Extending GDFA to Nonseparable Framework

Sandeep Dasgupta[#1]

Computer Science And Engineering, IIT Kanpur
Kanpur, Uttar Pradesh, India

[1]dsand@cse.iitk.ac.in

***Abstract*— In this project we explore the possibility of extending *gdfa* to the data flow frameworks where data flow information can be represented using bit vectors but the frameworks are not bit vector frameworks because they are non-separable e.g., faint variable analysis, possibly undefined variables analysis, strongly live variable analysis. This require changing the local data flow analysis. And we considered statement as an independent basic block to achieve the same.**

***Keywords*— non-separable framework, bit vector framework, non-separability, faint variable analysis, possibly undefined variable analysis, impact chain, extending gdfa.**

I. Introduction

Gdfa uses a carefully chozen set of abstractions which makes it possible to execute a wide variety of specifications without having to know the name of the particular analysis being performed, provided the specifications are given within the limits of possible values of specification primitives. The source code of *gdfa* is distributed under GNU GPL v 2.0 or later. Currently gdfa supports bit vector framework where the dataflow values of different entities are independent. But there are analyses where data flow value of a given entity may depend on the data flow value of same entity or data flow value of some other entity. So there is a need to extend the existing framework.

II. NON-SEPARABLE FRAMEWORK

Separability is based on independence of data flow properties of entities for which data flow analysis is being performed. In order to model non-separable flow functions in terms of Gen and Kill components, instead of defining constant Gen and Kill we define dependent Gen and Kill as:

$$\text{Gen}_n(x) = \text{ConstGen}_n \cup \text{DepGen}_n(x) \quad (1)$$

$$\text{Kill}_n(x) = \text{ConstKill}_n \cup \text{DepKill}_n(x) \quad (2)$$

The flow function f_n is defined as:

$$f_n(x) = (x - \text{Kill}_n(x)) \cup \text{Gen}_n(x) \quad (3)$$

In bit vector frameworks, the dependent parts are absent resulting in constant Gen and Kill components. In non-separable frameworks, the dependence can be of two types: The data flow value of a given entity may depend on the data flow value of the same entity or on data flow value of some other entity. Our work focuses on the former case.

The presence of dependent parts in Gen and Kill makes it difficult to summarize the effect of multiple statements in a flow function. Hence, basic blocks for non-separable analyses consist of single statements. However, multiple consecutive statements which do not have any data dependence between them can still be combined into a basic block subject to the usual control flow restriction. If two consecutive statements can be executed in any order without affecting program semantics, then they can be grouped into the same basic block for data flow analysis of nonseparable flows. Further, a conditional or unconditional jump need not always be a separate block. If it is included in a block, it must be the last statement of the block.

The statements relevant to data flow analysis are divided in the following categories:

(a) assignment statements $x = e$ where $x \in$ Var, $e \in$ Expr,
(b) input statements *read*(x) which assign a new value to x,
(c) use statements *use*(x) which model uses of x for condition checking, printing and parameter passing etc., and
(d) other statements.

A. *Faint Variable Analysis*

A Variable x ∈ Var is faint at a program point u if along every path from u to End, it is either not used before being defined or is used to define a faint variable.

Clearly, this is a backward data flow problem. However, unlike liveness analysis this is an all-paths analysis.

B. *Possibly Uninitialized Variable Analysis*

A variable x ∈ Var is possibly uninitialized at a program point u if there exists a path from Start to u along which either no definition of the variable has been encountered or the definition uses a possibly uninitialized variable on the right hand side of the assignment.

Clearly this is a forward data flow problem.

III. CURRENT STATE OF GDFA

The following is the main data structure for specification:

```
struct gimple_pfbv_dfa_spec
{
        entity_name             entity;
        initial_value           top_value_spec;
        initial_value           entry_info;
        initial_value           exit_info;
        traversal_direction     traversal_order;
        meet_operation          confluence;
        entity_manipulation     gen_effect;
        entity_occurrence       gen_exposition;
        entity_manipulation     kill_effect;
        entity_occurrence       kill_exposition;
        dfi_to_be_preserved     preserved_dfi;

        dfvalue  (*forward_edge_flow)
                        (basic_block src, basic_block dest);
        dfvalue  (*backward_edge_flow)
                        (basic_block src, basic_block dest);
        dfvalue  (*forward_node_flow) (basic_block bb);
        dfvalue (*backward_node_flow)(basic_block bb);
};
```

By specifying concreate values to these abstractions gdfa specifies any analysis.

Also the above structure is used when the definition of Gen and Kill are constants. Hence this structure also need to be improved so as to tackle dependent parts of Gen and Kill in a non-separable framework.

IV. EXTENDING GDFA

We take the following actions to extend the existing architecture

A. *Basic Block Contains Single Statments*

1) *local dfa*: We calculated the const Gen and Kill of each statement.

2) *Global dfa:*

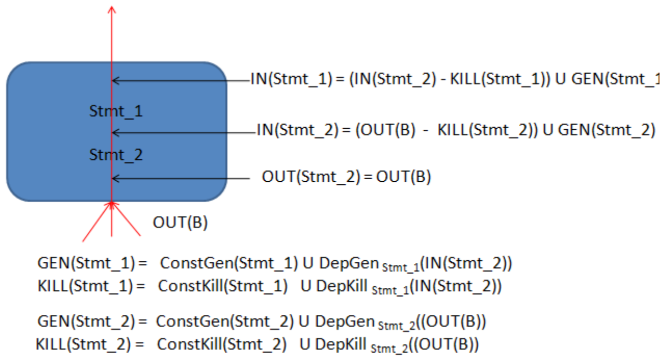
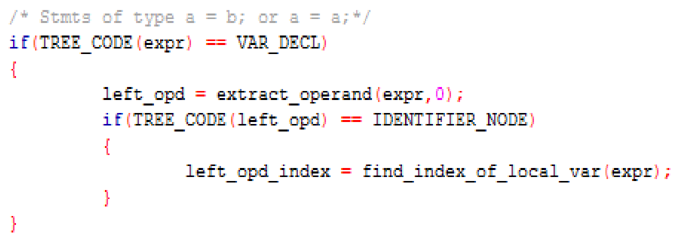

IN(Stmt_1) = (IN(Stmt_2) - KILL(Stmt_1)) U GEN(Stmt_1
Stmt_1
IN(Stmt_2) = (OUT(B) - KILL(Stmt_2)) U GEN(Stmt_2)
Stmt_2
OUT(Stmt_2) = OUT(B)
OUT(B)
GEN(Stmt_1) = ConstGen(Stmt_1) U DepGen Stmt_1(IN(Stmt_2))
KILL(Stmt_1) = ConstKill(Stmt_1) U DepKill Stmt_1(IN(Stmt_2))
GEN(Stmt_2) = ConstGen(Stmt_2) U DepGen Stmt_2((OUT(B))
KILL(Stmt_2) = ConstKill(Stmt_2) U DepKill Stmt_2((OUT(B))

Fig. 1 The figure shows how to calculate the global data flow value of statements within a basic block while doing the global data flow analysis of that basic block in a backward data flow problem.

As gdfa does global data flow analysis by iterating over each basic block in a specified sequence(backward or forward), so we do the global data flow analysis of the corresponding statements by computing the Dep Gen and Dep Kill at each statement. From Fig 1, it is clear that the OUT(stmt_2) = OUT(B) and IN(B) = IN(stmt_1).

B. *Including Other Statement Types*

1) *Copy statements*: Gdfa does not include copy statements of type (a = b or a = a) for data flow analysis. So we need to recognize and include that statement for consideration.



```
/* Stmts of type a = b; or a = a;*/
if(TREE_CODE(expr) == VAR_DECL)
{
        left_opd = extract_operand(expr,0);
        if(TREE_CODE(left_opd) == IDENTIFIER_NODE)
        {
                left_opd_index = find_index_of_local_var(expr);
        }
}
```

Fig. 2 The C-code showing how to recognize the copy statement and extract the operand from it.

2) *Assigning Indices to Statements*

All non-separable analysis work with individual statements and hence they need to be numbered uniquely. Now before invoking the gdfa driver, gdfa does this numbering for reaching definition analysis, but ignores the numbering of statements which are not of interest to that analysis, like condition checking statements.But we need to number those as well for non-separable analysis. So we numbered those statements keeping the numbering done for reaching definition analysis intact.

C. *Extending the Specification Structure*

The main data structure for specification is extended as follows:

```
struct gimple_pfbv_dfa_spec
{
        entity_name             entity;
        initial_value           top_value_spec;
        initial_value           entry_info;
        initial_value           exit_info;
        traversal_direction     traversal_order;
        meet_operation          confluence;
        entity_manipulation     gen_effect;
        entity_occurrence       gen_exposition;
        entity_manipulation     kill_effect;
        entity_occurrence       kill_exposition;
        dfi_to_be_preserved     preserved_dfi;

        dfvalue  (*forward_edge_flow)
                        (basic_block src, basic_block dest);
        dfvalue  (*backward_edge_flow)
                        (basic_block src, basic_block dest);
        dfvalue  (*forward_node_flow)   (basic_block bb);
        dfvalue  (*backward_node_flow)  (basic_block bb);
        /*@Newly Added Fields : START*/
        statement_type          constgen_statement_type;
        precondition            constgen_precondition;
        statement_type          constkill_statement_type;
        precondition            constkill_precondition;
        entity_dependence       dependent_gen;
        entity_dependence       dependent_kill;
        /*@Newly Added Fields : END*/
};

typedef enum statement_type {
         READ_X = 1,
         USE_X,
         IGNORE_STATEMENT_TYPE
         } statement_type;

typedef enum precondition {
         X_IN_OPERAND = 1,
         X_NOT_IN_OPERAND,              OPERAND_IS_
CONST, OPERAND_ISNOT_CONST,         IGNORE_PREC
ONDITION                          } precondition;

typedef enum entity_dependence{
         X_IN_GLOBAL_DATA_FLOW_VALUE=1,
         X_NOT_IN_GLOBAL_DATA_FLOW_VALUE,
         OPER_IN_GLOBAL_DATA_FLOW_VALUE,
OPER_NOT_IN_GLOBAL_DATA_FLOW_VALUE,
IGNORE_ENTITY_DEPENDENCE
         } entity_dependence;
```

We propose that with the above abstract structure specification of any non-separable analysis can be done.

D. *Specifying Faint Variable Analysis*

The constant and dependent parts of Gen and Kill for faint variable analysis are:

$$\mathit{ConstGen}_n = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ x \notin \mathit{Opd}(e) \\ \{x\} & n \text{ is } \mathit{read}(x) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{DepGen}_n(\mathsf{x}) = \emptyset$$

$$\mathit{ConstKill}_n = \begin{cases} \{x\} & n \text{ is } \mathit{use}(x) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{DepKill}_n(\mathsf{x}) = \begin{cases} \mathit{Opd}(e) \cap \mathbb{V}\mathrm{ar} & n \text{ is assignment } x = e,\ x \notin \mathsf{x} \\ \emptyset & \text{otherwise} \end{cases}$$

The above definitions are specified as follows:

```
   struct gimple_pfbv_dfa_spec gdfa_fv =
   {
        entity_var,             /* entity;          */
        ONES,                   /* top_value;       */
        ONES,                   /* entry_info;      */
        ONES,                   /* exit_info;       */
        BACKWARD,               /* traversal_order;*/
        INTERSECTION,           /* confluence;      */
        entity_mod,             /* gen_effect;      */
        up_exp,                 /* gen_exposition;*/
        entity_use,             /* kill_effect;     */
        any_where,              /* kill_exposition; */
        global_only,            /* preserved_dfi;   */
        stop_flow_along_edge,
        identity_backward_edge_flow
        stop_flow_along_node,
        backward_gen_kill_node_flow,
        READ_X,             /* constgen_statement_type */
        X_NOT_IN_OPERAND,
                            /* constgen_precondition   */
        USE_X,              /* constkill_statement_type*/
        IGNORE_PRECONDITION,
                            /* constkill_precondition  */
        IGNORE_ENTITY_DEPENDENCE,
                            /* dependent_gen           */
        X_NOT_IN_GLOBAL_DATA_FLOW_VALUE
                            /* dependent_kill          */
};
```

E. *Specifying Possibly Uninitialized Variable Analysis*

The constant and dependent parts of Gen and Kill for the variable analysis are:

$$\mathit{ConstGen}_n = \emptyset$$

$$\mathit{DepGen}_n(\mathsf{x}) = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ \mathit{Opd}(e) \cap \mathsf{x} \neq \emptyset\} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{ConstKill}_n = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ \mathit{Opd}(e) \subseteq \mathbb{C}\mathrm{onst} \\ \{x\} & n \text{ is } \mathit{read}(x) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{DepKill}_n(\mathsf{x}) = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ \mathit{Opd}(e) \cap \mathsf{x} = \emptyset\} \\ \emptyset & \text{otherwise} \end{cases}$$

The above definitions are specified as follows:

```
   struct gimple_pfbv_dfa_spec gdfa_puv =
   {
        entity_var,             /* entity;          */
        ZEROS,                  /* top_value;       */
        ONES,                   /* entry_info;      */
        ZEROS,                  /* exit_info;       */
        FORWARD,                /* traversal_order; */
        UNION,                  /* confluence;      */
        entity_mod,             /* gen_effect;      */
        up_exp,                 /* gen_exposition  */
        entity_mod,             /* kill_effect;     */
        any_where,              /* kill_exposition  */
        global_only,            /* preserved_dfi;   */
        identity_forward_edge_flow,
        stop_flow_along_edge,
        forward_gen_kill_node_flow,
        stop_flow_along_node,
        IGNORE_STATEMENT_TYPE,
                            /* constgen_statement_type */
        IGNORE_PRECONDITION,
                            /* constgen_precondition   */
       READ_X,              /* constkill_statement_type*/
       OPERAND_IS_CONST,
                            /* constkill_precondition  */
       OPER_IN_GLOBAL_DATA_FLOW_VALUE,
                            /* dependent_gen           */
       OPER_NOT_IN_GLOBAL_DATA_FLOW_VALUE
                            /* dependent_kill          */
   };
```

F. *Impact Chains*

The definition of Dep Gen and Dep Kill depicts the transistive effect between the entities. From the definition of Dep Gen the impact chain of variables is created, called Gen Impact Chain. Similarly, the Kill Impact Chain is created from the definition of Dep Kill. Also the chain are displayed using dot tool of graphViz software.

G. *Results*

Now for a given test case gdfa performs the faint variable and possibly uninitialized variable analysis and dump it in the file *.gdfa_fv and *.gdfa_puv respectively.
Also the current implementation generate gen_impact_chain.dot and kill_impact_chain.dot (in case any such impact chain forms during the analysis) so that Dot tool of graphViz software can generate impact chains in graphical format.

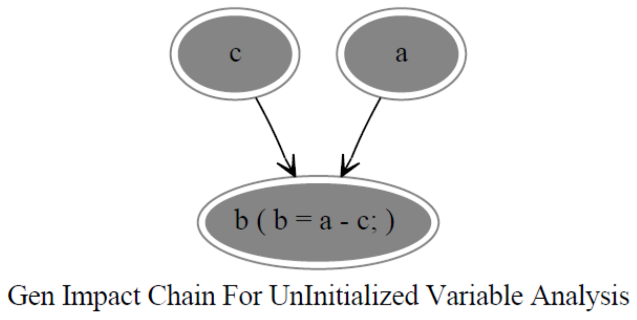


Fig. 3 An example Impact Chain for Possibly Uninitiazed variable analysis. The interpretation is as follows: “c impacts a through statement b = b-c”. It is called gen-impact chain because the impact is for generation of uninitialized variables, i.e. unintialized nature of c makes b uninitialized through statement b=a-c.

V. Conclusions

In this project we extended the architecture of gdfa to non-separable framework so as to support analyses like faint variable and possibly uninitialized variable analysis.

References

[1] Uday P.Khedker, Amitabha Sanyal, Bageshri Karkare, “Data Flow Analysis Theory And Practise”.