# Part III

# Implementing Data Flow Analysis

# 10

# *Implementing Data Flow Analysis in GCC*

This chapter presents a *g*eneric *d*ata *f*low *a*nalyzer for *p*er *f*unction (i.e., intraprocedural) *b*it *v*ector *d*ata *f*low *a*nalysis in GCC 4.3.0. We call this infrastructure *gdfa*. The analyzers implemented using *gdfa* are called *pfbvdfa*. *gdfa* has been used to implement several bit vector data flow analyses.

The design and implementation of *gdfa* is motivated by the following objectives:

- Demonstrating the practical significance of the following important generalization: Instead of implementing specific analyses directly, it is useful to implement a generic driver that is based on a carefully chosen set of abstractions. The task of implementing a particular analyzer then reduces to merely specifying the analysis by instantiating these abstractions to concrete values.

- Providing an easy to use and easy to extend data flow analysis infrastructure. The goal is to facilitate experimentation in terms of studying existing analyses, defining new analyses, and exploring different analysis algorithms.

Section 10.1 describes the specification mechanism of *gdfa* and shows how the resulting pass can be included in GCC 4.3.0. We illustrate it for the bit vector analyses implemented using *gdfa*. Section 10.2 demonstrates how *pfbvdfa* can be used. Section 10.3 describes the implementation of *gdfa*. This section also shows how local property computation can be driven by specifications. Finally Section 10.4 suggests some possible enhancements to *gdfa*.

The GCC related details in this chapter are interleaved with the description of *gdfa*. Appendix A provides a short introduction to GCC, its installation, and how to obtain its patch for *gdfa*.[*] The code presented in this chapter is a slightly edited version of the original code. This was required to fit a page size constraints.

## 10.1 Specifying a Data Flow Analysis

In this section we look at how we can use the generic data flow analysis driver to implement a data flow analysis pass in GCC. The implemented pass has to be registered

[*]We use GCC to denote the GNU compiler generation framework using which a compiler can be built for a given processor. The compiler so generated is denoted by `gcc`.

with the pass manager in GCC so that it can be executed by the compiler.

### 10.1.1 Registering a Pass With the Pass Manager in GCC

*gdfa* works on the gimple version of the intermediate representation used by GCC. We have included *pfbvdfa* passes such that they are invoked by default when `gcc` is used for compiling a program. When `gcc` is built, this causes *pfbvdfa* passes to run on the entire source of `gcc` which consists of over a million lines of C code. This helps in ensuring that these do not cause any exception in the compilation sequence.

After constructing the gimple representation, `gcc` views the rest of the compilation as sequential execution of various passes. This is carried out by traversing a linked list whose nodes contain pointers to the entry functions of these passes. A pass is registered with the pass manager through the following steps:

- Instantiating a variable as an instance of `struct tree_opt_pass` in some file.
- Declaring this variable as an `extern` variable in header file `tree-pass.h`.
- Inserting this variable in the linked list of passes using the macro `NEXT_PASS` in function `init_optimization_passes` in file `passes.c`.

Here is the declaration of `struct tree_opt_pass`. For convenience comments have been removed and are used in the explanation that follows.

```
 0 struct tree_opt_pass
 1 {
 2   const char *name;
 3   bool (*gate) (void);
 4   unsigned int (*execute) (void);
 5   struct tree_opt_pass *sub;
 6   struct tree_opt_pass *next;
 7   int static_pass_number;
 8   unsigned int tv_id;
 9   unsigned int properties_required;
10   unsigned int properties_provided;
11   unsigned int properties_destroyed;
12   unsigned int todo_flags_start;
13   unsigned int todo_flags_finish;
14   char letter;
15 };
```

The `name` of the pass (line 2) is used as a fragment of the dump file name. We have used the names like `gdfa_ave`. The `gate` function (line 3) is used to check whether this pass and all its sub-passes should be executed or not. They are executed only if this function returns `true`. If no such checking is required, this function pointer can be NULL. The execute function (line 4) is entry function of the pass. If this function pointer is NULL, there should be sub-passes otherwise this pass does nothing. The return value tells gcc what more needs to be done. The variable sub (line 5) is a list of sub-passes that should be executed depending upon the gate predicate. If there are sub-passes that must be executed unconditionally, then they are listed in next (line 6). The static pass number (line 7) is used as a fragment of the dump file name. If it is specified as 0, the pass manager computes its value depending on the position of the pass. It is this that generated numbers 15, 16, 17, 18, and 19 for our data flow analyses. Variable tv_id is the variable that can be used as a time variable. The rest of the variables are self-explanatory. The last variable letter is used to annotate RTL code that is emitted.

We have registered available expressions analysis by creating a structure variable called pass_gimple_pfbv_ave_dfa as shown below.

```
struct tree_opt_pass pass_gimple_pfbv_ave_dfa =
{
  "gdfa_ave",             /* name */
  NULL,                   /* gate */
  gimple_pfbv_ave_dfa,    /* execute */
  NULL,                   /* sub */
  NULL,                   /* next */
  0,                      /* static_pass_number */
  0,                      /* tv_id */
  0,                      /* properties_required */
  0,                      /* properties_provided */
  0,                      /* properties_destroyed */
  0,                      /* todo_flags_start */
  0,                      /* todo_flags_finish */
  0                       /* letter */
};
```

This variable is declared as follows in file tree-pass.h

```
extern struct tree_opt_pass pass_gimple_pfbv_ave_dfa;
```

The next step in registering this pass is to include it in the list of passes. We show below the relevant code fragment from function init_optimization_passes in file passes.c:

```
  NEXT_PASS (pass_build_cfg);
/* Intraprocedural dfa passes begin */
  NEXT_PASS (pass_init_gimple_pfbvdfa);
  NEXT_PASS (pass_gimple_pfbv_ave_dfa);
  NEXT_PASS (pass_gimple_pfbv_pav_dfa);
  NEXT_PASS (pass_gimple_pfbv_ant_dfa);
  NEXT_PASS (pass_gimple_pfbv_lv_dfa);
  NEXT_PASS (pass_gimple_pfbv_rd_dfa);
  NEXT_PASS (pass_gimple_pfbv_pre_dfa);
/* Intraprocedural dfa passes end */
```

Finally, we need to include the new file names in the GCC build system. This is done by listing the file names and their dependencies in `Makefile.in` in the `gcc-4.3.0/gcc` directory. Appendix A provides the steps for building `gcc`.

### 10.1.2 Specifying Available Expressions Analysis

The specification mechanism supported by *gdfa* is simple and succinct. It follows the GCC mechanism of specification by using a `struct` as a hook and by requiring the user to create a variable by instantiating the members of the `struct` defined for the purpose.

For available expressions analysis, we define a variable called `gdfa_ave` which is of the type `struct gimple_pfbv_dfa_spec gdfa_ave`.

```
struct gimple_pfbv_dfa_spec gdfa_ave =
{
     entity_expr,                  /* entity             */
     ONES,                         /* top_value          */
     ZEROS,                        /* entry_info         */
     ONES,                         /* exit_info          */
     FORWARD,                      /* traversal_order    */
     INTERSECTION,                 /* confluence         */
     entity_use,                   /* gen_effect         */
     down_exp,                     /* gen_exposition     */
     entity_mod,                   /* kill_effect        */
     any_where,                    /* kill_exposition    */
     global_only,                  /* preserved_dfi      */
     identity_forward_edge_flow,   /* forward_edge_flow  */
     stop_flow_along_edge,         /* backward_edge_flow */
     forward_gen_kill_node_flow,   /* forward_node_flow  */
     stop_flow_along_node          /* backward_node_flow */
};
```

Before we explain the above, we present the rest of the code required to complete the specification.

```
18 pfbv_dfi ** AV_pfbv_dfi = NULL;
19
20 static unsigned int
21 gimple_pfbv_ave_dfa(void)
22 {
23
24         AV_pfbv_dfi = gdfa_driver(gdfa_ave);
25
26         return 0;
27 }
```

Nothing more is required for specifying available expressions analysis apart from registering it with the pass manager with function `gimple_pfbv_ave_dfa` as its entry point as described in Section 10.1.1. This function calls the *gdfa* driver passing the specification variable `gdfa_ave` as actual parameter. The data flow information computed by the driver is stored in a pointer to an array called `AV_pfbv_dfi`; each element of this array represents the data flow information for a basic block and is an instance of the following type defined by *gdfa*.

```
typedef struct pfbv_dfi
{
        dfvalue gen;
        dfvalue kill;
        dfvalue in;
        dfvalue out;
} pfbv_dfi;
```

The semantics expressed by `struct gimple_pfbv_dfa_spec gdfa_ave` is as described below: Line 2 declares that the relevant entities for this analysis are expressions (`entity_expr`). Line 3 specifies that $\top$ is "all `ONES`" implying the universal set $\mathbb{E}\mathsf{xpr}$. The specification "all `ZEROS`" on line 4 initializes the $BI_{Start}$ to $\emptyset$ whereas `ONES` on line 5 renders $BI_{End}$ irrelevant because it is same as $\top$. Line 6 declares the direction of traversal to be `FORWARD`. Note that this is independent of the direction of flow and only influences the number of iterations. If we choose the direction of traversal as `BACKWARD`, the resulting data flow information will remain same except that it may take a much larger number of iterations. Line 7 declares the $\sqcap$ to be $\cap$. Line 12 directs the driver to preserve only the global data flow information (*In* and *Out*); the driver can reclaim the space occupied by the local data flow information (*Gen* and *Kill*).

The most interesting elements of the specification are the specifications of local properties and flow functions:

- *Local property specification.*

  Lines 8 to 11 define the *Gen* and *Kill* kill sets for a block. Observe that this mechanism closely follows the description in Section 2.2.

  - Lines 8 and 9 say that when a downwards exposed (`down_exp`) use of an entity (`entity_use`) is found in a basic block, it is included in the *Gen* set of the block. From line 2 we know that the entity under consideration is an expression (`entity_expr`).
  - Lines 10 and 11 say that when a modification of an entity (`entity_mod`) is found in a basic block, it is included in the *Kill* set of the block. This modification need not be upwards exposed or downwards exposed, it can appear `any_where`.

  This is possible because the *gdfa* driver is aware of the fact that the use of an entity could be affected by its modification and hence the notion of exposition of an entity is explicated in the specification.

- *Flow function specification.*

  Lines 13 to 16 specify the flow functions for available expressions analysis as required by the generic data flow Equations (5.1) and (5.2). The forward edge flow function $\overrightarrow{f_{n \to m}}$ in available expressions analysis is $\phi_{id}$ (line 13) whereas the forward node flow function $\overrightarrow{f_n}$ is the conventional *Gen*-*Kill* function $f(X) = Gen \cup (In - Kill)$. Further, there is no backward flow i.e., $\overleftarrow{f_n}$ and $\overleftarrow{f_{n \to m}}$ are $\phi_\top$ (Section 5.1). This is specified by lines 14 and 16. All these functions are supported by *gdfa* and it is enough to associate the function pointers with appropriate functions.

  When the nature of data flow is different from the default flows, it is also possible to write custom functions—we show how it is done for partial redundancy elimination in Section 10.1.3.

### 10.1.3 Specifying Other Bit Vector Data Flow Analyses

Given the specification of available expressions analysis, it is easy to visualize specifications for other bit vector frameworks. We describe the required changes in the following:

- *Partially available expressions analysis.*

  Confluence should be `UNION`, $\top$ and $BI_{End}$ should be `ZEROS`.

- *Reaching definitions analysis.*

  Entity should be `entity_defn`, confluence should be `UNION`, $\top$ and $BI_{End}$ should be `ZEROS`.

- *Anticipable expressions analysis.*

  The data flow equations for anticipable expressions analysis are Equations (2.9) and (2.10). In this case it is desirable, though not necessary, to choose the direction of traversal as `BACKWARD`. The exposition for *Gen* should be changed to `up_exp`. $BI_{Start}$ should be `ONES` and $BI_{End}$ should be `ZEROS`. Flow functions would change as follows:

  - forward edge flow function $\overrightarrow{f}_{n\to m}$ should be `stop_flow_along_edge`,
  - forward node flow function $\overrightarrow{f}_n$ should be `stop_flow_along_node`, and
  - backward node flow function $\overleftarrow{f}_n$ should be the default *Gen-Kill* function `backward_gen_kill_node_flow`.

- *Live variables analysis.*

  This specification would be similar to that of anticipable expressions analysis except that the entity should be `entity_var`, confluence should be `UNION`, $\top$ and $BI_{End}$ should be `ZEROS`.

- *Partial redundancy elimination.*

  Here it would useful to change the `gate` function to this pass to check that available expressions analysis and partially available expressions analysis has been performed.

  The specification of data flow analysis would be similar to that of anticipable expressions analysis except that the flow functions would change. The data flow equations for anticipable expressions analysis are Equations (2.9) and (2.10) whereas the data flow equations for partial redundancy elimination are Equations (2.11) and (2.15). Clearly, the change is only in the flow function in the equation for $In_n$. In particular, the forward edge flow function $\overrightarrow{f}_{n\to m}$ and the backward node flow function $\overleftarrow{f}_n$ cannot be chosen from the default functions supported by *gdfa*. We define the required functions as shown below.

```
dfvalue
forward_edge_flow_pre(basic_block src, basic_block dest)
{
        dfvalue temp;

        temp = union_dfvalues (OUT(AV_pfbv_dfi,src),
                               CURRENT_OUT(src));

        return temp;
}
```

In this function, `src` and `dest` indicate the source and destination of an edge. Since this flow function is used in computing $\mathit{In}_n$, `dest` represents $n$ and `src` represents the given predecessor node $p$. Under the assumption that the data flow information of available expressions analysis is stored in the variable `AV_pfbv_dfi`, the term `OUT(AV_pfbv_dfi,src)` represents $\mathit{AvOut}_p$ whereas the $\mathit{Out}_p$ is represented by the term `CURRENT_OUT(src)`. Thus this flow function computes $\mathit{AvOut}_p \cup \mathit{Out}_p$ for a given predecessor $p$.

The definition of backward node flow is similar to that of the default node flow except that we need to include the value of $\mathit{PavIn}_n$. This is easily achieved by the function defined below:

```
dfvalue
backward_node_flow_pre(basic_block bb)
{
        dfvalue temp1, temp2;

        temp1 = backward_gen_kill_node_flow(bb);

        temp2 = intersect_dfvalues (IN(PAV_pfbv_dfi,bb),
                                    temp1);

        if (temp1)
                free_dfvalue_space(temp1);

        return temp2;
}
```

Here `bb` is the current node $n$. The default backward node flow function is used to compute the data flow information in the variable `temp1`. Under the assumption that the data flow information of partially available expressions analysis is stored in the variable `PAV_pfbv_dfi`, the term `IN(PAV_pfbv_dfi,bb)` represents $\mathit{PavIn}_n$. All that further needs to be done is to intersect them.

This completes the specification of partial redundancy elimination.

## 10.2 An Example of Data Flow Analysis

We use the example program from Figure 2.1 on page 27 in Chapter 2 to demonstrate the use of analyzer implemented using *gdfa*. We show the result of live variables analysis and available expressions analysis. A C program that represents the CFG in Figure 2.1 is given below.

```
int x, y, z;

int exmp(void)
{   int a, b, c, d;

    b = 4;
    a = b + c;
    d = a * b;
    if (x < y)
         b = a -c;
    else
    {   do
        {    c = b + c;
             if (y > x)
             {    do
                  {    d = a + b;
                       f(b + c);
                  } while(y > x);
             }
             else
             {    c = a * b;
                  f(a - b);
             }
             g (a + b);
        } while(z > x);
    }
    h(a-c);
    f(b+c);
}
```

Since the original example does not show conditions explicitly, we have used global variables in conditions; these variables are ignored by intraprocedural data flow analysis. Further, the functions `f`, `g`, and `h` are unspecified. Since C uses call by value mechanism, we have ignored the effects of function calls under the assumption that arrays and addresses of variables are not passed as parameters.

### 10.2.1 Executing the Data Flow Analyzer

Our example program is not a complete program hence we cannot compile it into an executable program. For such programs we must use the `-c` option that creates only an object file for the given input C file. Alternatively, we can use the `-S` option that stops the compilation after generating the corresponding assembly file. We use the following command to generate text files that provide the results of our passes.

```
$ gcc -S -fdump-tree-all -fgdfa  exmp.c
```

The option `-fdump-tree-all` enables generation of the dump files for passes implemented on gimple representation. The option `-fgdfa` emits the results of our data flow analysis passes in respective dump files. The dump files that are of interest to us are:

| Name | Description of the output |
|---|---|
| `exmp.c.013t.cfg` | CFG |
| `exmp.c.015t.gdfa_ave` | available expressions analysis |
| `exmp.c.016t.gdfa_pav` | partially available expression analysis |
| `exmp.c.017t.gdfa_ant` | anticipable expressions analysis |
| `exmp.c.018t.gdfa_lv` | live variables analysis |
| `exmp.c.018t.gdfa_rd` | reaching definitions analysis |
| `exmp.c.019t.gdfa_pre` | partial redundancy elimination |

The numbers indicate the position of the pass in the sequence of passes. Pass number 014 processes the CFG to discover the entities of interest to us and performs depth first numbering of basic blocks so that post order or reverse post order traversal can be used by our data flow analysis passes. These numbers would change depending upon the exact sequence of passes in a given version of GCC.

## 10.2.2 Examining the Gimple Version of CFG

The gimple representation used by GCC consists of three address code statements. The CFG version of gimple representation identifies basic blocks and explicates control flow between basic blocks. It also shows the declarations of temporary variables. There are two categories of temporary variables in gimple:

- *Artificial variables.* These variables are created to store the values of global variables. Subsequently, these variables are used in expressions. Any assignment to a global variable uses the original global variable so that the latest value can be read into a new artificial variable for a subsequent use.

  Artificial variables are also created for those instances of local variables that are assigned a value returned by a function call. The value of these artificial variables is then assigned to the local variables.

- *Temporary variables.* These are the traditional temporary variables which hold the intermediate results of expression computations. The parameters passed to functions are also represented by temporary variables.

The declaration part of gimple CFG in `exmp.c.013t.cfg` is:

```

;; Function exmp (exmp)

exmp ()
{
  int d;
  int c;
  int b;
  int a;
  int D.1205;
  int D.1204;
  int x.7;
  int z.6;
  int D.1201;
  int D.1200;
  int x.5;
  int y.4;
  int D.1197;
  int x.3;
  int y.2;
  int y.1;
  int x.0;
```

The gimple representation of our program initially contains eight artificial variables: `x.7`, `z.6`, `x.5`, `y.4`, `x.3`, `y.2`, `y.1`, and `x.0`. Each use of a global variable causes a distinct number to be suffixed to the variable. The temporary variables are: `D.1205`, `D.1204`, `D.1201`, `D.1200`, and `D.1197`. They represent the parameters of the five calls made in our program. There are no temporary variables for holding intermediate results of computations because our expressions consist of a single operation—temporaries are created for expressions containing more than one operation.

The CFG contains a unique `ENTRY` block which does not contain any computation and does not have any predecessor block. Similarly, there is an `EXIT` block which does not contain any computation and does not have any successor. An unconditional control transfer from a block to another block is recorded as `fallthru` whereas a conditional transfer is labeled `true` or `false`. All auxiliary information about a block e.g., block number, list of successors and predecessors, nature of control flow etc. is shown with a # mark as the first symbol on a line.

`ENTRY` and `EXIT` blocks are not listed explicitly in the dump. Internally they are numbered block 0 and block 1 respectively. Hence the first block that appears in the CFG is block 2 as shown below. It corresponds to block $n_1$ in Figure 2.1 on page 27. Observe the use of artificial variables `x.0` and `y.1` in the block.

```
# BLOCK 2
# PRED: ENTRY (fallthru)
b = 4;
a = b + c;
d = a * b;
x.0 = x;
y.1 = y;
if (x.0 < y.1)
  goto <bb 3>;
else
  goto <bb 4>;
# SUCC: 3 (true) 4 (false)
```

This block has a conditional control transfer at the end of it. Its successor blocks are blocks 3 and 4 which correspond to blocks $n_2$ and $n_3$ respectively in the CFG in Figure 2.1. Note that the predecessors of a block are also labeled to indicate the nature of control transfer (i.e., `fallthru`, `true`, or `false`).

```
# BLOCK 3
# PRED: 2 (true)
b = a - c;
goto <bb 9>;
# SUCC: 9 (fallthru)

# BLOCK 4
# PRED: 2 (false) 8 (true)
c = b + c;
y.2 = y;
x.3 = x;
if (y.2 > x.3)
  goto <bb 5>;
else
  goto <bb 7>;
# SUCC: 5 (true) 7 (false)
```

The structure of the control flow between the remaining blocks is a little different from the CFG shown in Figure 2.1. Block 5 in the `gcc` generated CFG combines blocks $n_5$ and $n_6$ of Figure 2.1 because there is a strictly sequential control flow between them. Block 6 consists of a single `goto` that will be optimized away later. Figure 2.1 does not have this block. Block 7 corresponds to block $n_4$ and block 8 corresponds to block $n_7$ in Figure 2.1. The last block containing some program code is block 9 which corresponds to $n_8$ in Figure 2.1. Observe that it has `EXIT` as its successor. The details of these blocks are as follows:

```
  # BLOCK 5
  # PRED: 4 (true) 5 (true)
  d = a + b;
  D.1197 = b + c;
  f (D.1197);
  y.4 = y;
  x.5 = x;
  if (y.4 > x.5)
    goto <bb 5>;
  else
    goto <bb 6>;
  # SUCC: 5 (true) 6 (false)

  # BLOCK 6
  # PRED: 5 (false)
  goto <bb 8>;
  # SUCC: 8 (fallthru)

  # BLOCK 7
  # PRED: 4 (false)
  c = a * b;
  D.1200 = a - b;
  f (D.1200);
  # SUCC: 8 (fallthru)

  # BLOCK 8
  # PRED: 6 (fallthru) 7 (fallthru)
  D.1201 = a + b;
  g (D.1201);
  z.6 = z;
  x.7 = x;
  if (z.6 > x.7)
    goto <bb 4>;
  else
    goto <bb 9>;
  # SUCC: 4 (true) 9 (false)

  # BLOCK 9
  # PRED: 3 (fallthru) 8 (false)
  D.1204 = a - c;
  h (D.1204);
  D.1205 = b + c;
  f (D.1205);
  return;
  # SUCC: EXIT

}
```

In essence, the CFGs constructed by `gcc` are quite similar to the CFGs that we have seen in the earlier parts of the book.

### 10.2.3 Examining the Result of Data Flow Analysis

The results of an analysis are available in internal data structures in a ready to use form. Section 10.1.3 shows how they can be used when we describe the implementation of partial redundancy elimination which needs the result of available expressions analysis and partially available expressions analysis. Here we present the textual dump of the results produced by the options `-fdump-tree-all` and `-gdfa`.

File `exmp.c.018t.gdfa_lv` contains the result of liveness analysis. It indicates that for this example $\mathbb{V}\mathsf{ar} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ intraprocedural liveness analysis. It also indicates the bit position for each variable. Variable `d` is the first to be considered. This is because internally, the variables are added to the head of the list of variables rather than its tail. Observe that the other three category of variables (global, artificial, and local) have been eliminated from consideration.[†]

```
;; Function exmp (exmp)

Number of relevant entities: 4

 Bit position and entity mapping is  ********************
        0:(d),1:(c),2:(b),3:(a)

 Initial values **************************************

Basic Block 2. Preds:  ENTRY. Succs:  3 4
        ----------------------------
        GEN Bit Vector:   0100
        GEN Entities:     (c)
        ------------------------------
        KILL Bit Vector:  1011
        KILL Entities:    (d),(b),(a)
        ------------------------------
        IN Bit Vector:    0000
        IN Entities:
        ------------------------------
        OUT Bit Vector:   0000
        OUT Entities:
        ------------------------------
```

The $\mathit{In}_n$ and $\mathit{Out}_n$ properties have been initialized to $\emptyset$ which is $\top$ for live variables analysis. In the following, we produce only the lines that enumerate the $\mathit{Gen}_n$ and

[†]At the moment, our implementation does not consider formal parameters.

$Kill_n$ in terms of entity names rather than bit vectors.

```
Basic Block 2. Preds:  ENTRY. Succs:  3 4
        ----------------------------
        GEN Entities:     (c)
        KILL Entities:    (d),(b),(a)
        ------------------------------
Basic Block 3. Preds:  2. Succs:  9
        ----------------------------
        GEN Entities:     (c),(a)
        KILL Entities:    (b)
        ------------------------------
Basic Block 4. Preds:  2 8. Succs:  5 7
        ----------------------------
        GEN Entities:     (c),(b)
        KILL Entities:    (c)
        ------------------------------
Basic Block 5. Preds:  4 5. Succs:  5 6
        ----------------------------
        GEN Entities:     (c),(b),(a)
        KILL Entities:    (d)
        ------------------------------
Basic Block 6. Preds:  5. Succs:  8
        ----------------------------
        GEN Entities:
        KILL Entities:
        ------------------------------
Basic Block 7. Preds:  4. Succs:  8
        ----------------------------
        GEN Entities:     (b),(a)
        KILL Entities:    (c)
        ------------------------------
Basic Block 8. Preds:  6 7. Succs:  4 9
        ----------------------------
        GEN Entities:     (b),(a)
        KILL Entities:
        ------------------------------
Basic Block 9. Preds:  3 8. Succs:  EXIT
        ----------------------------
        GEN Entities:     (c),(b),(a)
        KILL Entities:
        ------------------------------
```

It can be readily verified from the table in Example 2.3 on page 27 that the local data flow values given below are identical to the values discovered earlier.

The final values are also generated in the same format. We show selected lines from the final result of liveness analysis of our example program:

```
Total Number of Iterations = 2 *********

Basic Block 2. Preds:  ENTRY. Succs:  3 4
        ----------------------------
        IN Entities:      (c)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 3. Preds:  2. Succs:  9
        ----------------------------
        IN Entities:      (c),(a)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 4. Preds:  2 8. Succs:  5 7
        ----------------------------
        IN Entities:      (c),(b),(a)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 5. Preds:  4 5. Succs:  5 6
        ----------------------------
        IN Entities:      (c),(b),(a)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 6. Preds:  5. Succs:  8
        ----------------------------
        IN Entities:      (c),(b),(a)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 7. Preds:  4. Succs:  8
        ----------------------------
        IN Entities:      (b),(a)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 8. Preds:  6 7. Succs:  4 9
        ----------------------------
        IN Entities:      (c),(b),(a)
        OUT Entities:     (c),(b),(a)
        ------------------------------
Basic Block 9. Preds:  3 8. Succs:  EXIT
        ----------------------------
        IN Entities:      (c),(b),(a)
        OUT Entities:
        ------------------------------
```

We leave it for the reader to verify that these values are identical to the values in the table in Example 2.3 on page 27.

If the option `-fgdfa` is replaced by `-fgdfa-details`, data flow values after each iteration are generated.

The result of data flow analyses involving expressions is produced much the same way. File `exmp.c.015t.gdfa_ave`. contains the details of available expressions analysis. The initial information in this file is:

```
;; Function exmp (exmp)

Number of relevant entities: 5

 Bit position and entity mapping is  **********************
    0:(b + c),1:(a * b),2:(a - c),3:(a + b),4:(a - b)

 Initial values **************************************

Basic Block 2. Preds:  ENTRY. Succs:  3 4
    ----------------------------
    GEN Bit Vector: 11000
    GEN Entities:   (b + c),(a * b)
    ----------------------------
    KILL Bit Vector:11111
    KILL Entities:  (b + c),(a * b),(a - c),(a + b),(a - b)
    ----------------------------
    IN Bit Vector:  11111
    IN Entities:    (b + c),(a * b),(a - c),(a + b),(a - b)
    ----------------------------
    OUT Bit Vector: 11111
    OUT Entities:   (b + c),(a * b),(a - c),(a + b),(a - b)
    ----------------------------
```

Unlike live variables analysis for which bit vectors of four bits are created, *gdfa* has created a bit vector of five bits for available expressions analysis of our example because our example has five expressions that qualify as local expressions. Observe that the expressions have been numbered in a different order compared to the order in Figure 2.1 on page 27. This is because *gdfa* forms the set $\mathbb{E}$xpr by making a forward pass over the program.

The initialization for available expressions analysis uses the entire $\mathbb{E}$xpr set which represents the $\top$ value. The value of *BI* is $\emptyset$. Although basic block 2 corresponds to block $n_1$ for which we had chosen *In* as *BI* for initialization, for the CFG constructed by `gcc`, *BI* is associated with the fictitious blocks `ENTRY` and `EXIT` as the case may be.

The local data flow properties for available expressions analysis of our example program for all blocks are:

```
Basic Block 2. Preds:  ENTRY. Succs:  3 4
    ----------------------------
    GEN Entities:   (b + c),(a * b)
    KILL Entities:  (b + c),(a * b),(a - c),(a + b),(a - b)
    ----------------------------
Basic Block 3. Preds:  2. Succs:  9
    ----------------------------
    GEN Entities:     (a - c)
    KILL Entities:    (b + c),(a * b),(a + b),(a - b)
    ------------------------------
Basic Block 4. Preds:  2 8. Succs:  5 7
    ----------------------------
    GEN Entities:
    KILL Entities:    (b + c),(a - c)
    ------------------------------
Basic Block 5. Preds:  4 5. Succs:  5 6
    ----------------------------
    GEN Entities:     (b + c),(a + b)
    KILL Entities:
    ------------------------------
Basic Block 6. Preds:  5. Succs:  8
    ----------------------------
    GEN Entities:
    KILL Entities:
    ------------------------------
Basic Block 7. Preds:  4. Succs:  8
    ----------------------------
    GEN Entities:     (a * b),(a - b)
    KILL Entities:    (b + c),(a - c)
    ------------------------------
Basic Block 8. Preds:  6 7. Succs:  4 9
    ----------------------------
    GEN Entities:     (a + b)
    KILL Entities:
    ------------------------------
Basic Block 9. Preds:  3 8. Succs:  EXIT
    ----------------------------
    GEN Entities:     (b + c),(a - c)
    KILL Entities:
    ------------------------------
```

Since block 6 consists of only an unconditional `goto` statement, $Gen_6 = Kill_6 = \emptyset$. For other block, the *Gen* and *Kill* values are same as in Example 2.9 on page 34. The final data flow values for available expressions analysis have been shown below.

```
Total Number of Iterations = 3 *********

Basic Block 2. Preds:  ENTRY. Succs:  3 4
    ----------------------------
    IN Entities:
    OUT Entities:     (b + c),(a * b)
    ------------------------------
Basic Block 3. Preds:  2. Succs:  9
    ----------------------------
    IN Entities:      (b + c),(a * b)
    OUT Entities:     (a - c)
    ------------------------------
Basic Block 4. Preds:  2 8. Succs:  5 7
    ----------------------------
    IN Entities:      (a * b)
    OUT Entities:     (a * b)
    ------------------------------
Basic Block 5. Preds:  4 5. Succs:  5 6
    ----------------------------
    IN Entities:      (a * b)
    OUT Entities:     (b + c),(a * b),(a + b)
    ------------------------------
Basic Block 6. Preds:  5. Succs:  8
    ----------------------------
    IN Entities:      (b + c),(a * b),(a + b)
    OUT Entities:     (b + c),(a * b),(a + b)
    ------------------------------
Basic Block 7. Preds:  4. Succs:  8
    ----------------------------
    IN Entities:      (a * b)
    OUT Entities:     (a * b),(a - b)
    ------------------------------
Basic Block 8. Preds:  6 7. Succs:  4 9
    ----------------------------
    IN Entities:      (a * b)
    OUT Entities:     (a * b),(a + b)
    ------------------------------
Basic Block 9. Preds:  3 8. Succs:  EXIT
    ----------------------------
    IN Entities:
    OUT Entities:     (b + c),(a - c)
    ------------------------------
```

We leave it for the reader to verify that these values are identical to the values obtained in Example 2.9 on page 34.

## 10.3 Implementing the Generic Data Flow Analyzer *gdfa*

We describe the implementation in terms of the specification primitives, interface with GCC, the generic functions for global property computation, and generic functions for local property computation.

### 10.3.1 Specification Primitives

The main data structure used for specification is:

```
 0 struct gimple_pfbv_dfa_spec
 1 {
 2         entity_name                entity;
 3         initial_value              top_value_spec;
 4         initial_value              entry_info;
 5         initial_value              exit_info;
 6         traversal_direction        traversal_order;
 7         meet_operation             confluence;
 8         entity_manipulation        gen_effect;
 9         entity_occurrence          gen_exposition;
10         entity_manipulation        kill_effect;
11         entity_occurrence          kill_exposition;
12         dfi_to_be_preserved        preserved_dfi;
13
14         dfvalue (*forward_edge_flow)(basic_block src,
15                                         basic_block dest);
16         dfvalue (*backward_edge_flow)(basic_block src,
17                                          basic_block dest);
18         dfvalue (*forward_node_flow)(basic_block bb);
19         dfvalue (*backward_node_flow)(basic_block bb);
20
21 };
```

The types appearing on lines 2 to 12 are defined as enumerated types with the following possible values.

| Enumerated Type | Possible Values |
|---|---|
| entity_name | entity_expr, entity_var, entity_defn |
| initial_value | ONES, ZEROS |
| traversal_direction | FORWARD, BACKWARD, BIDIRECTIONAL |
| meet_operation | UNION, INTERSECTION |
| entity_manipulation | entity_use, entity_mod |
| entity_occurrence | up_exp, down_exp, any_where |
| dfi_to_be_preserved | all, global_only, no_value |

The type dfvalue is just another name for the type sbitmap supported by GCC. We have used a different name to allow for the possibility of extending *gdfa* to other kinds of data flow values.

The entry point of each data flow analysis invokes the driver with its specification. The driver creates space for current data flow values in current data flow analysis in a variable current_pfbv_dfi which is declared as shown below:

```
typedef struct pfbv_dfi
{
        dfvalue gen;
        dfvalue kill;
        dfvalue in;
        dfvalue out;
} pfbv_dfi;

pfbv_dfi ** current_pfbv_dfi ;
```

For a basic block bb, different members of the data flow information are accessed using the following macros:

| Data flow variable | current_pfbv_dfi | Given dfi |
|---|---|---|
| *Gen* | CURRENT_GEN(bb) | GEN(dfi,bb) |
| *Kill* | CURRENT_KILL(bb) | KILL(dfi,bb) |
| *In* | CURRENT_IN(bb) | IN(dfi,bb) |
| *Out* | CURRENT_OUT(bb) | OUT(dfi,bb) |

Now we can describe the default functions that can be assigned to the function pointers on lines 14 to 19 in struct gimple_pfbv_dfa_spec. Alternatively, the users can define their own functions which have the same interface. The default functions supported by *gdfa* are:

| Function | Returned value |
|---|---|
| identity_forward_edge_flow(src, dest) | CURRENT_OUT(src) |
| identity_backward_edge_flow(src, dest) | CURRENT_IN(dest) |
| stop_flow_along_edge(src, dest) | top_value |
| identity_forward_node_flow(bb) | CURRENT_IN(bb) |
| identity_backward_node_flow(bb) | CURRENT_OUT(bb) |
| stop_flow_along_node(bb) | top_value |
| forward_gen_kill_node_flow(bb) | CURRENT_GEN(bb) ∪ ( CURRENT_IN(bb) - CURRENT_KILL(bb) ) |
| backward_gen_kill_node_flow(bb) | CURRENT_GEN(bb) ∪ ( CURRENT_OUT(bb) - CURRENT_KILL(bb) ) |

where `top_value` is of the type `initial_value` and is constructed based on the value of `top_value_spec` (line 3 in `struct gimple_pfbv_dfa_spec`).

This completes the description of the specification primitives.

## 10.3.2 Interface with GCC

The top level interface of *gdfa* with GCC is through the pass manager as described in Section 10.1.1. At the lower level, *gdfa* uses the support provided by GCC for traversals over CFGs, basic blocks etc.; discovering relevant features of statements, expressions, variables etc.; constructing and manipulating data flow values; and printing entities appearing in statements.

### Traversal Over CFG and Basic Blocks

In a round-robin iterative traversal, the basic blocks in a CFG are usually visited in the order of along control flow or against the order of control flow. In GCC, this is achieved as follows:

```
basic_block bb;

FOR_EACH_BB_FWD(ENTRY_BLOCK_PTR)
{     /* process bb */
}
FOR_EACH_BB_BKD(EXIT_BLOCK_PTR)
{     /* process bb */
}
```

In the above code, `basic_block` is a type supported by GCC. `ENTRY_BLOCK_PTR` and `EXIT_BLOCK_PTR` point to `ENTRY` and `EXIT` blocks of the current function being compiled. These macros have been defined by GCC. The two other macros used above are defined as follows:

```
#define FOR_EACH_BB_FWD(entry_bb)      \
    for(bb=entry_bb->next_bb;          \
        bb->next_bb!=NULL;             \
        bb=bb->next_bb)
#define FOR_EACH_BB_BKD(exit_bb)       \
    for(bb=exit_bb->prev_bb;           \
        bb->prev_bb!=NULL;             \
        bb=bb->prev_bb)
```

Given a basic block `bb`, its predecessor and successor blocks are traversed using an `edge_iterator` variable, an `edge` variable, and the macro `FOR_EACH_EDGE` as described below. All these are directly supported by GCC.

```
edge_iterator ei ;
edge e ;
basic_block succ_bb, pred_bb;

FOR_EACH_EDGE(e,ei,bb->preds)
{     pred_bb = e->src;
      /* process the predecessor pred_bb */
}
FOR_EACH_EDGE(e,ei,bb->succs)
{     succ_bb = e->dest;
      /* process successor succ_bb */
}
```

A statement is of the type `tree`. Further, all entities appearing in a statement are also of the type `tree`. All statements in a basic block can be traversed using a `block_statement_iterator` variable.

```
basic_block bb;
block_stmt_iterator bsi;
tree stmt;

FOR_EACH_STMT_FWD
{    stmt = bsi_stmt(bsi);
     /* process stmt */
}
FOR_EACH_STMT_BKD
{    stmt = bsi_stmt(bsi);
     /* process stmt */
}
```

The macros used in the above code are defined as follows:

```
#define FOR_EACH_STMT_FWD            \
    for(bsi=bsi_start(bb);           \
        !bsi_end_p(bsi);             \
         bsi_next(&bsi))

#define FOR_EACH_STMT_BKD            \
   for(bsi=bsi_last(bb);             \
       bsi.tsi.ptr!=NULL;            \
       bsi_prev(&bsi))
```

## Discovering the Entities in a Statement

Statements can be of many types but only a few types are relevant to local data flow analysis. The lvalue and rvalue of a given statement `stmt` are of the type `tree` and are extracted as shown below:

```
tree expr=NULL, lval=NULL;

switch(TREE_CODE(stmt))
{  case COND_EXPR:
            expr = TREE_OPERAND(stmt,0);
            break;
   case MODIFY_EXPR:
            lval = TREE_OPERAND(stmt,0);
            expr = TREE_OPERAND(stmt,1);
   case GIMPLE_MODIFY_STMT:
            lval = GIMPLE_STMT_OPERAND(stmt,0);
            expr = GIMPLE_STMT_OPERAND(stmt,1);
            break;
   default:
            break;
}
```

The operands of relevant expressions are extracted as shown below:

```
tree op0=NULL, op1=NULL;

switch(TREE_CODE(expr))
{  case MULT_EXPR:
   case PLUS_EXPR:
   case MINUS_EXPR:
   case LT_EXPR:
   case LE_EXPR:
   case GT_EXPR:
   case GE_EXPR:
   case NE_EXPR:
   case EQ_EXPR:
         op1 = TREE_OPERAND(stmt,1);
         op0 = TREE_OPERAND(stmt,0);
         break;
   default:
          break;
}
```

Observe that this covers the set of expressions that is currently supported by *gdfa*.

Clearly, extending this set is easy.

Local variables are discovered by traversing `cfun->unexpanded_var_list` using `TREE_VALUE` and `TREE_CHAIN` macros supported by GCC. Here `cfun` represents the current function being compiled.

```
tree var,list;

list = cfun->unexpanded_var_list;
while (list)
{    var = TREE_VALUE (list);
      /* process variables *
      list = TREE_CHAIN(list);
}
```

Discovering definitions is easy: A statement with `TREE_CODE` as `MODIFY_EXR` or `GIMPLE_MODIFY_STMT` is detected as a definition.

**Constructing and Manipulating Data Flow Values**

We define the type `dfvalue` as follows:

```
typedef sbitmap dfvalue;
```

`sbitmap` is a type supported by GCC to represent sets. We use the following `sbitmap` functions to construct and manipulate bitmaps. Note that these functions are not directly used in *gdfa*. Instead, *gdfa* code calls `dfvalue` functions that are defined in terms of these functions.

| Name of the Function | Action |
|---|---|
| `sbitmap_equal(v_a,v_b)` | is `v_a` equal to `v_b`? |
| `sbitmap_a_and_b(t, v_a, v_b)` | `t` = `v_a` ∩ `v_b` |
| `sbitmap_union_of_diff(t, v_a, v_b, v_c)` | `t` = `v_a` ∪ ( `v_b` − `v_c`) |
| `sbitmap_a_or_b(t, v_a, v_b)` | `t` = `v_a` ∪ `v_b` |
| `sbitmap_ones(v)` | set every bit in `v` to 1 |
| `sbitmap_zero(v)` | set every bit in `v` to 0 |
| `sbitmap_alloc(n)` | allocate a bitmap of `n` bits |
| `sbitmap_free(v)` | free the space occupied by `v` |

**Facilities for Printing Entities**

We use the function `dump_sbitmap` to print bitmaps. For printing a statement, the function `print_generic_stmt` is used whereas function `print_generic_expr` prints an expression `expr`.

### 10.3.3 The Preparatory Pass

Before the *gdfa* driver is invoked, some preparatory work has to be performed by an earlier pass. The top level function of this pass is:

```
static unsigned int
init_gimple_pfbvdfa_execute (void)
{
        local_var_count=0;
        local_expr_count=0;
        number_of_nodes = n_basic_blocks+2;

        assign_indices_to_var();
        assign_indices_to_exprs();
        assign_indices_to_defns();

        dfs_ordered_basic_blocks = NULL;
        dfs_numbering_of_bb();

        return 0;
}
```

Function `assign_indices_to_var` assigns a unique index to each local variable by traversing `cfun->unexpanded_var_list` as explained in Section 10.3.2. These indices represent the bit position of a local variable. This requires adding an `integer` field to the `tree` data structure. The variables which are not interesting are assigned index `-1`. Function `assign_indices_to_defns` assigns a unique index to each statement that is a definition.

Function `assign_indices_to_exprs` assigns a unique index to each expression whose operands are restricted to constants and variables that have been assigned a valid index. These indices represent the bit position of relevant expressions. Other expressions are assigned index `-1`. Unlike local variables, there is no ready list of expressions. Hence function `assign_indices_to_exprs` traverses the CFG visiting each statement and examining the expressions appearing in relevant statements. If the expression used in a statement qualifies as a local expression, it is first checked whether an index has already been assigned to it. This could happen because an expression could appear multiple times in a program.

Finally, function `dfs_numbering_of_bb` performs depth first numbering of the blocks in a CFG.

### 10.3.4 Local Data Flow Analysis

In production compilers, implementing global data flow analyzers is much easier compared to implementing local data flow analyzers. This is because local data flow analysis has to deal with the lower level intricate details of the intermediate representation and intermediate representation are the most complex data structures in practical compilers. Global data flow analyzers are insulated from these lower level details; they just need to know CFGs in terms of basic blocks. Thus most data flow analysis engines require the local property computation to be implemented by the user of the engine.

This situation can change considerably if we view local data flow analysis as a special case of global data flow analysis. The objective of local data flow analysis is to compute $\mathit{Gen}_n$ and $\mathit{Kill}_n$ of a block $n$. This computation can be performed by traversing statements in block $n$ in a manner similar to traversing blocks in a CFG. The only difference is that statements in a block cannot have multiple predecessors of successors.

The way $\mathit{In}_{\mathit{Start}}$ (or $\mathit{Out}_{\mathit{End}}$) is computed by incorporating the effect of blocks in a CFG, $\mathit{Gen}_n$ and $\mathit{Kill}_n$ can also be computed by incorporating the effects of individual statements in block $n$. The effect of statement $s$ can be defined in terms of $\mathit{Gen}_s$ and $\mathit{Kill}_s$. However, we need to overcome the following conceptual difficulty: When we compute $\mathit{Gen}_n$ for block $n$, $\mathit{Gen}_s$ of a statement $s$ must be added to the cumulative effect of the statements processed so far. However, when we compute $\mathit{Kill}_n$, $\mathit{Kill}_s$ of statement $s$ should be *added* to the cumulative effect instead of being removed. This deviates from the normal meaning of *Kill* which represents the entities to be removed.

We overcome this conceptual difficulty by renaming $\mathit{Gen}_s$ and $\mathit{Kill}_s$ as $\mathit{Add}_s$ and $\mathit{Remove}_s$ respectively. Now local data flow analysis does not depend on knowing whether the data flow property being computed is $\mathit{Gen}_n$ or $\mathit{Kill}_n$. Given a local property specification such as below:

```
typedef struct lop_specs
               {
                       entity_name entity;
                       entity_manipulation stmt_effect;
                       entity_occurrence exposition;
               } lp_specs;
```

Local data flow analysis searches for the effect of a given statement specified through `stmt_effect` and stores it in `add_entities`. If the specified `stmt_effect` is `entity_use`, the entities that qualify for `entity_mod` are stored in the variable `remove_entities`. Depending upon the `exposition`, the final decision of removal is taken.

Thus computation of $\mathit{Gen}_n$ and $\mathit{Kill}_n$ depends upon setting up a variable of the type `lp_specs` and the solving the following recurrence

$$\begin{aligned}\texttt{accumulated\_entities} = (&\texttt{accumulated\_entities} - \texttt{remove\_entities})\\ &\cup \texttt{add\_entities}\end{aligned}$$

Function `effect_of_a_statement` performs the above computation for a given statement. It is called by the top level function `local_dfa_of_bb`. The relevant code fragment for downwards exposed entities is:

```
FOR_EACH_STMT_FWD
{   stmt = bsi_stmt(bsi);
    accumulated_entities = effect_of_a_statement(lps_given,
                                        stmt, accumulated_entities);
}
```

For upwards exposed entities, the accumulation is against the control flow and the above traversal is performed using the macro `FOR_EACH_STMT_BKD`.

The main limitation of this approach is that it requires independent traversal of a basic block for computing *Gen* and *Kill*. However, by using a slightly more complicated data structure that passes both *Gen* and *Kill* to function `local_dfa_of_bb`, will solve this problem. The other limitation is that due to the generality, there are many checks that are done in the underlying functions. There are two possible solutions to this problem of efficiency:

- This is used as a rapid prototyping tool for a given data flow analysis. Once the details are fixed, one could spend time writing a more efficient data flow analyzer.
- Instead of interpreting the specifications, a program can generate a customized C code that is compiled with GCC source.

### 10.3.5 Global Data Flow Analysis

As observed earlier, implementation of global data flow analyzer is much simpler once local data flow analysis and interface with the underlying compiler infrastructure is in place. The fact that *gdfa* use generic data flow Equations (5.1) and (5.2) makes it possible to execute a wide variety of specifications without having to know the name of a particular analysis being performed. In other words, *gdfa* driver is not a collection of data flow analysis implementations but is capable of executing any specification within the limits of the possible values of specification primitives.

At the top level, the *gdfa* driver needs to perform the following tasks:

- Create special values like $\top$, $BI_{Start}$, and $BI_{End}$.
- Create space for data flow values
- Perform local data flow analysis
- Select flow functions
- Perform global data flow analysis

Function `gdfa_driver` performs the above tasks:

```
 0 pfbv_dfi **
 1 gdfa_driver(struct gimple_pfbv_dfa_spec dfa_spec)
 2 {
 3     if (find_entity_size(dfa_spec) == 0)
 4             return NULL;
 5     initialize_special_values(dfa_spec);
 6     create_dfi_space();
 7     traversal_order = dfa_spec.traversal_order;
 8     confluence = dfa_spec.confluence;
 9
10     local_dfa(dfa_spec);
11
12     forward_edge_flow = dfa_spec.forward_edge_flow;
13     backward_edge_flow = dfa_spec.backward_edge_flow;
14     forward_node_flow = dfa_spec.forward_node_flow;
15     backward_node_flow = dfa_spec.backward_node_flow;
16
17     perform_pfbvdfa();
18
19     preserve_dfi(dfa_spec.preserved_dfi);
20     return current_pfbv_dfi;
21 }
```

Lines 12 to 15 select the flow functions from the specifications. Below we show the code fragment of function `perform_pfbvdfa` when the direction of traversal is `FORWARD`.

```
do
{   iteration_number++;
    change = false;
    FOR_EACH_BB_IN_SPECIFIED_TRAVERSAL_ORDER
    {   bb = VARRAY_BB(dfs_ordered_basic_blocks,visit_bb);
        if(bb)
        {   if (traversal_order == FORWARD)
            {   change_at_in = compute_in_info(bb);
                change_at_out = compute_out_info(bb);
                change = change||change_at_out||change_at_in;
            }
            else  /* compute in the opposite order */
        }
    }
} while(change);
```

The main code fragment of function `compute_in_info` is as shown below. It calls function `backward_node_flow` which is extracted from the specification.

```
if (!bb->preds)
    temp = combine(entry_info, backward_node_flow(bb));
else
    temp = combine(combined_forward_edge_flow(bb),
                                 backward_node_flow(bb));
old = CURRENT_IN(bb);
change = is_new_info(temp,old);

if (change)
{
     CURRENT_IN(bb) = temp;
     if (old)
         free_dfvalue_space(old);
}
return change;
```

Function `combined_forward_edge_flow` computes the following term

$$\prod_{p \in \mathit{pred}(n)} \overrightarrow{f}_{p \to n}(\mathit{Out}_p)$$

Its main code fragment is shown below. It calls function `forward_edge_flow` which is extracted from the specification.

```
edge_vec = bb->preds;
temp = make_initialized_dfvalue(top_value_spec);

if (forward_edge_flow == &stop_flow_along_edge)
    return temp;

FOR_EACH_EDGE(e,ei,edge_vec)
{   pred_bb = e->src;
    new = combine(temp,forward_edge_flow(pred_bb,bb));
    if (temp)
        free_dfvalue_space(temp);
    temp = new;
}
return temp;
```

The code sequence corresponding to function `compute_out_info` is an exact dual of the above code sequence. This completes the description of generic global data flow analysis in *gdfa*.

## 10.4 Extending the Generic Data Flow Analyzer *gdfa*

Many extensions and enhancements of *gdfa* are possible. We suggest some of them by dividing them into the following categories.

- *Extensions that do not require changing the architecture of gdfa.*
  - Include space and time measurement of analyses.
  - Consider scalar formal parameters for analysis.
  - Support a work list based driver.
  - Extend *gdfa* to support other entities such as statements (e.g., for data flow analysis based program slicing), and basic blocks (e.g., for data flow analysis based dominator computation). Both these problems are bit vector problems.
  - Improve the implementation of *gdfa* to make it more space and time efficient. This may require compromising on the simplicity of the implementation but generality should not be compromised.
- *Extensions that may require minor changes to the architecture of gdfa.*
  - Implement incremental data flow analysis and measure its effectiveness by invoking in just before gimple is expanded into RTL.

    This would require a variant of a work list based driver.
  - 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 variables analysis, possibly undefined variables, analysis, strongly live variables analysis.

    This would require changing the local data flow analysis. One possible option is using matrix based local property computation [53]. The other option is to treat a statement as an independent basic block.
- *Extensions that may require major changes to the architecture of gdfa.*
  - Extend *gdfa* to non-separable frameworks in which data flow information cannot be represented by bit vectors e.g., constant propagation, signs analysis, points-to analysis, alias analysis, heap reference analysis etc. Although the main driver would remain same, this would require making fundamental changes to the architecture.
  - Extend *gdfa* to support some variant of context and flow sensitive interprocedural data flow analysis.