Functional Approach to Interprocedural DFA

M.Tech. R and D Report

by

Harshada A. Gune

07305904

under the guidance of

Prof. Uday Khedker

Dept. of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Contents

Abstract

A generic data flow analyzer (gdfa) 1.0 supports per function bit vector data flow analysis in gcc. This document describes two of the possible extensions to *gdfa 1.0*. One of them is per function data flow analysis using worklist method and other is extending intraprocedural data flow analysis to interprocedural data flow analysis using functional approach.

In this report, we explain the functional approach to interprocedural data flow analysis. We begin with an example of interprocedural data flow analysis and show the computation of data flow values using functional approach. In section 2, we describe how summary information can be calculated at per function level. Section 3 describes interprocedural driver written in gcc which extends per function summary information to interprocedural level. Theory of functional approach can be found at [2]. Most of the APIs from [1] are reused while extending gdfa with a functional approach.

1 Example Illustrating Functional Approach

In this section, we present live variable analysis using functional approach to interprocedural data flow analysis. The call-graph and CFGs of each function are given below:

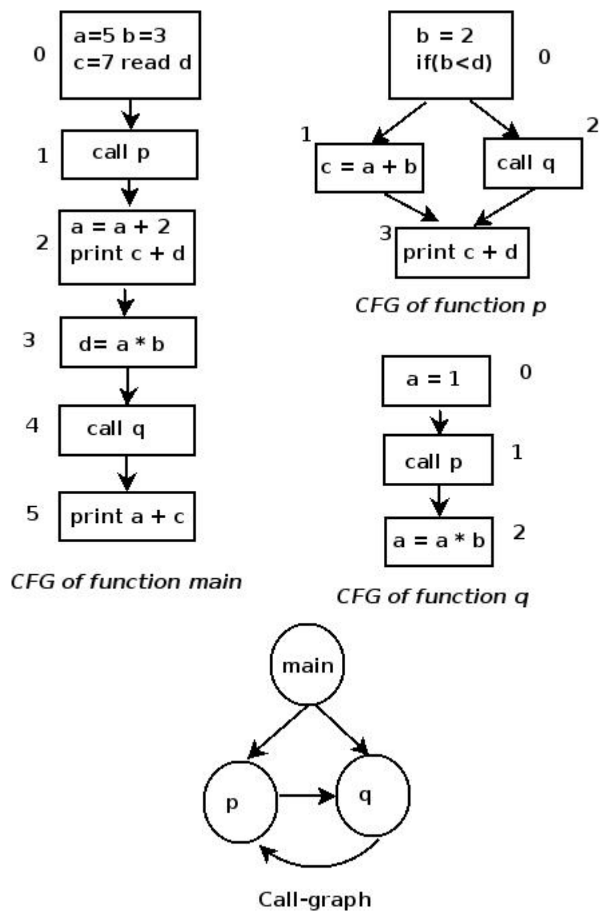

Figure 1: Example program for interprocedural live variable analysis.

1.1 Initializing Data Flow Values

Gen and kill values at each basic block of every function are initialized with gen-top and kill-top respectively. For live variable analysis, gen top is empty set, while kill top is universal set of entities. Summary gen and kill information of every function is also initialized with respective top values.

1.2 Using Worklists

The analysis is carried out using worklist method of data flow analysis. For this, we have maintained two types of worklists, one is called as outer worklist(OWL) and other is inner worklist(IWL). There is only one global outer worklist and its members are nothing but the functions in the call-graph. While there are as many number of inner worklists as number of functions, each worklist corresponding to unique function. Initialization of worklists is done as follows:

- Outer worklist is initialized with entry for every function present in the call-graph. The call-graph is traversed in DFS order and leaf nodes are added first to the worklist. This avoids the recomputing of data flow values of non-leaf nodes.
- Inner worklists of every function are initialized with basic blocks from respective CFGs. Here also CFGs are traversed in DFS order and basic blocks are added to worklists in descending order(forward data flow problems) or ascending order(backward data flow problems) of DFS numbers assigned to basic blocks. For our example, initialization of worklists is done as follows:

$$\begin{gathered}\text{Outer worklist: } \langle p, q, main\rangle \\ \text{IWL for p: } \langle 3,2,1,0\rangle \\ \text{IWL for q: } \langle 2,1,0\rangle \\ \text{IWL for main: } \langle 5,4,3,2,1,0\rangle\end{gathered}$$

Step by step computation of data flow information is shown in table?. The algorithm works as follows:

1. Take out the first function from outer worklist.
2. Process its inner worklist to compute its summary gen-kill information.
3. If there is change in previous summary values of this function, find out its caller functions and call nodes within them. Each such caller along with call nodes added to their respective inner worklists is added to outer worklist if not already present. If the caller is already present in outer worklist, but call nodes are not present in its inner worklist then inner worklist is updated by appending call nodes.

In our example, we can see that when p's computation is over, its summary values are changed. Callers of p are main(callnode: 1) and q(callnode: 1). Note that both the functions are already present in outer worklist and inner worklists of both of them are alredy having their respective call blocks. So no need to add any node to outer worklist. When computation of q is done, its summary values are also changed. Its callers are main(callnode: 4) and p(callnode: 2). Out of these, main is already present in outer worklist and inner worklist of main is having entry for basic blcok 4. Hence no need to add entry for main in outer worklist. But p is not present and hence new node of p added to outer worklist with inner worklist initiated with basic block 2. Again p is taken out and processed, but this time its summary doesn't change. Hence no addition to outer worklist.

4. Repeat these three steps until outer worklist is empty.

1.3 Per Function Data Flow Information

Second step in above algorithm is described in detail here. In the table 1, this step is distinguished by adding blank line between the rows. While calculating per function summary information, basic blocks of a given function are accessed using worklist method only. For this purpose, we have maintained per function worklists, that is, inner worklists. Summary gen and kill are computed as follows:

- Get the basic block at the head of an inner worklist.
- Compute gen and kill at OUT of this block using following equation:

$$\sqcap f_{n \to s}\,(GenIn_s)$$
$$\sqcap f_{n \to s}\,(KillIn_s)$$

- Compute the gen and kill at IN of this basic block using following equation:

$$(GenOUT(bb) - KILL(bb)) \cup GEN(bb)$$
$$KillOUT(bb) \cup KILL(bb)$$

  Here GenOUT and KillOut are gen and kill values at OUT of basic block bb. These equations are nothing but equations of function compositions. Here KILL(bb) and GEN(bb) are the local data flow values of basic block bb. In case of a basic block containing a call to some function, local data flow values of such a basic block are nothing but the summary values of the called function.

- If the newly computed IN values differ from previous values of gen and/or kill at IN, add predecessors of basic block bb to inner worklist, if not present already.
- Repeat above steps while inner worklist is not over.
- Finally when inner worklist gets over, Gen and Kill values at IN of a entry block are nothing but summary values for this function.

In our example, final summary values of all functions are given below. These values are gen and kill values at IN of entry block. In case of forward data flow problem, summary gen-kill values are nothing but gen and kill values at OUT of exit block.

Function	Summary Gen	Summary Kill
Main	$\emptyset$	$\{a, b, c, d\}$
p	$\{a, d\}$	$\{b, c\}$
q	$\{d\}$	$\{a, b, c\}$

2 Computing Per Function Summary Information

In this section, we present the function compute_pf_summarygenkill which is written to compute per function summary gen and kill information. As mentioned in section 1.3, this function uses worklist method to traverse the basic blocks of a function. Data structure used to save gen and kill information at each basic block is shown below:

```
-----------------------------------------------
typedef struct bvfunc_bb_genkill
{
        dfvalue gen_in;
        dfvalue kill_in;
        dfvalue gen_out;
        dfvalue kill_out;
}bvfunc_bb_genkill;
bvfunc_bb_genkill ** current_bvfunc_pf_genkill;
-----------------------------------------------
```

We present code frgament of compute_pf_summarygenkill to show the calculation of gen and kill when traversal direction is forward. If there is change in gen out or kill out of a given block, successors of a block are added to the worklist of a given function. Successors of a block are traversed using following code:

Table 1: Computation of summary data flow values of the program in Figure.1

Original WL		Propogation			Acc. DF Value		Updated WL	
Outer	Inner	Func:Block	Gen	Kill	Gen	Kill	Outer	Inner
$\langle p, q, main\rangle$	$\langle p : 3, 2, 1, 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	p:3	$\emptyset$	a,b,c,d	c,d	$\emptyset$	$\langle q, main\rangle$	$\langle p : 2, 1, 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle q, main\rangle$	$\langle p : 2, 1, 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	p:2	$\emptyset$	a,b,c,d	$\emptyset$	a,b,c,d	$\langle q, main\rangle$	$\langle p : 1, 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle q, main\rangle$	$\langle p : 1, 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	p:1	$\emptyset$	a,b,c,d	a,b,d	c	$\langle q, main\rangle$	$\langle p : 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle q, main\rangle$	$\langle p : 0\rangle$ $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	p:0	$\emptyset$	a,b,c,d	a,d	b,c	$\langle q, main\rangle$	$\langle p :\rangle$[a] $\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle q, main\rangle$	$\langle q : 2, 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	q:2	$\emptyset$	a,b,c,d	a,b	a	$\langle main\rangle$	$\langle q : 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle main\rangle$	$\langle q : 1, 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	q:1	$\emptyset$	a,b,c,d	a,d	a,b,c	$\langle main\rangle$	$\langle q : 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle main\rangle$	$\langle q : 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	q:0	$\emptyset$	a,b,c,d	d	a,b,c	$\langle main\rangle$	$\langle q :\rangle$[b] $\langle m : 5, 4, 3, 2, 1, 0\rangle$ $\langle p : 2\rangle$
$\langle p, main\rangle$	$\langle p : 2\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	p:2	$\emptyset$	a,b,c,d	d	a,b,c	$\langle main\rangle$	$\langle p : 0\rangle+$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle main\rangle$	$\langle p : 0\rangle$ $\langle m : 5, 4, 3, 2, 1, 0\rangle$	p:0	a,d	b,c	a,d	b,c	$\langle main\rangle$	$\langle p :\rangle$[c] value $\langle m : 5, 4, 3, 2, 1, 0\rangle$
$\langle main\rangle$	$\langle m : 5, 4, 3, 2, 1, 0\rangle$	m:5	$\emptyset$	a,b,c,d	a,c	$\emptyset$	$\langle\rangle$	$\langle m : 4, 3, 2, 1, 0\rangle$
$\langle\rangle$	$\langle m : 4, 3, 2, 1, 0\rangle$	m:4	$\emptyset$	a,b,c,d	d	a,b,c	$\langle\rangle$	$\langle m : 3, 2, 1, 0\rangle$
$\langle\rangle$	$\langle m : 3, 2, 1, 0\rangle$	m:3	$\emptyset$	a,b,c,d	a,b	a,b,c,d	$\langle\rangle$	$\langle m : 2, 1, 0\rangle$
$\langle\rangle$	$\langle m : 2, 1, 0\rangle$	m:2	$\emptyset$	a,b,c,d	a,b,c,d	a,b,c,d	$\langle\rangle$	$\langle m : 1, 0\rangle$
$\langle\rangle$	$\langle m : 1, 0\rangle$	m:1	$\emptyset$	a,b,c,d	ad	a,b,c,d	$\langle\rangle$	$\langle m : 0\rangle$
$\langle\rangle$	$\langle m : 0\rangle$	m:0	$\emptyset$	a,b,c,d	$\emptyset$	a,b,c,d	$\langle\rangle$	$\langle m :\rangle$[d]
$\langle\rangle$	$\langle\rangle$						$\langle\rangle$	$\langle\rangle$

[a] p's IWL is over here and change in p's summary values. But callers of p's are already in OWL with their all blocks already present in their respective IWLs

[b] q's IWL is over here and change in q's summary values. Callers of q's are p with bb 2 and main with bb 4. Out of these main is already in OWL. Hence addition of p only with its IWL containing 2

[c] No change in p's summary values. No addition to OWL

[d] main's IWL is over. No change in main's summary values. No addition to OWL. Here OWL also gets empty. Hence we stop here.

```
---------------------------------------------
edge_vec = current_block->succs;
FOR_EACH_EDGE(e,ei,edge_vec)
{
    succ_bb = e->dest;
    add_bb_to_IWL(succ_bb,IWL_head);
}
---------------------------------------------
```

Here is the code fragment of function compute_pf_summarygenkill:

```
------------------------------------------------------------------
void
compute_pf_summarygenkill()
{
  do
  {
    change = false;
    current_block = get_bb_from_IWL(IWL_head);
    if(traversal_order == FORWARD)
    {
      change_at_GenKillIn =
                compute_func_genkill_forward(current_block);
      index = find_index_bb(current_block);
      if(current_bvfunc_pf_genkill[index])
      {
          sbitmap_copy(GenIn,
                       current_bvfunc_pf_genkill[index]->fGenIn);
          sbitmap_copy(KillIn,
                       current_bvfunc_pf_genkill[index]->fKillIn);
      }
      change_at_GenOut = composite_func_gen(current_block,GenIn);
      change_at_KillOut = composite_func_kill(current_block,KillIn);
      change = change_at_GenOut || change_at_KillOut;
      if(change) //successors should be added
      {
        //Successors are added to worklist
      }
    }
  }while(IWL_head);
}
------------------------------------------------------------------
```

Function get_bb_from_IWL parameterized with a pointer to head of a work-list returns the basic block at the head of this worklist. The function compute_func_genkill_forward calculates gen and kill information at the IN of a given block. It computes the function confluences as given below:

$$
\begin{gathered}
f2 \sqcap f1 = f3 \\
if \sqcap is \cup \\
Kill_3 = Kill_1 \cap Kill_2 \\
Gen_3 = Gen_1 \cup Gen_2 \\
if \sqcap is \cap \\
Kill_3 = Kill_1 \cup Kill_2 \\
Gen_3 = Gen_1 \cap Gen_2
\end{gathered}
$$

The code fragement of this function is given below:

```
-------------------------------------------------------------------
bool compute_func_genkill_forward(basic_block bb)
{
   if (!bb->preds)
   {
     sbitmap_copy(newGenIn,funcgentop);
     sbitmap_copy(newKillIn,funckilltop);
   }
   else
   {
     newGenIn = combined_forward_edge_flow_func_genkill(bb,GEN);
     newKillIn = combined_forward_edge_flow_func_genkill(bb,KILL);
   }
   sbitmap_copy(oldGenOut,
                current_bvfunc_pf_genkill[index]->fGenOut);
   sbitmap_copy(oldKillOut,
                current_bvfunc_pf_genkill[index]->fKillOut);
   changeGen = is_new_info(newGenOut,oldGenOut);
   if(changeGen)
     sbitmap_copy(current_bvfunc_pf_genkill[index]->fGenOut,
                  newGenOut);
   if(oldGenOut)
     free_dfvalue_space(oldGenOut);

   changeKill = is_new_info(newKillOut,oldKillOut);
   if(changeKill)
     sbitmap_copy(current_bvfunc_pf_genkill[index]->fKillOut,
                  newKillOut);
   if(oldKillOut)
     free_dfvalue_space(oldKillOut);

   change = changeGen || changeKill;
   return change;
}
-------------------------------------------------------------------
```

If there is change in old gen and/or kill values, those values are overwritten with newly computed values and function returns true. The functions composite_func_gen and composite_func_kill compute the following terms:

$$Gen_3 = (Gen_1 - Kill_2) \cup Gen_2$$
$$Kill_3 = Kill_1 \cup Kill_2$$

The function combined_forward_edge_flow_func_genkill is called from compute_func_genkill_forward and calculates the following term:

$$\overrightarrow{\sqcap}_{p \in preds(n)} (GenOut_p)$$

It calls identity_forward_edge_flow_func_genkill function which simply returns gen or kill at OUT of a given block. The code fragment is shown below.

```
-------------------------------------------------------------------
dfvalue
combined_forward_edge_flow_func_genkill(int fid,basic_block bb,
                          GENORKILL genkillflag)
{
  edge_vec = bb->preds;
  if(genkillflag==GEN)
     temp = make_initialised_dfvalue(gen_init);
  else
     temp = make_initialised_dfvalue(kill_init);
  FOR_EACH_EDGE(e,ei,edge_vec)
  {
    pred_bb = e->src;
    if(genkillflag==GEN)
      new=combine_func_genkill(temp,
          identity_forward_edge_flow_func_genkill(succ_bb,GEN),
          gen_confluence);
    else
      new=combine_func_genkill(temp,
          identity_forward_edge_flow_func_genkill(succ_bb,KILL),
    if (temp)
      free_dfvalue_space(temp);
    temp = new;
  }
}
-------------------------------------------------------------------
```

3 Driver For Interprocedural DFA Using Functional Approach

The function described in section 2 computes summary information on per function level. The driver is written to extend this summary information on interprocedural level. Following data structure is added to save per function summary information.

```
-------------------------------------------------------------
typedef struct bvfunc_summary_genkill
{
        int funcid;
        dfvalue summary_gen;
        dfvalue summary_kill;
}bvfunc_summary_genkill;
bvfunc_summary_genkill ** current_bvfunc_summary_genkill;
-------------------------------------------------------------
```

The functional approach to interprocedural data flow analysis requires the top level driver to perform following tasks:

- Assign unique indices to functions.
- Compute the number of nodes per function.
- Initialise special values such as confluence operators and top values for gen and kill.
- Create space for per function gen and kill data flow values.
- Initialise these gen and kill data flow values with gen top and kill top values respectively.
- Create space for per function summary gen and kill information.
- Initialise these summary values with corresponding top values.
- Initialise inner worklist of each function with the corresponding basic blocks and the outer worklist(OWL) with entry for each function node.
- Compute interprocedural summary information. This is done using worklist method of data flow analysis.

Function func_ipdfa_driver performs these tasks.

```
---------------------------------------------------------------
void
func_ipdfa_driver(struct gimple_pfbv_dfa_spec dfa_spec)
```

```
{
        assign_indices_to_function_decl();
        compute_pf_no_of_nodes();
        init_special_functional_values(dfa_spec);
        create_pf_func_dfi_space();
        init_pf_func_genkill();
        create func_summary_genkill_space();
        init_func_summary_genkill();
        init_worklists();
        perform_ipfuncdfa();
        delete_pf_func_dfi_space();
        delete_func_summary_genkill();
}
--------------------------------------------------------------
```

The function assign_indices_to_function_decl assigns the unique indices to the function declarations. This index is added in the declaration of structure tree_function_decl in tree.h. These indices then can be used to access per function data flow information.

```
------------------------------------------------------------------------
void
assign_indices_to_function_decl(void)
{
  struct cgraph_node *current_node = cgraph_nodes;

  for (current_node = cgraph_nodes;
       current_node != NULL;current_node = current_node->next)
  {
     if(current_node->analyzed &&cgraph_is_master_clone(current_node))
         DECL_FUNCTION_ID(current_node->decl) = number_of_funcs++;
  }
}
------------------------------------------------------------------------
```

Function compute_pf_no_of_nodes calculates number of nodes per function which is required while creating space for data flow values for every function. The function init_special_functional_values uses the specifications provided in gimple-pfbvdfa-specs.c to decide special values such as gen top, kill top and confluence operators for gen, kill. Functions init_pf_func_genkill and init_func_summary_genkill initialize the gen and kill data flow values with respective top values.

The function init_worklists creates per function worklist, which we call as inner worklist(IWLs). Each inner worklist is initialized with all basic blocks from corresponding function. While outer worklist is initiated with nodes for each function. Declarations of nodes of each of these worklist are shown below:

```
-------------------------------------------------
typedef struct pf_worklist_node
{
    basic_block bb;
    struct pf_worklist_node *next;
}pf_worklist_node;

typedef struct func_worklist_node
{
    int funcid;
    struct cgraph_node *cg_node;
    pf_worklist_node *pf_worklist_head;
    struct func_worklist_node *next;
}func_worklist_node;
-------------------------------------------------
```

The function perform_ipfuncdfa does the interprocedural data flow analysis using functional approach. It is uses the worklist method to compute data flow values. The pseudo code is shown below:

```
-------------------------------------------------------------
perform_ipfuncdfa()
{
  struct cgraph_node *cgraph_NODE;
  do
  {
    func_node = get_func_from_OWL();
    cgraph_NODE = func_node->cg_node;
    change = compute_pf_summary_genkill(func_node);
    if(change)
    {
       Find out the callers and correspnding call-blocks
       if(cgraph_NODE)
       {
          for(edge=cgrpah_NODE->callers;edge;edge=edge->next_caller)
          {
               //for each such caller,
                 find out call sites and add them into IWL.
               //add each such function node to OWL.
          }
       }
    }
  }while(func_outer_worklist_head);
}
------------------------------------------------------------
```

The worklist algorithm works on outer worklist of function nodes. It performs the following steps while outer worklist is not empty.

- Take out the first node from outer worklist.
- Compute summary gen and kill information of this function using function compute_pf_summary_genkill described in section 2. This function returns boolean value indicating if there is change in old summary values of a passed function.
- If there is change in old summary gen and/or kill values of this function, its callers are found out along with their call nodes. These function nodes are then added to outer worklist.

When the outer worklist gets over, the final summary values of each function are nothing but gen and kill IN at entry block for backward data flow problems, while gen and kill OUT at exit block for forward data flow problems.

4 Support from GCC

Implementation needs following support from GCC:

- Data Structures:
  - Structure cgraph_node
    - cfg
    - x_n_basic_blocks
    - edge
    - caller
  - Structure sbitmap
  - Pass structure of gcc
  - DFA specifications from gdfa
  - enums defined in gdfa
  - Relevant entity and entity count from gdfa
- APIs
  - All sbitmap related APIs, such as sbitmap_copy, sbitmap_alloc etc.
  - API for traversing successors/predecessors: FOR_EACH_EDGE
  - Traversing call-graph using cgraph_edge of structure cgraph_node
  - Traversing CFG using VARRAY_BB
  - APIs from gdfa: GEN and KILL

5 Further Work

- Moving gdfa from intraprocedural levek to interprocedural level which includes
  - Assigning indices to global entities. Current gdfa processes only local entities.
  - Analysis of global entities. When analysis is to be carried along interprocedural level, effect of global entities must be taken into consideration.
  - Current gdfa accesses GEN, KILL data structures using block indices only as it is per function level analysis. But modification of data sturctures to index the basic blocks using basic block indices as well as function indices is required.
  - Moving current gdfa from intra level passes to inter level passes in gcc is to be done.
- API for traversing call-graph in DFS order and initializing outer worklist is to be implemented.
- APIs for initializing per function worklists with corresponding basic blocks are needed.
- API for finding out call nodes in a caller is to be done.

References

[1] Uday Khedker. gdfa: A generic data flow analyzer for gcc. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology Bombay, 2008.

[2] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group), 2009.