Experiences in Analyzing Data Dependences for Programs with Pointers and Structures

W. Amme, E. Zehendner

Abstract. We describe a method to derive safe approximations for data dependences in programs with pointers and structures. In our approach, alias information and reaching definitions' information at each program point is simultaneously covered by a single representation. We perform a single-pass data dependence analysis for imperative programs by solving a monotone data flow system. Advantages of our method are improved accuracy, economical storage use, and reduced analysis time.

1 Introduction

Determination of data dependences is mandatory when restructuring imperative programs [1]. An essential step in data dependence analysis is the calculation of reaching definitions. Once the reaching definitions have been determined, we are then able to infer def-use associations, i.e., dependences. The accuracy of the performed data dependence analysis directly affects the efficacy of its application: underestimation usually becomes disastrous in context of code restructuring techniques that rely on semantics-preserving transformations, overestimation is safe albeit it degrades the merits of the analysis. We are thus striving for safe approximations that are as accurate as possible.

Our method is based on expressing the data dependence analysis for programs with pointers and structures as a *monotone data flow system* [2], that can be solved by a well-known algorithm from data flow analysis [3, 4]. In this methodical context, it is easy to change the source language or the approximation strategy, and to estimate its specific merits and drawbacks; the chosen approach also enables us to easily prove the correctness of our method. For each kind of approximation, we use a specific *data flow framework* $(L, \vee, F)$, where L is called the *data flow information set*, $\vee$ is the *union operator*, and F is the set of *semantic functions*.

The data flow analysis presented in the following sections, in principle works for imperative languages like Pascal, Fortran 90, Modula-2, or C, when excluding type casting, arithmetic on pointers and other special features as setjmp, longjmp or exception handling. Target programs can contain pointers and dynamically allocated structures, allowing an unbounded number of levels of indirection; pointer variables may obtain their values via assignment or by allocation. Our current implementation accepts Modula-2 programs, that work on linked lists which are also allowed to be cyclic.

```
 1: New(p);
 2: i := 0;
 3: q := p;
 4: WHILE i < 10 DO BEGIN
 5:    IF i < 5 THEN
 6:       q^.data := i
 7:    ELSE
 8:       q^.data := 2 * i;
 9:    New(q^.next);
10:    q := q^.next;
11:    i := i + 1
12: END;
13: x := p^.next^.data;
```

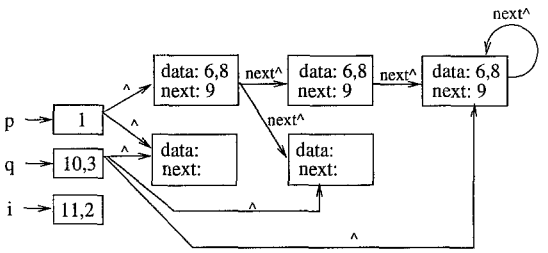

Fig. 1. A program segment and one of its corresponding A/D graphs

2 Data flow information sets

We use *approximate data dependence description graphs* (*A/D graphs*) as data flow information sets. Each A/D graph denotes, in a finite form, the structure of the store and some aspects of its state at a certain point in program execution. From A/D graphs we can derive essentially the following information: First, the combination of paths through which we can access a memory cell at one program point; second, the program statements which defined respectively used—which depends on the kind of data dependence, that is calculated—the contents of a memory cell last. The actual data dependences for the statement in consideration can then be read off by looking for relevant def-use combinations concerning the same memory cell.

The nodes of an A/D graph represent the objects present in memory. A reference of one of these storage objects to another storage object by use of a pointer is expressed by an edge in the A/D graph. There may be nodes of two kinds in an A/D graph: A *simple node* represents objects that can be accessed through any of the paths by which we can reach this node in the A/D graph; the sets of objects represented by distinct simple nodes need not be disjoint. Introduction of *condensation nodes* is necessary to force the finiteness of our representation. Condensation nodes are distinguished from simple nodes by loops emanating from them; a condensation node stands for objects that can be accessed through the infinite set of paths targeting the condensation node in the A/D graph.

Figure 1 shows a program segment and a corresponding A/D graph that describes the store immediately before execution of statement 13. The nodes of this A/D graph record write accesses to memory cells. The program segment presented here generates a linked list with 10 elements and initializes these as a function of the index value. In the A/D graph, two list elements are shown as distinct nodes; all further list elements are represented by a condensation node. Since statement 13 reads the contents of memory cells accessed through p,

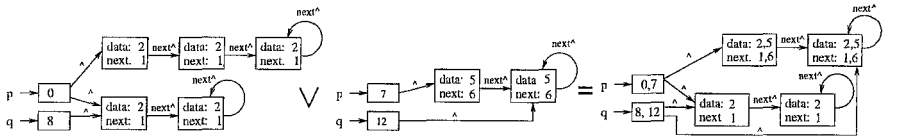

Fig. 2. Sample application of the union operator

`p^.next`, and `p^.next^.data`, we infer from this A/D graph a true dependence of statement 13 on statements 1, 6, 8, and 9.

An A/D graph A is called l-bounded if, after deleting the edges emanating from any condensation node, there is no directed path of length greater than l. To avoid some ambiguities in context with l-bounded A/D graphs, edges originating at a condensation node must point back to this same node. Also, we require that for any condensation node n_2 there should not exist any other node n_1, so that $Pre(n_1) \subseteq Pre(n_2)$, where $Pre(n)$ is the set of all predecessor nodes of a node n.

3 Data dependence analysis with A/D graphs

In data flow frameworks, the effects of joining paths in the flow graph is implemented by the union operator. We may invent several variants of union operators, that would be consistent with a chosen data flow information set. Our current version of the union operator $\vee$ works as following:
First, we merge the nodes that are reachable in both graphs by the same set of paths; new labels are calculated as the union of the corresponding old labels. Since this procedure does not always yield an l-bounded A/D graph, we may have to further *reduce* the resulting graph. Reduction continuously merges a condensation node n with all successors as well as with all nodes that could be exclusively reached via some predecessors of n.
We have proven, that for a fixed $l \in \mathbb{N}$, the set of all l-bounded A/D graphs in conjunction with this union operator constitutes a bounded semi-lattice with an one element and a zero element. Figure 2 shows a sample application of the union operator.

We can unambiguously assign a *semantic function* to every node; this semantic function serves to modify an A/D graph when processing the node. Figure 3 specifies the semantic functions of our method; AD_{in} and AD_{out} stand for an A/D graph before resp. after the application of the semantic function. We limit our description to the case of true dependences and output dependences.

Our semantic functions are composed of auxiliary functions, each performing a transformation on some l-bounded A/D graph A. *Field*(X) yields the longest suffix of path X not containing any pointer reference, and is used to access a single field within a record structure. *Define*(A, X, s, f) updates with s field f of every simple node, that is reached by path X; if X leads to a condensation node n, s will be joined to the label of field f in n. *GenerateNode*(A, X) marks

Assignment to a pointer variable

- *s: X := Y*
  AD_{out} = *Define(InsertAndComplete(DeleteEdge(AD_{in},X,Y),X,Y),X,s,Field(X))*
- *s: X := NIL*
  AD_{out} = *Define(DeleteEdge(AD_{in},X,NIL),X,s,Field(X))*
- *s: New(X)*
  AD_{out} = *Define(GenerateNode(DeleteEdge(AD_{in},X,NIL),X),X,s,Field(X))*
- *s: Dispose(X)*
  AD_{out} = *Define(DeleteEdge(AD_{in},X,NIL),X,s,Field(X))*

Assignment to a non-pointer variable

- *s: X := ...*
  AD_{out} = *Define(AD_{in},X,s,Field(X))*

Other statements

$AD_{out} = AD_{in}$

Fig. 3. Semantic functions

all simple nodes in A that are reachable via path X, starts a new edge on every marked node that cannot be reached via a path of length l, and attaches a blank node to the new edge. In contrast, each marked node reachable via a path of length l becomes a condensation node, by insertion of a loop.

DeleteEdge(A, X, Y) first transforms A into an *expanded* A/D graph—in such a graph, none of the nodes can be reached via different paths starting at the same variable. Then we mark all simple nodes that are reachable via X, but not via a prefix of Y; we ignore Y, when set to NIL. We erase all edges emanating from any marked node, and delete unreachable nodes. Finally, we merge all nodes that are reachable exactly by the same set of paths, and reduce the emerging graph to an l-bounded A/D graph.

InsertAndComplete(A, X, Y) first expands A. Now, nodes that are reachable via path X and via a prefix of Y must be treated differently from nodes that are reachable via X, but not via a prefix of Y. Let v be the variable where path Y starts. First, we mark all nodes that are reachable via X and via a path starting at v, as well as all these nodes' outgoing edges. We insert edges from any marked node n to all successors of n that are reachable via $Y\hat{}$, and then delete all marked edges emanating from n. Finally, we delete all unreachable nodes.

In a second step, we have to mark all nodes that can be reached via X, but not via a path starting at v. For any marked node n we do the following: Let *node* be the set of nodes that are reachable via path $Y\hat{}$, but not reachable via a path starting at a variable for which a path to n exists, starting at the same variable. Then we copy each subgraph that starts by an element $m \in$ *node* and insert an edge from each predecessor of m, and from n, to the entry node of this subgraph.

Figure 4 illustrates, step by step, the application of a semantic function that

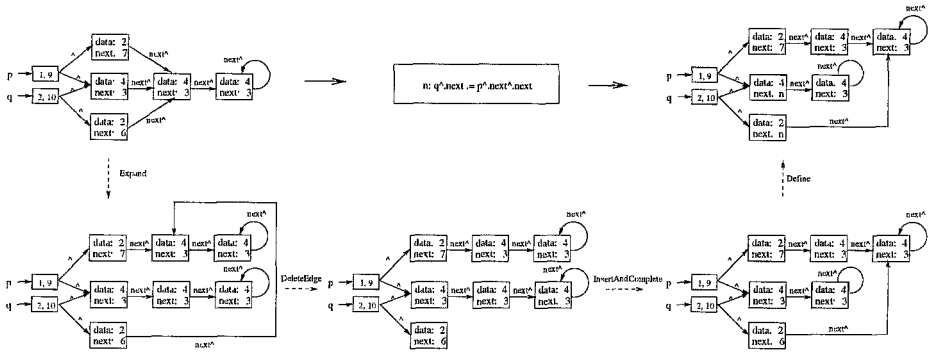

Fig. 4. Sample application of a semantic function

corresponds to a node of the control flow graph representing an assignment to a pointer variable.

We have implemented the use of A/D graphs into our data dependence analysis and have received the first empirical results. Analyzing our test programs resulted in an average of 2.24 data dependences per statement. In an other investigation we analyzed the time behavior of our method. The analysis time required from our technique was in average about 1% of the time the *WRL* compiler (from *DEC*) needed for the total compilation of the same programs.

4 Conclusions

We have presented a single-pass method to derive data dependences in imperative programs with pointers and structures that is safe, accurate, fast, and storage economical. Our method solves a monotone data flow system, and is based on representations called A/D graphs, that cover both reaching definitions and alias information simultaneously. The use of A/D graphs for data dependence analysis promises to be a significant improvement over other known methods: data dependence analysis becomes more accurate, and therefore useful information can be transmitted to subsequent processing phases.

References

1. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
2. H. Zima and B. Chapman. *Supercompilers for parallel and vectors computers.* ACM Press, New York, 1991.
3. M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.
4. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, January 1977.