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:

```c
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: $F_i : G^* \rightarrow \mathcal{P}(G^*)$; 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 $F : S \times A \rightarrow S$, 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