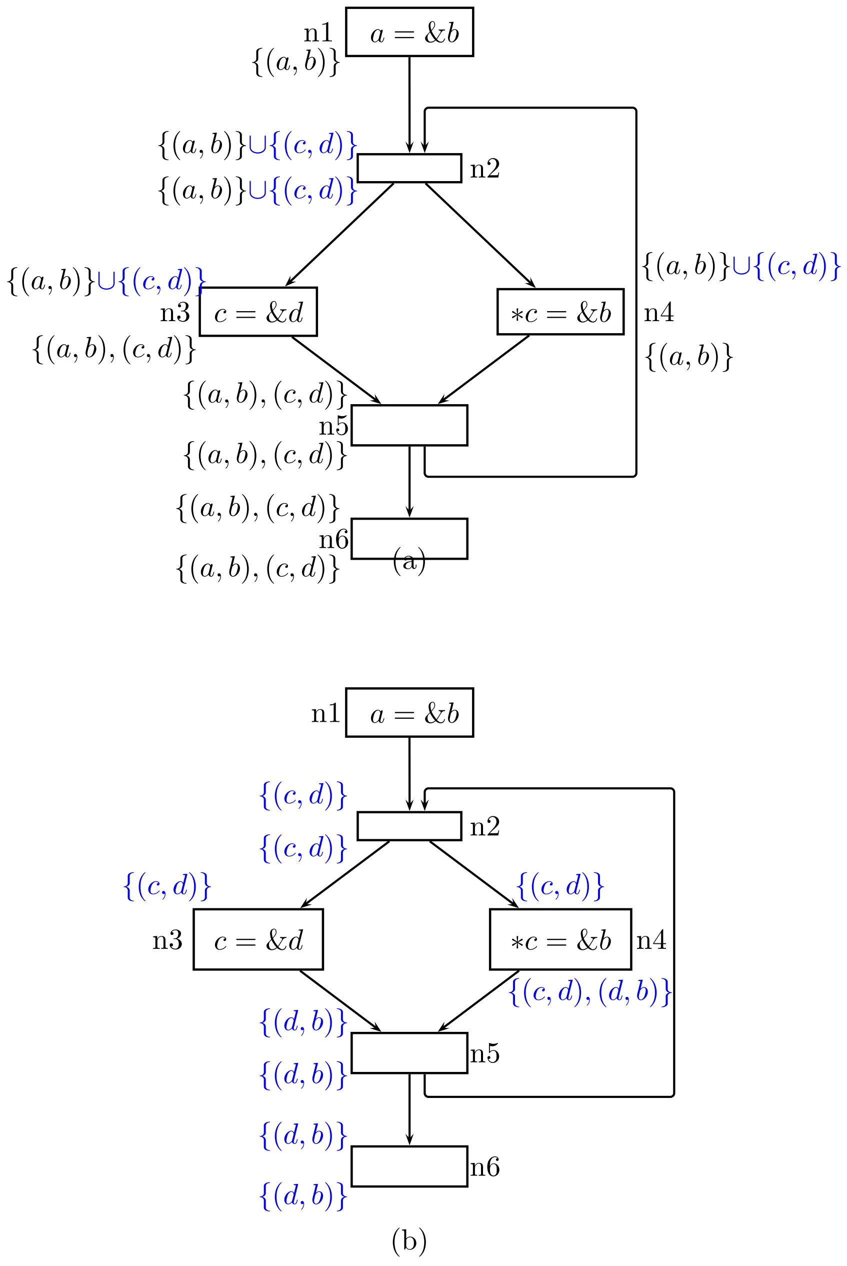


Figure 1.2: Motivating example for incremental points-to analysis across different iterations. In order to show the increments in the sets, we have used $\cup$ instead of writing the values after performing set union.

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 with each iteration computing only the new values not computed before, or modifying the values that needed to change. This eliminates redundant computation of old values.

Consider points-to analysis in fig 1.2. After first iteration points-to pair (c,d) is added to the pointer set at the IN of node $n2$. At node $n4$, we modify only the pointees of c and hence, at the IN of node $n4$ we require only pointer information of c in order to generate new points-to pairs. After first iteration, computation of pairs (a, b) at IN of node $n4$ is redundant. The data flow values generated in an iteration such that they were not present in the previous 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 scope of doing constant propagation. After first pass of available expression analysis, at the OUT of $n2$ expressions $x + 1$ and $a + b$ are available. The value of x is 10 at the OUT of $n1$, 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 value of variable y in print statement at node $n3$. Now if we further perform dead code elimination on the modified graph then the dead statements at node $n1$ and $n2$ will be removed as shown in Fig 1.3(c) and 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.

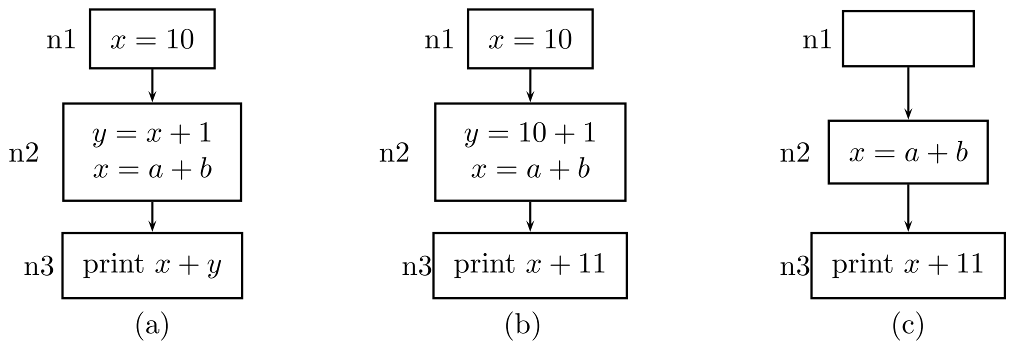


Figure 1.3: Incremental analysis across different applications.

1.1 PRISM Framework

PRISM is a program analyzer generator developed by Tata Research Development and Design Center (TRDDC). PRISM provides a platfrom that is being used for generating various program analysis tools for documentation, reverse engineering, domain specific model checking etc. Various data flow analyses have been implemented using PRISM, such as Strongly Live-variable analysis, LFCPA, Copy Constant propagation, Liveness analysis etc.

PRISM has two basic components, Kulang compiler and Solver. Kulang compiler parses analyzer specifications and generates some java classes. These classes are used by solver to solve data flow problem. It also uses many other packages which provide many utility functions.

Figure 1.4 [3] shows the architecture of PRISM. Source code is passed to the **IR Generator** which generates a language independent IR, which is then fed to **Generated analyzer**. Kulang compiler takes "**.klg**" file as an input and generates **java** classes. Then **Generated analyzer** takes IR as an input, and produces desired output. More details are presented in Chapter 3.

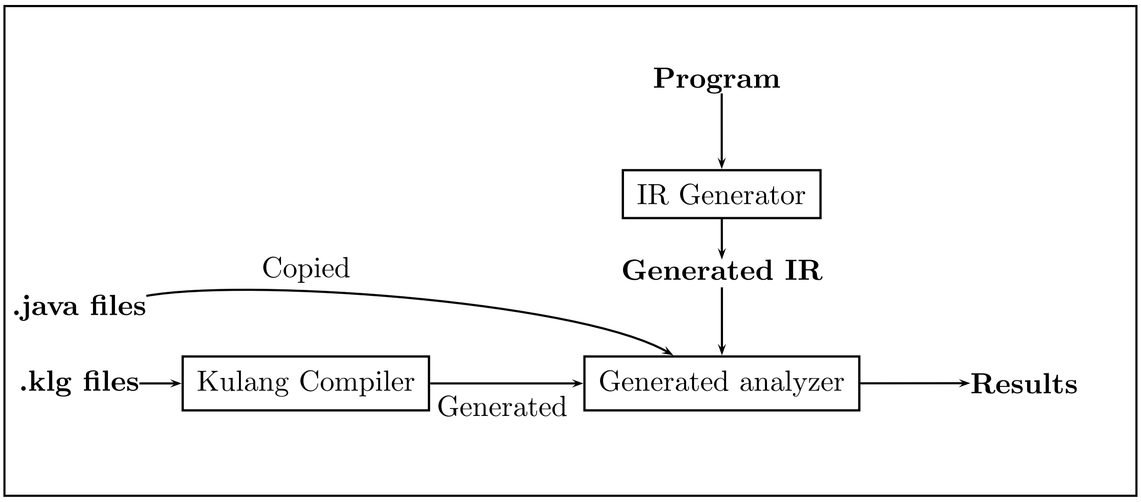


Figure 1.4: Architecture of PRISM

1.2 Scope of the project

This project aims at providing support for incremental data flow analysis in PRISM. This involves changing the solver. Current version of PRISM performs context sensitive analysis. It can also perform both unidirectional and bidirectional analysis. But it does not perform incremental data flow analysis.

This report presents some issues in implementing incremental data flow analysis for Constant propagation analysis. This report also discribes about the implemented Liveness-based Intra-procedural Reaching Definition Analysis in PRISM.

1.3 Organization of the report

The report is organized as follows. Chapter 2 describes i ncremental analysis and issues in doing incremental analysis in Constant Propagation analysis. Chapter 3 focuses on overview of PRISM. Chapter 4 describes our implementation of Liveness-based Intra-procedural Reaching Definition analysis in PRISM and performance measurements. Chapter 5 discusses plans for future 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 of data flow information computed earlier becomes invalid. Updating the 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 is used, which modifies only the data flow information of a program that has been affected by the change rather than recomputing entire data flow information of a program. It is more cost-effective than an exhaustive analysis.

When a program gets modified, the changes in the data flow information computed earlier 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 and 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 in Bit-vector framework

2.1.1 Flow functions in bit-bector framework

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

- Raise : Results 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 $\top$ ($\text{OUT}_n = \text{GEN} \cup (\text{IN}_n - \text{KILL})$). Flow function at n2 is a Raise function whose result is always $\top$
- Lower : Results is always bot($\perp$). 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 n2, GEN is 0 and KILL is 0. Therefore, $OUT_2 = IN_2$. Flow function at n2 propagates the value at the IN to OUT of the node n2

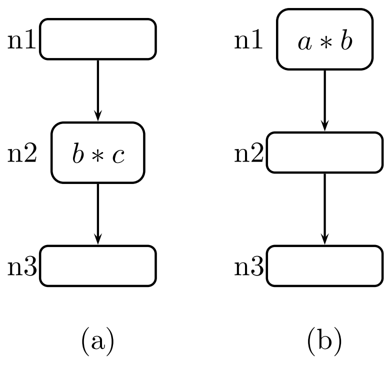


Figure 2.1: Flow functions in Bit-vector framework

In bit-vector framework [1], for a single entity 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 may remain same.

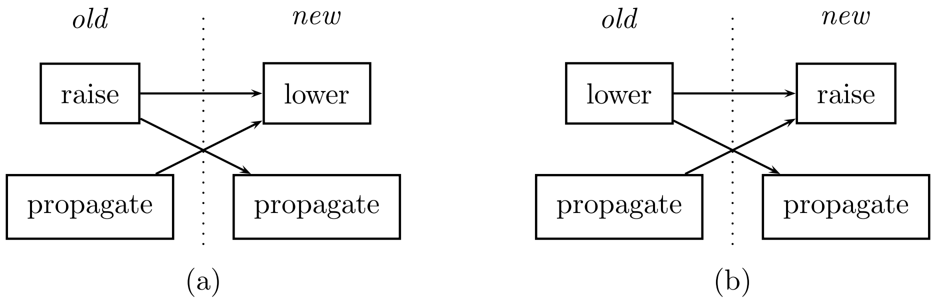


Figure 2.2: Flow functions in Bit-vector framework (a) Possible changes in flow function for top to bottom change. (b) Possible changes in flow functions for bottom to top change.

2.1.2 Handling Top to Bottom change

Since, $\forall x \in L : x \sqcap \top = x$, a top value for a data flow property is an intermediate value until the data flow analysis is completed. But in case of bottom, $\forall x \in L : x \sqcap \bot = \bot$,a bottom is a final value even during analysis. So whenever there is 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 Fig 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 $a * b$ 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 Fig 2.2) at the OUT of $n3$ which in turn makes bottom at the IN of $n2$ since $IN_2 = OUT_1 \cap OUT_3$, 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.

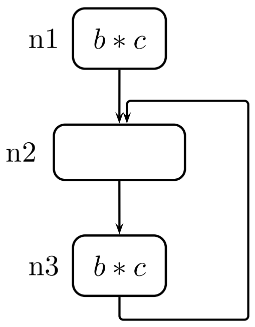


Figure 2.3: Top to Bottom change in control flow graph

2.1.3 Handling Bottom to Top change

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

Consider control flow graph in figure 2.3, if there is a bottom to top change(refer to Fig 2.2) at the OUT of $n3$. Then the value of IN_2 cannot be directly determined by OUT_3 alone since now it depends on the value of OUT_1 also. Thus, to incorporate 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.

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.

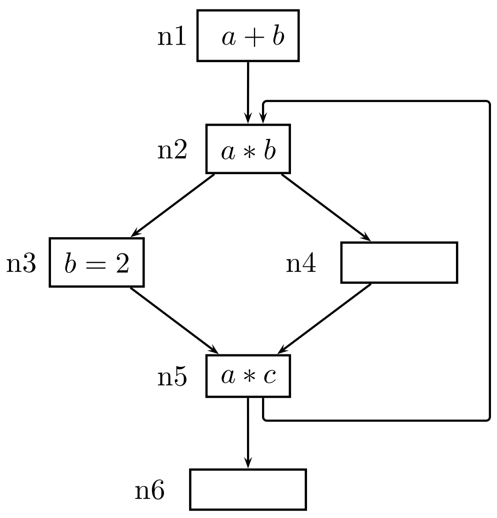


Figure 2.4: Control flow graph for available expression analysis

	$a+b$		$a*b$		$a*c$	
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

Table 2.1: Available expression analysis for Fig. 2.4

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 are 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.

	$a+b$		$a*b$		$a*c$	
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		

Table 2.2: The data flow values which may become 1.

- From the data flow values shown in Table 2.2, the data flow vales 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 the 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 is shown in Table 2.3

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

Table 2.3: The data flow values which must remain 0.

Thus, the final solution is shown in Table 2.4

	$a + b$		$a * b$		$a * c$	
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

Table 2.4: Final updated information.

2.1.4 Node listing as affected region

A *node listing* [6] 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

$$l = (n_1, n_2, ...n_m) \qquad (2.1)$$

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

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.

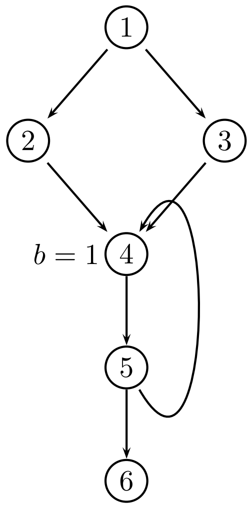


Figure 2.5: Control flow graph for *node listing*

2.1.5 Modelling Control Flow Changes

Control flow graph will change due to the following:

- Adding a node
- Deleting a node
- Adding an edge
- Deleting an edge

For adding a node/edge, we need not create an affected region since the effect of adding a node/edge will always go down the graph. Otherwise, we need to create an affected region.

2.1.6 Validation

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

2.1.7 Complexity

If n is the total number of nodes in a graph then in the worst case the size of the affected region is equal to n. If there are m nodes in an affected region then in the worst case it requires *O(m)* time to identify the boundary nodes since we need to check each node in an affected region.

2.2 Other Approaches of Incremental Data Flow Analysis

One approach [7] 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 [8] is to first compare the old and new code version and generates a structural diff. This diff will gives information about which nodes and edges in the program's inter-procedural control-flow graph (ICFG) were added or removed. Then we will call all nodes that were added or removed as *changed* nodes and infer from all the changed the set of *affected* nodes. Affected nodes are all nodes which are reachable from changed nodes which is an over approximation. Then it follows a clear-and-propagate strategy: for each affected node it first clears the analysis information computed and then re-propagates the information from all the nodes predecessors. Basically affected region is created always.

2.3 Incremental Analysis in Constant Propagation analysis

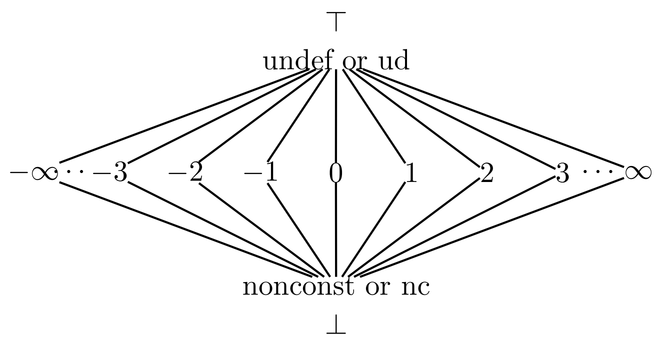


Figure 2.6: Component Lattice for Integer Constant Propagation

Consider a component lattice for Constant Propagation analysis in fig 2.6. Following are the possible flow functions in Constant Propagation analysis:

- Top : It is similar to the raise function in bit-vector framework. It always results in Top value.
- Bottom : It is similar to the lower function in bit-vector framework. It always results in Bottom value.
- Constant : Function whose result is always constant. Consider a constant propagation analysis in fig 2.7(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 2.7(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 then a will become top and a will become bottom if any of the operands is bottom. Flow functions for these type of statements whose value depends on the operands of the statement is known as side level function.

2.3.1 Changes in Constant Propagation analysis

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

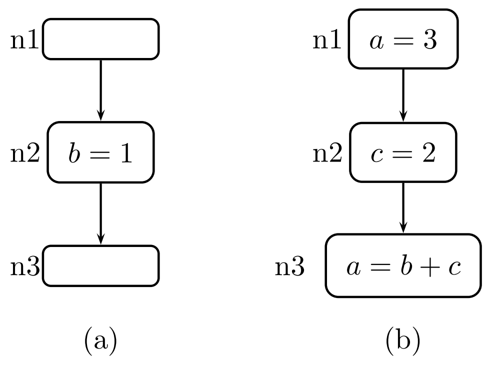


Figure 2.7: Flow functions in Constant Propagation analysis

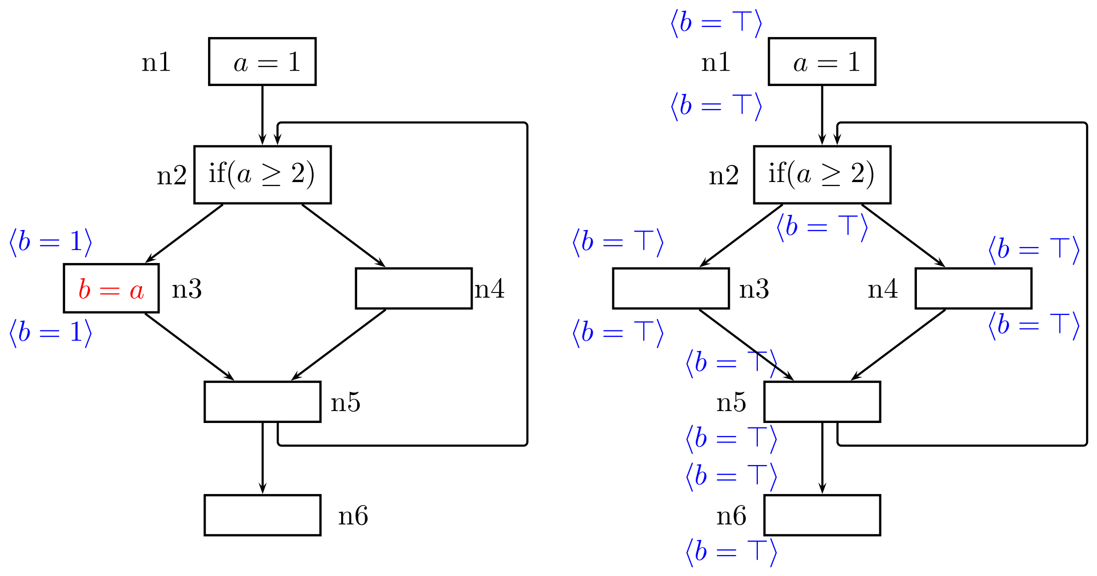
- a change to top,
- a change to bottom, and
- a side level change

Change from intermediate level to top

Consider a Constant propagation analysis in control flow graph shown in figure 2.8. In this case, if expression $b = a$ is removed from the node, then b will become $\top$ at the OUT of node $n3$ since the earlier value of b was 1, this a change to top.

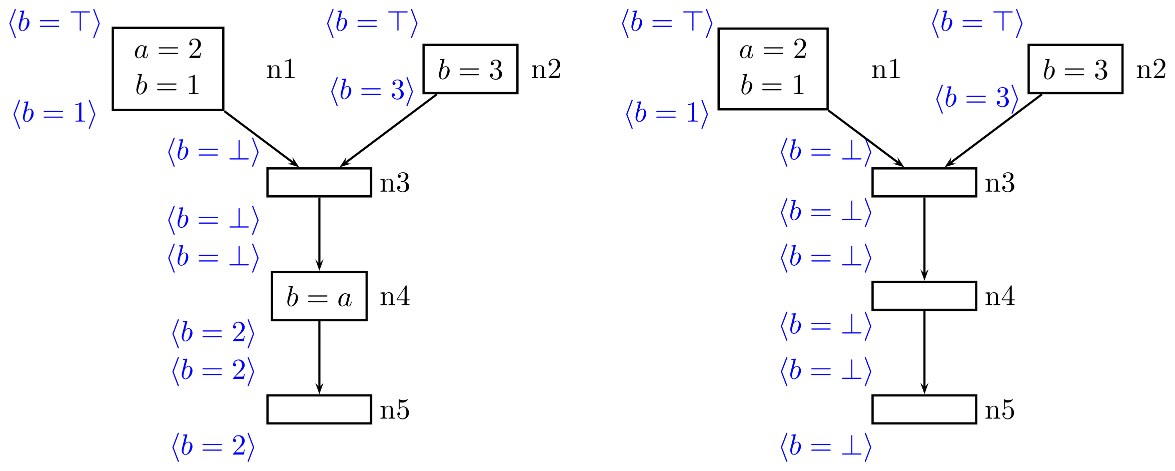
Change from intermediate level to bottom

Consider a control flow graph as shown in figure 2.9. After removal of expression $b = a$ at node $n5$, b becomes bottom($\perp$) at the OUT of node $n4$ (as shown in figure 2.9(b)) since the earlier value of b was 2 at OUT of $n4$, this is a change to bottom($\perp$).



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

Figure 2.8: Change from intermediate level to top



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

Figure 2.9: Change from intermediate level to bottom

Side level change

Consider a control flow graph as shown in figure 2.10. Suppose, expression $b = a$ at node $n3$ has been removed as shown in figure 2.10(b). Due to this change b becomes 2 at the OUT of node $n3$ since the earlier value of b was 1, this is a side level change in the lattice.

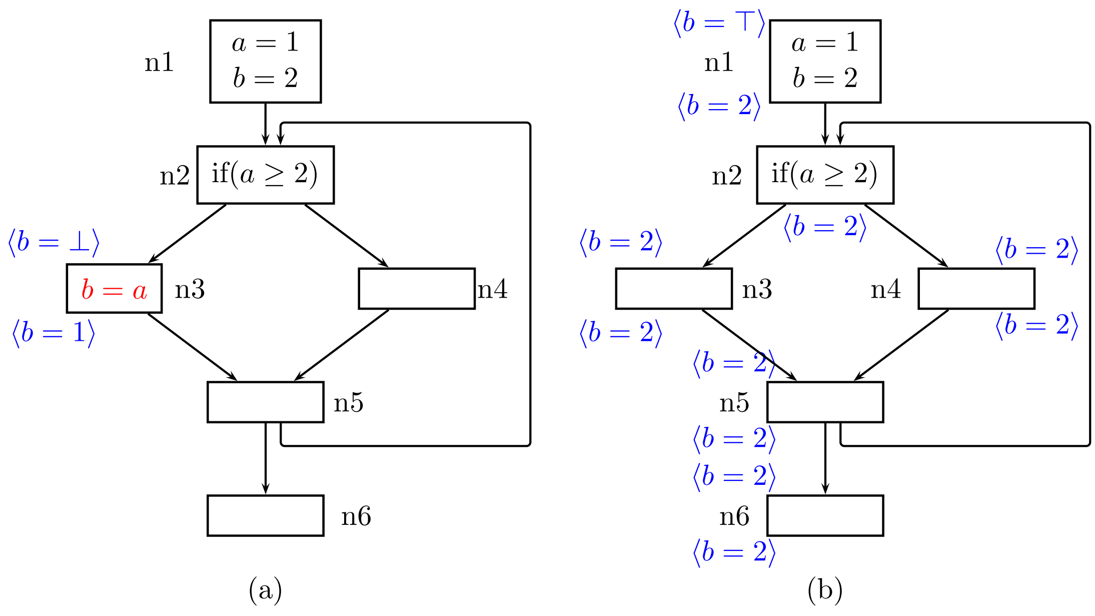


Figure 2.10: Side level change

2.4 Issues in Incremental Analysis for Constant Propagation

In Constant propagation analysis when there is a change to bottom(i.e undefined) due to a program change we need not to create an affected region, otherwise we need to create an affected region. Possible changes in the flow function when we need not create an affected region is shown in Figure 2.11.

Bottom function corresponds corresponds to reading values from input. Therefore, a change to bottom due to a program change is very infrequent and hence we may need to create an affected region for almost all the cases. Unlike bit-vector framework, bottom is also produced due to the meet operation. Therefore, it may be possible to restrict the size of affected region even if we are unable to avoid it.

Consider an example in Figure 2.12. Let at node n1 expression "$b = 1$" as in fig 2.12(a) is changed to "$b = 2$" due to a program change as shown in fig 2.12(b). Now at IN of node $n3$, two different values of b are reaching which makes b=$\perp$. In this case we can restrict the size of affected region. Therfore, we need some method to restrict ourself in creating an affected region.

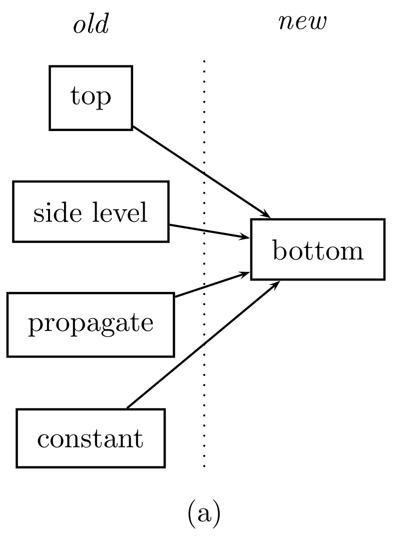


Figure 2.11: Possible changes in flow function when we need not create an affected region

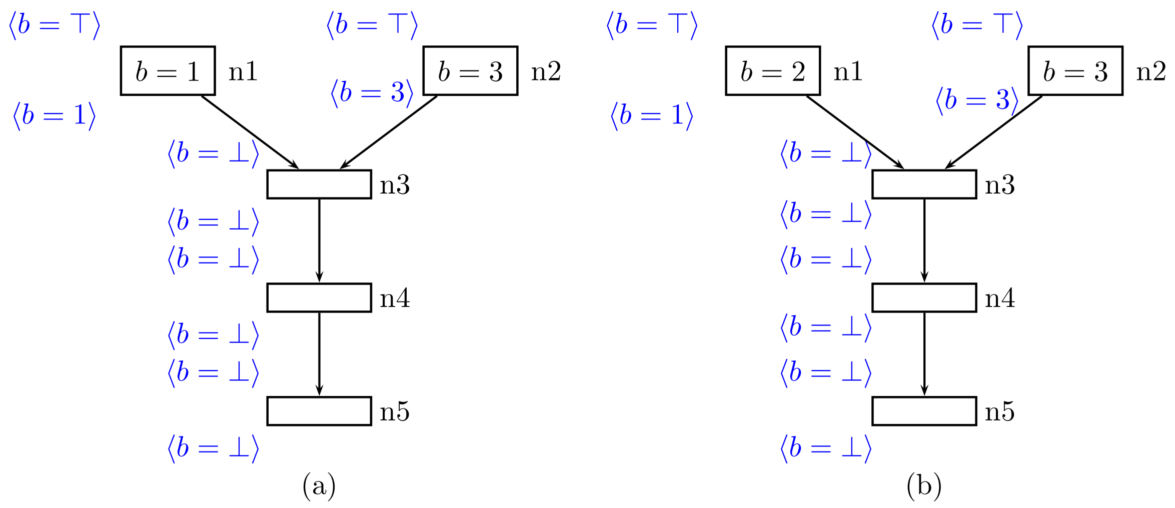


Figure 2.12: Change from intermediate level to bottom

2.5 Summary

When 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 very cost inefficient since it may compute redundant values. So incremental data flow analysis is used, which modifies only the data flow information of a program that has been affected by the change rather than recomputing entire data flow information of the program. It is more cost effective than an exhaustive analysis.

In Bit-vector framework, due to program change the data flow value may change either from top to bottom or bottom to top. Top to bottom change can be incorporated by directly propagating the values to its neighbors. But for bottom to top change, we need to create an affected region to incorporate the change to its neighbors.

In Constant Propagation unlike bit-vector framework, we can avoid computing an affected region only when there is a change to bottom otherwise we need to create an affected region. But bottom change is very infrequent and hence we will end up in creating an affected region for almost all the cases. Bottom values is also produced by the meet function, so it may be possible to restrict the size of affected region even if we are unable to avoid it. We need some method to restrict the size of the affected region which will be a part of our future work.

The method described in this chapter performs incremental analysis for a single change in a program. In general multiple changes in a program is also possible, so we need a method which can also perform incremental analysis for multiple changes in a program.

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.

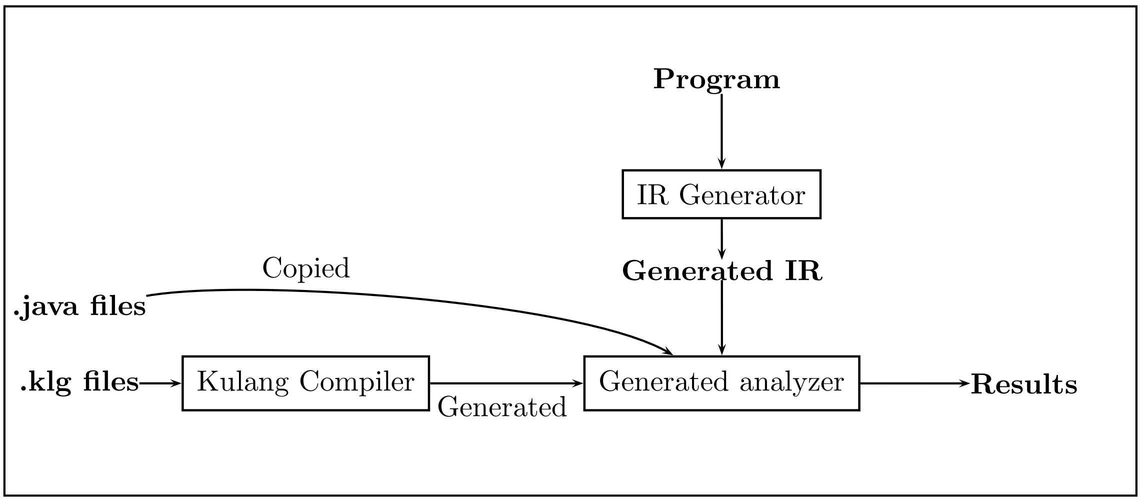


Figure 3.1: Architecture of PRISM

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 `Epxr` 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 $x \to y$ 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 $p \to q$ 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 [5]
- 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;
- *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 framework as input and generates a result.

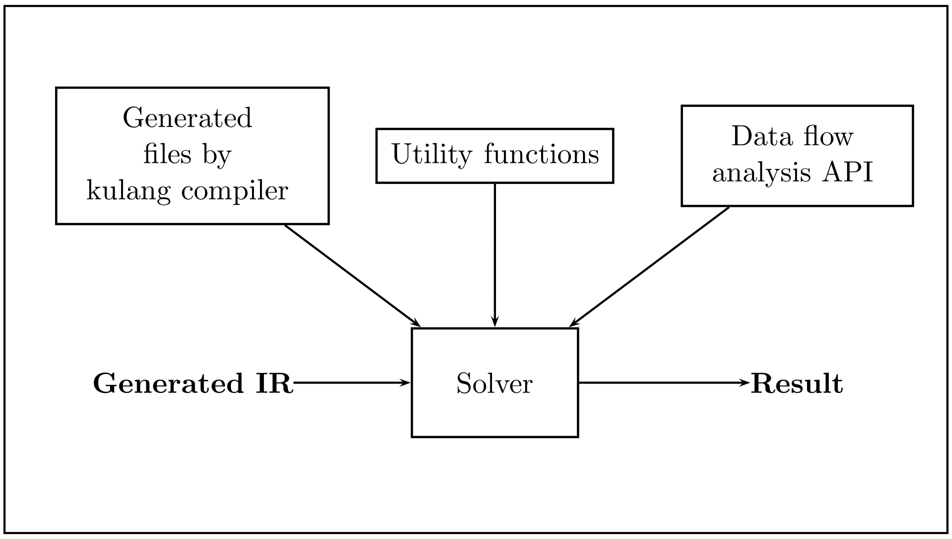


Figure 3.2: Architecture of analyzer generator

3.2.3 Running the generated analyser

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

3.2.4 Limitations in current version of PRISM

This section describes some imitations in current version of PRISM. Some of which we are trying to overcome in our implementation. Following are some of the limitations :

- It does not support Incremental data flow analysis
- Meet function needs to be explicitly specified in kulang specifications or defined as a java function. The meet function can be inferred from the lattice of the data flow problem
- There is no proper way to debug the kulang specifications
- Very hard to understand the specification language

Chapter 4

Liveness Based Reaching Definition Analysis using PRISM

In order to understand PRISM, we have implemented a query for Intra-procedural 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 C. This section describes flow functions for reaching definition analysis with and without liveness.

4.1 Intra-procedural 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 in 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 Intra-procedural reaching definition analysis

This section describes the flow functions for Liveness-Based Intra-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 $LOut_n$)

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

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.

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

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

Table 4.1: Benchmark results for normal reaching definition analysis.

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

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

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

Figure 4.1 shows a comparision between two analysis.

4.4 Summary

We have implemented a query for reaching definition analysis with and without liveness 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.

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. In general, we observe that anerage size of the set at each program point in liveness-based analysis is much smaller then that of the normal reaching definition analysis.

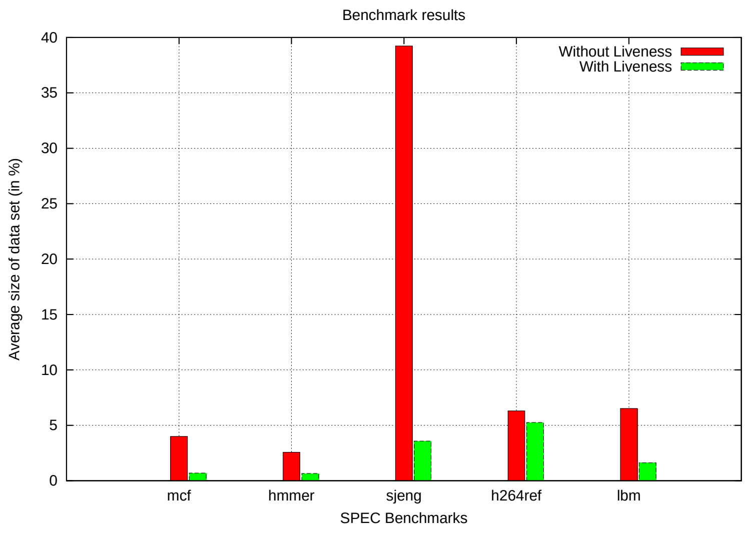


Figure 4.1: Percentage reduction in size of data set for Reaching definition analysis with and without liveness

Chapter 5

Future Work

This report describes the method of doing incremental analysis in bit-vector framework. This method performs incremental analysis for a single change in a program but in general multiple changes are also possible, so we need a method which can also perform incremental analsis for multiple changes. It also describes some issues in doing incremental analysis in Constant propagation. In Consant propagation we need to create affected region for almost all the cases. We may restrict the size of the affected region even if we cannot avoid it. We need some method so that we can restrict the size of the affected region which will be one of our future work.

Current PRISM framework supports context sensitive bi-directional analysis. Incremental analysis is not yet implemented in current framework. As a part of first stage of the project, queries for both reaching definition analysis with and without liveness have been implemented.

The current specification does not allow the user to specify data flow variables and equations as seen in literature. It places contraints on the type of analysis defined. It allows those analysis which has single data flow variable. Therefore, the anlysis writer needs to convert multi-variable equation to single-variable equation which is an extra work done by the writer. The specifications can be extended to allow analysis containig multiple variables.

Appendix A

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.

A.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.

A.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

A.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

Appendix B

PRISM APIs

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

B.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

B.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 C

Specifications of developed kulang queries

C.1 Intra-procedural 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 printSet(set NamedEntity),
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;
```

C.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 printSet(set NamedEntity),
boolean CHECK(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;
```

References

[1] Uday P. Khedkar. *A Generalised Theory of Bit Vector Data Flow Analysi.* 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] Pritam Gharat. *Interprocedural analysis in PRISM.* Mtech report (Stage 1), Computer Science and Engineering, IIT Bombay, 2012.

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

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

[7] 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.*

[8] 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.*

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