Symbolic Reaching Definitions Analysis of Ada Programs

Johann Blieberger and Bernd Burgstaller

Department of Automation (183/1),
Technical University Vienna,
Treitlstr. 1/4, A-1040 Vienna, Austria
{blieb,bburg}@auto.tuwien.ac.at

Abstract. A data-flow framework for symbolic reaching definitions analysis is presented. It produces a more accurate solution of the reaching definitions problem than can be achieved with "classic" data-flow analysis. This is crucial for applications in the area of real-time, embedded, and safety-related systems.

1 Introduction

The *Reaching Definitions Problem* is a data-flow problem used to answer the following questions: Which definitions of a variable X reach a given use of X in an expression? Is X used anywhere before it is defined?

The underlying program presentation with data-flow problems is usually the *control flow graph (CFG)*, a directed labelled graph. Its nodes are the program's basic blocks (a basic block is a single entry, single exit, sequence of statements), whereas its edges represent transfers of control between basic blocks. *Entry* and *Exit* are distinguished nodes used to denote start and terminal node.

Traditional treatments of the Reaching Definitions Problem (confer [9]) utilise a set-based approach that aims at determining the set *Reach*(B) of variable definitions that reach a given CFG node B. We have the following equations:

$$\begin{aligned}\mathrm{Reach}(B) &= \bigcup_{B' \in \mathrm{Preds}(B)} [(\mathrm{Reach}(B') \cap \mathrm{Pres}(B')) \cup \mathrm{Gen}(B')] \\ \mathrm{Reach}(\rho) &= \emptyset,\end{aligned}$$

where $\mathrm{Reach}(B)$ is the set of definitions reaching the top of B, $\mathrm{Preds}(B)$ is the set of *predecessors* of B, $\mathrm{Pres}(B)$ is the set of definitions *preserved* through B (that is, not superseded by more recent definitions), and $\mathrm{Gen}(B)$ is the set of definitions *generated* in B.

Figure 2 shows the CFG for the program fragment given in Figure 1. The comments added at statements in Figure 1 indicate in which node of the control flow graph they are contained. Note that an extraneous edge from node *Entry* to node *Exit* has been inserted which has no correspondence to the actual data-flow in procedure **Aha**, it is only present to simplify algorithms based on the CFG.

The set of equations for the reaching definitions problem of the example program is shown in Table 1 where we have written X_i instead of $\mathrm{Reach}(B_i)$.

```
procedure Aha is
   h,j : integer;                      -- Node 1
begin
  if false then                        -- Node 1
     j := 0;                           -- Node 2
  end if;
  for i in 1 .. 10 loop          -- Node 3 and 4
     h := h + i;                       -- Node 5
  end loop;
end Aha;
```

Fig. 1. Simple Example Procedure

We will use this set of equations in the following to solve the reaching definitions problem with an *iteration algorithm* (confer [1]), i.e., we initialise the variables to $\emptyset$, insert these values on the right side of the equations and use the new values for the same purpose until the process stabilises.

The results are shown in Table 2. As can be seen the algorithm "converges" very fast, which is a great advantage of iteration algorithms. However, it is too optimistic in its assumptions:

- The definition of j at node 2 never occurs.
- Variable h is undefined at the beginning. Thus its value after executing procedure **Aha** is undefined too.

Both cases represent serious programming errors which cannot be detected with present data-flow algorithms. The reason for this is that "classic" data-flow analysis assumes that each edge of the CFG definitely is followed during execution of the program.

For this reason we define a data-flow framework that incorporates more information in order to allow more precise program analysis. Our data-flow framework is superior to standard techniques (compare [9]) as well as to the more involved information-flow analysis incorporated in SPADE (see [2]). Approaches like ANNA and that of [4] cannot be directly compared to ours since they are primarily based on annotations. In contrast our approach extracts all information

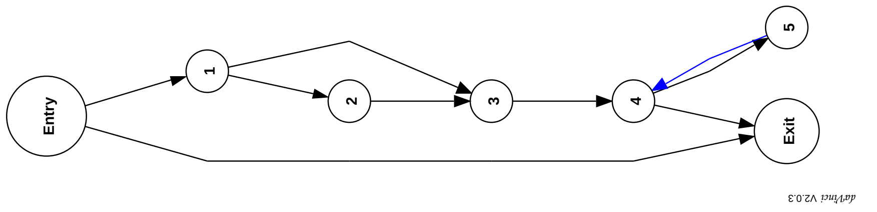

Fig. 2. Control Flow Graph of Simple Example Procedure

$$\begin{aligned}
X_{\text{Entry}} &= \emptyset \\
X_1 &= (X_{\text{Entry}} \cap \{h,j\}) \cup \emptyset \\
X_2 &= (X_1 \cap \{h,j\}) \cup \emptyset \\
X_3 &= ((X_1 \cap \{h,j\}) \cup \emptyset) \cup ((X_2 \cap \{h\}) \cup \{j\}) \\
X_4 &= ((X_3 \cap \{h,j\}) \cup \{i\}) \cup ((X_5 \cap \{j\}) \cup \{h,i\}) \\
X_5 &= (X_4 \cap \{h,i,j\}) \cup \emptyset \\
X_{\text{Exit}} &= (X_4 \cap \{h,j\}) \cup \emptyset
\end{aligned}$$

Table 1. Equations for Reaching Definitions Problem of Simple Example Program

from the source code alone. Note also that [4] is based on SPARK (i.e. SPADE) too.

Our approach can easily be incorporated in existing Ada compilers. In fact we have integrated a prototype into GNAT for our purposes.

2 Symbolic Evaluation

Symbolic evaluation is a form of static program analysis in which symbolic expressions are used to denote the values of program variables and computations (cf. e.g. [5]). In addition a path condition describes the impact of the program's control flow onto the values of variables and the condition under which control flow reaches a given program point.

Conditions and the Control Flow Graph

An edge $e = (B', B)$ of the CFG has assigned a condition Cond(B', B) which must evaluate to true for the control flow to follow this edge (e.g. in case of the then-branch of an if-statement Cond is the condition of the if-statement).

Program State and Context

The *state* $\mathcal{S}$ of a program is described by a set of pairs $\{(v_1, e_1), \ldots, (v_m, e_m)\}$ where v_i is a program variable and e_i is a symbolic expression describing the

X_{Entry}	$\emptyset$	$\emptyset$
X_1	$\emptyset$	$\emptyset$
X_2	$\emptyset$	$\emptyset$
X_3	$\emptyset$	$\{j\}$
X_4	$\emptyset$	$\{h,i,j\}$
X_5	$\emptyset$	$\{h,i,j\}$
X_{Exit}	$\emptyset$	$\{h,j\}$

Table 2. Results of Reaching Definitions Problem of Simple Example Program

value of v_i for $1 \leq i \leq m$. For each variable v_i there exists exactly one pair (v_i, e_i) in $\mathcal{S}$.

A program consists of a sequence of statements that may change $\mathcal{S}$.

A *path condition* specifies a condition that is valid at a certain program point. If conditional statements are present, there may be several different valid program states at the same program point. A different path condition is associated with each of them.

States $\mathcal{S}$ and path conditions $\mathcal{C}$ specify a *program context* which is defined by

$$\bigcup_{i=1}^{k} [\mathcal{S}_i, \mathcal{C}_i]$$

where k denotes the number of different program states valid at a certain program point. (The $\cup$ and $\bigcup$-operators are used to enumerate different program states.) A program context completely describes the variable bindings at a specific program point together with the associated path conditions.

3 A Data-Flow Framework for Symbolic Evaluation

We define the following set of equations for the symbolic evaluation framework:

$$\mathrm{SymEval}(B_{\mathrm{Entry}}) = [\mathcal{S}_0, \mathcal{C}_0],$$

where $\mathcal{S}_0$ denotes the initial state containing all variables which are assigned their initial values, and $\mathcal{C}_0$ is true,

$$\mathrm{SymEval}(B) = \bigcup_{B' \in \mathrm{Preds}(B)} \mathrm{PrpgtCond}(B', B, \mathrm{SymEval}(B')) \mid \mathrm{LocalEval}(B),$$

where $\mathrm{LocalEval}(B) = \{(v_{i_1}, e_{i_1}), \ldots, (v_{i_m}, e_{i_m})\}$ denotes the symbolic evaluation local to basic block B. The variables that get a new value assigned in the basic block are denoted by $v_{i_1}, \ldots, v_{i_m}$. The new symbolic values are given by $e_{i_1}, \ldots, e_{i_m}$. The *propagated conditions* are defined by

$$\mathrm{PrpgtCond}(B', B, \mathrm{PC}) = \begin{cases} \mathrm{Cond}(B', B) \odot \mathrm{PC}, & \text{if } B' \text{ has} \geq 1 \text{ successors,} \\ \mathrm{PC}, & \text{otherwise.} \end{cases}$$

Denoting by PC a program context, the operation $\odot$ is defined as follows:

$$\begin{aligned} \mathrm{Cond}(B', B) \odot \mathrm{PC} = \mathrm{Cond}(B', B) \odot [\mathcal{S}_1, p_1] \cup \ldots \cup [\mathcal{S}_k, p_k] = \\ [\mathcal{S}_1, \mathrm{Cond}(B', B) \wedge p_1] \cup \ldots \cup [\mathcal{S}_k, \mathrm{Cond}(B', B) \wedge p_k], \end{aligned}$$

i.e., the $\odot$-operator is used as a placeholder for path conditions of currently unknown program contexts.

In the following we state certain rules which have to be applied to perform symbolic evaluation[†]:

[†] Rules 1, 2, and 3 constitute an informal definition of the |-operator used in the SymEval equations.

1. If a situation like
$$\{\dots,(v,e_1),\dots\} \mid \{\dots,(v,e_2),\dots\},$$
is encountered during symbolic evaluation, we replace it with
$$\{\dots,(v,e_2),\dots\}.$$
The pair (v,e_1) is not contained in the new set.
2. If a situation like
$$\{\dots,(v_1,e_1),\dots\} \mid \{\dots,(v_2,e_2(v_1)),\dots\},$$
where $e(v)$ denotes an expression involving variable v, is encountered during symbolic evaluation, we replace it with
$$\{\dots,(v_1,e_1),\dots,(v_2,e_2(e_1)),\dots\}.$$
3. If a situation like
$$\{\dots,(v,e),\dots\} \mid \{\dots,(v,v(\perp,\omega)),\dots\}$$
is encountered during symbolic evaluation, we replace it with
$$\{\dots,(v,v(e,\omega)),\dots\}.$$
The pair (v,e) is not contained in the new set.
The notation $v(v_0,\omega)$ is defined in Section 4.
For the situations discussed above it is important to apply the rules in the correct order, which is to elaborate the elements of the right set from left to right.
4. If a situation like
$$[\{\dots,(v,e),\dots\},C(\dots,v,\dots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\dots,(v,e),\dots\},C(\dots,e,\dots)].$$
5. If a situation like
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v,\dots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v_0,\dots)].$$
6. If a situation like
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v(\perp,\omega),\dots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\dots,(v,v(v_0,\omega)),\dots\},C(\dots,v(v_0,\omega),\dots)]$$

This data-flow framework has been introduced in [3].

4 Solving the Symbolic Evaluation Data-Flow Framework

Unfortunately the definition of PrpgtCond(...) prevents the symbolic evaluation framework from being bounded. Thus it cannot be solved by iteration algorithms (compare [9, 7, 8]).

Nevertheless we can solve symbolic evaluation frameworks with help of *elimination algorithms* (see [11, 10]). Note that the set of equations ($i = 1, \ldots, n$)

$$\{E_i : x_i = W_i(x_{i_1}, \ldots, x_{i_{n_i}})\} \tag{1}$$

implies a dependency relation on the variables x_i. We say that x_i on the left side *depends* on all variables on the right side. If the corresponding dependency graph is acyclic, set (1) can be solved by simple insertions, thereby eliminating one variable after the other. If it contains cycles, insertions alone are not enough to obtain a solution. However, if a rule is available for replacing such an equation with one in which the left variable does not appear on the right, with a guarantee that any solution to this new equation set will satisfy the original, then it becomes possible to move the elimination process forward. Such a rule is called *loop-breaking rule.*

The loop-breaking rule for SymEval equations is defined as follows ([3]): Assume we have the following equation (we use X_i for SymEval(B_i) as a shorthand)

$$\begin{aligned}E_i : X_i = &\bigcup_{1 \le j \le r} (C_j \odot X_i) | \{(v_{j_1}, e_{j_1}), \ldots, (v_{j_s}, e_{j_s})\} \cup \\ &\bigcup_{1 \le k \le t,\ 1 \le m \le n,\ m \ne i} (C_k \odot X_m) | \{(v_{k_1}, e_{k_1}), \ldots, (v_{k_u}, e_{k_u})\},\end{aligned}$$

then we replace it with

$$\begin{aligned}e_i : X_i = &\neg\text{LoopEntryCond}_i \odot \\ &\bigcup_{1 \le k \le t,\ 1 \le m \le n,\ m \ne i} (C_k \odot X_m) | \{(v_{k_1}, e_{k_1}), \ldots, (v_{k_u}, e_{k_u})\} \cup \\ &\text{LoopEntryCond}_i \odot \\ &\bigcup_{1 \le k \le t,\ 1 \le m \le n,\ m \ne i} ((C_k \odot X_m) | \{(v_{k_1}, e_{k_1}), \ldots, (v_{k_u}, e_{k_u})\}) | \text{LoopExit},\end{aligned}$$

where LoopEntryCond$_i$ denotes the condition which has to be true to enter the loop body starting at basic block B_i, and

$$\text{LoopExit} = \{(v_{J_1}, v_{J_1}(\bot, \omega_\ell)), \ldots, (v_{J_t}, v_{J_t}(\bot, \omega_\ell))\}$$

for all variables v_{J_p} being contained in

$$\bigcup_{1 \le j \le r} \{(v_{j_1}, e_{j_1}), \ldots, (v_{j_s}, e_{j_s})\}.$$

Note that the first term of e_i mirrors the case when the loop body is not executed at all, and the second term treats the case when the loop body is executed at least one times.

The purpose of our loop-breaking rule is to replace a loop by a *set of recurrence relations.* Each induction variable (cf. [1]) gives raise to an (indirect) recursion. Let v be such a variable, then $v(v_0, \omega_\ell)$ denotes the symbolic solution of the recursion, where v_0 is a suitable initial value and ω_ℓ denotes the number of iterations of loop ℓ‡. If no initial value is known or the initial value is irrelevant to the solution, we write $v(\perp, \omega_\ell)$.

Setting up recurrence relations during loop-breaking is described in the following. If there are nested loops in the source code of interest, we start by setting up recurrence relations from the innermost loop and proceed to the outermost§.

Let v denote a variable, then we call $v(k)$ its *recursive counterpart*, where k is a variable that does not occur in the program being evaluated..

According to the notation above we set up a recurrence relation for all $1 \leq j \leq r$, $1 \leq q \leq s$ and for $k \geq 0$ by

$$v_{j_q}(k+1) = e_{j_q}(k) \qquad \text{if } C_j(k) \text{ evaluates to true,}$$

where $e_{j_q}(k)$ and $C_j(k)$ means that all variables contained in e_{j_q} and C_j are replaced with their recursive counterparts.

Note that we have not specified initial values for the recursion; these are supposed to be supplied by situations handled by the rules given in Section 3.

We have used and implemented an algorithm described in [12] for solving symbolic evaluation frameworks. It solves data-flow equations in $O(\log |N| \cdot |E|)$ insertions and loop-breaking operations, where $|N|$ denotes the number of nodes in the CFG and $|E|$ is the number of edges of the CFG. The CFG is supposed to be reducible, which is true for all Ada programs. Because of the undecidability of the halting problem, however, we cannot give time bounds for solving the recurrence relations produced by the loop-breaking rule.

5 Symbolic Evaluation of the Simple Example

According to the algorithm given in [12] we now solve the equations given in Table 3. We write "$a \to b$" for indicating that equation E_a is inserted into E_b and we write "$c \circlearrowleft\!\!\!\!/$" for loop-breaking equation E_c.

We denote an undefined value by "$\perp$". Furthermore we assume that an undefined value involved in an operation such as "+" or "−" results in an undefined value again. Assigning an undefined value to variable v results in v being undefined.

The equations of Table 3 are derived in a straight-forward manner. We would only like to mention that variable i is implicitly declared in the for-loop and thus does only appear at nodes 3 and 5.

In the following derivation terms with path conditions equal to $false$ can be ignored, which we do sometimes without further notice.

‡ Each loop gets assigned a unique number $\ell \in \mathbb{N}$.

§ This is guaranteed by the algorithm described in [12].

$$\begin{array}{ll}
X_{\text{Entry}} & = [\{(h,\perp),(j,\perp)\},\text{true}] \\
X_1 & = X_{\text{Entry}} \\
X_2 & = \mathit{false} \odot X_1 \mid \{(j,0)\} \\
X_3 & = \mathit{true} \odot X_1 \mid \{(i,1)\} \cup \mathit{true} \odot X_2 \mid \{(i,1)\} \\
X_4 & = X_3 \cup X_5 \\
X_5 & = (1 \le 10) \odot X_4 \mid \{(h,h+i),(i,i+1)\} \\
X_{\text{Exit}} & = \mathit{true} \odot X_4
\end{array}$$

Table 3. Set of SymEval Equations for Simple Example Program

$5 \to 4$

$$X_4 = X_3 \cup (1 \le 10) \odot X_4 \mid \{(h,h+i),(i,i+1)\}$$

$4 \not\circlearrowleft$

$$X_4 = \neg(1 \le 10) \odot X_3 \cup (1 \le 10) \odot X_3 \mid \{(h,h(\perp,\omega))\}$$

$4 \to$ **Exit**

$$X_{\text{Exit}} = \neg(1 \le 10) \odot X_3 \cup (1 \le 10) \odot X_3 \mid \{(h,h(\perp,\omega))\}$$

$3 \to$ **Exit,** $1 \to$ **Exit, Entry** $\to$ **Exit**

$$\begin{aligned}
X_{\text{Exit}} &= \neg(1 \le 10) \odot [\{(h,\perp),(j,\perp)\},\mathit{true}] \cup \\
&\quad (1 \le 10) \odot [\{(h,h(\perp,\omega)),(j,\perp)\},\mathit{true}] \\
&= [\{(h,\perp),(j,\perp)\},\mathit{false}] \cup [\{(h,h(\perp,\omega)),(j,\perp)\},\mathit{true}]
\end{aligned}$$

Next we solve the recurrence relation for variable i. The recursion is

$$\begin{aligned} i(1) &= 1, \\ i(k+1) &= i(k)+1. \end{aligned}$$

Clearly its solution is

$$i(k) = k$$

for $k \ge 1$.

The recurrence relation for variable h reads

$$\begin{aligned} h(1) &= \perp, \\ h(k+1) &= h(k)+i(k) = h(k)+k \end{aligned}$$

Its solution is

$$h(k) = \perp$$

for $k \ge 1$.

Hence we finally get

$$X_{\text{Exit}} = [\{(h,\perp),(j,\perp)\},\mathit{true}]$$

which correctly mirrors the fact that both j and h are undefined after procedure **Aha** has been executed.

Performing further insertions more detailed information can be derived. For example inserting Entry $\to$ 1, $1 \to 3$, and $3 \to 4$, we see that variable h is used before it is defined in Node 4.

```
procedure Foo (E : in Integer; Y : out Integer) is
   Index, I, R : Integer;                    -- Node 1
begin
   Index := E;                               -- Node 1
   while Index > 0 loop                      -- Node 2
      if Index mod 2 = 0 then                -- Node 3
         I := E + 1;                         -- Node 4
      end if;
      Index := Index - 1;                    -- Node 5
      R := I + Index;                        -- Node 5
   end loop;
   Y := R + 1;                               -- Node 6
end Foo;
```

Fig. 3. Hypothetical Example Procedure

6 A More Complicated Example

Figure 4 shows the CFG for the program fragment given in Figure 3.

The SymEval equations for this hypothetical example are shown in Table 4. Note that we can restrict our interest to the variables *Index*, *I*, *R*, and *Y* because *E* cannot be overwritten within procedure **foo** (compare [6] for semantic details).

Again we solve this set of equations according to the algorithm given in [12].

$4 \rightarrow 5$

$$\begin{aligned} X_5 = & (((Index \bmod 2 = 0) \odot X_3) \mid \\ & \quad \{(I, E+1), (Index, Index-1), (R, I+Index-1)\}) \\ & \cup((\neg(Index \bmod 2 = 0) \odot X_3) \mid \\ & \quad \{(Index, Index-1), (R, I+Index-1)\}) \end{aligned}$$

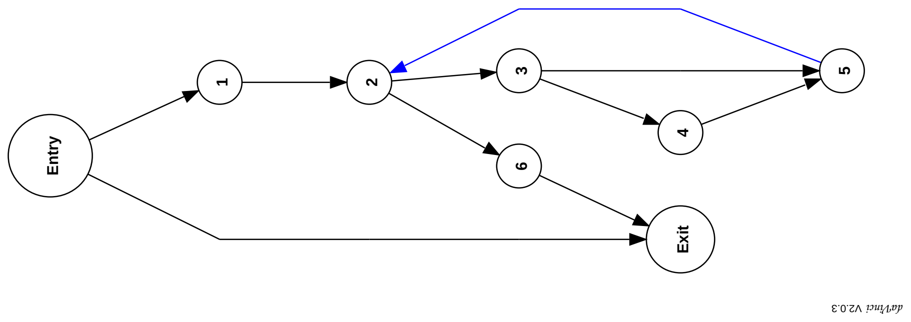

Fig. 4. Control Flow Graph of Hypothetical Example Procedure

$$\begin{array}{ll}
X_{\text{Entry}} & = [\{(\mathit{Index},\bot),(I,\bot),(R,\bot),(Y,\bot)\},\text{true}] \\
X_1 & = X_{\text{entry}} \mid \{(\mathit{Index},E)\} \\
X_2 & = X_1 \cup X_5 \\
X_3 & = ((\mathit{Index} > 0) \odot X_2) \\
X_4 & = ((\mathit{Index} \bmod 2 = 0) \odot X_3) \mid \{(I, E+1)\} \\
X_5 & = (X_4 \mid \{(\mathit{Index}, \mathit{Index}-1), (R, I+\mathit{Index}-1)\}) \cup \\
 & \quad (\neg(\mathit{Index} \bmod 2 = 0) \odot X_3 \mid \{(\mathit{Index}, \mathit{Index}-1), \\
 & \quad\quad (R, I+\mathit{Index}-1)\}) \\
X_6 & = (\neg(\mathit{Index} > 0) \vee \neg(\mathit{Index}(\bot,\omega) > 0)) \odot X_2 \mid \{Y, R+1\} \\
X_{\text{Exit}} & = X_6
\end{array}$$

Table 4. Set of SymEval Equations for Hypothetical Example Program

$5 \rightarrow 2$

$$\begin{aligned}
X_2 = X_1 \cup (((\mathit{Index} \bmod 2 = 0) \odot X_3) \mid \\
\{(I, E+1), (\mathit{Index}, \mathit{Index}-1), (R, I+\mathit{Index}-1)\}) \\
\cup((\neg(\mathit{Index} \bmod 2 = 0) \odot X_3) \mid \\
\{(\mathit{Index}, \mathit{Index}-1), (R, I+\mathit{Index}-1)\})
\end{aligned}$$

$3 \rightarrow 2$

$$\begin{aligned}
X_2 = X_1 \cup (((\mathit{Index} \bmod 2 = 0) \wedge (\mathit{Index} > 0)) \odot X_2) \mid \\
\{(I, E+1), (\mathit{Index}, \mathit{Index}-1), (R, I+\mathit{Index}-1)\}) \cup \\
((\neg(\mathit{Index} \bmod 2 = 0) \wedge (\mathit{Index} > 0)) \odot X_2) \mid \\
\{(\mathit{Index}, \mathit{Index}-1), (R, I+\mathit{Index}-1)\})
\end{aligned}$$

$6 \rightarrow$ **Exit**

$$X_{\text{Exit}} = (\neg(\mathit{Index} > 0) \vee \neg(\mathit{Index}(\bot,\omega) > 0)) \odot X_2 \mid \{Y, R+1\}$$

$2\ \not\circlearrowleft$

$$\begin{aligned}
X_2 = \neg(\mathit{Index} > 0) \odot X_1 \cup \\
(\mathit{Index} > 0) \odot X_1 \mid \{(\mathit{Index}, \mathit{Index}(\bot,\omega)), (I, I(\bot,\omega)), (R, R(\bot,\omega))\}
\end{aligned}$$

$2 \rightarrow$ **Exit,** $1 \rightarrow$ **Exit, Entry** $\rightarrow$ **Exit**

$$\begin{aligned}
X_{Exit} = [\{(\mathit{Index}, E), (I,\bot), (R,\bot), (Y,\bot)\}, \neg(E > 0)] \cup \\
[\{(\mathit{Index}, \mathit{Index}(E,\omega)), (I, I(\bot,\omega)), (R, R(\bot,\omega)), \\
(Y, R(\bot,\omega)+1)\}, (E > 0)]
\end{aligned}$$

We now set up the recurrence relations involved ($k \geq 0$). For *Index* we obtain:

$$\begin{aligned}
\mathit{Index}(0) &= E, \\
\mathit{Index}(k+1) &= \mathit{Index}(k) - 1 \qquad \text{if } \mathit{Index}(k) > 0
\end{aligned}$$

which has the closed form

$$\mathit{Index}(k) = E - k \qquad \text{for } 0 \leq k \leq E,$$

from which we conclude that $\omega = E$.

For variable I we get the following recurrence relation:

$$\begin{aligned} I(0) &= \perp, \\ I(k+1) &= \begin{cases} \text{E+1 if } (\mathit{Index}(k) > 0 \wedge (\mathit{Index}(k) \bmod 2 = 0), \\ \text{I(k) otherwise.} \end{cases} \end{aligned}$$

Its solution is:

$$\begin{aligned} \text{I}(1) &= \begin{cases} E+1 \text{ if } (E > 0) \wedge (E \bmod 2 = 0), \\ \perp \quad \text{otherwise,} \end{cases} \\ \text{I(k)} &= \text{E+1} \qquad \text{for } k \geq 2. \end{aligned} \tag{2}$$

For variable R we derive

$$\begin{aligned} &R(0) = \perp, \\ &R(1) = I(1) + \mathit{Index}(1) = \begin{cases} 2E \text{ if } (E > 0) \wedge (E \bmod 2 = 0), \\ \perp \text{ otherwise,} \end{cases} \\ &R(k) = I(k) + \mathit{Index}(k) = (E+1) + (E-k) = 2E - k + 1 \qquad \text{for } k \geq 2. \end{aligned}$$

This implies

$$R(\omega) = \begin{cases} \perp & \text{if } E = 1, \\ E+1 & \text{if } E \geq 2. \end{cases} \tag{3}$$

Finally, we derive

$$Y = R(\omega) + 1 = \begin{cases} \perp & \text{if } E = 1, \\ E+2 & \text{if } E \geq 2. \end{cases} \tag{4}$$

Inserting these results into equation X_{Exit} we obtain the following context

$$\begin{aligned} X_{Exit} = {} & [\{(\mathit{Index}, E), (I, \perp), (R, \perp), (Y, \perp)\}, \neg(E > 0)] \cup \\ & [\{(\mathit{Index}, 0), (I, (2)), (R, (3)), (Y, (4))\}, (E > 0)], \end{aligned}$$

where the results from equations (2), (3), and (4) are incorporated at the indicated places.

From this we see that

1. I, R, and Y are undefined if $\neg(E > 0)$, which means that the loop body has not been executed at all,
2. I, R, and Y are undefined if $E = 1$ and
3. I is undefined if $\neg(E \bmod 2 = 0)$ during the *first* iteration of the loop.

Note that case (3) is a transient fault which is propagated to Y only if $E = 1$. If for example $E = 3$, I and R are undefined during the first iteration. However this is "repaired" during the second iteration where I and R are assigned proper values.

7 Conclusion

Employing symbolic evaluation for reaching definitions analysis produces more accurate solutions than can be achieved with "classic" data-flow algorithms. We are currently investigating the implications of the *interprocedural* reaching definitions problem on our approach.

Symbolic evaluation can also be used for detecting dead paths and for improving other important data-flow properties of programs. It can also be employed for determining the *worst-case execution time* of (real-time) programs (see [3]). These properties are crucial for applications in the area of real-time, embedded, and safety related systems.

We have implemented the algorithm presented in [12] for almost all control flow affecting language features of Ada. At the current stage our implementation does not support exceptions and tasking. An implementation of the symbolic evaluation data-flow framework is under way. As already mentioned our prototype implementation is integrated into GNAT.

References

1. A. V. Aho, R. Seti, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.
2. J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):37–61, 1985.
3. J. Blieberger. Data-flow frameworks for worst-case execution time analysis. (submitted), 1997.
4. R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.
5. T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans. on Software Engineering*, 5(4):403–417, July 1979.
6. ISO/IEC 8652. *Ada Reference manual*, 1995.
7. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.
8. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
9. T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks – a unified model. *Acta Informatica*, 28:121–163, 1990.
10. M. C. Paull. *Algorithm Design – A Recursion Transformation Framework.* Wiley Interscience, New York, NY, 1988.
11. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, 1986.
12. V. C. Sreedhar. *Efficient Program Analysis Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995.

This article was processed using the LaTeX macro package with LLNCS style