Dependency Analysis of Recursive Data Structures Using Automatic Groups

D. K. Arvind and T. A. Lewis

Department of Computer Science, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland.

Abstract. This paper describes a framework for the dependence analysis of complex, pointer-based data structures. The analysis method can handle any structure which can be represented as an automatic group. A simple recursive function is analysed to illustrate the method. The paper demonstrates how structures described in languages such as Graph Types and ASAP can be integrated in the framework.

1 Motivation

The static analysis of dependencies in complex, pointer-based data structures is presented a view towards better informing the parallelisation process. The approach in this paper assumes that the programmer provides additional, explicit structural information in the definition of the data structures. The method of analysis that is proposed can handle any such structure which can be represented as an automatic group [ECH+92].

Figure 1 encapsulates the proposed framework. The problem conveniently divides itself into three parts - specification of structural information, conversion of this information into an internal representation and finally the analysis of dependencies. We are primarily concerned with the latter two parts.

The rationale behind the use of automatic groups for internal representation can be explained as follows. [Mic95] introduces the idea of a graph of nodes and pointers as being the Cayley graph of a group. The nodes represent group elements and the edges/pointers represent the action of one element on another. We extend this idea to enable programmers to specify the structure within the group, so we require a class of groups where patterns of linkages could be expressed. These specifications also need to be manipulated so that equality of group elements (the *word problem*) could be decided. Automatic groups fit these requirements well. We are not strict about the group structure of these data structures; for example, we do not require every pointer to have a corresponding inverse pointer. The group is used primarily as an underlying representation of linkages within a structure.

The manual creation of such automatic group representations is potentially awkward and error prone, even for simple linkages. We believe that a more suitable approach would be to use established 'front-end' description languages to specify the structures and then automatically convert them to an AG representation. In such a framework, expressions denoting dependencies can be automatically converted to produce an equivalent AG with the correct properties. We will demonstrate that this provides a sufficiently general and powerful framework for dependence analysis.

Some methods of describing structures that we have considered so far are: Graph Types [KS93], Shape Types [FM97], ASAP [HHN94] and our format based descriptions [LA97]. We can mechanically translate the structural information in the latter two description types into AGs. We cannot, at present, do the same for the earlier two types of descriptions.

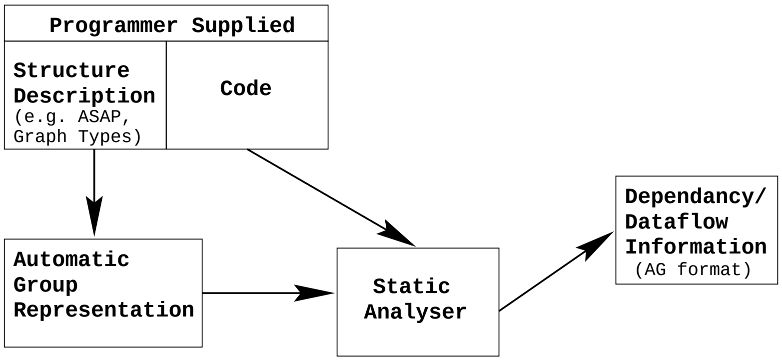


Fig. 1. Framework

2 Definitions

This section provides a brief introduction to Automatic Groups, in the context of understanding the method of analysis. The reader is directed to [ECH$^+$92], for a more detailed exposition of automatic groups. An automatic group is a structure where each node is named by a string of symbols. We call such a string a *path name*: it describes the path of pointers from the root of the structure. These symbols are the *generating* directions and correspond to the underlying tree pointers of the data structure. For each additional link direction, a description exists in the form of a finite state automaton, specifying the set of nodes which are linked by this direction.

A (deterministic) finite state automaton (FSA) is a simple model of computation, where an automaton reads a string of symbols one at a time and moves from one state to another. The string is said to have been accepted when the automaton ends in one of the accept states. The automaton consists of a finite set of states, S, and a transition function, $F : S \times A \rightarrow S$, which gives the next state for each of the possible input symbols in A. A *two-variable* FSA differs, in that the automaton attempts to accept a pair of strings and inspects one symbol from each of them at each transition. It can also be thought of as a one-variable FSA, but with the set of symbols extended to pairs of symbols- the set $\{A \times A\}$. There is a subtlety however, in that we may wish to accept strings of unequal lengths, so the shorter one is padded with the additional '$-$' symbol (but we do not allow the double padded symbol $(-,-)$). This makes the actual set of symbols $((A \cup \{-\}) \times (A \cup \{-\})) \backslash (-,-)$: for example, if we consider a tree structure with two pointers l and r, the set of symbols will be $\{(l,l),(l,r),(l,-),(r,l),(r,r),(r,-),(-,l),(-,r)\}$.

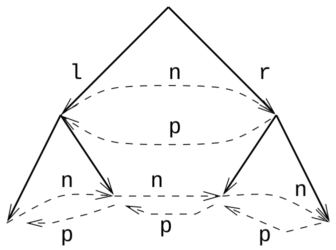
The description with each link direction, l, is a two-variable finite state automaton, called the multiplier of that link, and denoted M_l. There is also a word acceptor automaton, W, that accepts the path names that correspond to valid parts of the data structure.

Automatic groups are a good choice of description, since they can be readily manipulated by using existing regular language theory. In particular, we can produce FSAs for words of directions from the individual FSAs for each direction. We can also perform basic logical operations (AND, OR, NOT, $\forall$ and $\exists$) with the FSAs and hence the sets of nodes.

We can also use other constructions from Automatic Group theory. Given FSAs for individual directions, we would wish to combine the multiplier FSAs into a multiplier for words of directions that appear in the code. For instance, we may wish to find those parts of the structure which are accessed by the word, `a.b`, given the appropriate FSAs for `a` and `b`. This can be naturally performed in the AG approach when given two FSAs, $\mathcal{F}_1$ and $\mathcal{F}_2$, their composition is denoted by $\mathcal{F}_1.\mathcal{F}_2$ - see [ECH+92] for the details of its construction. We also mention the inverse FSA, denoted by $\mathcal{F}^{-1}$, which is built by swapping the pair of letters in each transition of $\mathcal{F}$.

3 The Specification

The specification using automatic groups representation will be illustrated by following through an example of a pair of simple recursive functions (Fig. 4) over the given structure. Fig.2 illustrates a binary tree, each node of which has a left and right subtree, and in addition the nodes in each level are connected as a doubly-linked list, with 'next' and 'previous' pointers. The description declares each direction as either a generator or a link, followed by the definitions of each. The l and r directions are generators, and so do not require an explicit FSA description. The n direction is given as a FSA with two states, one start and one accepting, followed by a list of transitions from each state. The p direction is defined as the inverse of the n direction, with the syntax '`(n)^`'. Fig. 3 gives the structure of the AG representation which is built automatically from this description.




```
Generator l, r;
Link n, p;
{l:  }
{r:  }
{n:
  States 2;    Start 1;    Accept 2;
  Transitions[
            [(l,l)->1;(r,r)->1;(l,r)->2;]      //From state 1
            [(r,l)->2;]                        //From state 2
  ]
}
{p: (n)^}
```

Fig. 2. A binary tree with additional ‘next’ and ‘previous’ pointers, with its code description.

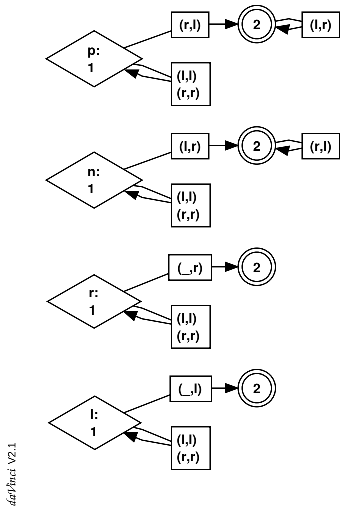


Fig. 3. AG representation of the description. The rhombus nodes are start nodes and are labelled with the direction name e.g. (l). Other states are circles and are labelled with the state number. The boxes hold the pair of symbols for that transition. Double borders indicate an accept state.

```
Traverse(w) {
     if (w ==NULL) return;
  A: Update(w.l);
  B: Struct(w) = Struct(w.p);
  C: Traverse(w.r);
}
Update(w) {
     if (w ==NULL) return;
  D: Struct(w) = Struct(w.p.l.n);
  E: Update(w.l);
  F: Update(w.r);
}
```

Fig. 4. Sample recursive functions. The A, B, C, D, E, F are statement labels and not part of the original code.

4 The Analysis

The recursive functions in Fig. 4 operate over this binary tree structure. The function `Traverse` moves down the left hand side of the tree, copying values to adjacent nodes. It then calls `Update` on the right hand sub-tree, which updates each node in that whole tree by copying from a nearby node.

The descriptions for single directions can be used to build composite directions. For instance, the 'read' in statement D requires the composite *read word*: $p.l.n$. Each runtime statement can be labelled by a *control word*, a word from the set given by the regular expression $(A|C|E|F)^*.(B|D)$. Each time we call the function recursively from statement A, we append a l direction to the parameter w and for statement C we similarly append a r. The same is true for statements E and F. This information can be captured in a FSA that converts a given control word into the value of the parameter w, for that function call. This FSA is given in Fig. 5.

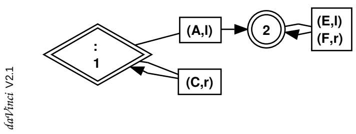


Fig. 5. FSA that maps control words to values of w.

We should mention that in order to perform this analysis, the code should not recurse in a direction that is not a generator; otherwise the construction of this particular FSA is a difficult problem, in some cases impossible, since the required language is not a regular one. This is a restriction of the method. We can still produce some information, however, by *unrolling* the recursion, and analysing within the function.

For each read statement we can append the statement symbol and the read word for that statement. This produces a FSA $\mathcal{F}_r$ that maps from any control word to all the nodes of the structure that can be read by that statement. Similarly, we can produce a FSA that maps from control words to nodes which are written, denoted by $\mathcal{F}_w$. These read and write FSAs are given in Fig. 6.

We can now describe all the nodes which are read from and written to, by any statement. By combining the read and write FSAs we create an FSA that links statements when one writes a value that the other reads, *i.e* the

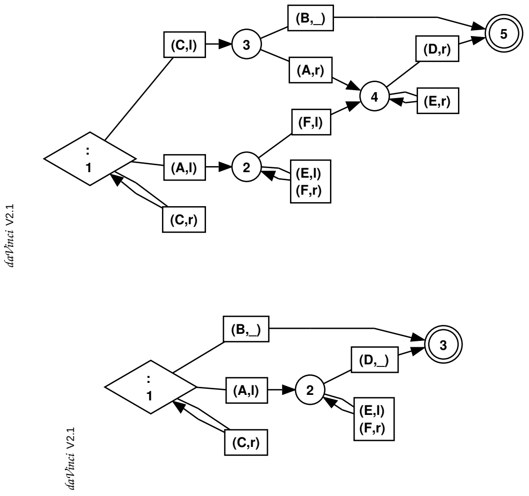


Fig. 6. Read and write FSAs.

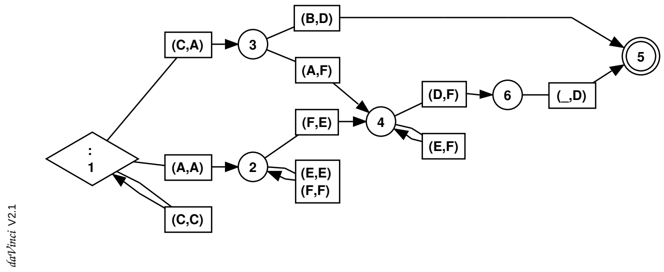


Fig. 7. Conflict FSA.

statements *conflict*. The conflict FSA $\mathcal{F}_{conf}$ is given by $\mathcal{F}_r.(\mathcal{F}_w)^{-1}$. The FSA in Fig.7 is the conflict FSA which has also been combined (ANDed) with an FSA $\mathcal{F}_{causal}(X,Y)$ which accepts control words X,Y, only if Y occurs before X in the sequential running of the code. This removes any false sources that actually occur after the statement.

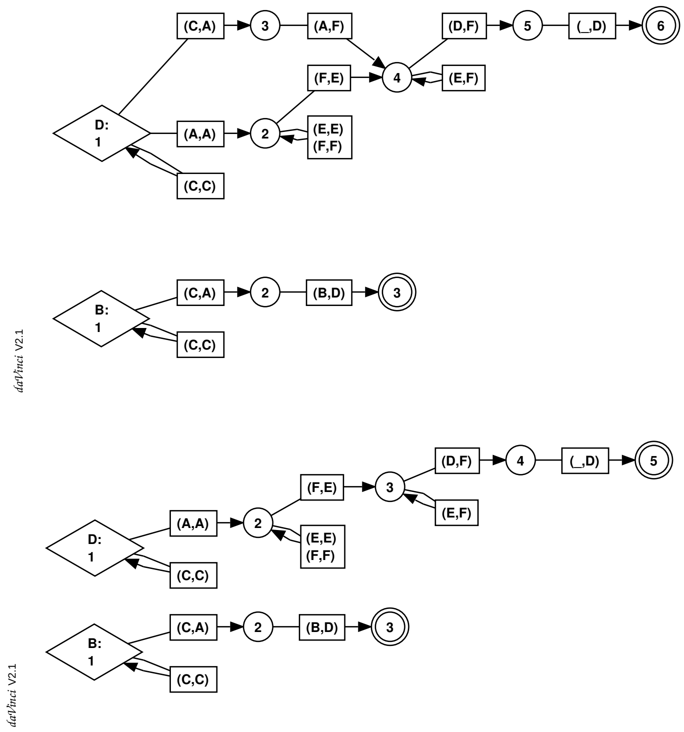


Fig. 8. (Above) Potential sources. (Below) Pruned to produce dataflow sources.

The final step is to extract dataflow information. If the statement X depends on Y and Z, and Z occurs after Y, then we can remove Y as a potential source. Removing all such sources, will provide the dataflow information. We can prune these false sources by splitting the conflict FSA up into components- one for each read statement, then removing multiple sources. Consider the FSA for the potential sources for statement B. in Fig.8, in particular the (C,A) and (C,C) transitions from the start node. For a given sink with a C statement, the (C,C) transition will always produce a potential source later than the (C,A) transition, and hence will overwrite the value. The earlier (C, A) transition can therefore be removed. In general, in any state with transitions, (X, Y) and (X, Y'), with $Y < Y'$, (X, Y) can be removed. We can then minimise the resulting FSA, by possibly removing unreachable states. When this is done on the entire FSA, the resulting dataflow FSA is shown in Fig.8.

5 Conditional Statements

In general we do not assume whether a statement is actually executed. This means we cannot produce full dataflow information, since we cannot prune the FSA as mentioned in the last section, if we are unsure as to whether a read or write occurs at run-time. This section describes how we could incorporate some knowledge into our model.

We assume that the function begins with a *terminating condition*, a statement of the form, `if T(w) then return`, where `T(w)` is a predicate on the function parameter `w`. We restrict T to be a regular language with an associated FSA which we will denote by $\mathcal{T}$. Note that we do not require the programmer to supply these FSAs explicitly, and for certain simple classes of predicate (such as the ones that select paths of length greater than a fixed constant) we could generate them automatically. We do restrict these predicates, and require the property: $T(x) \Rightarrow T(x.y) \forall y$. This is needed so that we can determine whether a particular control word is executed without having to examine all shorter words for the one that has terminated the recursion.

In addition, each statement or function call C can have a *conditional guard* P_C, which determines if the statement is executed. For a statement C, this is equivalent to the code fragment '`if (P(w)) then C;`'.

Our aim is to combine this conditional information into one FSA, $\mathcal{E}$, that decides whether the statement corresponding to a given control word is executed. Firstly, we need to convert the FSAs that operate on path names to ones that operate on control words. Then we merge the conditionals that guard the recursive calls into the termination FSA. Finally, we create the FSA, $\mathcal{E}$, by combining the termination and conditional guards of the read/write statements.

5.1 Merging call guards and the termination FSA

Our recursive function has a set of recursive call statements labelled RC_i of the form:-

$$\texttt{if} \quad (RP_i(w)) \quad \texttt{then} \quad F(w.g_i)$$

We convert the $\mathcal{R}P_i(w)$ to $\mathcal{RP}'_i(W)$ that operate on control words, by applying S, then applying NOT so that it accepts the words that halt the recursion. We now append $S(g_i)$ to this FSA to produce $\mathcal{RP}''_i(W)$, so that $\mathcal{RP}''_i$ accepts $W.S(g_i)$ if, and only if, $\mathcal{RP}''_i$ accepts W. The set of $\mathcal{RP}''_i$ now acts as additional termination conditions, and can be ANDed together with $\mathcal{T}$.

5.2 Completion of the executing FSA

To determine whether a statement corresponding to a particular control word does get executed at runtime, we need to ensure that the actual recursive call is executed, and the conditional guard for that particular statement is true. For each statement labelled C_i, we convert the conditional guard $\mathcal{P}_i$ to $\mathcal{P}'_i$ using S. We append the label C_i to both $\mathcal{P}_i$ and $\mathcal{T}$ and then AND the pair. Finally, the executing FSA $\mathcal{E}$ is produced by ORing all these FSAs together.

6 Front-end Description Languages

The paper so far has described the analysis of the dependences in the framework of AG descriptions of structural information. In this section we will elaborate on how description languages, in particular ASAP, Graph Type and Shape Types can be integrated in the framework. In order to achieve this the AG description is augmented with extra information, *i.e.* we define two further FSAs $\mathcal{F}_a$ to hold the aliasing information, and $\mathcal{F}_d$ to hold the disjointedness information.

6.1 Integrating ASAP description

Linkage information in a structure is expressed in ASAP via a series of axioms. The axioms come in two basic types, either describing when nodes are equal or unequal. The equal axioms describe aliasing information, the unequal ones disjointedness.

There are three kinds of axioms:

$$\begin{array}{ll} 1) & \forall p \quad p.RE1 \neq p.RE2 \\ 2) & \forall p \neq q \quad p.RE1 \neq q.RE2 \\ 3) & \forall p \quad p.RE1 = p.RE2 \end{array}$$

where p, q are any path names and $RE1, RE2$ are regular expressions over A^* .

Given a regular expression, $RE1$, it is trivial to convert it into a two variable 'appended' FSA that accepts $(p.RE1, p)$ for all p, and is denoted by Ap_{RE1}. In addition, using $\mathcal{F}_{\neq}$ (the FSA that accepts a pair of paths if they are not equal) we can produce expressions that convert axioms of each type to FSAs that accept the equivalent pairs of words.

$$\begin{array}{ll} 1) & \mathrm{Ap}_{RE1}.(\mathrm{Ap}_{RE2}^{-1}) \\ 2) & \mathrm{Ap}_{RE1}.F_{\neq}.(\mathrm{Ap}_{RE2}^{-1}) \\ 3) & \mathrm{Ap}_{RE1}.(\mathrm{Ap}_{RE2}^{-1}) \end{array}$$

We then build $\mathcal{F}_d$ by ORing together all the FSAs produced from axioms 1 and 2, and $\mathcal{F}_a$ by ORing the ones from axiom 3.

6.2 Graph Types

We cannot convert automatically from a Graph Types description to our AG description. Instead we have focussed on extending our system to handle structures that can be described using this formalism. We need to introduce the concept of different types at each node. As it stands, the structures we have been considering are *homogeneous*, the allowed number and type of pointers at each node are identical. We extend the descriptions to allow each pointer direction to be of a different type, and introduce the concept of type variant. Each of these is encoded into the path name, so that queries on type variants can be expressed. As an example, consider how we could describe a binary tree with linked leaf nodes in C.

```
union TreeNode {                  struct Internal {
      Internal* intern;                   int ival;
      Leaf*    leaf;                      TreeNode *left;
};                                        TreeNode *right;
                                        };

struct Leaf {
      int lval;
      TreeNode *next;
};
```

The path name for a node in the structure is then a word over the alphabet $\{$*intern, leaf, left, right, next*$\}$. Of course, some path names are invalid, *intern* must always be followed by either a *left* or *right*. Such properties can be readily expressed using FSAs and incorporated into the $\mathcal{W}$ part of the AG description, and derived automatically from the above style of code. In addition, we will need the programmer to supply descriptions of where the directions- $\{$*left, right, next*$\}$- actually point.

6.3 Shape Types

We are presently investigating the integration of Shape Types [FM97] in the framework. Structure linkages in Shape Types are represented by a multi-set of field and node names. Each field name is followed by the pair of nodes that it links. The structure of the multi-set is given by a context-free grammar. In addition, operations on these structures are presented as *transformers*, *i.e.* rewrite rules that update the multi-sets. The authors present the method as a way of proving program correctness: deciding whether the structure of the grammar is preserved by the rewrite rules.

Since Shape Types are based upon context-free languages, and Automatic Groups use less general regular languages, the class of structures that they describe will be larger. As a result, we cannot convert automatically from a Shape Types description into a AG description; there will be some structures that just will not be representable. The authors use example structures such as skip lists of level two, binary trees with linked leaves and left-child, right-sibling trees. These can all be expressed in AG descriptions.

7 Complexity

Since finite state machines are the fundamental object being manipulated using this method, the complexity of our analysis relate to the complexity of the underlying FSA operations.

If n_g is the number of generating directions, n_l the number of lines in the code, then the size of the alphabet of each two-variable FSA is $(n_g + n_l + 1)^2 - 1$. For a reasonably-sized program, the number of lines in the code will dominate, making the complexity $O(n_l^2)$. The memory requirements for such a FSA with n_s states will be $O(n_s.n_l^2)$; storing the information sparsely will decrease this. However, the number of states can potentially increase dramatically: given two FSAs with n and m states, the FSA built by the operations AND, OR can have up to $n \times m$ states. When combining two FSAs to form a composite, the combined FSA can have up to $2^{n \times m}$ states. There is therefore opportunity to produce very large FSAs, with large memory requirements, and computation time.

For the FSAs automatically obtained from ASAP axioms, as described previously, the descriptions for a sparse matrix produce an FSA of approximately 150 states. Obviously, such large FSAs would pose a problem and restrict the amount of analysis that could be carried out on real programs.

In practice, though, the number of states often remains manageable. For example, using the worst case estimates described previously, the combined word, $p.l$, required for analysis of the read in statement D in Fig. 4, could be expected to require 16 states. In fact it requires only 3. The reason is that these simple FSAs are sparse, there being only a couple of transitions from each state.

8 Conclusion

This paper has described a framework based on an underlying Automatic Group representation to analyse the dependencies in complex, pointer-based data structures. The approach assumes that the programmer supplies explicit structural information. We have demonstrated that description languages such as Graph Types and ASAP can be integrated into this framework.

We have demonstrated that this analysis can be carried out on a class of programs which consist of many possibly recursive functions of a single parameter. We are presently working towards extending this method to handle functions that recurse with multiple parameters.

Our framework allows us to combine ASAP descriptions, with stricter AG descriptions. Thus we allow a broader range of information to be supplied by the programmer. In return, the dependency information given can be more detailed, with a finer grained resolution of dependencies between individual runtime statements.

Acknowledgements

Tim Lewis was supported by a postgraduate studentship from Engineering and Physical Sciences Research Council (EPSRC). We would like to thank Christian Lengauer for useful discussions - the visits to Passau were funded under British Council/ARC grant entitled *Formal Parallelisation of Program Specifications*.

References

[ECH+92] David B. A. Epstein, J. W. Cannon, D. E. Holt, S. V. F. Levy, M. S. Paterson, and W. P. Thurston. *Word Processing in Groups.* Jones and Bartlett, 1992.

[FM97] P. Fradet and D. Le Metayer. Shape types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[HHN94] Joseph Hummel, Laurie J Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[LA97] Tim Lewis and D. K. Arvind. Static analysis of recursive data structures. In Z. Li et al, editor, *Languages and Compilers for Parallel Computing, 10th International Workshop LCPC'97*, number 1366 in Lecture Notes in Computer Science, pages 423–426, Minneapolis, Minnesota, USA, August 1997. Springer-Verlag.

[Mic95] Olivier Michel. Design and implementation of 81/2, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, December 1995.