Efficient Implementation of Detection of Undefined Variables

Z. J. CZECH

Institute of Computer Science, Technical University of Silesia, Pstrowskiego 16, 44–100 Gliwice, Poland

Two algorithms for solving the problem of detecting undefined (or uninitialised) variables during compilation are considered. The first, well-known algorithm solves the problem by computing the use-definition chains for a program flow graph. Its time complexity is $O(|N|^2)$ where $|N|$ is a number of nodes of the flow graph. An $O(|N|)$ algorithm is proposed that analyses the direct acyclic graph of a reducible flow graph. The implementations of both algorithms in an Ada compiler are evaluated and compared.

> The number of programming languages in use today is very large ... The number of high-quality language implementations, however, is quite small. – W. A. Wulf[26]

Received October 1985, revised April 1987

1. INTRODUCTION

One of the most common programming bugs is the use of undefined (or uninitialised) variables. However, it is rare to find a language implementation that detects references to undefined variables, or support programmers to detect them. Even in the Ada language, which was designed with program reliability as one of the (three) overriding concerns, the demand for recognition of undefined variables was eventually withdrawn. (According to the requirement contained in the Steelman document,[7] the use of an undefined value raised the NO_VALUE_ERROR exception in the Green language[18] and the preliminary Ada,[11] but this exception has been removed from the later versions of the language.[19,21])

There are two methods of solving the problem of detecting undefined variables (abbreviated as *UV problem*). Both of them are run-time methods; that is, they detect the undefined variables during the execution of a program. In the first method, a compiler inserts into the object program run-time checks for testing whether a value of a variable is defined every time it is referenced. The second method is based on using special hardware to detect that a variable being referenced during the execution of a program has not been previously defined.

Unfortunately, neither method is suitable for most practical implementations. The first causes a great degradation of a program's performance, because of the amount of run-time checking; the second is not general enough, as only a few machines have hardware that allows detection of undefined variables.[17]

This paper considers an efficient method of detecting undefined variables during compilation. Although, as we will see, the method does not detect all references to undefined variables, it can help a programmer to discover most of them before a program execution.

The paper consists of six sections. Section 2 contains the basic definitions and results for program flow graphs. The *UV* problem is formulated in Section 3. Section 4 considers the method of solving the *UV* problem by computing the use-definition chains for a flow graph. An efficient method of solving the problem by analysing the direct acyclic graph of a flow graph is described in Section 5. Section 6 contains an evaluation of implementations of these methods in an Ada compiler.

Present address: University of California, Santa Barbara, California 93106.

2. PRELIMINARIES

For detecting undefined variables at compile-time we will use a model of a program, called a *program control flow graph* (or *flow graph*). First, we partition the program into *basic blocks*; that is, maximal sequences of consecutive statements that may be entered only at the beginning, and when entered are executed sequentially, without branches. A *flow graph* is a triple $G = (N, E, n_o)$, where

N is a finite set of *nodes*, representing the basic blocks of the program;

E is a finite set of edges, i.e. ordered pairs of nodes (n_i, n_j) representing the flow of control;

node $n_o \in N$ is the *initial node*; it represents the block containing the first statement of the program.

There is a directed edge from node n_i to node n_j if the basic block represented by n_j could be executed immediately after that represented by n_i We say that n_i is a *predecessor* of n_j, and n_j is a *successor* of n_i. Fig. 1 shows an example program and its flow graph $G = (\{1, 2, 3, 4,$

(*a*)

```
begin
  GET(A); GET(B);
  if A ≥ B then
    D: = A – B;
  elsif A < B then
    D: = B – A;
  else
    E: = D;
  end if;
end;
```

(*b*)

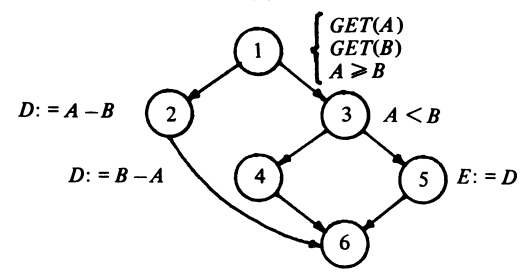

Fig. 1. The example program *(a)* and its flow graph *(b)*

5, 6}, {(1, 2), (1, 3), (2, 6), (3, 4), (3, 5), (4, 6), (5, 6)}, 1).

Let us define the sets of predecessors $P[n]$ and successors $S[n]$ of node n in a flow graph G as follows: $P[n] = \{x \in N \mid (x, n) \in E\}$, $S[n] = \{y \in N \mid (n, y) \in E\}$.

A *path* from n_1 to n_k is a sequence of nodes $(n_1, n_2, \ldots, n_k)$, where each n_i is an immediate predecessor of n_{i+1} for $1 \leqslant i \leqslant k-1$.

In the definition of a flow graph G we require that there should be a path from n_0 to every node in N. We say that node m *dominates* node n in G if every path from the initial node of G to n contains m. Edge (n, m) in G is *backward* if and only if either $n = m$ or m dominates n in G.

A *cycle* is a path that begins and ends at the same node. The path is a *simple* (or *cycle-free*) *path* if the nodes in the path are distinct. A *directed acyclic graph* (*dag*) *of a flow graph* $G = (N, E, n_0)$ is an acyclic flow graph $D = (N, E', n_0)$, such that $E' \subseteq E$ and for any edge e in $E-E'$, $(N, E' \cup \{e\}, n_0)$ is not a *dag*. That is, D is a maximal acyclic subflowgraph.[8]

We define a flow graph $G = (N, E, n_0)$ to be a *reducible flow graph* (*rfg*) if and only if it can be decomposed *uniquely* into a *dag* $D = (N, E', n_0)$ and a set of backward edges $E-E'$; otherwise, the flow graph G is *non-reducible*. (The empirical studies have shown that flow graphs arising from 'real' computer programs are almost always reducible, i.e. more than 95% of the time.[2,16,14] It has also been shown that any non-reducible graph can be transformed to a reducible one by a process known as *node splitting*.[2,3,20])

A *depth-first spanning tree* (DFST) of a flow graph G is a directed, rooted, ordered spanning tree grown by Algorithm[11,21,4,9] given below. This algorithm also defines an ordering on the nodes of DFST of G that we call *rPostorder*, i.e. reverse Postorder.

Algorithm 1

rPostorder computation by depth-first search

Input: A flow graph G whose nodes are numbered from 1 to $|N|$ in an arbitrary manner.

Output: (1) A DFST for G. (2) A numbering rPostorder of the nodes from 1 to $|N|$, in array *rPOSTORDER*, indicating the reverse of the order in which each node was last visited during a depth-first search of G.

Method: Initially, all nodes are marked 'unvisited'. There is a global integer array *rPOSTORDER* (1 . . $|N|$) and a global integer i with initial value n. The algorithm consists of a call to $DFS(n_0)$, where *DFS* is the recursive procedure defined as follows:

```
recursive procedure DFS(x);
  Mark x 'visited';
  while S[x] / = ∅ loop
    Select and delete a node y from S[x];
    if y is marked 'unvisited' then
      Add edge (x, y) to DFST;
      call DFS(y);
    end if;
  end loop;
  rPOSTORDER(x):= i;
    i: = i−1;
return; □
```

It may be shown[5] that after the execution of Algorithm 1 the condition $rPOSTORDER(n) \geqslant rPOSTORDER(m)$ holds for each backward edge (n, m) of G.

We will evaluate each algorithm given in this paper by establishing the time needed to solve a problem as a function of the quantity of input data. This function is called the *time complexity* of the algorithm. The time will be expressed in terms of the number of elementary operations (computational steps) executed in the algorithm, for example logical operations on sets. Because all algorithms operate on graphs, the number of nodes and/or edges will be the measure of the quantity of input data. For the approximate evaluation of time complexity functions we will use O-notation.[15,4]

It has been shown that Algorithm 1 requires $O(MAX(|N|, |E|))$ steps on a graph with $|N|$ nodes and $|E|$ edges.[4]

4. FORMULATION OF THE UV PROBLEM

By a *use* of (or reference to) variable A we mean any occurrence of A as an operand. By a *definition* of A we mean a statement that can modify the value of A, such as an assignment or a *GET* statement.

Let us consider the use of a variable A within node n that is not preceded by any definition of A in n. A variable A used in n *might* be undefined if there is a simple (cycle-free) path from the entry of the initial node of a flow graph to the entry of node n, such that no definition of A appears on the path.

Note that we cannot assert that variable A referenced in n *is* undefined because the path that does not contain any definition of A may involve some tests that can never be simultaneously satisfied, so that path could never be taken during the program execution. For example, in Fig. 1(*b*) there is a path from the initial node 1 to the node 5 in which variable D is used, and there is no definition of D on this path. Thus, one might say that variable D in node 5 is undefined. However, the path (1, 3, 5) contains tests $A \geqslant B$ and $A < B$ that cannot be simultaneously satisfied, so this path will never be traversed.

To simplify the problem we shall assume that all paths in a flow graph are traversable. As the result, the warning will be given for the program of Fig. 1(*b*) that variable D used in node 5 might be undefined, though this is not true in practice.

A more severe drawback of this method of solving the UV problem during compilation is that it is virtually impossible to find all references to undefined variables. Consider the following program:

```
declare
  I: INTEGER
  A: array (1 .. 10) of INTEGER;
begin
  A (5):= 0;
  GET (I);
  PUT A(I);
end;
```

Because the value of I is unknown at compile time, we cannot say whether the indexed variable $A(I)$ referenced in the *PUT* statement is defined or not. It will be defined, of course, only when the value of I is 5.

In the sequel we assume that an indexed variable might be undefined if there is a simple path from the initial node of a flow graph to the use of the indexed variable, such that no definition of any indexed variable for the

same array appears on the path. Consequently, no warning will be given for the last program.

4. SOLVING THE UV PROBLEM BY COMPUTING THE UD-CHAINS

Uses of undefined variables might be found by computing the *use-definition* (or *ud-*) *chains*.[5] For a given use of variable A in node n, the ud-chain is a list containing all definitions that define the value of A used in n. If we introduce dummy definitions of all variables prior to the initial node of a flow graph, and the ud-chain for a given use of A in n contains the dummy definition of A, it means that the value of A might be undefined at this node.

In order to compute the ud-chains we have to solve one of the *data-flow analysis* problems, namely, the *reaching definitions problem*. Its goal is to determine, for each node n of a flow graph, the set $IN[n]$ of definitions that reach the entry of n. A definition d of variable A in node n_1 is said to *reach* the entry of node n_2 if (a) variable A is not redefined in n_1, and (b) there is a path from (the exit of) n_1 to the entry of n_2 such that no definition of A appears on the path.

It follows from these conditions that while determining the IN sets we need consider only the last definition of a variable in each node, because none of the definitions appearing before the last definition in a node can be contained in any of the IN sets.

Given a node n in a flow graph, the set $IN[n]$ can be defined in terms of the following sets.[5]

$GEN[n]$ is the set of definitions generated within n, i.e. the set of last definitions of variables within node n; and

$KILL[n]$ is the set of definitions that are killed within n, i.e. the set of those definitions outside n that define variables that also have definitions within n.

The sets $GEN[n]$ and $KILL[n]$ which can be computed from local (basic block) information, lead to $2|N|$ simultaneous equations that relate $IN[n]$s and $OUT[n]$s for a flow graph G of $|N|$ nodes:

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$
$$IN[n] = \bigcup_{p \in P[n]} OUT[p] \qquad (1)$$

The sets $OUT[n]$ are analogous to the $IN[n]$ but contain the definitions that reach the exit of node n.

The simplest algorithm to solve this set of equations is an iterative algorithm given below. (According to M. S. Hecht,[10] this algorithm was probably first used by V. A. Vyssotsky as part of a compile-time diagnostic facility for a Bell Laboratories IBM 7090 FORTRAN II compiler in 1961. The analysis of the algorithm can be found in Refs 22, 9 and 13.)

Algorithm 2

The iterative algorithm for the reaching definitions problem

Input: (1) A flow graph $G = (N, E, n_0)$. (2) $KILL[n]$ and $GEN[n]$ for each $n \in N$.

Output: $IN[n]$ *and* $OUT[n]$ for each $n \in N$.

Method:

```
begin
  Initialisation
  for each n∈N loop
    IN[n]:= ∅;
    OUT[n]:= GEN[n];
  end loop;
  CHANGE:= TRUE;  To get the while-loop going.
  while CHANGE loop
    CHANGE:= FALSE;
    for each n∈N loop
      NEWIN:= ∪ OUT[p];   Hold IN[n] in a tem-
              p∈P[n]      porary to check for a
                          change.
      if NEWIN/= IN[n] then
        CHANGE:= TRUE;
        IN[n]:= NEWIN;
        OUT[n]:= (IN[n]-
          KILL[n]) ∪ GEN[n];
      end if;
    end loop;
  end loop;
end; □
```

The algorithm starting with $IN[n] = \varnothing$ for all n 'propagates' definitions to nodes by repeatedly visiting them until the flow of definitions stabilises. The algorithm terminates when none of IN sets has been changed during a pass through nodes of a flow graph.

Once the reaching definitions problem is solved, the ud-chain for a given use of a variable A in node n can be computed as follows.[5]

(1) If a use of A is preceded by a definition of A in n, then only the last definition of A in n prior to this use reaches the use. Thus, the ud-chain for this use consists of only this one definition.

(2) If a use of A is preceded by no definition of A in n, then the ud-chain for this use consists of all definitions of A in $IN[n]$.

As has been mentioned earlier, the ud-chains give the solution to the UV problem. That is, if the ud-chain for any use of variable A contains the dummy definition of A, we conclude that A might be undefined at this point of use.

The main problem in detecting undefined variables by computing the ud-chains is to determine the reaching definitions sets. Although the iterative algorithm is easy to implement, it generally finds those sets in several passes through the nodes of a flow graph. In the worst case the algorithm could take time $O(|N|^2)$ to process a flow graph with $|N|$ nodes. (The average time complexity of Algorithm 2 can be decreased by passing over the nodes of a flow graph in rPostorder,[9] but for some pathological graphs it still takes time $O(|N|^2)$.)

5. EFFICIENT METHOD FOR SOLVING THE UV PROBLEM

An efficient method for solving the UV problem is based on the following observations.

During the computation of ud-chain for a use of variable A in a given node n in G, we take into account all paths that begin in nodes of G containing a definition of A and end in node n. Such paths may consist of any edges of G; in particular, any backward edges. However, in discovering whether the variable A is undefined in node n, we are interested only in those definitions of A that can reach node n along the simple, i.e. acyclic, paths from the initial node of G to node n. In other words, in solving the UV problem we can omit all backward edges (which form cycles) and consider only the *dag* of a flow graph. This simplification allows us to find the reaching

definitions sets for a *dag* of G performing only one pass through its nodes. It must be noted, however, that such an approach is possible only for reducible flow graphs, because only these graphs have the unique *dags*.

Algorithm 3 given below builds the *dag* for a reducible flow graph G.

Algorithm 3

Finding a *dag*

Input: A reducible flow graph $G = (N, E, n_o)$.

Output: The *dag* $D = (N, E', n_o)$ of G.

Method: Our task is to find the set E' of the *dag* by eliminating all backward edges of G from the set E. This can be done in two steps: (1) Compute a number rPostorder of the nodes of G using Algorithm 1. (2) Eliminate the backward edges of G as follows.

begin
  $E' := E$;
  for *all* $(n, m) \in E$ **loop**
    if $rPOSTORDER(n) \geqslant rPOSTORDER(m)$ **then**
      $E' := E' - (n, m)$;
    end if;
  end loop;
end; □

It is easy to see that this algorithm takes time $O(|E|)$ for a flow graph with $|E|$ edges.

Let $IN'[n]$ and $OUT'[n]$ be the sets of definitions that reach the entry and exit of node n in a *dag*, respectively. Those sets can be easily computed by applying equation (1), provided that a node n is processed after all its predecessors. This condition will be satisfied if the nodes of a *dag* are processed in *linear order*, i.e. in order that embeds the partial order established by predecessor–successor relationships. Linear orders are usually found by *topological sorting*[15,3] of nodes of a flow graph (or a *dag*), but it has been shown that rPostorder topologically sorts the *dag* of an *rfg*.[9] Thus we can use this order to find the sets $IN'[n]$ and $OUT'[n]$ in a *dag* as follows:

Algorithm 4

One-pass algorithm for the reaching definitions problem for the *dag* of a flow graph G

Input: (1) The *dag* $D = (N, E', n_o)$ of G. (2) $KILL[n]$ and $GEN[n]$ for $n \in N$.

Output: $IN'[n]$ and $OUT'[n]$ for $n \in N$.

Method:

begin
  $IN'[n_o] := \varnothing$; (1)
  $OUT'[n_o] := GEN[n_o]$; (2)
  for *each* $n \in N - \{n_o\}$ *in rPostorder* **loop**
    $IN'[n] := \bigcup_{p \in P[n]} OUT'[p]$; (3)
    $OUT'[n] := (IN'[n] - KILL[n]) \cup GEN[n]$; (4)
  end loop;
end; □

It is easy to prove that the time complexity of Algorithm 4 is $O(|E| + |N| - 1)$. Namely, assume as the elementary operations in the algorithm the logical operations on sets, such as $\cup$ and $-$. At each node of a flow graph other than the initial node, the computation of

$$\bigcup_{p \in P[n]} OUT'[p]$$

in step 3 requires one less set operation than the number of edges entering that node; and so the entire flow graph requires $|E| - (|N| - 1)$ operations. Two set operations are required to compute $(IN'[n] - KILL[n]) \cup GEN[n]$ in step 4, for a total of $2(|N| - 1)$ operations. Summing the operations in step 3 and 4 we get

$$[|E| - (|N| - 1)] + [2(|N| - 1)] = |E| + |N| - 1.$$

Finally, the UV problem can be solved by executing Algorithms 1, 3 and 4. Algorithm 1 finds a numbering rPostorder of nodes of a flow graph G. This numbering is used in Algorithm 3 to build the *dag* of G, and in Algorithm 4 to find the reaching definitions sets $IN'[n]$ and $OUT'[n]$. The ud-chains for the *dag* of G that are computed from those sets give the immediate solution to the UV problem. Obtaining this solution takes time $O(|N|)$, as the complexity of each of Algorithms 1, 3 and 4 is $O(|N|)$.

6. EVALUATION OF IMPLEMENTATIONS OF THE ALGORITHMS

The algorithms for solving the UV problem (presented in Sections 4 and 5) have been tested using the York Ada workbench compiler.[6] The compiler was written for a VAX 11 computer system running the UNIX operating system. The implementation language was C. The algorithms were implemented on an intraprocedural level; that is, each procedure of a program under analysis was considered separately, and the worst-case assumption about the effects of procedure calls was made. So it was presumed that all global variables and the pass-by-reference actual parameters were changed due to each procedure call. The algorithms were the parts of the Control and Data Flow Analyser designed for the York Ada compiler. Using this analyser the speed of algorithms for solving the UV problem has been measured. The results are as follows:

Table 1. The times of solving the UV problem for sample programs

	No. of Ada source lines	Aho–Ullman method	Efficient method
Program 1	151	18.9	15.3
Program 2	275	79.5	64.7
Program 3	449	292.9	225.6

Table 1 contains the times of solving the UV problem for three sample Ada programs. Each time comprises the time necessary for executing the algorithms (Algorithm 2 in the Aho–Ullman method and Algorithms 1, 3 and 4 in the efficient method) and the time spent on detecting undefined variables. The times (in fiftieths of a second) were measured using the UNIX *time(1)* command. Programs 1, 2 and 3 solve the 'Tower of Hanoi problem', the 'Scientific calculator' and the 'Mastermind game', respectively.

The comparison shows that the efficient method is about 20% faster than the Aho–Ullman method.

7. CONCLUSIONS

In the paper the efficient algorithm for solving the problem of detecting undefined variables during compilation is presented. It has been proved that its time complexity is $O(|N|)$, where $|N|$ is a number of nodes of the flow graph. The comparison for some example programs showed that the efficient algorithm is about 20% faster than the Aho–Ullman algorithm. It must be noted however that the former algorithm does not compute the use-definition chains that might be valuable for other goals of compilation, i.e. the object code optimisation.

Acknowledgements

The work presented in this paper was carried out during the author's sabbatical leave at the University of York with the assistance of a scholarship from the British Council. The author wishes to thank Professors Ian C. Wand and Ian C. Pyle for their support, and John A. Murdie and Lewis Tsao for their helpful collaboration. Special thanks are due to Dr Mark R. Manning for valuable comments on an early draft of this paper.

REFERENCES

1. F. E. Allen, Control flow analysis. *ACM SIGPLAN Notices* (7), 1–19 (1970).
2. F. E. Allen, *Graph theoretic constructs for program control flow analysis*. Research Report RC-3923, IBM T. J. Watson Research Center, Yorktown Heights, New York (1972).
3. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling*, vol. I: *Parsing*, vol. II: *Compiling*. Prentice-Hall, Englewood Cliffs, N. J. (1972).
4. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass. (1974).
5. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Addison-Wesley, Reading, Mass. (1977).
6. J. S. Briggs, C. H. Forsyth, C. W. Johnson, M. R. Manning, J. A. Murdie, I. C. Pyle, C. Runciman, I. C. Wand and A. J. Williams, *Ada Workbench Compiler Project 1982*, Department of Computer Science Report YCS.59, University of York (1983).
7. *Department of Defense Requirements for High Order Computer Programming Languages, STEELMAN*. Defence Advanced Research Projects Agency, Arlington, Virginia (1978).
8. M. S. Hecht and J. D. Ullman, Characterizations of reducible flow graphs. *J. ACM* (3), 367–375 (1974).
9. M. S. Hecht and J. D. Ullman, A simple algorithm for global data flow analysis problems. *SIAM J. on Computing* **4** (4) 519–532 (1975).
10. M. S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York (1977).
11. J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine and B. A. Wichmann, Preliminary Ada reference manual, *ACM Sigplan Notices*, **14** (6), part A (1979).
12. J. D. Ichbiah *et al.*, *Reference Manual for the Ada Programming Language*. ANSI/MIL-1815A (1983).
13. K. Kennedy, A comparison of two algorithms for global data flow analysis. *SIAM J. on Computing* **5** (1), 158–180 (1976).
14. W. Kennedy and L. Zucconi, Applications of graph grammars for program control flow analysis. *Conf. Rec. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, CA*, pp. 72–85 (1977).
15. D. E. Knuth, *The Art of Computer Programming*, vol. I: *Fundamental Algorithms*. Addison-Wesley, Reading, Mass. (1968).
16. D. E. Knuth, An empirical study of FORTRAN programs, *Software – Practice and Experience*, **1** (2) 105–133 (1971).
17. E. I. Organick, *Computer System Organization. The B5700/B6700 Series*. Academic Press, London (1973).
18. *Reference Manual for the GREEN Programming Language*. Honeywell Inc. and Cii Honeywell Bull (1979).
19. Reference Manual for the Ada Programming Language. United States Department of Defense (1980).
20. M. Schaefer, *A Mathematical Theory of Global Flow Analysis*. Prentice-Hall, Englewood Cliffs, N.J. (1973).
21. R. E. Tarjan, Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1** (2), 146–160 (1972).
22. J. D. Ullman, Fast algorithms for the elimination of common subexpressions. *Acta Informatica* **2**, 191–213 (1973).
23. J. Welsh, Economic range checks in Pascal. *Software – Practice and Experience* **8** (1), 85–97 (1978).
24. J. Welsh, W. J. Sneeringer and C. A. R. Hoare, *Ambiguities and Insecurities in Pascal*. In *Pascal – the Language and its Implementation*, edited D. W. Barron. John Wiley, Chichester (1981).
25. B. A. Wichmann, Is Ada too big? A designer answers the critics. *Comm. ACM* **27** (2), 98–103 (1984).
26. W. A. Wulf, Trends in the design and implementation of programming languages. *IEEE Computer*, pp. 14–24 (1980).