Static Analysis of Recursive Data Structures

D. K. Arvind and T. A. Lewis

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

Abstract. The static analysis of imperative languages is made difficult by the unrestricted pointer structures which makes their analysis imprecise. A static analysis approach is proposed for a class of graph structures, where the regularity is described by the programmer via a Markov algorithm. The method is illustrated with an example.

1 Motivation

The static analysis of programs involving arrays with a view towards their automatic parallelisation has been studied in [2]. In contrast, the static analysis of pointer-based data structures is relatively more complicated. These structures can be viewed as a graph of values, with no restrictions placed on which values the pointers point to. In practice, they may often resemble trees or lists, but there is no requirement for them to conform to these structures. Consequently, any analysis cannot make such assumptions, unless the properties of the structures can be derived during the analysis of the code.

The approach adopted here is for the programmer to infuse extra structural information in the description of the data structure, which is strictly adhered to by the program. We differ from the ASAP approach [3] in that the nodes of our structure need to be given unique names at compile-time, so the description must provide information about when any pair of nodes are equal. The ASAP approach, however, allows incomplete information to be used, including identifying parts of the structure that *cannot* be aliased. Our approach is to explore more fully the analysis that can be achieved in the presence of complete knowledge, and aim to produce correspondingly more accurate analysis.

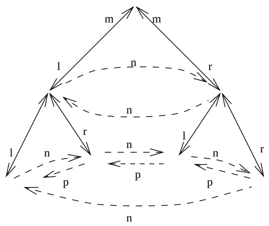
Since the aim of the analysis will be to enable the automatic parallelisation of programs written in this style, we would also want to make their parallel implementation as transparent as possible. Our approach is consistent with this philosophy. The additional information expresses properties of the data structure used in the algorithm, and are *not* annotations that describe how the program should be distributed on a parallel architecture.

2 Background

The data structures considered are graphs with the data elements stored at the nodes. Each node has a fixed set of outgoing directions D. The nodes can be indexed by the language L over the alphabet D, each word being a *path name* from a distinguished root node denoted by ϵ.

There is also an equivalence relation, $\equiv$, over the path names, which describes when two different paths actually access the same data node. We can define a normalising function, n, on such path names, so that all paths can be mapped to a unique equivalent one, *i.e.* it has the property that $x \equiv y \iff n(x) = n(y)$. The set of names v such that $u \equiv v$ is the set of *aliases* for u. We use the normalising function as the definition of the regularity in the structure. A *Markov algorithm*, [1], is proposed as a formal way of doing this.

A Markov algorithm is a system with an ordered set of string rewriting rules. Given an input string, s, the algorithm searches through the set, and when a rule is matched then s is updated by applying the rule. This procedure is repeated by returning to the first rule. The loop is terminated either when the process is *blocked* (*i.e.* no rules can be applied), or when one of a set of marked *termination* rules is applied. In our case, we accept blocking as termination and ignore the use of explicit termination rules.



$ln \to r$	$lp \to pr$	$lm \to \epsilon$
$rn \to nl$	$rp \to l$	$rm \to \epsilon$
$n \to \epsilon$	$p \to \epsilon$	$m \to \epsilon$

Fig. 1. Binary tree with associated Markov algorithm

We will take as an example a binary tree with additional pointers. Apart from the main l and r directions, we have a parent pointer, m, and the nodes on each level are joined into a doubly linked cyclic list, with n and p directions. The p directions, (not all illustrated) travel in the opposite direction to the n. The algorithm for normalisation is shown in Fig. 1. Note that we can always travel in all directions from all nodes, the parent of the root node is defined as the root node, and the n and p fields wrap around to create a cyclic list.

We restrict our program model to recursive functions with a path name as the parameter w, as shown in the example in Section 2.2. Statements in the body of the function can read and write to elements of the structure obtained by prefixing the parameter path name by a word from L, *i.e.* to access the current node we use `Struct(w)`. Additionally, we can call the function recursively with the new parameter $w.x$, for any direction x. During the analysis each recursive call in the function body is assigned a label from an alphabet $\mathcal{C}$. Also, each read and write statement is given a label from $\mathcal{I}$; we will often only consider one read and one write denoted by R and W. A *control word* in $\mathcal{C}^*.I$ will then uniquely describe a run-time instance of either a read or a write statement. We will also assume that there is a predicate on the size of the w parameter that halts the recursion, causing the function invocation to terminate. If P is a word from $\mathcal{C}^*$, then we define $\text{Current}(P)$ to be the current parameter of the function call. Also, for a control word V we can define $\text{Data}(V)$ to be the path name of the data element actually accessed by that statement, whether for a read or a write. The aim is to convert the properties present in the data structure to dataflow dependencies between control words.

There is a total ordering on all the control words obtained from the ordering on the call labels and the statement order within the function. This total order is described by extending the order on these symbols to a lexicographic order on the strings of symbols. This ordering corresponds to the actual execution order of the statements for a sequential execution of the function.

The aim of the dataflow analysis is, given a particular statement $P.R$ that reads from the structure, to locate the *source*, the write statement that produced that value, if such a statement exists. This characterises the data dependencies within the function. Define two statements U, V as *conflicting* if $\mathrm{Data}(U) \equiv \mathrm{Data}(V)$. We define a *potential source* T as a statement that conflicts with $P.R$ and occurs before it, $T < P.R$. Thus the source is the maximum of all potential sources.

2.1 Example Analysis

Since we can simply define an inverse for each direction, finding the source of a read is simplified. For a pair of read/write statements with directions r and w, we apply r to the current path of the particular statement and then apply w^{-1} to find the potential source. We can then check that it occurs before the read, and reject it otherwise. We follow through a simple example function, and check for a flow of data between statements S and T.

```
F(w) =      {
      S:    Struct(w.m) = ...
      T:    ... = Struct(w.n)
      L:    F(w.l);
      R:    F(w.r);      }
```

Instead of using the Markov description directly, we reformulate the description so that the addition of a direction n to an existing path is viewed as an operation N that maps a path in a particular format to a path in a related format. For example, the addition of direction n will locate the last l direction in a path, and flip it and the preceding directions from $l \to r$ and $r \to l$. In the situation where there are no l directions, we flip all r to l; this is the scenario where the list of nodes wraps around. So the name $rlrr$ is converted to $rrll$ and rr to ll. In general we can denote this operation as, $x.l.r^p \to x.r.l^p$, and $r^p \to l^p$, where p is any integer $p \geq 0$ and x is any path name. We can similarly describe the full set of directions as follows.

$$
\begin{array}{lll}
 & N_1 : x.l.r^p \to x.r.l^p & \\
L : x \to x.l & N_2 : r^p \to l^p & M_1 : x.l \to x \\
R : x \to x.r & P_1 : x.r.l^p \to x.l.r^p & M_2 : x.r \to x \\
 & P_2 : l^p \to r^p & M_3 : \epsilon \to \epsilon
\end{array}
$$

To apply a direction to an existing path we simply find the first format that matches the path and apply the rule. We can also apply the inverse of a direction, as we do when working with the write word, by applying the rules from right to left.

The aim is then, to perform any sequence of operations (corresponding to rw^{-1}), on a general path name. The method works as follows. We begin with two formats, the *restriction*, R and the *current*, C. When we apply an operation, we match up each possible rule with C, and then apply the rule to C. More than one rule may match C in more than one way, so the process will branch to produce many different outcomes. If a rule does not match, we may need to enforce restrictions on the original path, and update R accordingly.

We work through this in the context of our example. We want to apply the operations $N.M^{-1}$ to a general path x. First we apply N, which restricts x to be of the format $x'.l.r^p$ or r^p. We apply the rules and branch into two subcases. We continue the process by applying the inverse of direction M. The evaluation tree is depicted below, the dashed boxes represent the restriction, the unbroken boxes the current path names.

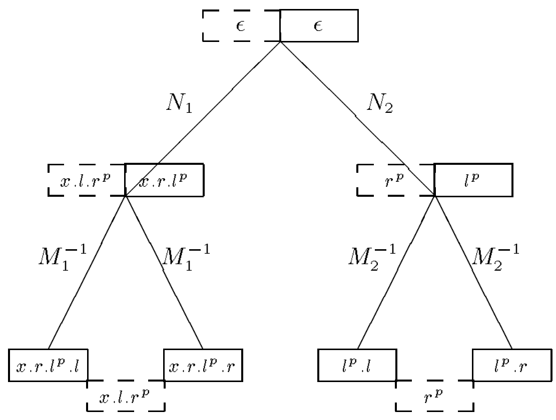


Fig. 2. Example analysis tree

We can now convert these dependencies between parts of the data structure to dependencies between different control words of the function. When this has been done for all the possible pairs of reads and writes in the function, this information can be combined into dataflow information that describes the source/sink of each value.

3 Conclusions

A static analysis approach has been proposed for a class of graph structures whose structure is described by a Markov algorithm. The method hinges on a path normalisation scheme which can be described as a Markov algorithm. This provides a general framework for static analysis of other structures with different relations over the edges

We are currently extending the class of recursive functions to handle a more general form of predicate on the individual recursive calls. In a sense the predicate describes a subset of the data structure, which renders the static analysis more complex as this subset may be run-time dependent.

Another line of investigation is in considering static analysis of the larger class of structures described by Markov algorithms.

References

1. A. T. Berztiss. *Data Structures - Theory and Practice.* New York : Academic Press, 1971.
2. Albert Cohen, Jean-Francois Collard, and Martin Griebl. Array data-flow analysis for imperative recursive programs. Technical Report 96/035, Laboratory PRiSM, University of Versailles, France, 1996.
3. 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.