Safe Approximation of Data Dependencies in Pointer-Based Structures

D.K. Arvind and T.A. Lewis

Institute for Computing Systems Architecture,
Division of Informatics, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
dka|tl@dcs.ed.ac.uk

Abstract. This paper describes a new approach to the analysis of dependencies in complex, pointer-based data structures. Structural information is provided by the programmer in the form of two-variable finite state automata (2FSA). Our method extracts data dependencies. For restricted forms of recursion, the data dependencies can be exact; however in general, we produce approximate, yet safe (*i.e.* overestimates dependencies) information. The analysis method has been automated and results are presented in this paper.

1 Introduction

We present a novel approach to the analysis of dependencies in complex, pointer-based data structures which builds on our earlier work [AL98]. The input to our analysis is structural information of pointer-based data structures, and algorithms that work over those structures. We consider algorithms that update the data in the nodes of the structure, although we disallow structural changes that would result from pointer assignment. The structural specification is given by identifying a tree backbone and additional pointers that link nodes of the tree. These links are described precisely using two-variable finite state automata (2FSA).

We can then produce dependency information for the program. This details for each runtime statement the set of statements that either read or write into the same node of the structure. Some of this information may be approximate, but we can check that it is conservative in that the correct set of dependencies will be a subset of the information we produce. For even quite small sections of program the output may be dauntingly complex, but we explore techniques for reducing it to a tractable size and extracting useful information.

The paper is organised as follows: we outline the notion of a 2FSA description in Section 2, and describe the restricted language that we use in Section 2.1. In Section 3.1 we look at an example of a recursive rectangular mesh, and follow through with the description and analysis of a simple piece of program. We deal with a more complex example in Section 4. We describe related works in Section 5, with conclusions and plans for future work in Section 6.

2 Structure Descriptions using 2FSA

We first observe how dynamic data structures are handled in a language such as C, and then relate this to our approach. Consider the following example of a tree data structure:

```
struct Tree {
    int data;
    Tree * d1;
    Tree * d2;
    Tree * d3;
    Tree * d4;
    Tree * r;
};
```

The items `data`, `d1`, `d2`, `d3`, `d4` and `r` are the *fields* of the structure, and may contain items of data (such as `data`) or pointers to other parts of the structure. We assume here that `d1`, `d2`, `d3`, `d4` point to four disjoint subtrees, and the `r` pointer links nodes together across the structure.

We next explain how this structure is represented. We have a fixed list of symbols, the *alphabet* A, that corresponds to the fields in the structure. We define a subset, $G \subseteq A$, of *generators*. These pointers form a tree backbone for this structure, with each node in the structure being identified by a unique string of symbols, called the *pathname* (a member of the set G^*), which is the path from the root of the tree to that particular node. Therefore `d1`, `d2`, `d3` and `d4` are the generators, in the example.

Our description of the structure also contains a set of relations, $\rho_i \subseteq G^* \times G^*$, one for each non-generator or *link* field i. This relation links nodes that are joined by a particular pointer. A node may be joined to more than one target node via a particular link. This allows approximate information to be represented. It is useful to consider each relation as a function from pathnames to the power set of pathnames: $F_i : G^* \to \mathcal{P}(G^*)$; each pathname maps to a set of pathnames that it may link to. In our example `r` is such a link. A *word* is a string of fields; we append words to pathnames to produce new ones.

We represent each relation as a *two-variable finite state automaton* (2FSA). These are also known as *Left Synchronous Transducers* (see for example [Coh99]). We recall that a (deterministic) Finite State Automaton (FSA) reads a string of symbols, one at a time, and moves from one state to another. The automaton consists of a finite set of states S, and a transition function $F : S \times A \to S$, which gives the next state for each of the possible input symbols in A. The string is accepted, if it ends in one of the *accept* states when the string is exhausted.

A *two-variable* FSA attempts to accept a pair of strings and inspects a symbol from each of them, at each transition. It can be thought of as a one-variable FSA, but with the set of symbols extended to $A \times A$. There is one subtlety, in that we may wish to accept strings of unequal lengths, in which case the shorter one is padded with the additional '$-$' symbol. This results in the actual set of symbols to be: $((A \cup \{-\}) \times (A \cup \{-\})) \backslash (-,-)$, since the double padding symbol is never needed.

We can utilise non-deterministic versions of these automata. As already described, deterministic 2FSAs allow only one transition from a fixed state with a fixed symbol; non-deterministic ones relax this condition by allowing many. Most manipulations work with deterministic 2FSAs, but on occasions it may be simpler to define a non-deterministic 2FSA with a particular property, and determinise it afterwards.

We also use 2FSAs to hold and manipulate the dependency information that we gather. There are other useful manipulations of these 2FSAs which we use in our analysis.

- *Logical Operations.* We can perform basic logical operations such as AND ($\wedge$), OR ($\vee$), and NOT on these 2FSAs.
- The *Exists and Forall* automata. The one variable FSA $\exists(F)$, accepts x if there exists a y such that $(x, y) \in F$. Related to this is the 2FSA $\forall(R)$ built from the one variable FSA R, that accepts (x, y) for all y, if R accepts x.
- *Composition.* Given 2FSAs for individual fields, we wish to combine the multiplier 2FSAs into one multiplier for each word of fields 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 2FSAs for `a` and `b`. This *composition* 2FSA can be computed: given two FSAs, $\mathcal{R}$ and $\mathcal{S}$, their composition is denoted by $\mathcal{R}.\mathcal{S}$. This is defined as : $(x, y) \in \mathcal{R}.\mathcal{S}$, if there exists a z, such that $(x, z) \in \mathcal{R}$ and $(z, y) \in \mathcal{S}$. See [ECH+92] for the details of its construction.
- The *inverse* of 2FSA $\mathcal{F}$, denoted $\mathcal{F}^{-1}$, which is built by swapping the pair of letters in each transition of $\mathcal{F}$.
- The *closure* of 2FSA $\mathcal{F}$, denoted $\mathcal{F}*$, which we discuss later in Section 3.4.

With the exception of the *closure* operation, these manipulations are exact for any 2FSAs.

2.1 Program Model

We work with a fairly restricted programming language with C-like syntax that manipulates these structures. The program operates on one global data structure. Data elements are accessed via pathnames, which are used in the same manner as conventional pointers. The program consists of a number of possibly mutually recursive functions. These functions take any number of pathname parameters, 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 '`Func(w->g)`' where `g` is any field name.

The basic statements of the program are *reads* and *writes* to parts of the structure. A typical read/write statement is '`w->a = w->b`' where `w` is a variable and `a` and `b` are words of directions, and denotes the copying of an item of data from `w->b` to `w->a` within the structure. Note that we do not allow structures to be changed dynamically by pointer assignment. This makes our analysis only valid for sections of algorithms where the data in an existing structure is being updated, without structural changes taking place.

3 The Analysis

3.1 A Rectangular Mesh Structure

We work through the example of a rectangular mesh structure, as shown in Figure 1 to demonstrate how a 2FSA description is built up from a specification. These mesh structures are often used in finite-element analysis; for example, to analyse fluid flow within an area. The area is recursively divided into rectangles, and each rectangle is possibly split into four sub-rectangles. This allows for a greater resolution in some parts of the mesh. We can imagine each rectangle being represented as a node in a tree structure. Such a variable resolution mesh results in an unbalanced tree, as shown in Fig. 1. Each node may be split further into four subnodes. Each rectangle in the mesh has four adjacent rectangles that meet along an edge in that level of the tree. For example, the *rectangle* 4 on the bottom right has *rectangle* 3 to the left of it and *rectangle* 2 above. This is also true for the smallest *rectangle* 4, except that it has a *rectangle* 3 to its right. We call these four directions l, r, d and u. The tree backbone of the structure has four generators $d1, d2, d3$ and $d4$ and linking pointers l, r, d and u. We assume that these links join nodes at the same level of the tree, where these are available, or to the parent of that node if the mesh is at a lower resolution at that point. So moving in direction u from node $d3$ takes us to node $d1$, but going up from node $d3.d1$ takes us to node $d1$, since node $d1.d3$ does not exist.

We next distill this information into a set of equations that hold this linkage information. We will then convert these equations into 2FSA descriptions of the structure. Although they will be needed during analysis, the generator descriptions do not have to be described by the programmer as they are simple enough to be generated automatically.

Let us now consider the r direction. Going right from a $d1$ node takes us to the sibling node $d2$. Similarly,we reach the $d4$ node from every $d3$. To go right from any $d2$ node, we first go up to the parent, then to the right of that parent, and down to the $d1$ child. If that child node does not exist then we link to the parent. For the $d4$ node, we go to the $d3$ child to the right of the parent. This information can be represented for the r direction by the following set of equations (where x is any pathname) :

$$r(x.d1) = x.d2 \tag{1}$$
$$r(x.d2) = r(x).d1|r(x) \tag{2}$$
$$r(x.d3) = x.d4 \tag{3}$$
$$r(x.d4) = r(x).d3|r(x) \tag{4}$$

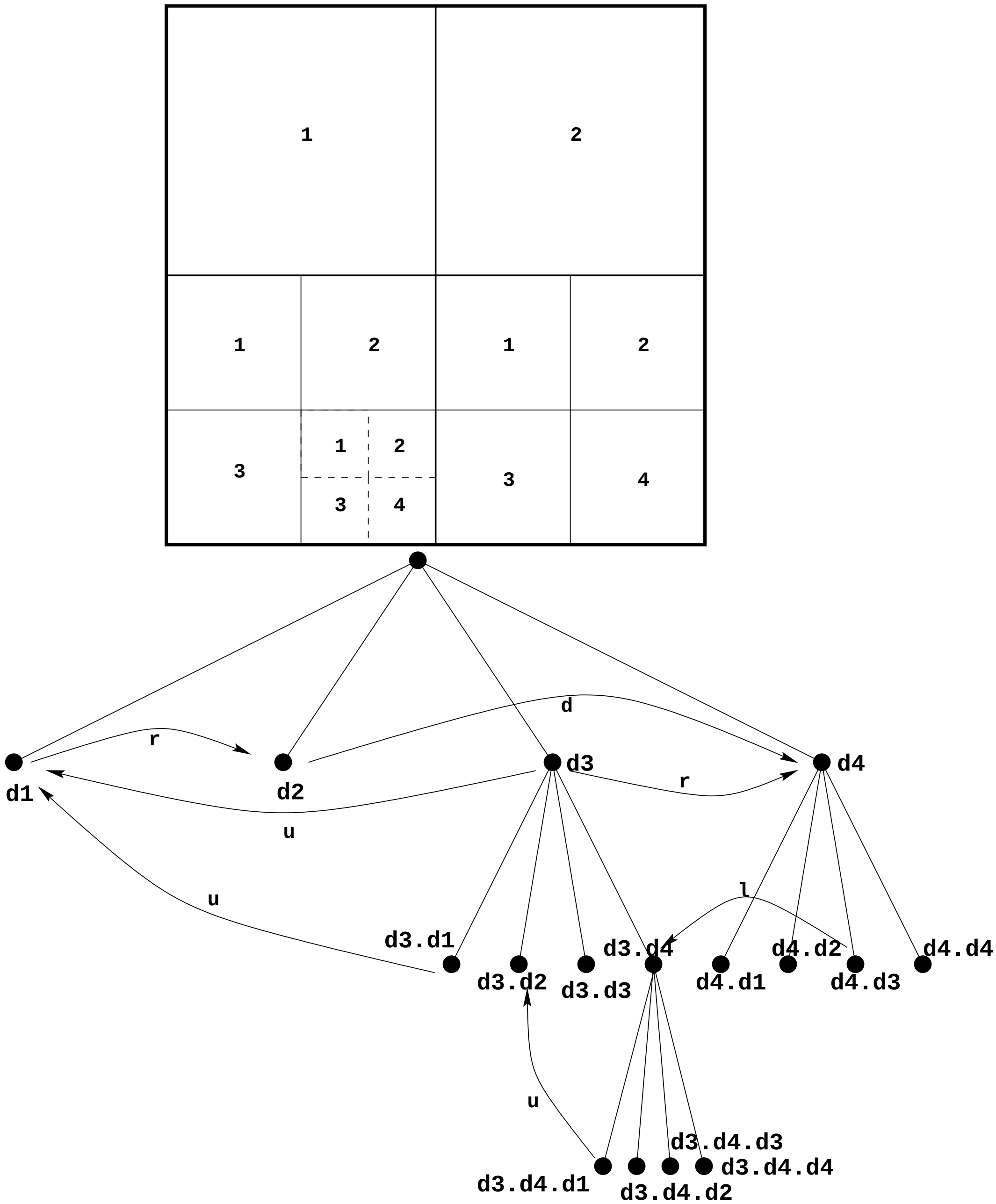

Fig. 1. Above: the variable resolution rectangular mesh. Below: The mesh is represented as a tree structure with the l, r, d, u links to adjacent rectangles. (Not all the links are shown, for the sake of brevity.)

$$
\begin{aligned}
d(x.d1) &= x.d3 \\
d(x.d2) &= x.d4 \\
d(x.d3) &= d(x).d1|d(x) \\
d(x.d4) &= d(x).d2|d(x)
\end{aligned}
$$

$$
\begin{aligned}
u(x.d1) &= u(x).d3|u(x) \\
u(x.d2) &= u(x).d4|u(x) \\
u(x.d3) &= x.d1 \\
u(x.d4) &= x.d2
\end{aligned}
$$

$$
\begin{aligned}
l(x.d1) &= l(x).d2|l(x) \\
l(x.d2) &= x.d1 \\
l(x.d3) &= l(x).d4|l(x) \\
l(x.d4) &= x.d3
\end{aligned}
$$

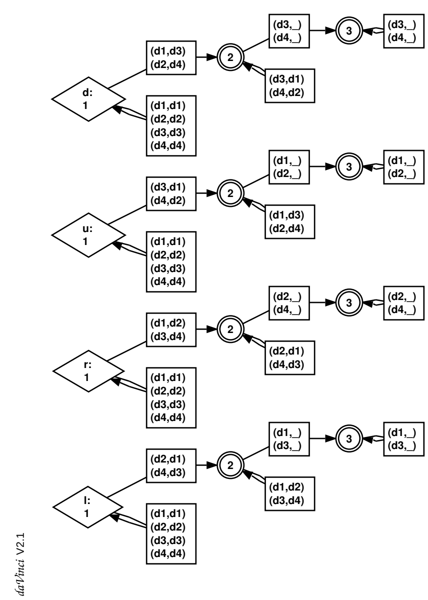

Fig. 2. Equations and 2FSA descriptions for the link directions in the mesh structure.

These equations are next converted into a 2FSA for the r direction. Note that one pathname is converted into another by working from the end of the pathname back to the beginning. The rule '$r(x.d4) = r(x).d3|r(x)$' does the following: if $d4$ is the last symbol in the string, replace it with a $d1$ or an ϵ, then apply the r direction to the remainder of the string. This is similarly true for the '$r(x.d2) = r(x).d1|r(x)$' rule. In the rule '$r(x.d1) = x.d2$' the last $d1$ symbol is replaced by a $d2$, and then outputs the same string of symbols as its input. Viewed in this way, we can create a 2FSA that accepts the paths, but in reverse order.

The correct 2FSA is produced by reversing the transitions in the automata and exchanging initial and accept states. Note that the presence of transitions labelled with ϵ may make this reversal impossible. We can, however, use this process to produce 2FSA descriptions for the other link directions l, d and u. Figure 2 illustrates the remaining set of equations and 2FSA descriptions.

3.2 The Program

```
main (Tree *root) {
  if (root != NULL) {
A:    traverse(root->d2);
B:    traverse(root->d4);
  }
}
traverse(Tree *t) {
  if (t!=NULL) {
C:    sweepl(t);
D:    traverse (t->d1);
E:    traverse (t->d2);
F:    traverse (t->d3);
G:    traverse (t->d4);
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
H:    x->l->data = x->data + x->r->data;
I:    sweepl(x->l);
  }
}
```

Fig. 3. Sample recursive functions. The $A, B, C, D, E, F, G, H, I$ are statement labels and are not part of the original code.

The recursive functions in Fig. 3 operate over the rectangular mesh structure illustrated in Fig. 1. The function `main` branches down the right hand side of the tree calling `traverse` to traverse the whole sub-trees. The function `sweepl` then propagates `data` values out along the `l` field.

The descriptions for single fields can be used to build composite fields. For instance, the update in statement H requires the composite fields: $r \rightarrow data$ and $l \rightarrow data$, both built up from the basic descriptions by composing the 2FSAs. Each runtime statement is uniquely identified by a *control word*, defined by labelling each source code line with a symbol, and forming a word by appending the symbol for each recursive function that has been called. Each time the `traverse` function is called recursively from statement D, we append a D symbol to the control word. Source code line H therefore expands to the set of runtime control words $(A|B).(D|E|F|G)^*.C.I^*.H$.

A pathname parameter to a recursive function is called an *induction parameter*. Each time we call the function recursively from statement D, we append a $d1$ field to the induction parameter t and for statement E we similarly append a $d2$ field. The same is true for statements F and G. This information can be captured in a 2FSA that converts a given control word into the value of the parameter t, for that function call. The resulting 2FSA is shown in Fig. 4.

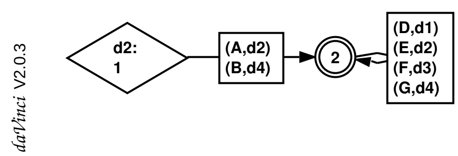

Fig. 4. A 2FSA that maps control words to values of t.

3.3 Building a general induction parameter 2FSA

In general for each induction parameter V_i in each function, we need to create a 2FSA, F_{V_i}, that maps all control words to the pathname value of that parameter at that point in the execution of the program. Fig. 5 outlines the construction of such a 2FSA.

If a function is called in the form `F(...,Vk,...)` where `Vk` is an induction parameter, then the substitution 2FSA will contain transition symbols of the form (A, ϵ). We aim to remove these by repeated composition with a 2FSA that will remove one ϵ at a time from the output. Provided that the transition is not in a recursive loop, *i.e.*, we can bound the number of times that the function can be called in any run of the program, applying this repeatedly will remove all such transitions. The construction of this epsilon-removal 2FSA is outlined in Fig. 6.

Since the program in Fig. 3 has a function `sweepl` that recurses by updating its induction parameter by a non-generator direction `l`, we have to approximate the dependency information for this function. This is because, in general, the required induction parameter relation will not necessarily be a 2FSA; hence we approximate with one.

- Create a non-deterministic 2FSA with a state for each induction variable, plus an additional initial state. State $i+1$ (corresponding to variable i) is the only accepting state.
- For each pathname parameter j of the `main` function, add an epsilon transition from state 1 to state $j+1$.
- For each induction variable k, we seek all call statements of the form `A: F(...,Vk->g,...)`. If `Vk->g` is passed as variable m in function F, then we add transition from state $k+1$ to state $m+1$, with symbol (A, g). Here we assume that g is a generator. If g is empty, we use an epsilon symbol in its place, then attempt to remove it later. If g is a non-generator we still use the symbol (A, g), but we need to apply the closure approximation described later.
- Determinise the nondeterministic 2FSA.

Fig. 5. The Substitution 2FSA for the variable V_i

- Create a 2FSA with one state for each generator (and link), plus another three states, an initial state, an accepting final state and an epsilon state.
- Add transitions from initial state. (d, d) symbols where d is a field (generator or link), make the transition back to the initial state. (ϵ, d) symbols move to the state for that field $(d+1)$. $(\epsilon, -)$ symbols move to the final state.
- Add transitions for each field state, (field i corresponds to state $i+1$). Symbol (i, i) leave it at state $i+1$. (i, d) moves it to state $d+1$. $(i, -)$ moves to the final state.
- Add transitions for the epsilon state. (ϵ, d) moves to $d+1$. (ϵ, ϵ) leaves it at the epsilon state. $(\epsilon, -)$ moves it to the final state.
- There are no transitions from the final state.

Fig. 6. The remove ϵ 2FSA

3.4 Formation of Approximate Closures of 2FSA

We next consider the problem of producing a 2FSA that safely approximates access information of functions that recurse in the non-generator fields. This sort of approximation will give coarse-grained access information for a whole nest of recursive calls.

We first take the simplest example of a function that calls itself recursively, which appends a non-generator field to the induction parameter at each call. We then show how this can be used to produce approximate information for any nest of recursive calls. Consider the following program fragment:

```
sweepl(Tree *x) {
  if (x!=NULL) {
H:     x->l->data = x->data + x->r->data;
I:     sweepl(x->l);
  }
}
```

Recursion in one field can be approximated by a number of steps in that field with a 2FSA that approximates any number of recursions. In the second iteration of the function, the value of x will be `l` appended to its initial value. Iteration k can be written as $\mathtt{n}^{k-1}$, and can be readily computed for any finite value of k, although the complexity of the 2FSA becomes unmanageable for a large k. We wish to approximate all possible combinations in one 2FSA.

Definition 1. *The relation* $[=]$ *is the equality relation,* $(x, y) \in [=] \iff x = y$.

Definition 2. *The* closure *of the field* p*, written* p^**, is defined as*

$$p^* = \bigvee_{k=0}^{\infty} p^k$$

where p^0 *is defined as* $[=]$

In the example of the `l` field, the closure can be represented as a 2FSA, and the approximation is therefore exact. In general, however, this is not always the case, but we aim to approximate it safely as one. A *safe* approximation S to a relation R, implies that if $(x, y) \in R$, then $(x, y) \in S$.

We have developed a test to demonstrate that a given 2FSA R is a safe approximation to a particular closure: we use a heuristic to produce R, and then check that it is safe.

Theorem 1. *If* R *and* p *are relations such that* $R \supseteq R.p \vee [=]$*, then* R *is a safe approximation to* p^*.

Proof. Firstly, $R \supseteq R.p \vee [=] \Rightarrow R \supseteq R.p^{k+1} \bigvee_{i=1}^{k} p^i$, for any k (Proof: induction on k). If $(x, y) \in p^*$, then $(x, y) \in p^r$ for some integer r. Applying above with $k = r$ implies that $(x, y) \in R$.

Therefore, given a 2FSA, R, that we suspect might be a safe approximation, we can test for this by checking if $R \supseteq R.p \vee [=]$. This is done by forming $R \wedge (R.p \vee [=])$, and testing equality with $R.p \vee [=]$. It is worth noting that safety does not always imply that the approximation is useful; a relation that accepts every pair (x, y) is safe but will probably be a poor approximation.

To prove an approximation exact we need equality in the above comparison and an additional property; that for every node x travelling in the direction of p we will always reach the edge of the structure in a finite number of moves.

Theorem 2. *If R and p are relations such that $R = R.p \vee [=]$, and for each x there exists a k_x such that for all y, $(x, y) \notin p^{k_x}$, then $R = p^*$, and the approximation is exact.*

Proof. If $(x, y) \in R$, then $(x, y) \in R.p^{k_x} \bigvee_{i=1}^{k_y - 1} p^i$. Since (x, y) cannot be in $R.p^{k_x}$, it must be in p^i for some $i < k_x$. So $(x, y) \in p^*$.

Unfortunately we cannot always use this theorem to ascertain that an approximation is exact, as we know of no method to verify that the required property holds for an arbitrary relation. However, this property will often hold for a link field (it does for most of the examples considered here), so an approximation can be verified for exactness.

The method that we use to generate these closures creates a (potentially infinite) automaton. In practice we use a number of techniques to create a small subset of this state space:

– Identifying fail states.
– 'Folding' a set of states into one approximating state. The *failure transitions* for a state is the set of symbols that leads to the failure state from that state. If two states have the same set of failure transitions, we assume they are the same state.
– Changing transitions out of the subset so that they map to states in the subset.

We have tested the closures of around 20 2FSAs describing various structures. All have produced safe approximations, and in many cases exact ones.

3.5 Generalising to any loop

We can use these closure approximations in any section of the function call graph where we have a recursive loop containing a link field in addition to other fields. Starting from the 2FSA for the value of an induction variable v, we build up an expression for the possible values of v using *Arden's rules* for converting an automaton to a regular expression. This allows us to solve a system of i equations for regular expressions E_i . In particular, we can find the solution of a recursive equation $E_1 = E_1.E_2|E_3$ as

$$E_1 = E_3.(E_2)^*$$

The system of equations is solved by repeated substitution and elimination of recursive equations using the above formula. We thus obtain a regular expression for the values of v within the loop. We can then compute an approximation for this operation using OR, *composition* and *closure* manipulations of 2FSAs.

3.6 Definition and Use 2FSAs

For each *read* statement, X, that accesses `p->w`, we can append to $\mathcal{F}_p$ the statement symbol, X, and the read word, w, for that statement. Thus if $\mathcal{F}_p$ accepts (C, y), then this new 2FSA will accept $(C.X, y \to w)$, and is formed from two compositions. The conjunction of this set of 2FSAs produces another 2FSA, $\mathcal{F}_r$, 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 2FSA that maps from control words to nodes which are written, denoted by $\mathcal{F}_w$. These write and read 2FSAs, are derived automatically by the dependency analysis.

1. The *definition* 2FSA, that accepts the pair (control word, path-name) if the control word writes (or defines) that node of the structure.
2. The *use* 2FSA for describing nodes that are read by a particular statement.

We can now describe all the nodes which are read from and written to, by any statement. By combining the read and write 2FSAs we create a 2FSA that link statements when one writes a value that the other reads, *i.e* the statements *conflict*. The conflict 2FSA, $\mathcal{F}_{conf}$, is given by $\mathcal{F}_r.(\mathcal{F}_w)^{-1}$. We can now describe all the nodes which have been read from, and written to, by any statement. By combining these read and write 2FSAs, we can now create 2FSAs that links statements for each of the possible types of dependency (read after write, write after read and write after write). The conjunction of these 2FSAs forms the *conflict* 2FSA:

$$\mathcal{F}_{conf} = \mathcal{F}_r.(\mathcal{F}_w)^{-1} \cup \mathcal{F}_w.(\mathcal{F}_r)^{-1} \cup \mathcal{F}_w.(\mathcal{F}_w)^{-1}$$

We may be interested in producing *dataflow* information, *i.e.* associating with each read statement, the write statement that produced that value. We define the *causal* 2FSA, $\mathcal{F}_{causal}$ which accepts control words X, Y, only if Y occurs before X in the sequential running of the code. The 2FSA in Fig.7 is the *conflict* 2FSA for the example program, which has also been ANDed with $\mathcal{F}_{causal}$, to remove any false sources that occur after the statement.

3.7 Partitioning the computation

We now consider what useful information can be gleaned from the conflict graph with regard to the parallel execution of the program. Our approach is as follows:

- Define execution threads by splitting the set of control words into separate partitions. Each partition of statements will execute as a separate thread. At present these are regular expressions supplied by the programmer.
- Use the *conflict* 2FSA to compute the dependencies between threads.

We apply this approach to the running example. We can take as our two partitions of control words:

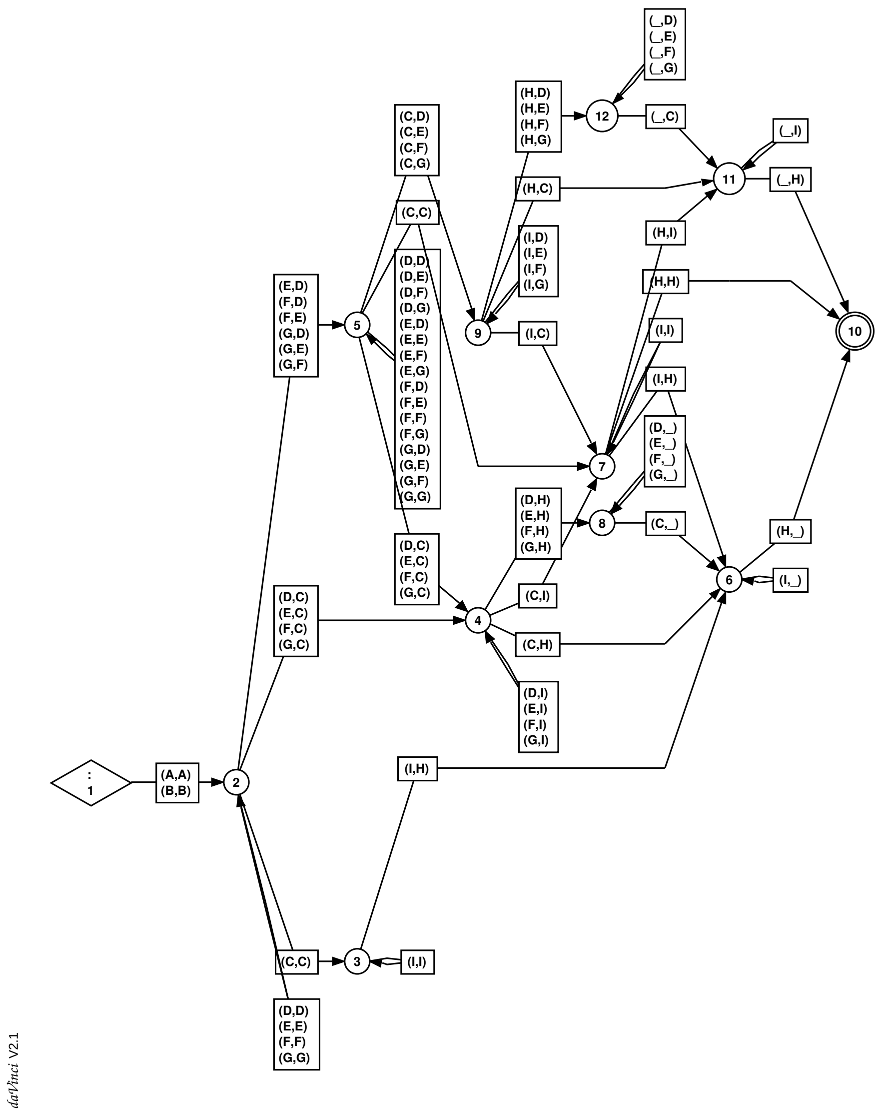

Fig. 7. The conflict 2FSA for the example code.

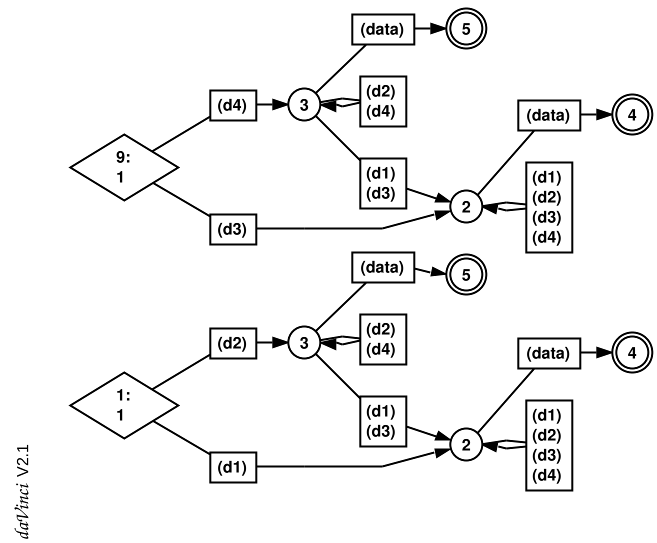

Fig. 8. The values read by and written to by each of the threads.

1. $(A).(D|E|F|G)^*.C.I^*.H$
2. $(B).(D|E|F|G)^*.C.I^*.H$

We can then compute the total set of values read by and written to each statement in each thread. This is shown in Fig.8. Since there is no overlap the two threads are completely independent and can therefore be spawned safely.

4 A Second Example

We next describe how this approach can be applied to a larger program, as shown in Fig. 9. Consider the function calls in statements `A` and `B` being spawned as separate threads. The resulting *conflict* 2FSA for this example has 134 nodes, too large to interpret by manual inspection. We describe an approach for extracting precise information that can be used to aid the parallelisation process.

A particular statement, X say, in the second thread may conflict with many statements in the first thread. Delaying the execution of X until all the conflicting statements have executed ensures that the computation is carried out in the correct sequence. We can therefore compute information that links a statement to the ones whose execution it must wait for.

```
main (Tree *node) {
  if (node!=NULL) {
A:    update(node);
B:    main (node->d1);
C:    main (node->d2);
  }
}
update(Tree *w) {
  if (w!=NULL) {
D:    sweepl(w);
E:    propagate(w);
  }
}
propagate(Tree *p) {
  if (p!=NULL) {
    p->data = p->l->data + p->r->data
F:      + p->u->data + p->d->data;
G:    propagate(p->d1);
H:    propagate(p->d2);
I:    propagate(p->d3);
J:    propagate(p->d4);
  }
}
sweepl(Tree *x) {
  if (x!=NULL) {
K:    x->l->data = x->data;
L:    sweepl(x->l);
  }
}
```

Fig. 9. The second example program

We map X to Y, such that if X conflicts with Z, then Z must be executed earlier than Y. We produce the *wait-for* 2FSA by manipulating the *conflict* and *casual* 2FSAs:

$$\mathcal{F}_{\textit{wait-for}} = \mathcal{F}_{\textit{conf}} \wedge \mathrm{Not}(\mathcal{F}_{\textit{conf}} . \mathcal{F}_{\textit{causal}})$$

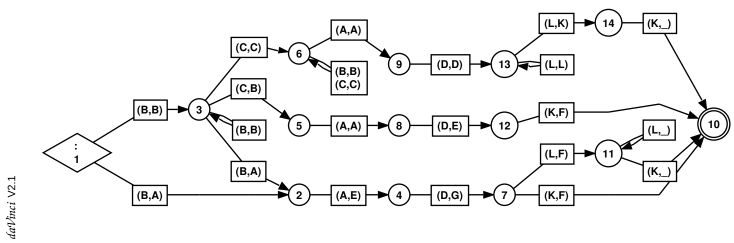

Fig. 10. The 'waits-for' information

We use this approach to trim the conflict information and produce the one in Fig. 10. We are interested in clashes *between* the two threads, so only the portion that begins with the (B, A) symbol is relevant. We extract the information that the statements given by $B.A.D.L.(L)^*.K$ must wait for $A.E.G.F$ to execute. This implies that to ensure the threads execute correctly we would need to insert code to block these statements from executing until the statement $A.E.G.F$ has done so.

4.1 Implementation

We have implemented the techniques described in this paper in C++ using the 2FSA manipulation routines contained in the 'kbmag' library [Hol]. In addition we use the graph visualisation application 'daVinci' [FW] for viewing and printing our 2FSA diagrams. We ran our analysis codes on an 400MHz Pentium II Gnu/Linux machine. The smaller example took 283 seconds, and the larger one 497 seconds. Much of this time is spent in the closure computation routines, and in the case of the larger example the production of the waits-for information from the large conflict graph.

5 Related Work

The ASAP approach [HHN94b] uses three different types of axiom to store information about the linkages in the structure.

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

where p, q are any path names and $RE1, RE2$ are regular expressions over the alphabet of fields . These axioms are then used to prove whether two pointers can be aliased or not. In comparison, our properties are of the form $p.RE1.n = p.RE2$ where $RE1$ and $RE2$ are regular expressions in the generator directions and n is a link. They are slightly more powerful in that the 2FSA description allows $RE1$ and $RE2$ to be dependent. Provided that the regular expressions do not include link directions, we can express ASAP axioms as 2FSAs and combine them into our analysis. In fact, even if the expression has link directions inside a Kleene star component, we can use the closure approximation method to approximate this information and use it in our analysis. ASAP descriptions are an improvement on an earlier method, ADDS, which used different dimensions and linkages between them to describe structures. Thus ASAP is more suited to structures with a multi-dimensional array-like backbone.

Comparison with the ADDS/ASAP description for a binary tree with linked leaves (see [HHN94a]), shows that our specification is much more complicated. To justify this, we demonstrate that the 2FSA description can resolve dependencies more accurately. Consider the ADDS description of the binary tree,

```
type Bintree [down][leaves] {
  Bintree *left,*right is uniquely forward along down;
  Bintree *next        is uniquely forward along leaves;
}
```

This description names two dimensions of the structure `down` and `leaves`. The `left` and `right` directions form a binary tree in the `down` direction, the `next` pointers create a linked list in the `leaves` dimension. The two dimensions are not described as disjoint, since the same node *can* be reached via different routes along the two dimensions. Now consider the following code fragment, where two statements write to some subnodes of a pointer 'p'.

```
p->l->next->next = ...
p->r = ...
```

Dependency analysis of these statements will want to discover if the two pointers on the left sides can ever point to the same node. Since the sub-directions contain a mixture of directions from each dimension `[down]` and `[leaves]`, ADDS analysis must assume conservatively that there may be a dependence. The 2FSA description however can produce a 2FSA that accepts all pathnames p for which these two pointers are aliased. This FSA is empty, and thus these writes are always independent.

The 'Graph Types' [KS93] are not comparable to ours since they allow descriptions to query the type of nodes and allow certain runtime information, such as whether a node is a leaf to be encoded. If we drop these properties from their descriptions then we can describe many of their data structures using 2FSAs. This conversion can probably not be done automatically. Their work does not consider dependency analysis; it is mainly concerned with program correctness and verification. [SJJK97] extends this to verification of list and tree programs, and hints that this analysis could be extended to graph types.

Shape types [FM97] uses grammars to describe structures, which is more powerful than our 2FSA-based approach. Structure linkages 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*, rewrite rules that update the multi-sets. However, many of the questions we want to ask in our dependence analysis may be undecidable in this framework. If we drop their convention that the pointers from a leaf node point back to itself, we can describe the dependence properties of their example structures (e.g. skip lists of level two, binary trees with linked leaves and left-child, right-sibling trees) using our formalism.

In [CC98], the authors use formal language methods (pushdown automata and rational transducers) to store dependency information. The pushdown automata are used for array structures, the rational transducers for trees; more complex structures are not considered. Rational transducers are more general than 2FSAs, since they allow ϵ transitions that are not at the end of paths. This extension causes problems when operations such as intersections are considered. Indeed only ones which are equivalent to 2FSAs can be fully utilised. Handling multidimensional arrays in their formalism will require a more sophisticated language. Our method is not intended for arrays, although they can be described using 2FSAs. However, the analysis will be poor because we have to approximate whichever direction the code recurses through them.

[BRS99] allows programs to be annotated with reachability expressions, which describe properties about pointer-based data structures. Their approach is generally more powerful than ours, but they cannot, for example, express properties of the form $x.RE1 = y.RE2$. Our 2FSAs can represent these sort of dependencies. Their approach is good for describing structures during phases where their pointers are manipulated dynamically, an area our method handles poorly at best. Their logic is decidable, but they are restricted in that they do not have a practical decision algorithm.

[Fea98] looks at dependencies in a class of programs using *rational transducers* as the framework. These are similar, but more general than the 2FSAs we use here. The generality implies that certain manipulations produce problems that are undecidable, and a semi-algorithm is proposed as a partial solution. Also the only data structures considered are trees, although extension to doubly linked lists and trees with upward pointers is hinted at. However, even programs that operate over relatively simple structures like trees can have complex dependency patterns. Our method can be used to produce the same information from these programs. This indicates a wider applicability of our approach from the one involving structures with complex linkages.

In [Deu94] the author aims to extract aliasing information directly from the program code. A system of symbolic alias pairs *SAPs* is used to store this information. Our 2FSAs form a similar role, but we require that they are provided separately by the programmer. For some 2FSAs the aliasing information can be expressed as a SAP. For example, the information for the n direction in the binary tree can be represented in (a slightly simplified version of) SAP notation as:-

$$(< X(\to l)(\to r)^{k_1} \to n, X(\to r)(\to l)^{k_2} >, k_1 = k_2)$$

The angle brackets hold two expressions that aliased for all values of the parameters k_1, k_2 that satisfy the condition $k_1 = k_2$. SAPs can handle a set of paths that are not regular, so they are capable of storing information that a finite state machine cannot. These expressions are not, however, strictly more powerful than 2FSAs. For example, the alias information held in the 2FSA descriptions for the rectangular mesh cannot be as accurately represented in SAP form.

In [RS98] the authors describe a method for transforming a program into an equivalent one that keeps track of its own dataflow information. Thus each memory location stores the statement that last wrote into it. This immediately allows methods that compute alias information to be used to track dataflow dependencies. The method stores the statement as a line number in the source code, rather than as a unique run time identifier, such as the control words we use. This means that dependencies between different procedures, or between different calls of the same procedure, will not be computed accurately. Although we describe how dataflow information can be derived by our approach, we can still produce alias information about pointer accesses. So the techniques of [RS98] could still be applied to produce dependency information.

6 Conclusions and Future Work

This paper builds on our earlier work in which the analysis was restricted to programs which recursed in only the generator directions. We have now extended the analysis, to non-generator recursion. The approximation is safe in the sense that no dependencies will be missed. The next step would be to extend the analysis to code that uses pointer assignment to dynamically update the structure.

As it stands the computation of the waits-for dependency information involves a large intermediate 2FSA, which will make this approach intractable for longer programs. We are currently working on alternative methods for simplifying the conflict 2FSA that would avoid this.

We are also pursuing the idea of attaching probabilities to the links in the structure description. A pointer could link a number of different nodes together, with each pair being associated with a probability. We could then optimise the parallelisation for the more likely pointer configurations while still remaining correct for all possible ones.

References

[AL98] D. K. Arvind and Tim Lewis. Dependency analysis of recursive data structures using automatic groups. In Siddhartha Chatterjee et al, editor, *Languages and Compilers for Parallel Computing, 11th International Workshop LCPC'98*, Lecture Notes in Computer Science, Chapel Hill, North Carolina, USA, August 1998. Springer-Verlag.

[BRS99] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In S. D. Swierstra, editor, *ESOP '99: European Symposium on Programming, Lecture Notes in Computer Science*, volume 1576, pages 2–19, March 1999.

[CC98] A. Cohen and J.-F. Collard. Applicability of algebraic transductions to data-flow analysis. In *Proc. of PACT'98*, Paris, France, October 1998.

[Coh99] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, University of Versailles, 1999.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, New York, NY, 1994. ACM Press.

[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.

[Fea98] Paul Feautrier. A parallelization framework for recursive programs. In *Proceedings of EuroPar*, volume LNCS 1470, pages 470–479, 1998.

[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.

[FW] Michael Frohlich and Mattias Werner. The davinci graph visualisation tool. http://www.informatik.uni-bremen.de/daVinci/.

[HHN94a] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 100–104, June 1994.

[HHN94b] 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.

[Hol] Derek Holt. Package for knuth-bendix in monoids, and automatic groups. http://www.maths.warwick.ac.uk/ dfh/.

[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.

[RS98] J. L. Ross and M. Sagiv. Building a bridge between pointer aliases and program dependences. *Nordic Journal of Computing*, (8):361–386, 1998.

[SJJK97] Michael I. Schwartzbach, Jakob L. Jensen, Michael E. Jorgensen, and Nils Klarlund. Automatic verification of pointer programs using monadic second order logic. In *Proceedings of the Conference on Programming Language design and implementation*. ACM, 1997.