ELSEVIER

Parallel Computing 24 (1998) 505–525

PARALLEL COMPUTING

Data dependence analysis in programs with pointers

Wolfram Amme [*], Eberhard Zehendner [1]

Computer Science Department, Friedrich Schiller University D-07740 Jena, Germany

Received 10 April 1997; revised 2 September 1997

Abstract

This paper offers an introduction to the field of data dependence analysis in programs that handle pointers and dynamically allocated structures. We are principally concerned with methods that use monotone data flow systems, and in particular present in detail a one-pass approach we recently developed. The new method is appropriate for imperative languages such as C, Java, Fortran 90, Pascal, or Modula-2. We can perform an interprocedural analysis on recursive or nonrecursive programs, and deal with arbitrary acyclic data structures, covering any level of indirection; an extension to cyclic data structures is currently under development. Our method is based on storage representations called A/D graphs, that cover both reaching definitions and alias information simultaneously. May-alias information, as well as must-alias information, is implicitly kept in the graphs, allowing strong updates and thus leading to a high precision. Reaching definitions are represented in the graphs by means of an instrumented semantics. The new method is safe, accurate, fast, and storage-economical, and therefore promises to be a significant improvement over other known methods. A prototype implementation showed very encouraging results with regard to accuracy and analysis time. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Data dependence analysis; Alias analysis; Data flow analysis; Pointer analysis

1. Introduction

Data dependence analysis is indispensable in the ever-widening field of automatic parallelization [1], and is, as well, widely applicable towards various other software development techniques [2]; for example, source-level debugging, automated data flow testing [3], semantic browsing, integration of independently altered versions of pro-

[*] Corresponding author. E-mail: wolfram.amme@uni-jena.de
[1] E-mail: zehendner@acm.org

0167-8191/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.
PII S0167-8191(98)00023-4

grams, and, last but not least, most of the code optimization tools that are part of a compiler [4].

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 but it degrades the merits of the analysis. We are thus striving for safe approximations that are as accurate as possible. However, it can be expected that increased accuracy will require more time to analyze the code and more storage during the analysis.

There are well-known mature techniques for the data dependence analysis of programs that are exclusively based on loops as control structures and arrays as data structures. In contrast, there has been little research on data dependence analysis for programs that also use pointer variables; analysis techniques for complete imperative languages like C or Fortran 90 are rare in the literature. In the presence of pointer variables, we have to consider aliasing, i.e., a memory location might be accessed through several variable names, or, more generally, via different access paths. As the value of a pointer variable can be defined by assignment or allocation, the alias relation may change in any statement of the program. Also, there no longer exists a fixed relation between variable names resp. access paths used within a program, and memory objects handled by the program, since the address of a memory object can appear as the value of a pointer variable and thus may be passed to other pointer variables by assignment, thereby overwriting a previous reference.

2. Data dependences in programs with pointers

In programs with pointers it is necessary to express data dependences in terms of storage objects rather than program variables. A precise definition of data dependences is given as follows: A statement S_2 has a data dependence on a statement S_1 if S_2 can be reached after the execution of S_1, and the following conditions hold:

(i) Both statements access a common memory location *loc*.
(ii) At least one of them writes to *loc*.
(iii) Between the execution of S_1 and S_2, there is no statement S_3 that also writes to *loc*.

A data dependence is called a *true dependence* if S_1 writes to *loc* whereas S_2 reads from *loc*. It is called an *anti-dependence* if S_1 reads from *loc* and S_2 writes to *loc*; it is an *output dependence* if both statements write to *loc*. S_1 and S_2 are in *conflict*, if conditions (i) and (ii) hold, whereas (iii) may or may not be fulfilled. Conflict analysis is a frequently used approximation of data dependences. The techniques to perform data dependence analysis for each of these kinds of dependence are very similar, so we concentrate on true dependences.

In the presence of pointer variables, or when performing interprocedural analysis with reference formals, a storage object might be accessed through several variable names or,

more generally, through different access paths. This phenomenon is called aliasing. The basic problem of alias analysis is to determine *may-alias* information, i.e., access paths that during some execution history of a program refer to the same storage object at the same program point. A variant of the may-alias problem is the calculation of *must-alias* information; a must-alias is an alias that at a specific program point holds on any execution path covering this point. May-alias information is needed to ensure safety, and must-alias information can be used to improve the accuracy of the data dependence analysis. This is illustrated by the program in Fig. 1. Without may-alias information we cannot detect the true data dependence of statement 6 on statement 5; whereas, without must-alias information we cannot detect that the contents of the object which is reachable via *PA^* in statement 6 is defined by statement 5, resulting in a spurious true data dependence of statement 6 on statement 4.

Currently, data dependence analysis is performed by static analysis. During static analysis we abstract from the particular values contained in, or pointed to by, program variables. Hence we are not able to determine the direction of branches taken during program execution. Therefore, we must consider all legal paths in the control flow graph of a program. These legal paths might be a proper superset of the execution paths actually appearing at run time, thus rendering our analysis inaccurate. Nevertheless, within the context of static analysis, a dependence relation (or alias relation, or the like) usually is called *precise* if it is exact with respect to the legal paths in the control flow graph.

There are two main directions for performing static analysis: abstract interpretation [5] and data flow analysis [6]. *Abstract interpretation* describes the problem from the point of view of formal semantics. After constructing an abstract semantics, which is based on a concrete semantics, the interpretation of the program yields the result of the analysis. In contrast, *data flow analysis* (described later in further detail) is a form of static analysis that uses control flow. Our method is based on treating the data dependence analysis as a data flow problem.

The determination of true data dependences can be achieved by different means. The most commonly used is the calculation of reaching definitions for all statements. This can be described as the problem of determining, for a specific program point and memory location, all program points where the value of this memory location has been (or could have been) written last. Once the reaching definitions have been determined, we are able to infer def-use associations; a def-use pair of statements indicates a true dependence between them. For scalar variables, the determination of reaching definitions can be performed by a well-known standard algorithm described in Ref. [7]. This

```
1: NEW(PA);
2: NEW(PB);
3: PC := PA;
4: PA^:= 1;
5: PC^:= 2;
6: PB^:= 3 * PA^;
```

Fig. 1. A simple program using pointers.

algorithm can be used also for the calculation of reaching definitions in programs with pointers, when treating several instances of an object that are generated at the same program point, as a single object (cf. Refs. [2,28]). This calculation of true data dependences encompasses four phases of analysis: may-alias information, must-alias information, reaching definitions, and def-use associations.

Tracking the shape of dynamic data structures is another means of calculating data dependences in programs with pointers. For each program statement, a finite graph is derived which describes the shape of the storage prior to the execution of the statement. It can be determined from these graphs whether two statements might be able to access the same storage object. This approach is particularly important when parallelizing programs. Our method described below implicitly uses this kind of *shape analysis*.

Clearly an accurate determination of data dependences in programs with pointers is a complex process. The intraprocedural alias problem in the presence of multiple-level pointers, as well as a precise determination of intraprocedural reaching definitions in the presence of single level or multiple-level pointers, are both *NP-hard* [8].

3. Monotone data flow systems

Our method is based on expressing the data dependence analysis for programs with pointers and structures as a *monotone data flow system*. In this methodical context, it is easy to change the source language or the approximation strategy, and to estimate the particular merits and drawbacks of a chosen strategy. 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 (sometimes called the meet operator in the literature), and F is the set of semantic functions.

The data flow information set is a conceptual universe of objects upon which the analysis is working. In our case, an element of the data flow information set covers, simultaneously, both alias information and reaching definitions' information for a single program point at a specific time during the analysis. We translate each program into a set of recurrence equations whose formal variables denote elements from the data flow information set. Each operator in an equation is either a semantic function or the union operator. A semantic function corresponds directly to a statement in the program, and models the effect of executing this statement. The union operator implements joining paths in the control flow graph.

The least fix-point, with respect to an implicitly given partial order, of the set of recurrence equations, if one exists, describes all dependence information on the program we are able to obtain under the chosen approximation policy. By an inductive process that starts with all variables initialized to a trivial element of the data flow information set, we successively approach the fix-point of the system. If the semantic functions are monotone and $(L, \vee)$ forms a bounded semi-lattice with a one element (which is the trivial element mentioned above) and a zero element, we can use a general iterative algorithm, that always terminates and yields the least fix-point of the data flow system [4,24,25].

We perform a static analysis, and thus must abstract from the particular values contained in, or pointed to by, program variables. Hence we are not able to infer the

direction of branches taken during program execution. Therefore, we must consider all paths in the control flow graph of a program, which might be a proper superset of the execution paths actually appearing at run time, thus rendering the analysis inaccurate. This is a general problem of static analysis, and not due to our specific model.

When using dynamically allocated storage objects in combination with multiple level pointer indirection, we may get data structures whose size is not bounded in any number inferable from the program code. Hence the size of the alias relation also cannot be bounded statically, and the alias analysis has to assume an unbounded number of objects. This leads to data flow frameworks with unbounded information sets which may have no finite fix-point solution, and thus would not be computable by an inductive approach. There we rely on an approximate static analysis that uses a finite representation to describe a virtually unbounded store. This approximation may lead to some loss of accuracy during our data dependence analysis. Nevertheless, our solution is safe, i.e., all real data dependences are detected, but the analysis may also report some spurious dependences.

In our approach we use a data flow information set that simultaneously covers alias information and reaching definitions' information at a program point by a single data structure. The technique to note the recently accessing statement directly in the data structure figures as an *instrumented semantics* in Ref. [9].

4. Programming model

The data flow analysis presented in this paper is appropriate for imperative languages such as C, Java, Fortran 90, Pascal, or Modula-2. (However, we exclude type casting, arithmetic on pointers, and other special features such as setjmp, longjmp, or exception handling.) In Ref. [10], we described a data dependence analysis that was restricted to data structures with a single level of indirection. Presently we have expanded the programming model to cover arbitrary acyclic data structures. Thus programs may handle pointers and dynamically allocated structures, allowing for an unbounded number of levels of indirection. Pointer variables may obtain their values via assignment or by allocation. We assume that programs have been fully type-checked, and that all references are correct.

Currently we use Modula-2 as our source language. This choice has been motivated by the fact that Modula-2 comes closest to our programming model when we do not include module *system.def*. Procedures are allowed to access their arguments through call-by-value or through call-by-reference. We suppose that functions are modeled by procedures, with results passed back via reference parameters.

Prior to the essential data dependence analysis, our prototype implementation performs the following transformations on the source code to streamline the analysis (this concerns the definition of the semantic function *InsertEdge* described below): (i) each pointer assignment $l := r$ (where l and r may be arbitrary expressions of conforming pointer type, and $r \neq NIL$) is replaced by a statement sequence $t := r;\ l := t;\ t := NIL$, where t is a temporary variable; (ii) by introducing additional reference parameters, procedures that access global variables are converted into procedures with access to local variables and parameters only.

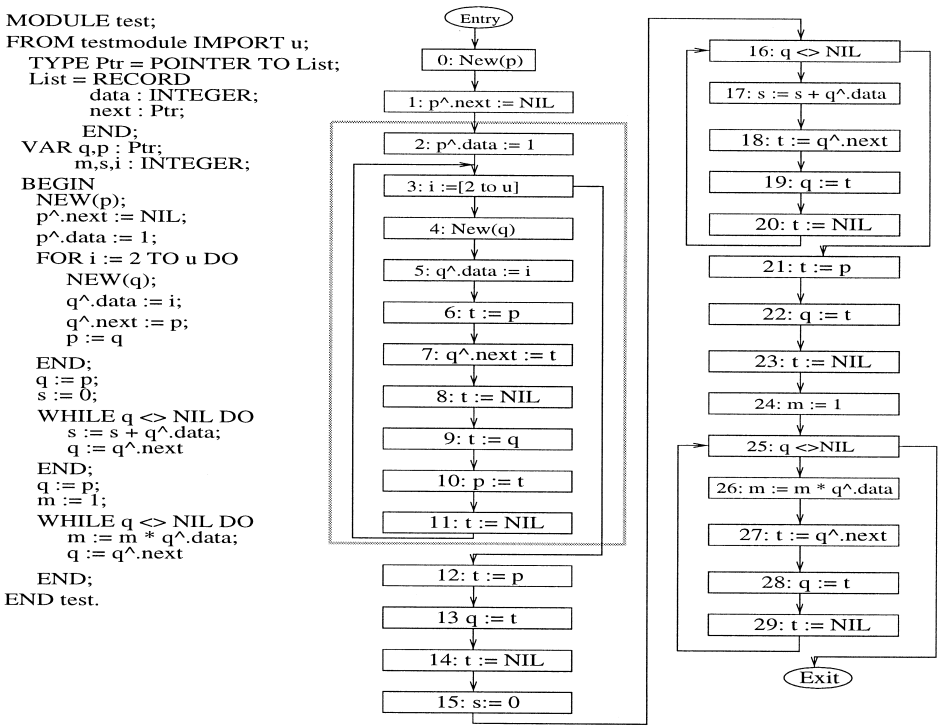

Fig. 2. Sample source program and control flow graph of the transformed program.

Fig. 2 shows a Modula-2 source program, together with the control flow graph of the program after transformation according to these rules. The sample program presented here generates a linked list with us and initializes these as a function of the index value. After the initialization, sum and product of the items are calculated within two separate loops.

5. Data flow information set

We use approximate data dependence description graphs (A/D graphs) as the data flow information set. Each A/D graph denotes, in a finite form, possible structures of the store and some aspects of its state (as described below) at a certain point in program execution. We emphasize that our representation falls into the category that usually is called a *storeless model* [11], as opposed to *memory-based models*.

From A/D graphs we can derive the following information: First, the set of paths with which we can access an object at one program point (i.e., the alias information); second, the program statements which defined resp. used the contents of a storage object last (i.e., the reaching definitions resp. the reaching uses). The actual data dependences for a statement can then be determined by looking for relevant def-use, def-def, and use-def associations concerning the same object. We limit our description to the case of

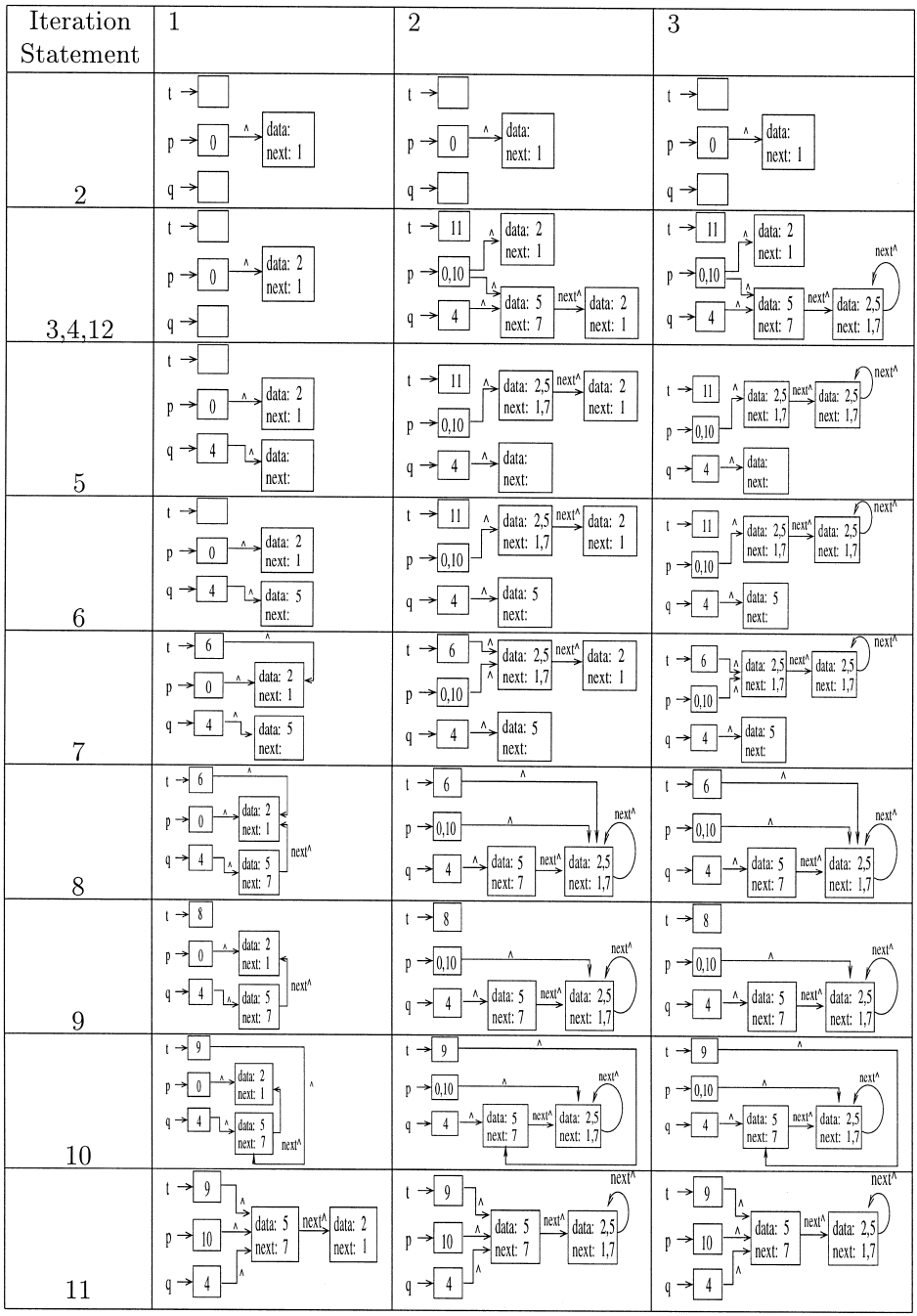

Fig. 3. Data dependence analysis with A/D graphs for the framed part in Fig. 2.

definitions, which is sufficient to determine true dependences and output dependences. For handling anti-dependences, the semantic functions presented in this paper must be adjusted to note read accesses instead of write accesses in the A/D graphs.

The nodes of an A/D graph represent the objects present in the store. A reference of one of these storage objects to another storage object by use of a pointer is expressed by a labeled edge in the A/D graph. There may be two kinds of nodes in an A/D graph: simple nodes and condensation nodes. A simple node represents the objects that can be accessed through any of the paths by which this node is reachable in the A/D graph. Condensation nodes are distinguished from simple nodes by at least one outgoing edge that leads back to the condensation node—we call this kind of edge a *loop*, whereas all other edges are called *simple edges*. A condensation node stands for all objects that can be accessed through the infinite set of paths targeting the condensation node in the A/D graph. The concept of condensation nodes becomes inevitable in context of data structures whose size cannot be decided during compile time. Precise control of when to use a condensation node is achieved by working with l-bounded A/D graphs. An A/D graph is called l-bounded, if no directed path composed of simple edges is longer than l. The sets of objects represented by nodes in an A/D graph may intersect and can change during the analysis.

Fig. 3 shows the results of a data flow analysis with three-bounded A/D graphs for the framed section of the program in Fig. 2. In this part of the program a list *ll* is generated. For each iteration of the general algorithm and each statement, we calculated an A/D graph that describes the store immediately before the execution of the statement under consideration. (Note: The analysis becomes stationary after the third iteration. The nodes for the non-pointer variables *i*, *s*, and *m* have been suppressed in the drawing.) We refer to Fig. 3 whenever explaining details of our method. To simplify our description, we denote by $A_{s,i}$ the A/D graph which describes the store immediately before execution of a statement s in iteration i in Fig. 3.

In the A/D graph $A_{12,3}$ the simple node that is reachable via the set of paths $\{p\hat{}, q\hat{}\}$ represents the first of *ll*, if the parameter *u* is greater than one; the remaining list elements are then represented by the condensation node. If $u \leq 1$, the first—and only—item of the list is represented by the node in $A_{12,3}$ that is reachable via the path $p\hat{}$ only. Both cases are included in the same A/D graph since the actual length of *ll* cannot be decided at compile time; but even when the value of parameter u would have been known to the compiler, the same representation would have resulted due to the inability of the static analysis to evaluate program predicates. Since statement 12 reads the contents of storage objects that can be accessed by path p, from the A/D graph $A_{12,3}$ we infer a true dependence of statement 12 on statements 0 and 10.

6. Intraprocedural data dependence analysis with A/D graphs

This section describes the union operator and the semantic functions of the data flow framework that we constructed for the data dependence analysis with A/D graphs. We use l-bounded A/D graphs as data flow information set, where $l \in \mathbb{N}$ may be chosen to guarantee the desired accuracy of the analysis.

The nodes of an A/D graph can be uniquely distinguished by the sets of access paths reaching the nodes. There we represent each node of an A/D graph as a tuple $(n,*)$, where $* \in \{S,C\}$ stands for the kind of node (simple or condensed), and n for the set of access paths composed from simple edges only. Let $STAT$ be the set of program statements, V the set of variables, F the set of all field names in structures, and $F\hat{}$ the set of dereferenced field names.

For simplicity, we will treat variables as structures that contain only the empty field ϵ. We define the set of paths in an A/D graph as $AD_{\text{paths}} = V(.F\hat{})*$. Consequently, an A/D graph A is a triple $A = (N,E,D)$, where (N,E) is a directed labeled graph, $N \subseteq \mathcal{P}(AD_{\text{paths}}) \times \{S,C\}$, $E \subseteq N \times F\hat{} \times N$, $D \subseteq N \times F \times \mathcal{P}(STAT)$. The sets N_S and N_C constitute a partition of N; they represent the simple nodes resp. the condensation nodes of A. An element $(n,f,s) \in D$ indicates that field f of node n is defined by the set s of statements. If no element $(n,f,*) \in D$ exists, then field f of node n has not been initialized up to this point. We consider a node $n \in N$, $n \cap V \neq \emptyset$ as an entry point of A; we call such a node an entry node for all v, $v \in n \cap V$. The concept of paths in A/D graphs differs from that used in programming languages. For conversion we introduce two functions. Let p be an access path in a program. $Field(p)$ yields the field that is accessed by p. In contrast, $Obj(p)$ converts that part of p, by which the object that contains $Field(p)$ is accessible, into the corresponding path of an A/D graph. We informally define $Node(p,A) = \{n \in A | n \text{ is reachable via } Obj(p)\}$, where $A \subseteq N$, as the set of all nodes in A that are reachable by the path $Obj(p)$.

Simple edges are not allowed to start at a condensation node. Additionally, we require that each node of an A/D graph is *valid*. We call a node $(n,*)$ valid, if there exist no $x,y \in n$ where x is a prefix of y. An A/D graph that contains only valid nodes is referred to as a *valid graph*. In our programming model, graphs that arise during a data dependence analysis with A/D graphs are always valid, since only acyclic structures are permitted, and thus it is impossible to access objects in storage via a path p and a prefix of p simultaneously. Finally, we call two valid nodes $(n,*)$, $(m,\overline{*})$ of an A/D graph *agreeable*, abbreviated as $((n,*)\Phi(m,\overline{*}))$, if there exists no $x \in n$, $y \in m$ where x is a prefix of y, or vice versa.

Fig. 5 shows the union operator *Union* on l-bounded A/D graphs. The union operator is defined as a union of the nodes, edges, and definition sets of two graphs. Additionally, definition sets that belong to a field of the same node will be collapsed to a single definition set. Due to the special form of labeling the nodes in an A/D graph, the union operator on l-bounded A/D graphs always yields an l-bounded A/D graph. In Fig. 3, $A_{12,3}$ is the result of applying the union operator to those graphs that result from applying the semantic functions, described in the sequel, to the graphs $A_{2,3}$ and $A_{11,2}$.

For a fixed $l \in \mathbb{N}$, the set of all l-bounded A/D graphs is denoted by AD_l. We have proven that the structure $(AD_l, Union)$ is a semi-lattice that is bounded, as the number of l-bounded A/D graphs is always finite. The one element of the semi-lattice is the empty graph, and the zero element is the graph that results by iterative application of the union operator to all l-bounded A/D graphs.

In the control flow graph that is used for the analysis, each node stands for a uniquely labeled program statement. Therefore we can unambiguously assign a *semantic function* to each of the nodes. This semantic function will be used to transform the A/D graph

when processing the node. Fig. 4 defines the semantic functions used by our method. In this definition, AD_{in} stands for an A/D graph before the application of the semantic function, and AD_{out} for the corresponding A/D graph after the application of the semantic function.

These semantic functions are composed of auxiliary functions that perform transformations on l-bounded A/D graphs. Fig. 5 contains a formal description of these functions. Splitting the semantic functions is not essential, but simplifies the presentation. In our prototype we use a more efficient implementation of these semantic functions, i.e., the number of graph traversals is minimized. *Declaration*(A, M) attaches a blank entry node to A for each $n \in M \subseteq V$. We use this function to introduce local procedure variables into an A/D graph.

Define(A, X, f, s) updates the field f of all simple nodes with the label s, reachable via *Obj*(X), in the A/D graph A. If *Obj*(X) leads to a condensation node, a strong update is not allowed, as a condensation node may represent a set of objects. To give a safe approximation in this case, the label of field f will be merged with s. *GenerateNode*(A, X, f) first marks all nodes in A, that are reachable via path *Obj*(X). For each marked condensation node, we add a loop labeled with $\hat{f}$, unless this loop is already there. On every marked simple node that cannot be reached via a path of length l,

Entry points

- *Entry*
  AD_{out} = *Declaration*(AD_{in},X), X *is the set of local procedure variables.*

Assignment to pointer variables

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

Assignment to non-pointer variables

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

FOR loop

- s: $I := [u_1$ *to* $u_2]$
  AD_{out} = *Define*(AD_{in},I,*Field*(I),s)

Other statements

$AD_{out} = AD_{in}$

Fig. 4. Semantic functions.

$Union : AD_l \times AD_l \rightarrow AD_l$, *where*
$Union((N,E,D),(N',E',D')) = (N \cup N', E \cup E', \{(n,f,\bigcup_i d_i) | (n,f,d_i) \in D \cup D'\})$.

$Declaration : AD_l \times \mathcal{P}(V) \rightarrow AD_l$, *where*
$Declaration((N,E,D),M) = (N \cup \{\{n\} | n \in M\}, E, D)$.

$Define : AD_l \times AD_{paths} \times F \times \mathcal{P}(STAT) \rightarrow AD_l$, *where*
$Define((N,E,D),X,f,s) = (N,E,D')$, *where*
$D' = D \backslash \{(n,f,d) | n \in Node(X,N)\} \cup \{(n,f,s) | n \in Node(X,N_S)\}$
$\cup \{(n,f,d \cup s) | (n,f,d) \in D, n \in Node(X,N_C)\}$.

$GenerateNode : AD_l \times AD_{paths} \times F \rightarrow AD_l$, *where*
$GenerateNode((N,E,D),X,f) =$
forall $n \in Node(X,N_C)$ **do** $E := E \cup \{(n,f\hat{},n)\}$ **od**
forall $n = (n_1,S) \in Node(X,N_S)$ **do**
  if $\exists s \in n_1, |s| = l$ **then**
    $N := N \backslash \{n\} \cup \{(n_1,C)\}$;
    $E := E \backslash \{(m,*,n) \in E\} \cup \{(m,*,(n_1,C)) | (m,*,n) \in E\} \cup \{((n_1,C),f\hat{},(n_1,C))\}$;
    $D := D \backslash \{(\overline{n},d,p) \in D\} \cup \{((n_1,C),d,p) | (n,d,p) \in D\}$
  else
    $N := N \cup \{(\{s.f\hat{} | s \in n_1\},S)\}; E := E \cup \{(n,f\hat{},(\{s.f\hat{} | s \in n_1\},S))\}$
  fi
od;
return A;

$DeleteEdge : AD_l \times AD_{paths} \times F \rightarrow AD_l$, *where*
$DeleteEdge((N,E,D),X,f) =$
$(\overline{N},\overline{E},\overline{D}) := Expand((N,E,D))$;
for all $n \in Node(X,\overline{N}_S)$ **do** $DeleteEdge_{Exp}((\overline{N},\overline{E},\overline{D}),n,f)$ **od**
return $Compress((\overline{N},\overline{E},\overline{D}))$;

$InsertEdge : AD_l \times AD_{paths} \times AD_{paths} \times F \rightarrow AD_l$, *where*
$InsertEdge((N,E,D),X,Y,f) =$
$\overline{A} := Expand((N,E,D))$;
$Node_X := Node(X,\overline{N}) \backslash Node(Y,\overline{N})$; $Node_Y := Node(Y,\overline{N}) \backslash Node(X,\overline{N})$;
forall $n \in Node_Y$ **do**
  $remove := true$; $source := \{k \in Node_X | k \in N_C \ or \ k \Phi n\}$;
  forall $m \in source$ **do**
    if $m \in \overline{N}_S$ **then** $\overline{A} := InsertEdge_{Exp}(\overline{A},m,n,f)$ **else** $\overline{A} := Reduce_{\Phi}(\overline{A},m,n,f)$ **fi**
    if $\exists l \in S^*(n), \neg l \Phi m$ **then** $remove := false$ **fi**
  od
  if $remove$ **then** $\overline{A} := DeleteSucc_{Exp}(\overline{A},n)$ **fi**
od
while $\exists n = (n_1,S) \in \overline{N}$ *and* $\exists p \in n_1, |p| = l$ **do** $\overline{A} := Reduce(\overline{A},n)$ **od**
return $Compress(A)$;

Fig. 5. Functions on A/D graphs.

GenerateNode(A,X,f) starts a new edge, labeled with $f\hat{}$, and attaches a blank node of the corresponding type to it. In contrast, all marked simple nodes that are reachable via a path of length l will be transformed into a condensation node, by inserting a loop labeled with $f\hat{}$ to it.

DeleteEdge(A,X,f) transforms the l-bounded A/D graph A into an expanded A/D graph. In this graph, nodes cannot be reached via different paths starting at entry nodes

for the same variable. (Expanding an A/D graph A is called a materialization of A, cf. Ref. [12].) This technique allows us to carry out strong updates when deleting edges emanating from simple nodes. *DeleteEdge*(A, X, f) deletes all edges labeled with $f^{\wedge}$ that start from a simple node reachable via path $Obj(X)$. Finally, we remove all unreachable nodes from A and compress the graph—we collapse all nodes that are reachable by the same set of paths to a single node, and attach to this node the union of the fields of the collapsed nodes. Fig. 6 illustrates, step by step, an application of *DeleteEdge*.

When inserting edges into an A/D graph, two situations might arise which would violate the well-definedness of an l-bounded A/D graph: simple edges emanating from a condensation node, or the appearance of a simple node that is reachable via a path that is longer than l. We treat both cases by replacing parts of the graph with condensation nodes. Unfortunately, the insertion of only one condensation node to solve this problem would destroy the monotonicity of the semantic functions. This undesirable effect occurs because we condensed, in only one data structure, information derived from different paths in the control flow graph. Therefore, we must conservatively decode information given by an A/D graph. We call a node n in an A/D graph A an *endpoint* of A, if there exists no simple edge emanating from n or, if there exists no successor node m where n is the only predecessor of m. *Reduce*(A, n) transforms an A/D graph A that violates the concept of l-bounding into a well-defined graph.

Let n be a node in A that is reachable via a path of length l, and that has some successor. For each path p (from node n to an endpoint of A), *Reduce*(A, n) adds a condensation node to A, which is constructed by the following procedure: Copy the subgraph of A which contains only nodes on path p, including all edges from outside which target a node within the subgraph. Reduce this copy of the subgraph by collapsing iteratively a pair of nodes n_1 and n_2 to create a single node n_3 as follows:

1. Mark the label fields of n_3 with the union of the label fields of n_1 and n_2.
2. Replace every edge emanating from n_1 or n_2, directed to a node r, with an edge from n_3 to r.
3. Replace every edge emanating from a node r, directed to n_1 or n_2, with an edge from r to n_3.

After the construction of condensation nodes, *Reduce*(A, n) removes the subgraph starting at node n, as well as all edges from outside which target a node within this subgraph.

We treat the insertion of an edge emanating from a condensation node by the function *Reduce*$_{\Phi}(A, n, m, f)$. This function differs slightly from that above, as only paths to endpoints of A are considered where nodes are agreeable with m, and no subgraph is removed from A.

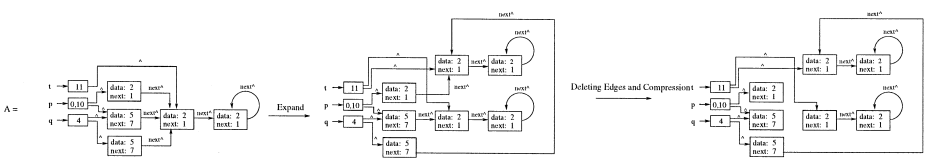

Fig. 6. An application of *DeleteEdge*$(A, p, \wedge next, next)$.

The insertion of edges into an A/D graph is an elaborate procedure. When applying *InsertEdge*(A, X, Y, f), there may be many nodes in A that are reachable via *Obj*(X) resp. *Obj*(Y), and we must construct an A/D graph that conservatively covers all

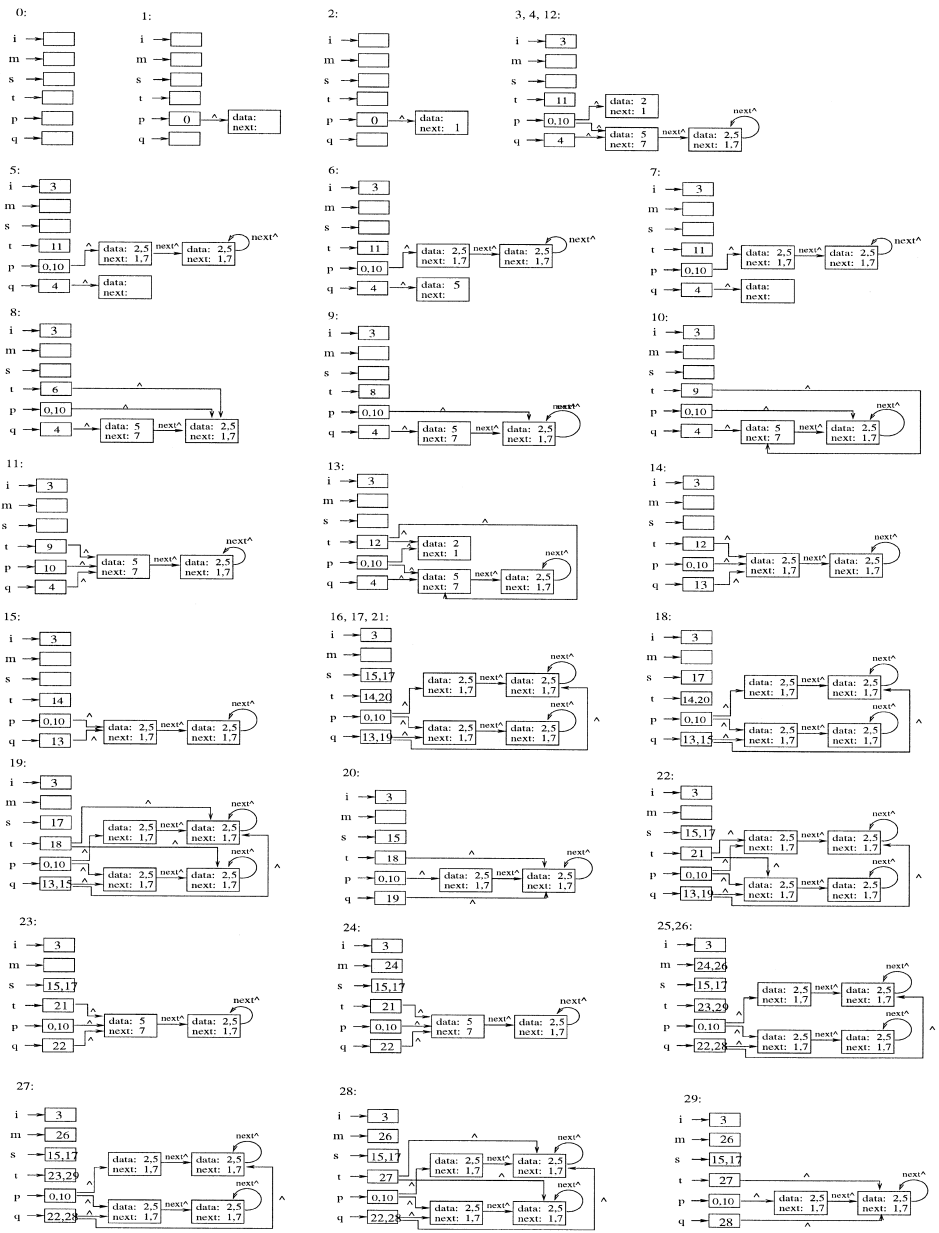

Fig. 7. A/D graph for example in Fig. 2.

possible sets of paths with which we may access objects in the store after an assignment. $InsertEdge(A, X, Y, f)$ disregards nodes of A that are both source and target of an edge to insert, because such pointer assignments are already expressed by the graph. Let $Node_X$, $Node_Y$ be defined as in Fig. 5. First, $InsertEdge(A, X, Y, f)$ expands the A/D graph A. Then for each node $n \in Node_Y$ we do the following: Let *source* be the set of nodes in $Node_X$ that are agreeable with n. For each simple node $m \in source$, we copy the subgraph which begins with n (as well as all edges from outside which target a node within the subgraph), remove from this copy all nodes and edges which are not agreeable with n, and insert an edge (labeled with $f^{\wedge}$) from m to the entry node of this copied graph. In contrast, for each condensation node $m \in source$, we apply the function $Reduce_{\Phi}(A, m, n, f)$. We can delete all successors from n and the node itself, only if all $m \in source$ are agreeable with the successors from n. Permanently performed garbage collection will lead to an unsafe approximation, i.e., we could delete nodes that are not agreeable with an element of *source*.

Before compressing the graph $\bar{A}$, we have to examine whether there is a simple node n in the graph with outgoing edges and which can be reached via a path of length l; in such a case, we must apply the function $Reduce(\bar{A}, n)$.

We have proven that the semantic functions are monotone. For all programs considered here, $(AD_l, Union)$ forms a bounded semi-lattice with one and zero element. Thus, to solve the data flow framework, we can use the previously cited general algorithm that always terminates and gives a safe solution. Fig. 7 shows the complete example of a data flow analysis with three-bounded A/D graphs for the given program in Fig. 2.

As we are able to realize strong updates, we can detect with our analysis that there exist no true data dependences between the second and third loop of this example. Such an accurate determination of data dependences is useful when transmitting the data dependence information to subsequent processing phases. For example, after an appropriate renaming of variables q and t, these two loops can be executed in parallel.

7. Interprocedural data dependence analysis with A/D graphs

We use a *partial in-line expansion* [23] for interprocedural analysis. This allows an accurate, but nevertheless easy interprocedural analysis for nonrecursive programs. To apply this method, we must convert the program to an *interprocedural control flow graph* (ICFG). An ICFG contains two different nodes for each call statement: The *call node* denotes the actual procedure call, i.e., it establishes the connection between the call statement and the body of the called procedure; the *global-effect node* models the effects of the procedure call on the calling program segment. In our prototype we use a simple algorithm to derive the global effects of a procedure without nested calls [10]. The global-effect nodes of a call statement in the ICFG will be replaced by a control flow graph, which is the output of this algorithm. We augment the intraprocedural system described in the preceding with a semantic function for the call nodes of an ICFG. The function *Transform*, used to define this semantic function, models stack

Transform$(A, [v_{expr1}/vp_1,\ldots, v_{exprk}/vp_k], [r_1/rp_1, \ldots, r_m/rp_m], s)$:

1. *For each formal reference parameter* $rp_1, \ldots, rp_m$ *insert an edge leading to those nodes that are pointed to by the corresponding arguments* r_i *for* $1 \leq i \leq m$. *Mark the variables* $rp_1, \ldots, rp_m$ *inserted into* A *as reference parameters.*

2. *For each formal value parameter* vp_i, $1 \leq i \leq k$, *allocate a blank node and insert an edge from* vp_i *to it. Mark the blank nodes as written to by statement* s. *If the formal parameter* vp_i *is of pointer type then insert for each edge that starts at a node reachable via the path* v_{expri} *another edge starting at the node pointed to by* vp_i *and terminating at the same target node. Mark the variables* $vp_1, \ldots, vp_k$ *inserted into* A *as value parameters.*

3. *Delete all variables not marked as reference or value parameter as well as the edges emanating from them. Delete all nodes not reachable from any variable and all edges attached to them. Merge all nodes that can be reached via identical sets of paths.*

Fig. 8. Treatment of a procedure call.

set-up and the parameter-passing mechanism. Fig. 8 contains an informal description of the function *Transform*. The semantic function of a call node is monotone, so that the data flow system can be used for safe interprocedural data dependence analysis.

8. An implemented graphical exploration tool

We have implemented the use of A/D graphs into our data dependence analysis and have received the first empirical results. The implementation is written in C, and analyzes a subset of Modula-2, which is defined by the chosen programming model. To obtain a better platform for further investigations, we developed our implementation in the form of a graphical exploration tool. The front end of our tool performs the syntax resp. semantic analysis of the program and transforms it into an abstract syntax tree. Thereafter it constructs a control flow graph of the program, upon which we then perform the data dependence analysis. The output of the system is a graphical representation of the control flow graph and data dependence graph of the program. We are also able to show the derived A/D graphs separately for each statement. Fig. 9 exhibits a snapshot of the graphical output of our exploration tool. Initial experiments were conducted to obtain preliminary data on the usefulness of our method. The empirical results are based on parts of four real application programs which we analyzed—the *Mesa* graphic tool, the computer algebra system *MAS*, the *DEC WRL* compiler, and a database program written in Modula-2.

As expected, we observed two sources of inaccuracy in the determination of data dependence with A/D graphs: *l*-bounding, resulting from the necessity for approximation in the handling of dynamic data structures, and control flow, resulting from safe assumptions about the actual execution paths. Preliminary experiments show that three-bounding of A/D graphs is good enough to obtain accurate results in most cases. Only in a few programs (mostly database applications) an increased l will lead to a

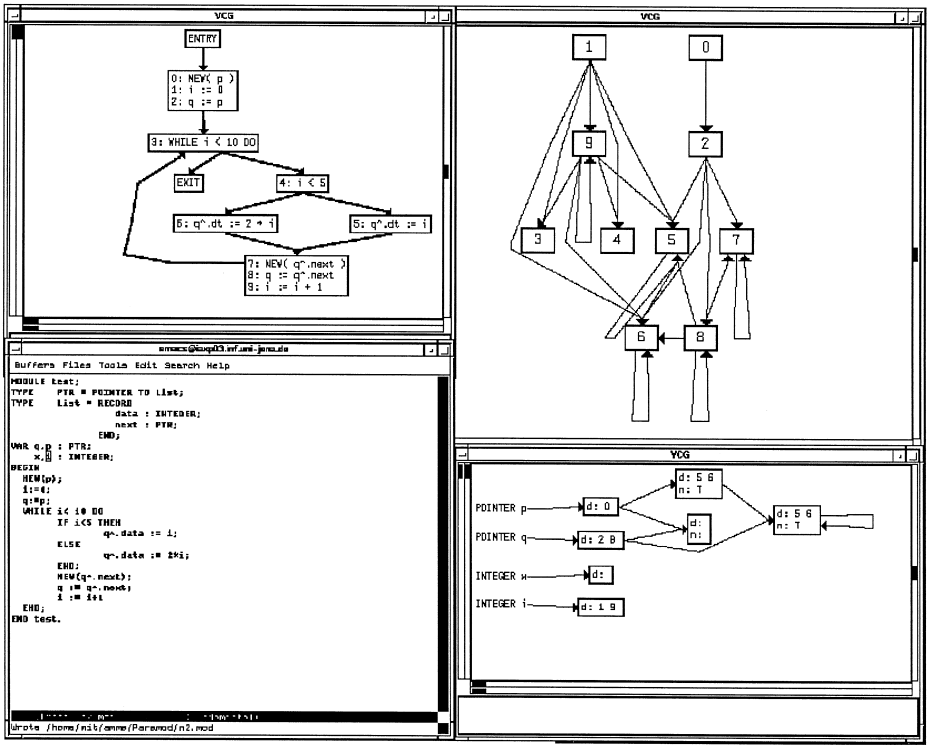

Fig. 9. Snapshot of the graphical exploration tool.

more accurate analysis. To measure the accuracy of our method, we chose the average number of data dependences per statement as our metrics. Analyzing our test programs resulted in an average of 2.24 data dependences per statement. Although the chosen measure is somewhat theoretical in nature, these results are very encouraging. In further research, we plan to do a more detailed study of the practical applications of our method.

In another investigation we analyzed the time behavior of our method. First we measured the required time for analyzing programs with this technique and then compared these results with the analysis used by a conventional optimizing compiler (*WRL* compiler from DEC). The analysis time required by our technique was on average about 1% of the time the *WRL* compiler needed for the total compilation of the same programs. This result reflects the practical significance of our method.

9. Extensions of A/D graphs

Without further restrictions, the number of nodes in an l-bounded A/D graph could grow exponentially with the number of variables. Although this case appears rarely in practice [13,12], we would like to rule out such a storage complexity. For this purpose

we use $k-l$-bounded A/D graphs. We call an l-bounded A/D graph $k-l$-bounded, if there does not exist any node with more than k outgoing edges, labeled with the same field name. As for the parameter l, we may globally fix the parameter k. When using the concept of $k-l$- bounded A/D graphs, we must extend the functions *Union* and *InsertEdge*: we must reduce each node that violates the well-definedness of a $k-l$-bounded A/D graph by insertion of condensation nodes.

We can expand the programming model to allow programs working on cyclic data structures. The simplest way to handle cyclic data structures is to replace each appearance of a cycle in an A/D graph by the insertion of condensation nodes. A more accurate treatment of cyclic data structures would be to label the nodes of an A/D graph with regular expressions instead of sets. Such a labeling technique is similar to that used in Ref. [14]. However, a labeling with regular expressions changes the admissibility of strong updates in an A/D graph. There, we have to introduce the concept of *safe* nodes and edges. By doing so, strong updates of our data structures are only allowed when reaching a node by a *safe* path.

The algorithm that we use for the determination of the global effect of a procedure call cannot handle recursive procedures in general. By using one of the known standard approaches (see, for example, Refs. [14,15]) and introducing special kinds of condensation nodes [12], we can extend our analysis method to handle recursive programs in a conservative way too.

10. Related work

Many algorithms have been developed to solve aliasing problems introduced through reference formals. A precise, polynomial-time algorithm for calculating alias information in FORTRAN programs is described for example in Refs. [16,26].

Conditional alias analysis has been suggested for interprocedural calculation of may-alias information in programs with pointers in Ref. [17]. This technique analyzes the code of a procedure under certain input assumptions and then combines the results of this conditional analysis with the situations that actually may occur during program execution. Based on this approach, algorithms for computing precise program-point-specific aliases in C programs with a single level of indirection [8] and approximate aliases for multiple levels of indirection [18] have been designed. Furthermore, in Ref. [8], a polynomial-time algorithm for computing must-alias information for data structures with a single level of indirection is presented. Based on conditional aliasing, in Ref. [2] the first polynomial-time approximate algorithm for interprocedural def-use associations for programs with single-level pointers is developed. Similar to the technique described above, the reaching definitions of a program statement are used to derive the def-use associations. Compared to our technique, this algorithm is more precise when recursive procedure calls are allowed. In contrast, for programs working on data structures with more than one level of indirection, our technique outperforms the latter, as they do not possess a practical algorithm for computing the must-alias information of a program.

An interesting technique for the calculation of may-alias information in programs with pointers is developed in Ref. [11]. This work proposes a memoryless model to

solve the may-alias problem, employing algebraic methods. Starting from a new semantic model for data aliasing, an algorithm is stated that allows extraction of precise may-aliases in a compact way for certain list manipulating programs. Due to the lack of a strong updating operation, this algorithm may yield spurious aliases. As mentioned above, the method offers no algorithm for the calculation of must-alias information. Therefore, this algorithm may be used for conflict analysis only.

A different approach is proposed by Hendren and Nicolau [19]. They construct an algorithm for interference analysis (a kind of conflict analysis) that manipulates so-called path matrices. This algorithm handles only acyclic data structures—lists and trees, and is unable to handle cyclic structures conservatively.

Jones and Muchnick [20] first analyzed how to track the shape of dynamic data structures in a Lisp-like language. They use graphs to represent the shape of the storage. The nodes in the graphs represent the objects in storage. A pointer reference between objects is represented by directed edges. In contrast to our technique, they associated each program point with a set of graphs. Most research in this area is based on this work.

Horwitz et al. [9] extended the graphs used in Ref. [20] by storing the most recent write- or read-access within the fields of the nodes. Our technique to note the last access to a storage object directly in the structure is taken from this work, where it is called *instrumented semantics*. They use a memory-based technique by labeling the nodes with location and use sets of graphs to represent the shape of the storage. In contrast to our work, they are only interested in intraprocedural analysis. Further, their method for data dependence analysis is, as a whole, mainly theoretical in nature and has never been implemented, obviously due to its complexity and its exorbitant storage demands.

To avoid some problems encountered by Horwitz et al. [9], we adopted techniques developed in Ref. [14] in the context of monotone data flow frameworks for alias analysis. In contrast to ours, this work is dedicated to conflict analysis only, and is applied to programs written in Lisp. Thus, their data flow information set and semantic functions are quite different. Larus and Hilfinger [14] extended the graphs used in Ref. [20] by adding unique labels to each node. They employ alias graphs as a data flow information set. Whereas we attach a set of paths to each node, they label the nodes with regular expressions. By using sets of variables, an implicit materialization (cf. Ref. [12]) is performed. Therefore, for a pointer assignment $X := Y$ we can do strong updates, i.e., the semantic function *DeleteEdge* always erases all edges emanating from a simple node. In this way our algorithm is more accurate. Due to their two-pass approach, their results are rather imprecise with respect to data dependence analysis, and the analysis is still very storage-inefficient.

Chase et al. [21] used storage shape graphs to represent the shape of the memory. The nodes of the graphs are labeled with the allocation site of the objects. In contrast to our technique and also to most techniques described above, they do no strong l-bounding to limit their graphs. To achieve the finiteness of their graphs, they control the number of nodes in a graph by the appearance of so-called deterministic variables.

Independently from our work, Sagiv et al. [12] developed an algorithm to determine the possible shapes that heap-allocated structures in a program can take on. Their method is quite accurate and can also be applied to cyclic data structures. As in our

method, manipulation of the representation is based on sets of access paths, distinguishing this work from previously published approaches. In contrast to our technique, they utilize a single condensation node in their data structure. They use an elegant form of materialization. Therefore, when information appears that is not denoted by the graph, a new node from the condensation node is materialized. However, they are not strictly interested in data dependence analysis. It is questionable if their technique is really applicable in compilers, because only may-alias information can be derived—although they denote a kind of must-alias information in their graphs.

Matsumoto et al. [22] developed a two-stage dependence analysis to handle Pascal or Fortran 90 programs with linearly linked lists or trees. The first stage is a safe alias analysis, where variables are supposed to be aliased except when definitely pointing to disjoint data structures. The second stage detects linearly linked lists, and then performs a dependence analysis between references pointing to the same list. This method cannot be used outside the scope of detecting linearly linked lists or trees. Even when handling these kinds of data structures, the accuracy of the method is poor due to very coarse approximations in the different phases, passed through the whole analysis. For instance, by this method one cannot infer that there is no true dependence between the second and the third loop in our example from Fig. 2.

11. 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. 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, and the analysis time required by our technique was on average about 1% of the time the *WRL* compiler needed for the total compilation of the same programs. The question of trade-offs between precision, storage use, and analysis time will be clarified after intensive experimental work in the future. As a further step we are attempting to extend the present research to programs with cyclic data structures. Our final aim is to be able to analyze any imperative program that does not use pointer arithmetic.

Acknowledgements

We would like to thank all members of the *ParaMod team*—in particular Peter Braun, Sven Bühling, Jürgen Frotscher, Sibylle Guder, and Thoralf Müller as well as Andreas Unger, for many helpful discussions on data dependence analysis in programs with pointers and for implementing the prototype. Moreover, we are grateful to

Elizabeth Kley for various comments on the English version. This paper is dedicated to the memory of Johannes Röhrich, who provided the initial support and supervision for this work.

References

[1] S. Amarasinghe, J. Ander, C.W. Tseng, An overview of the SUIF compiler for scalable parallel machines, Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, 1995, pp. 662–667.
[2] H.D. Pande, W.A. Landi, B.G. Ryder, Interprocedural def-use associations for C systems with single level pointers, IEEE Trans. Software Eng. 22 (1994) 385–403.
[3] D.I.S. Marx, P.G. Frankl, The path-wise approach to data flow testing with pointer variables, Proceedings of the 1996 International Symposium on Software Testing and Analysis, San Diego, 1996, pp. 135–146.
[4] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Computers, ACM Press, New York, 1990.
[5] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixed points, Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages, New York, 1977, pp. 238–252.
[6] M.S. Hecht, Flow Analysis of Computer Programs, Elsevier, Amsterdam, 1977.
[7] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.
[8] W. Landi, B.G. Ryder, Pointer-induced aliasing: a problem classification, Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages, Orlando, 1991, pp. 93–103.
[9] S. Horwitz, P. Pfeiffer, T. Reps, Dependence analysis for pointer variables, Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, Portland, 1989, pp. 28–40.
[10] W. Amme, E. Zehendner, A/D graphs: a data structure for data dependence analysis in programs with pointers, Proceedings of the 4th PASA Workshop Parallel Systems and Algorithms (Jülich, 1996), World Scientific Publishing, Singapore, 1997, pp. 259-278.
[11] A. Deutsch, Interprocedural may-alias analysis for pointers: beyond *k*-limiting, Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages Design and Implementation, Orlando, 1994, pp. 230–241.
[12] M. Sagiv, T. Reps, R. Wilhelm, Solving shape-analysis problems in languages with destructive updating, SIGPLAN-SIGACT Symposium on Principles of Programming Languages, FL, 1996, pp. 16–31.
[13] W. Landi, B.G. Ryder, S. Zhang, Interprocedural modification side effect analysis with pointer aliasing, Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (1993) 56–67.
[14] J.R. Larus, P.N. Hilfinger, Detecting conflicts between structure accesses, Proceedings of the ACM SIGPLAN '88 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices 21 (1988) 21–34.
[15] R.E. Tarjan, A unified approach to path problems, J. ACM 28 (1981) 577–593.
[16] K. Cooper, K. Kennedy, Fast interprocedural alias analysis, Proceedings of the 16th ACM Symposium Principles of Programming Languages, Austin, TX, 1989, pp. 49–59.
[17] W. Landi, Interprocedural aliasing in the presence of pointers, PhD thesis, The State University of New Jersey, Rutgers, 1992.
[18] W. Landi, B.G. Ryder, A safe approximation algorithm for interprocedural pointer aliasing, Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, 1992, pp. 235–248.
[19] L.J. Hendren, A. Nicolau, Parallelizing programs with recursive data structures, IEEE Trans. Parallel Distributed Syst. 1 (1) (1990) 35–47.
[20] N. Jones, S. Muchnick, Flow analysis and optimization of Lisp-like structures, in: S. Muchnick, N. Jones (Eds.), Program Flow Analysis: Theory and Application, Prentice-Hall, Englewood Cliffs, NJ, 1979, pp. 102–131.

[21] D.R. Chase, M. Wegman, F.K. Zadeck, Analysis of pointers and structures, Proceedings of the ACM SIGPLAN '90 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices 25 (1990) 296–310.
[22] A. Matsumoto, D.S. Han, T. Tsuda, Alias analysis of pointers in Pascal and Fortran 90: dependence analysis between pointer references, Acta Informatica 33 (1996) 99–130.
[23] C.P. Chu, L.D. Carver, Parallelizing subroutines in sequential programs, IEEE Software 11 (1) (1994) 77–85.
[24] J.B. Kam, J.D. Ullman, Global flow analysis and iterative algorithms, J. ACM 23 (1976) 158–171.
[25] J.B. Kam, J.D. Ullman, Monotone data flow analysis frameworks, Acta Informatica 7 (1977) 305–317.
[26] E.M. Myers, A precise interprocedural data flow algorithm, Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages, 1981, pp. 219–230.
[27] H.D. Pande, Compile time analysis of C and C++ systems, PhD thesis, Rutgers University, NJ, May 1996.