Dependency Analysis of Recursive Data Structures Using Automatic Groups

D. K. Arvind and T. A. Lewis

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

Abstract. A framework is described for the static analysis of less regular, pointer-based data structures which have been augmented with explicit structural information. The framework has three distinct parts to it - the specification of structural information, their translation into an internal representation as automatic groups, and the dependency analysis. The application of the method to a case study in fluid flow simulation is described.

1 Motivation

This paper deals with the static analysis of dependencies in complex, pointer-based data structures, which are augmented with explicit structural information. The method of analysis can handle structures which can be represented as an automatic group [ECH+92]. The proposed framework, illustrated in Figure 1, can be divided into three distinct parts - the specification of structural information in the data structure, their translation into an internal representation and the analysis of dependencies. This paper is principally concerned with the latter two parts.

The idea of a graph of nodes and pointers being the Cayley graph of a group was introduced in [Mic95]. The nodes represented group elements and the edges/pointers represented the action of one element on another. We have extended this idea to enable programmers to specify the structure within the group; we therefore 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 (AG) 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 prone to error, 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.

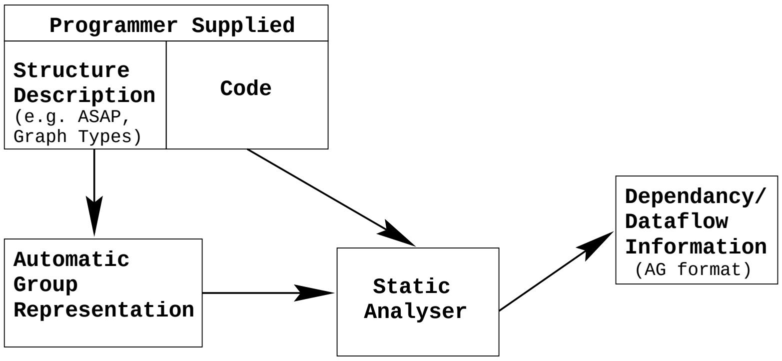
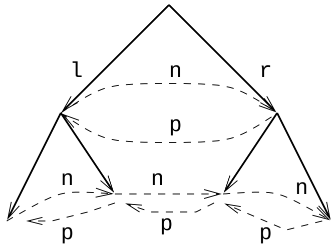


Figure 1. Framework

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 called 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 example, consider a simple binary tree, where the two sub-trees at each node are labelled l and r. The nodes can be described by a string of l and r symbols, indicating the path taken from the root node at the top of the tree.

We may, in addition, have *link* directions. For instance, in a binary tree, we may want to link the nodes in each level of the tree into a doubly-linked list. Then each node will have two link directions - n for the next (eastwards) and p for the previous (westwards) nodes. The nodes can still be uniquely described by strings of l and r symbols. A description exists, for each additional link direction, in the form of a two-variable, finite state automaton (FSA), which specifies the set of nodes which are linked by this direction. The *two-variable* FSA attempts to accept a pair of strings and inspects one symbol from each of them at each transition. It can also be viewed as a one-variable FSA, but with the set of symbols extended to pairs of symbols - the set




```
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)^}
```

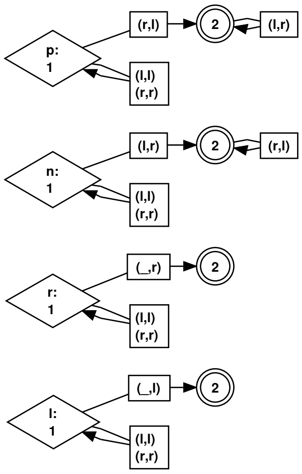


Figure 2. A binary tree with additional 'next' and 'previous' pointers, with its source code description. The graphical representation of the AG description is shown below. The diamonds are start nodes and are labelled with the direction name e.g. (l). Other states are circles and are labelled with the state number. The boxes hold the pair of symbols for that transition. Double borders indicate an accept state.

$\{A \times A\}$. 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 (we do not, however, allow the double padded symbol $(-,-)$). This makes the actual set of symbols $((A \cup \{-\}) \times (A \cup \{-\}))\backslash(-,-)$: for example, for our binary tree structure with two generators 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, d, is a two-variable finite state automaton, called the multiplier of that link, and denoted M_d. There is also a word acceptor automaton, W, that accepts the path names that correspond to valid parts of the data structure.

Consider the task of producing the multiplier FSA for the n link direction. When moving eastwards from a node with a pathname ending in a l symbol, the last l is changed to a r, to obtain the name of the target node. For a name that ends with a r symbol, $x.r$ say, the target node can be defined in a recursive manner: if the node next to x is known, then a l symbol is appended to it, to produce the final target node. This information can be expressed by stating that the next direction links nodes of the format $x.l.r^p$ to nodes given by $x.r.l^p$, where p is any integer (or zero), and x is any path name. This description can be converted to a FSA along with others for the l, r and p directions. The generator FSAs (l and r) are simpler to understand, since they accept pairs of paths that are indentical up to the the point where the first terminates and the second has an additional symbol (either an l or r). These generator descriptions do not need to be described by the programmer.

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 perform basic logical operations (AND, OR, NOT, $\forall$ and $\exists$) with the FSAs, and by extension, on the sets of nodes that they represent. Other constructions from Automatic Group theory can be used; for instance, given FSAs for individual directions, one may wish to combine the multiplier FSAs into a multiplier for words of directions that appear in the code, e.g. to find those parts of the structure which are accessed by the word, `a.b`, when given the appropriate FSAs for `a` and `b`. This can be naturally performed in the AG approach - 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. The inverse FSA, denoted by $\mathcal{F}^{-1}$, is built by swapping the pair of letters in each transition of $\mathcal{F}$.

3 The Programming Model

The automatic groups representation will be illustrated by applying it to an example involving a pair of simple recursive functions (Fig. 3) over a particular 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 - *start* and *acceptance*, 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. 2 gives the structure of the AG representation which is built automatically from this description.

The implementation analyses a fairly restricted programming language that manipulates these structures. The code accesses one global data structure. Data elements are accessed via pathnames, which are used as a replacement for conventional pointers. The code consists of a number of possibly mutually recursive functions. These functions take one parameter `w` (a pathname), and return `void`. It is assumed that the first (`main`) function is called with the root path name. Each function may make possibly recursive calls to other functions using the syntax '`FuncName(w.g)`' where `g` is any generator.

It should be clarified at this stage that the analysis assumes that the code does not recurse in a direction that is not a generator; otherwise the construction of the map from control words to pathnames ceases to be representable by a regular language. This is a restriction of the method. One can still produce some information, however, by *unrolling* the recursion, and analysing within the function. Another related problem is when a function calls another one with the parameter `w`, *e.g.* '`FuncName(w)`'. This can be handled provided that it does not occur within a mutually recursive set of functions, *i.e.* the function can only be called a finite number of times.

The basic statements of the code are *reads* and *writes* to different parts of the structure. A typical statement would be of the form '`Struct(w.a) = Struct(w.b)`' where `w` is a variable and `a` and `b` are words of directions. This statement denotes the copying of an item of data from `w.b` to `w.a`, within the structure `Struct`.

Another observation about these static control programs is that if run-time dependencies are disallowed in the code, then one may as well run the program and trace its dataflow directly. Therefore, variable sized structures are allowed, delimited by `NULL` pointers. Conditional statements of the form '`if (pathname == NULL) then ...`' or '`if (pathname != NULL) then ...`' are also allowed. This however restricts the type of analysis that can be done. For example, any dataflow analysis has the potential to become inaccurate when one cannot decide statically if a statement is executed or not. One can still find sets of nodes accessed by a particular statement and deduce coarser-grained parallelism.

4 The Analysis

The recursive functions given in Fig. 3 operate over a 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.

```
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);
}
```

Figure 3. Sample recursive functions. The A, B, C, D, E, F are statement labels and not part of the original code.

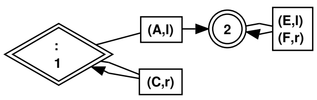


Figure 4. FSA that maps control words to values of w.

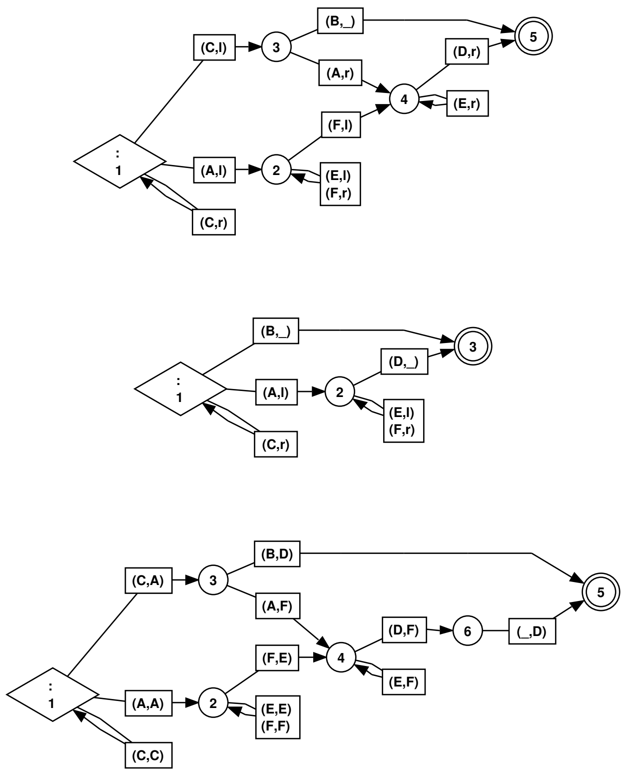


Figure 5. Read (Top) Write (Middle) and Conflict (Bottom) FSAs.

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 the function is called recursively from statement A, a l direction is appended to the parameter w, and similarly for statement C, a r is appended. 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. 4.

One can append, for each *read* statement, 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. 5.

We can now describe all the nodes which are read from and written to, by any statement. By combining the FSAs for read and write, a new FSA is created that links statements when one writes a value that the other reads, *i.e* the statements *conflict*. The conflict FSA $\mathcal{F}_{conf}$ is given by $\mathcal{F}_r.(\mathcal{F}_w)^{-1}$. The FSA in Fig.5 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.

The final step would be to extract dataflow information. If the statement X depends on Y and Z, and Z occurs after Y, then one can remove Y as a potential source. Removing all such sources, will provide the dataflow information. One can prune these false sources by splitting the conflict FSA up into components - one for each read statement, and then removing multiple sources. Consider the FSA for the potential sources for statement B in Fig.6, 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 will therefore overwrite the value. The earlier (C,A) transition can now 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. The resulting dataflow when this is performed on the entire FSA FSA is shown in Fig.6.

5 Conditional Statements

One cannot always predict at compile time whether a statement will actually be executed. This means that full dataflow information cannot be produced, since we cannot prune the FSA as mentioned in the last section. This section describes how one 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

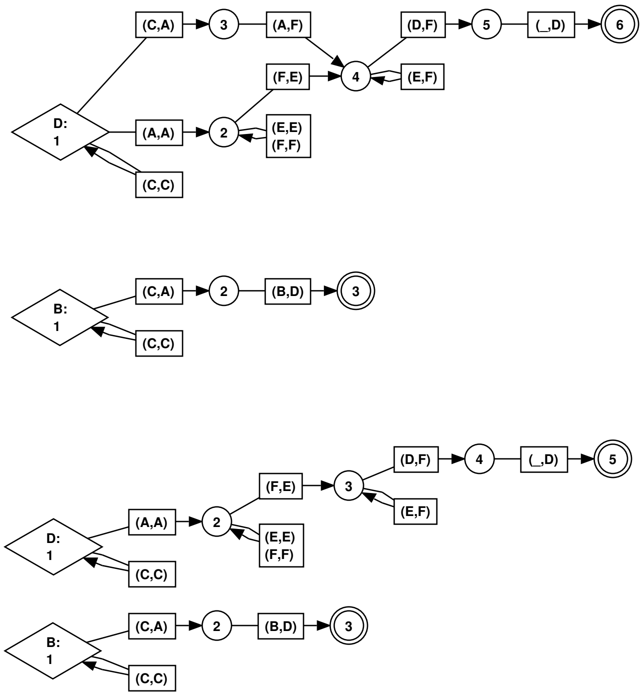


Figure 6. (Above) Potential sources. (Below) Pruned to produce dataflow sources.

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 and 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 briefly touch on a couple of description languages - ASAP [HHN94] and Graph Type [KS93] - which can be integrated in the framework. This will be achieved by augmenting the AG description with extra information, *i.e.* two further FSAs are defined - $\mathcal{F}_a$ and $\mathcal{F}_d$ to hold the aliasing and disjointedness information, respectively.

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, which describe when the 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

One cannot automatically convert from a Graph Types description to the AG one. We have instead focussed on extending our system to handle structures that can be described using this formalism. We need to introduce the idea of different types at each node. As it stands, the structures that have been considered are *homogeneous*, with the allowed number and type of pointers at each node being 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. Consider the case of a binary tree with linked leaf nodes in C language.

```
union TreeNode {
      Internal* intern;
      Leaf*    leaf;
};
struct Leaf {
      int lval;
      TreeNode *next;
};
```

```
struct Internal {
        int ival;
        TreeNode *left;
        TreeNode *right;
        };
```

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.

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 a potential to produce very large FSAs, with large memory requirements, and computation time.

In the case of FSAs obtained automatically from ASAP axioms, as described previously, the resulting FSAs can be significantly large. For example, the descriptions for a sparse matrix produce a disjointness FSA of approximately 150 states. Obviously, such large FSAs would pose a problem and impact on the analysis that could be carried out on real programs.

In practice, however, 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. 3, 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 A Case Study: Fluid Flow Simulation

We have applied this description and analysis to the Fast Multipole Method Vortex algorithm which is used in fluid flow simulations.

A tree is data structure of choice, with l and r pointers at each node. We also have additional pointers, n and p, that link nodes at each level into a linked list, and a *spiral* pointer s, that joins the nodes in a spiral from bottom left of the tree to the root.

```
FindLeftmostLeaf(w) {
   if (w.l.l == NULL) PhaseTwo(w);
   FindLeftLeaf(w.l);
}
PhaseOne(w) {
   Struct(w.phi) = Struct(w.l.phi) + Struct(w.r.phi);
   if (w.m == NULL)
   PhaseOne(w.s);
}
PhaseTwo(w) {
    if (w == NULL) return;
  A: Struct(w.psi) = Struct(w.m.psi)
     + Struct(w.m.p.l.phi) + Struct(w.m.p.r.phi)
     + Struct(w.m.n.l.phi) + Struct(w.m.n.r.phi);
  B: if (w.l == NULL) Struct(w.theta) = Struct(w.psi)
                    + Struct(w.p.gamma) + Struct(w.n.gamma);
  C: PhaseTwo(w.l);
  D: PhaseTwo(w.r);
}
```

Figure 7. Algorithm for fast multipole method

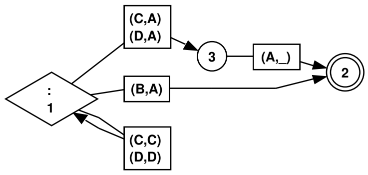


Figure 8. FSA for dependency of the second phase of fluid simulation algorithm

```
PhaseOne(w) {
   A: Struct(w.phi) = Struct(w.l.phi) + Struct(w.r.phi);
      if (w.m == NULL) return;
   B: Struct(w.s.phi) = Struct(w.s.l.phi) + Struct(w.s.r.phi);
      if (w.s.m == NULL) return;
   C: Struct(w.s.s.phi) = Struct(w.s.sl.phi) + Struct(w.s.s.r.phi);
      if (w.s.s.m == NULL) return;
   D: Struct(w.s.s.s.phi) = Struct(w.s.s.s.l.phi) + Struct(w.s.s.s.r.phi);
      if (w.s.s.s.m == NULL) return;
      PhaseOne(w.s.s.s.s.s);
}
```

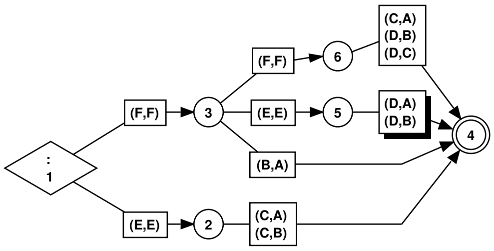


Figure 9. PhaseOne of the algorithm, unrolled and the FSA for dependencies within one function call of this function.

The nodes carry four items of data *psi*, *phi*, *theta* and *gamma*, all of which are multipole expansions of the energy potential due to various sets of vortices. The code shown in Fig. 7 has been simplified in two important ways:

– The summations represent more complex manipulations of these expansions.
– The code is a 'one-dimensional' version of the algorithm. In the two dimensional version there are four subtrees at each node, and the nodes in each level are linked into a grid, rather than a list.

Our method can handle all the complexities of the two dimensional version. We present the simplified version because it allows the essentials of the algorithm to be highlighted.

If we analyse the function `PhaseTwo`, we get the FSA in Fig. 8, for the possible sources of statements. This shows that the *reads* that occur in statement B are produced by the *write* in statement A. Also the *read* in statement A uses the value produced in the previous function call.

We need a different approach to producing information for the function 'PhaseOne' since it recurses in a non-generator direction. We have 'unrolled' three of the function calls into the body of the loop, and have looked for dependencies within one call, rather than between all calls of the function. The resulting dependancy FSA is given in Fig. 9. The main observation is that there are only dependencies between calls that access the first three levels of the tree. In all the other levels the four statements are independant and can therefore be executed simultaneously.

9 Conclusion

This paper has described a framework based on an underlying Automatic Group representation for analysing the dependencies in complex, pointer-based data structures which are less regular. The approach assumes that the programmer supplies explicit structural information. We have given a flavour as to how 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, and to allow recursion in non-generator directions.

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.

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.

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