Recursive Data Structures
and Parallelism Detection

Laurie Hendren*

TR 88-924
June 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported in part by NSF grant CCR 87-04367.
Recursive Data Structures and Parallelism Detection

Laurie Hendren*

June 23, 1988

Abstract

Interference estimation is a key aspect of automatic parallelization of programs. In this paper we study the problem of estimating interference in a language with dynamic data-structures. We focus on the case of binary trees to illustrate the approach. We develop a structural flow-analysis technique that allows us to estimate whether two statements influence disjoint sub-trees of a forest of dynamically-allocated binary trees. The method uses a regular-expression-like representation of the relationships between the nodes of the trees and is based on the algebraic properties of such expressions. We have implemented our analysis in Standard ML and have obtained some promising experimental results.

1 Introduction

The emergence of parallel architectures holds the promise of faster execution of programs. Unfortunately, dealing with parallelism adds a new di-

*This work was supported in part by NSF grant CCR 87-04367
mension to the design of algorithms and programs, thereby further increasing the complexity of programming. Traditional programming languages do not provide any mechanism for handling this added complexity. One approach to handling parallelism is to design a language that includes constructs for expressing parallel computations directly. With this approach, the onus is on the programmer to express which computations may be done in parallel. A second approach is to use a more conventional programming language, but to provide a parallelizing compiler to extract parallelism. Thus, a sequential program is converted into an equivalent parallel program in which some computations may be done in parallel.

When the underlying programming language is imperative, alias analysis and interference detection are central to both approaches. Two computations interfere if one computation writes to a location that the other computation reads or writes. Interfering computations may not be executed in parallel. In general, alias detection is required for interference detection. Various approaches to interference detection and alias analysis have been suggested: traditional flow analysis [Bar78,Ban79,BC86], syntactic control on interference [Rey78,Ten83], abstract interpretation for estimating aliases and side effects [NPD87,Nei88], and effect analysis [Luc87,LG88]. Although all of these methods handle atomic data structures, there remains a problem with dealing with large aggregate data structures such as lists, trees and arrays.

Alias analysis for programs containing array references is difficult in general. However, various techniques have been developed for analyzing simple, regular array references. As a result, significant progress has been made in parallelizing scientific programs in which the array references occur in a simple, regular manner [PW86,AK87]. Alias analysis for programs containing dynamic data structures is a problem that has only recently been attacked. Existing methods give only coarse-grain alias information. However, just as array alias analysis can be done in some regular cases, we propose that fine-grain alias analysis of dynamic structures can be done with recursive data structures such as trees. Furthermore, we propose that such techniques can be used for interference detection and parallelism extraction in programs which use dynamic data structures in a regular (recursive) manner. In the following section, an overview of the problems encountered in interference
analysis for dynamic data structures is presented, a description of previous work in the area is given, and a new interference analysis technique for binary trees is developed.

2 Interference Analysis for Dynamic Data Structures

Interference detection in the presence of dynamic data structures and pointers is a difficult problem that has been only partially solved. Intuitively, the difficulty lies in the lack of compile-time names for all allocated objects. Static scalar structures are easy to handle because they can be associated with an identifier name at compile-time. Aggregate structures such as arrays are more difficult to deal with. Although entire arrays can be associated with an identifier name, elements within an array have computed names. Thus, it is relatively easy to determine that two arrays X and Y are disjoint collections of objects or that the elements X[i] and Y[j] are different objects. However, it is substantially more difficult to determine that X[i] and X[j] are different objects. Since i and j are computed values, subscript analysis must be performed. With dynamic data structures, objects are allocated and linked together at run-time. Thus, not only is there no compile-time name for each individual object, there is also no simple way to name collections of objects.

2.1 Previous Work

Recently, two approaches have been suggested for interference analysis in the presence of dynamic data structures and pointers.

Lucassen and Gifford have proposed an approach whereby a programming language is defined that incorporates an effect system as well as a type system [Luc87, LG88]. The effect of a computation is a summary of the observable side-effects of the computation. For example, \{ \text{read}(X), \text{write}(Y),
alloc(Z)} represents the effect of a computation that reads from region X, writes to region Y, and allocates objects in region Z. A region describes the area of store in which side-effects might occur. Two computations interfere when they have interfering effects.

![Diagram of regions X and Y](image)

Figure 1: Collections of objects separated into regions

The effect system ensures that each dynamic object is allocated in a named region and objects may be linked together only if they are associated with the same region. Thus, objects in one region are guaranteed not to share any storage with objects in another region. Figure 1 represents situation where tree A has been created in region X and tree B has been created in region Y. Any computation on tree A will not interfere with a computation on tree B.

Although the effect system provides a method of statically associating collections of objects with region names, it does not provide a way of distinguishing between individual objects within a region. Note that even though subtrees C and D do not actually share any common storage, the effect system forces both subtrees to be associated with region Y. This lack of fine-grain information results in an overly conservative interference analysis for recursive structures such as trees.
Another approach to static analysis for dynamic structures has been proposed by Neirynck [Nci88]. This method uses abstract interpretation techniques to provide information about aliasing and side effects in a higher-order expression language. Within this framework, dynamic data structures are handled by estimating each linked structure in an abstract store. Each call to a recursive function which creates a linked data structure is approximated by one entry in the abstract store. Figure 2 illustrates the correspondence between the actual store and abstract store for tree p and list q. Note that this approach also fails to provide any information about the internal structure of collections. For example, the abstract store fails to distinguish between the left and right subtrees of p.

![Diagram](image)

(a) the real store  
(b) an approximation

Figure 2: Collections of objects represented by an abstract store

### 2.2 Interference analysis for binary trees

Existing interference analysis techniques for dynamic data structures gather only coarse-grain information about collections of objects which are linked together. However, the regular structure of recursive data structures such as lists and trees should allow more fine-grain analysis. As an illustration of
fine-grain analysis for regular recursive structures, an interference analysis method for binary trees is presented in this section.

The basic building blocks of binary trees are nodes. Each node consists of one or more scalar values, a left pointer to a node, and a right pointer to a node. In general, objects built by linking such nodes together are directed graphs. We can classify two special types of directed graphs: (1) a TREE is a directed graph in which each node has at most one parent, and (2) a DAG is a directed graph in which some node has more than one parent and the graph does not contain a directed cycle.

The potential for parallelism in programs that use binary trees arises from the following observation. If a program builds linked structures that are of type TREE, then the left and right sub-trees, $T_{left}$ and $T_{right}$, of tree $T$ are guaranteed to share no common storage. Thus, a computation on $T_{left}$ or any sub-tree of $T_{left}$ will not interfere with a computation on $T_{right}$ or any sub-tree of $T_{right}$. Using this observation, we conclude that a useful interference analysis tool would: (1) check that the linked structures created by a program are guaranteed to be of type TREE and (2) recognize when two trees are are unrelated (one tree is not a sub-tree of the other). In the following sections, a small imperative programming language is presented, an interference analysis scheme for programs using TREES is developed, and an extension of the scheme that allows the creation of DAGS is discussed.

### 2.2.1 A simple imperative language - SIL

A SIL program consists of a set of possibly mutually recursive procedures and a main program. The language is statically scoped and has call by value semantics. Two types are supported: integers and handles. A handle can be thought of as a name of a binary tree node. The recursive type for handles can be expressed as follows:

```
type handle = Nil | (value: int; left: handle; right: handle).
```

This is equivalent to the Pascal type:
type
    handle = \bintreenode;
    bintreenode = record
        value: integer;
        left: handle;
        right: handle
    end .

The language provides a built-in function new which takes an integer value as input. Each invocation of new allocates a new node in the store, sets the value field to the input value, and initializes the left and right fields to Nil. The return value from an invocation of new must be assigned to a variable with type handle. A skeleton of the abstract syntax for SIL is given in Figure 3.

<Program> ::= <ProcedureList> <Block>

<Procedure> ::= procedure <id> ( <ParamList> )
               <LocalList>
               <Block>

<Block> ::= begin <StmtList> end

<Stmt> ::= <ScalarAssignment>
         | <HandleAssignment>
         | <HandleUpdate>
         | if <Expr> then <Stmt> [else <Stmt>]
         | while <Expr> do <Stmt>
         | <Block>
         | <ProcedureName> ( <ArgList> )

Figure 3: Abstract Syntax of SIL

Two kinds of statements are of interest for the analysis: handle assignments and handle updates. Handle assignments are used to assign handles to nodes in a tree, while handle updates are used to modify the structure of a tree. Figure 4 outlines the handle assignment and update statements along with the equivalent Pascal statements.
2.2.2 Analysis

In this section a structure-based dataflow analysis is developed. The goals of the analysis are: (1) to guarantee that all binary trees are built as \textit{Trees} and (2) to determine if two handles refer to the roots of unrelated subtrees. Given an input program, the analysis will compute a set of possible relationships among handles live at each point in the program. A \textit{point} refers to a position between two statements in the program. A handle \( h \) is \textit{live} at a point \( p \) if there is some execution path starting at \( p \) that uses \( h \). The structure of the analysis is illustrated as follows.

\[
\begin{array}{c}
\text{Statement} \\
\downarrow \\
p'
\end{array} \quad \downarrow \quad \begin{array}{c}
\text{r} \\
\uparrow \\
r'
\end{array} 
\]

\[
p \quad \downarrow \quad r
\]
Given $r$, an estimate of the relationships among all handles live at point $p$, we wish to compute $r'$, an estimate of the relationships among all handles live at point $p'$.

The estimate of relationships among handles captures the relative position of handles within a tree (or forest). Relative information can be used to detect if a statement creates a data structure that is possibly not a TREE. For example, if node $a$ is a descendent of node $b$, then the statement $a.left := b$ will create a cycle. Relative information may also be used to determine if two handles refer to disjoint sub-trees. If node $a$ is not a descendent of node $b$ and node $b$ is not a descendent of node $a$, then $a$ and $b$ refer to disjoint sub-trees and a computation on $a$ cannot interfere with a computation on $b$.

The relationship between two handles $a$ and $b$, denoted by $r(a,b)$, is specified by a set of paths. A path is denoted either by $S$ (meaning that two handles refer to the same node) or by a path expression which describes the directed path between two nodes. A path expression is non-empty sequence of links. A link is one of: $L^i$ - $i$ left edges, $L^+$ - one or more left edges, $R^i$ - $i$ right edges, $R^+$ - one or more right edges, $D^i$ - $i$ edges, or $D^+$ - one or more edges. Each path is classified as definite, the path is guaranteed to exist, or possible, the path may not exist. The relationships among a set of handles are described by a path matrix. Each entry in the matrix describes the relationship between two handles. Figure 5 illustrates the path matrix corresponding to a tree with five live handles.

The structure of the analysis can now be more precisely stated. For each kind of statement, a function is defined that takes as input an instance of a statement $s$ and a path matrix $p$ and produces as output a new path matrix $p'$. The most basic functions to be defined are those for handle assignments and handle updates.

As an example, consider the function for a statement of the form $a := b.left$. Given the relationships between $b$ and all other live handles, the function must compute the relationships between $a$ and all other live handles. The function is specified as three rules (see Figure 6). In each rule, $p$ represents the input path matrix, $p'$ represents the output path matrix,
$H'$ represents the set of handles live after the statement, $x$ represents any path expression, and $d$ represents a boolean value that is true if the path is definite and false if the path is possible. Rule I specifies the definite relationships between handles $a$ and $b$. There is a definite path $L^1$ between $b$ and $a$, there is no path between $a$ and $b$, and $a$ is the same node as $a$. Rule II specifies the relationships between all handles $H_i$ and $a$, where $H_i$ is above or the same as $b$. For each path $x$ between $H_i$ and $b$, there is a path $xL^1$ between $H_i$ and $a$. Rule III specifies the relationships between all handles $H_j$ and $a$, where $H_j$ is below $b$. For each path $x$ between $b$ and $H_j$ that begins with a $L$ or $D$ link, there is a shorter path between $a$ and $H_j$. Note that when the first link of $x$ is $D^+$ or $L^+$, there are two possible shorter paths.

An example of an application of the handle assignment rules is given in Figure 7. Figure 7(a) illustrates an initial path matrix representing the relationships between the handles $a$, $b$, and $c$. Figure 7(b) shows the path matrix that would result from the statement $d := a.right$ and 7(c) illustrates the resulting path matrix after the statement $e := d.left$. Note that although the path matrices in 7(a) and 7(b) have only definite paths, the path matrix in 7(c) contains some possible paths (denoted by ?). Since the exact length of the path between handles $d$ and $a$ is not
\begin{itemize}

\item \textbf{Rule I}

\begin{align*}
 p'(b, a) &= \{ \langle L^1, \text{true} \rangle \} \\
p'(a, b) &= \{ \} \\
p'(a, a) &= \{ \langle S, \text{true} \rangle \}
\end{align*}

\item \textbf{Rule II}

\begin{align*}
 \forall H_i \in H'. (H_i \neq a) \land (H_i \neq b) \Rightarrow \\
 \langle S, d \rangle \in p(H_i, b) \Rightarrow \langle L^1, d \rangle \in p'(H_i, a) \\
 \langle x, d \rangle \in p(H_i, b) \Rightarrow \langle xL^1, d \rangle \in p'(H_i, a)
\end{align*}

\item \textbf{Rule III}

\begin{align*}
 \forall H_j \in H'. (H_j \neq a) \land (H_j \neq b) \Rightarrow \\
 (L^1, d) \in p(b, H_j) \Rightarrow \langle S, d \rangle \in p'(a, H_j) \\
 (L^n, d) \in p(b, H_j) \Rightarrow \langle L^{n-1}, d \rangle \in p'(a, H_j) \\
 (L^+, d) \in p(b, H_j) \Rightarrow \langle S, \text{false} \rangle, \langle L^+, \text{false} \rangle \in p'(a, H_j) \\
 (D^1, d) \in p(b, H_j) \Rightarrow \langle S, \text{false} \rangle \in p'(a, H_j) \\
 (D^n, d) \in p(b, H_j) \Rightarrow \langle D^{n-1}, \text{false} \rangle \in p'(a, H_j) \\
 (D^+, d) \in p(b, H_j) \Rightarrow \langle S, \text{false} \rangle, \langle D^+, \text{false} \rangle \in p'(a, H_j) \\
 (L^1x, d) \in p(b, H_j) \Rightarrow \langle x, d \rangle \in p'(a, H_j) \\
 (L^nx, d) \in p(b, H_j) \Rightarrow \langle L^{n-1}x, d \rangle \in p'(a, H_j) \\
 (L^+x, d) \in p(b, H_j) \Rightarrow \langle x, \text{false} \rangle, \langle L^+x, \text{false} \rangle \in p'(a, H_j) \\
 (D^1x, d) \in p(b, H_j) \Rightarrow \langle x, \text{false} \rangle \in p'(a, H_j) \\
 (D^nx, d) \in p(b, H_j) \Rightarrow \langle L^{n-1}x, \text{false} \rangle \in p'(a, H_j) \\
 (D^+x, d) \in p(b, H_j) \Rightarrow \langle x, \text{false} \rangle, \langle D^+x, \text{false} \rangle \in p'(a, H_j)
\end{align*}

\end{itemize}

\textbf{Figure 6: Rules for } a := b\.left
known, the path between of handles \( e \) and \( c \) is either \( S \) (\( e \) and \( c \) may be handles to the same node) or \( D^+ \) (\( c \) is one or more edges below \( e \)).

Functions have been defined for each kind of handle assignment and handle update statements. Using these basic functions as building blocks, functions for blocks (statement sequences), conditional statements, and while loops can be defined as follows.

Given an input path matrix \( p_0 \) and a sequence of statements \( s_1; s_2; \ldots; s_n \) from a block, the final path matrix \( p_n \) is produced as follows. For each statement, \( s_i, i = 1..n \), the statement analysis function is applied to \( p_{i-1} \) and \( s_i \) resulting in \( p_i \).

Given an input path matrix \( p \) and a conditional statement of the form \textbf{if} \( \langle expr \rangle \ \textbf{then} \ \langle stmt_1 \rangle \ \textbf{else} \ \langle stmt_2 \rangle \), a pair of path matrices \( \langle p_{\text{then}}, p_{\text{else}} \rangle \) is produced as follows. The statement analysis function is applied to \( p \) and \( stmt_1 \) to produce \( p_{\text{then}} \). The statement analysis function is applied to \( p \) and \( stmt_2 \) to produce \( p_{\text{else}} \).

Given an input path matrix and a while loop of the form \textbf{while} \( \langle expr \rangle \ \textbf{do} \ \langle stmt \rangle \), a pair of path matrices \( \langle p_0, p_+ \rangle \) is produced as follows. The output path matrix \( p_0 \) represents zero iterations of the while loop and is equal the input path matrix \( p \). The output path matrix \( p_+ \) approximates one or more iterations of of the while loop, and is computed using a fixed-point iterative approximation. The statement analysis function is applied to \( p \) and \( stmt \) to produce the first estimate, \( p_1 \). For each iteration \( i \), the analysis function is applied to \( p_i \) and \( stmt \) to produce \( p'_i \). The next iteration begins with \( p_{i+1} \), where \( p_{i+1} = \text{merge}(p_i, p'_i) \). The iterative approximation terminates when \( p_i = p_{i+1} \).

In order to perform the fixed-point calculation, it is necessary to: (1) test for equality of path matrices and (2) define a \textit{merge} function that guarantees termination of the iterative approximation. Two building blocks are required in order to express the equality test and merge operator: (1) an efficient equality test for path expressions and (2) an efficient squash function for path expressions.
Figure 7: An example of handle assignments
Two path expressions $p_1$ and $p_2$ are equal if their normal forms are identical (they have exactly the same sequence of links). Each path expression $p$ has a normal form that consists of a concatenation of subsequences of the form $L^i$, $L^+$, $L^iL^+$, $R^i$, $R^+$, $R^iR^+$, $D^i$, $D^+$, $D^iD^+$, such that no two subsequences of the same kind ($L$, $R$, or $D$) are adjacent. For example, the normal forms of $L^1L^+L^3R^+R^+D^+$ and $L^+L^+L^3R^1R^+D^+$ are both $L^4L^+R^1R^+D^+$. The complexity of the normal form computation for a path expression $p$, is $O(n)$, where $n$ is the number of links in $p$. The complexity of equality testing of two normalized path expressions $p_1$ and $p_2$ is $O(\min(n, m))$, where $n$ is the number of links in $p_1$ and $m$ is the number of links in $p_2$.

Path expression $p_1$ is more general than path expression $p_2$ if all paths in the set denoted by $p_2$ are also in the set denoted by $p_1$ and there exists some path in the set denoted by $p_1$ that is not in the set denoted by $p_2$.

The minimal length of a path expression is the sum of the minimal lengths of its links, where the minimal length of $D^+$, $R^+$, or $L^+$ is 1 and the minimal length of $D^i$, $R^i$, or $L^i$ is $i$.

The squash of two normalized path expressions $p_1$ and $p_2$ is a normalized path expression $q$ such that: (1) any path in the sets denoted by $p_1$ or $p_2$ is also in the set denoted by $q$ and (2) the minimal length of $q$ is no greater than $\min(n, m)$, where $n$ is the minimal length of $p_1$ and $m$ is the minimal length of $p_2$. Some examples of valid squashes are: $\text{squash}(R^+, R^1) = R^+$, $\text{squash}(L^2R^1, L^1R^2) = L^1D^1R^1$, and $\text{squash}(L^1L^+, L^1L^+L^1) = L^1L^+$. 

The definition of merge can now be given in terms of squash. The merge of two path matrices $m_1$ and $m_2$ is calculated as follows. Each entry in $m_1$ and $m_2$ is simplified by squashing all path expressions into one representative path expression. Thus, each entry in $m_1$ and $m_2$ will become one of: $\{\}$, $\{S\}$, $\{x\}$, or $\{S, x\}$ (where $x$ is the representative path expression). The calculation of $m_{\text{merged}} = \text{merge}(m_1, m_2)$, uses the simplified entries in $m_1$ and $m_2$ as follows. If $m_1(H_i, H_j)$ contains $S$ or $m_2(H_i, H_j)$ contains $S$, then $m_{\text{merged}}(H_i, H_j)$ must contain $S$. If $m_1(H_i, H_j)$ contains a path expression $x$ and $m_2(H_i, H_j)$ contains a path expression $y$, then $m_{\text{merged}}(H_i, H_j)$ must contain $\text{squash}(x, y)$. If only one of $m_1(H_i, H_j)$ and $m_2(H_i, H_j)$ contains a path expression, then $m_{\text{merged}}(H_i, H_j)$ must contain that path expression.
An example of the merge operation is illustrated in Figure 8.

Original path matrices

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1R^+$, $L^1L^+$</td>
<td>$L^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1D^+$, $L^1L^+R^1$</td>
<td>$R^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>$S$</td>
</tr>
</tbody>
</table>

Simplified matrices, each item squashed

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1D^+$</td>
<td>$L^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1D^+$</td>
<td>$D^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

Result of merging the two simplified matrices

Figure 8: An example of the merge operation

Given the merge operation, it is possible to show that the iterative approximation for the while loop terminates. Path matrix $m_1$ equals path matrix $m_2$ when all entries, $m_1(H_i, H_j) = m_2(H_i, H_j)$. Let $p_1, p_2, ..., p_n$ be the sequence of path matrix estimates calculated by the iterative approximation. Given the previous definitions of merge and squash, it can be shown that the representative path expression for any element $p_k(H_i, H_j)$ can only be the same or more general than the representative path expression of the same element in $p_{k-1}(H_i, H_j)$. It can also be shown that the minimal length of the representative path expression in $p_k(H_i, H_j)$ must
be no larger than the minimal length of the representative path expression in $p_{k-1}(H_i, H_j)$. Thus, the series of approximations for each element $p_1(H_i, H_j), p_2(H_i, H_j), ..., p_n(H_i, H_j)$ is finite and the iterative approximation for while loops terminates.

The analysis function for procedure calls is outlined below. First, consider the case of a call to a non-recursive procedure. Given an input path matrix $p$ and a procedure call of the form $f(H_1, ..., H_n)$, the resulting path matrix $p'$ is produced as follows. The body of procedure $f$ analysed with an input path matrix $q$, where $q$ is a path matrix that combines path information between handles live immediately after the call to $f$ and the handles which correspond to formal parameters of $f$. Consider as an example the following procedure which contains the procedure call $\texttt{SwapChildren(root)}$.

procedure $f(root:\ handle);$
  $l, r:\ handle$
begin
  $l := root.left;$
  $r := root.right;$
  $\texttt{SwapChildren(root); \{ swaps the left and right sub-trees of root \}}$
  ...
  \{ use $l$ and $r$ \}
  ...
end

In this example, the call to $\texttt{SwapChildren}$ changes the relationships between handles $root$, $l$ and $r$. Figure 9 illustrates four stages of the path matrix computation. Figure 9(a) represents the path matrix at the point just before the call $\texttt{SwapChildren(root)}$. The path matrix of 9(a) is combined with the formal parameter $h$ to produce 9(b), the input path matrix for analysis of the body of $\texttt{SwapChildren}$. Figure 9(c) illustrates the path matrix resulting from the analysis of the body, while 9(d) is the final result of the call.

Analysis of calls to recursive procedures requires a combination of the techniques used in non-recursive procedure calls and the techniques used
for fixed-point approximations.

In this section, an overview of an interference analysis method for TREES has been given. This analysis calculates the relative positions of live handles at each point in the program. The analysis relies on the fact that the data structure is a TREE and if at any point a possible DAG is detected, then the analysis must fail.

### 2.3 Interference analysis for DAGS

In this section an extension of the TREE analysis to handle DAGS is outlined. Simple imperative programs that process TREES may also create DAGS temporarily. For example, consider the following definition of reverse. Note that the statement a.left := r creates a situation in which
both $a.left$ and $a.right$ refer to $r$. The TREE analysis would fail at this point. However, note that the next statement $a.right := l$ recreates a TREE.

procedure reverse(a:handle)
    l, r: handle
begin
    l := a.left;
    r := a.right;
    reverse(l);
    reverse(r);
    a.left := r;
    a.right := l
end

In order to handle DAGS it is necessary to maintain more information in the path matrix. Information about nodes that have more than one parent is also required. A dag node is a node for which there may be more than one parent (more than one incoming edge). For each dag node, the information required is: (1) an estimate of its position relative to the live handles and other dag nodes and (2) an estimate of its reference count.

The basic functions for DAGS are quite similar the those for TREES. Since the number of dag nodes created is not known statically, dag nodes are approximated by associating a dag handle with each handle update statement in the program. For example, in the reverse function a dag handle would be associated with the statements $a.left := r$ and $a.right := l$. If the analysis of the statement indicates that a dag node is possible, then the dag handle associated with the statement is placed in the path matrix. If analysis of a later statement indicates that the dag handle now refers to a node that has at most one parent, the dag handle can be removed from the path matrix and the analysis can continue as in the TREE case. In the case of reverse, the analysis of the statement $a.left := r$ causes a dag handle to be added to the path matrix and then the analysis of $a.right := l$ removes the dag handle.
2.4 Implementation

A prototype of interference analysis for SIL has been implemented in Standard ML. The analysis functions for handle update statements, handle assignment statements, conditional statements and while loops have been implemented. An example of the output produced by this system may be found in Appendix A. Appendix B gives the output for a bitonic merge procedure.

3 Conclusions and Future Work

Since the interference analysis is an estimation scheme, we must demonstrate that the analysis provides useful information on real programs. Preliminary results indicate that the binary tree analysis does provide useful information for the extraction of parallelism.

We plan to continue the development of interference analysis tools for dynamic recursive data structures. As part of this project, we plan to extend the implementation to include fixed-point approximations for recursion, to handle n-ary trees, and to incorporate further effect analysis. In addition, we plan to collect a set of benchmarks that exhibit regular uses of dynamic data structures. These benchmarks will provide a basis for experimenting with variations on the interference analysis techniques and for developing parallelization tools based on the information produced by the analysis.
A  A Small Example

This example illustrates the computation of path matrices for a simple program containing a while loop and a conditional statement. The following are included in this example:

- The source program for procedure `test_while`.
- The abstract syntax tree and environment for `test_while`. Note that the syntax tree is decorated with the set of handles live at each point in the tree.
- Output produced by the analysis program. The output traces the analysis by printing each statement as it is analyzed, along with the resulting path matrix.
- An illustration for each of the two possible path matrices computed at the final point in the program. Note that the first path matrix corresponds to the handle relationships after zero iterations of the while loop, while the second path matrix corresponds to an approximation of the handle relationships for one or more iterations.

```plaintext
procedure test_while (h: handle; key:int)
   i: int;
   l, d, bot, temp: handle
begin
   l := h.left;
   d := h.right;
   while d ≠ nil do
      if d.value > key then
         begin
            temp := d.left;
            d := temp
         end
      else
```

20
begin
    temp := d.right;
    d := temp
end;
bot := d.left;
i := l.value + bot.value
end

PROCEDURE: test_while

ARGS:
    [ [h, handle], [key, int] ]

LOCALS:
    [ [i, int], [l, handle], [d, handle], [bot, handle], [temp, handle] ]

DAG HANDLES:
    [ ]

BODY:

Block({ h })
    HandleAssignment(l, HandleField(h, left))
    { h l };
    HandleAssignment(d, HandleField(h, right))
    { h l d };
    While({ h l d })
        BinaryExp(UnEqual,
            Var(HandleId(d)),
            Nil),
    Conditional(
        BinaryExp(Greater,
            Var(ScalarField(d, value)),
            Var(ScalarId(key))),
        Block({ h l d })
            HandleAssignment(temp, HandleField(d, left))
            { h l temp };
            HandleAssignment(d, HandleId(temp))
            { h l d });
        Block({ h l d })
            HandleAssignment(temp, HandleField(d, right))
            { h l temp };

21
HandleAssignment(d, HandleId(temp))
    { h l d }));
HandleAssignment(bot, HandleField(d, left))
    { h l bot };
ScalarUpdate(
    ScalarId(i),
    BinaryExp(Plus,
        Var(ScalarField(l, value)),
        Var(ScalarField(bot, value))))
    { h }

HandleAssignment(l, HandleField(h, left))
HSimple(
    h    l
h:    [S] [L1]
l:    [] [S]
)
HandleAssignment(d, HandleField(h, right))
HSimple(
    h    l    d
h:    [S] [L1] [R1]
l:    [] [S] []
d:    [] [] [S]
)

================================ BEGIN WHILE  ===============
================================ COND (THEN PART) ===============

HandleAssignment(temp, HandleField(d, left))
HSimple(
    h    l    temp
h:    [S] [L1] [R1L1]
l:    [] [S] []
temp: [] [] [S]
)
HandleAssignment(d, HandleId(temp))
HSimple(
    h    l    d
h:    [S] [L1] [R1L1]
l:    [] [S] []
d: [] [] [S]
)

------------------- (ELSE PART) -------------------

HandleAssignment(temp, HandleField(d, right))
HSimple(
    h  l  temp
h:  [S] [L1] [R2]
l:  [] [S] []
temp: [] [] [S]
)

HandleAssignment(d, HandleId(temp))
HSimple(
    h  l  d
h:  [S] [L1] [R2]
l:  [] [S] []
d:  [] [] [S]
)

------------------- END COND -------------------

------- results of iteration #1 -------

IN:
    h  l  d
h:  [S] [L1] [R1]
l:  [] [S] []
d:  [] [] [S]

OUT:
    h  l  d
h:  [S] [L1] [R1D1]
l:  [] [S] []
d:  [] [] [S]

================== ITERATION # 2 ===================

COND (THEN PART) ==================

HandleAssignment(temp, HandleField(d, left))
HSimple(
    h  l  temp
h:  [S] [L1] [R1D1L1]
\textbf{else part}  

\begin{verbatim}
HandleAssignment(temp, HandleField(d, right))
HSimple(
   h  l  temp
   h:  [S] [L1] [R1D1R1]
   l:  []  [S]  []
   temp:  []  []  [S]
)
\end{verbatim}

\begin{verbatim}
HandleAssignment(d, HandleId(temp))
HSimple(
   h  l  d
   h:  [S] [L1] [R1D1R1]
   l:  []  [S]  []
   d:  []  []  [S]
)
\end{verbatim}

\begin{verbatim}
\textbf{end cond}  
\end{verbatim}

\begin{verbatim}
\textbf{results of iteration #2}
\end{verbatim}

\begin{verbatim}
IN:
   h  l  d
   h:  [S] [L1] [R1D1]
   l:  []  [S]  []
   d:  []  []  [S]
\end{verbatim}

\begin{verbatim}
OUT:
   h  l  d
   h:  [S] [L1] [R1D2]
   l:  []  [S]  []
\end{verbatim}
\textbf{HandleAssignment}(temp, \text{HandleField}(d, \text{left}))
\text{HSimple}(h\ l\ \text{temp})
\begin{align*}
h: & \quad [S] [L1] [R1D+L1] \\
l: & \quad [] [S] [] \\
temp: & \quad [] [] [S] \\
\end{align*}
\}
\text{HandleAssignment}(d, \text{HandleId}(temp))
\text{HSimple}(h\ l\ d)
\begin{align*}
h: & \quad [S] [L1] [R1D+L1] \\
l: & \quad [] [S] [] \\
d: & \quad [] [] [S] \\
\end{align*}
\)
\textbf{HandleAssignment}(temp, \text{HandleField}(d, \text{right}))
\text{HSimple}(h\ l\ \text{temp})
\begin{align*}
h: & \quad [S] [L1] [R1D+R1] \\
l: & \quad [] [S] [] \\
temp: & \quad [] [] [S] \\
\end{align*}
\)
\text{HandleAssignment}(d, \text{HandleId}(temp))
\text{HSimple}(h\ l\ d)
\begin{align*}
h: & \quad [S] [L1] [R1D+R1] \\
l: & \quad [] [S] [] \\
d: & \quad [] [] [S] \\
\end{align*}
--- results of iteration #3 ---

IN:
   h  l  d
h:  [S] [L1] [R1D+]
l:  [] [S] []
d:  [] [] [S]

OUT:
   h  l  d
h:  [S] [L1] [R1D1D+]
l:  [] [S] []
d:  [] [] [S]

IN merge OUT => next(IN):
   h  l  d
h:  [S] [L1] [R1D+]
l:  [] [S] []
d:  [] [] [S]

===================== END WHILE ======================

HandleAssignment(bot, HandleField(d, left))
HWhile(HSimple(
   h  l  bot
h:  [S] [L1] [R1L1]
l:  [] [S] []
bot:  [] [] [S]
),
HSimple(
   h  l  bot
h:  [S] [L1] [R1D+L1]
l:  [] [S] []
bot:  [] [] [S]
))
B Bitonic Merge Example

This example illustrates the computation of path matrices for the bitonic merge algorithm presented in [BN86]. At each phase in the algorithm, a traversal is made down the tree. At each stage in the traversal values may be swapped and sub-trees may be swapped. The following are included in this example:

- The source program for bitonic merge, \textit{bimerge}.

- A sample of the output from the analysis. Note that dag handles appear in some path matrices that follow handle update instructions. These dag handles reflect the intermediate state of the swap of sub-trees. However, these dag handles are only temporary. The next instruction always reduces their reference count to 1 and the dag handle is not included in the next path matrix. Also note that the two recursive calls \textit{bimerge}(rl,root.value,dir) and \textit{bimerge}(rr,spare,dir) can be done in parallel because \textit{rl} and \textit{rr} refer to disjoint sub-trees.
procedure bimerge(root: handle; spare: int; dir: int)
  rightezchange, elementezchange, temp : int;
  pl, pr, rl, rr, tl, t2: handle
begin
  rightezchange := (root.value > spare) xor dir;
  if rightezchange then
    begin
      temp := root.value;
      root.value := spare;
      spare := temp
    end;
  pl := root.left;
  pr := root.right;
  while (pl ≠ nil) do
    begin
      elementezchange := (pl.value > pr.value) xor dir;
      if rightezchange then
        if elementezchange then
          { swap values and right subtrees, search path goes left }
          begin
            { SwapValue(pl,pr) }
            temp := pl.value;
            pl.value := pr.value;
            pr.value := temp;
            { SwapRight(pl,pr) }
            t1 := pl.right;
            t2 := pr.right; pl.right := t2; { pl.right := pr.right }
            pr.right := t1;
            { move pl and pr to the left }
            t1 := pl.left; pl := t1; { pl := pl.left }
            t2 := pr.left; pr := t2 { pr := pr.left }
          end
        else
          { search path goes left }
          begin
            t1 := pl.right; pl := t1; { pl := pl.right }
            t2 := pr.right; pr := t2 { pr := pr.right }
          end
    end
  end
else
  if elementexchange then
    { swap values and left subtrees, search path goes right }
    begin
      { SwapValue(pl,pr) }
      temp := pl.value;
      pl.value := pr.value;
      pr.value := temp;
      { SwapLeft(pl,pr) }
      t1 := pl.left;
      t2 := pr.left; pl.left := t2; { pl.left := pr.left }
      pr.left := t1;
      { move pl and pr to the right }
      t1 := pl.right; pl := t1; { pl := pl.right }
      t2 := pr.right; pr := t2 { pr := pr.right }
    end
  else
    { search path goes left }
    begin
      t1 := pl.left; pl := t1; { pl := pl.left }
      t2 := pr.left; pr := t2 { pr := pr.right }
    end
end; { while }
if (root.left ≠ nil) then
  begin
    rl := root.left;
    rr := root.right;
    bmerge(rl,root.value.dir);
    bmerge(rr,spare,dir)
  end
end { bmerge }

-------------------------------
HandleAssignment(pl, HANDLEField(root, left))
HSimple(
  root pl
root:  [S]  [L1]
pl:    []  [S]
29
HandleAssignment(pr, HandleField(root, right))
HSimple(
    root pl pr
    root: [S] [L1] [R1]
    pl:   [] [S] []
    pr:   [] [] [S]
)

BEGIN WHILE
COND (THEN PART)

COND (THEN PART)

HandleAssignment(t1, HandleField(pl, right))
HSimple(
    root pl pr t1
    root: [S] [L1] [R1] [L1R1]
    pl:   [] [S] [] [R1]
    pr:   [] [] [S] []
    t1:   [] [] [] [S]
)

HandleAssignment(t2, HandleField(pr, right))
HSimple(
    root pl pr t1 t2
    root: [S] [L1] [R1] [L1R1] [R2]
    pl:   [] [S] [] [R1] []
    pr:   [] [] [S] [] [R1]
    t1:   [] [] [] [S] []
    t2:   [] [] [] [] [S]
)

HandleUpdate(HandleField(pl, right), HandleName(t2, t200))
HSimple(
    root pl pr t1 t200
    root: [S] [L1] [R1] [L1R1] [R2, L1R1]
    pl:   [] [S] [] [] [R1]
    pr:   [] [] [S] [] [R1]
    t1:   [] [] [] [S] []
    t200: [] [] [] [] [S]
)

)
HandleUpdate(HandleField(pr, right), HandleName(t1, t1[i]))
HSimple(
    root pl pr
    root: [S] [L1] [R1]
    pl: [] [S] []
    pr: [] [] [S]
)
HandleAssignment(t1, HandleField(pl, left))
HSimple(
    root pr t1
    root: [S] [R1] [L2]
    pr: [] [S] []
    t1: [] [] [S]
)
HandleAssignment(pl, HandleId(t1))
HSimple(
    root pl pr
    root: [S] [L2] [R1]
    pl: [] [S] []
    pr: [] [] [S]
)
HandleAssignment(t2, HandleField(pr, left))
HSimple(
    root pl t2
    root: [S] [L2] [R1L1]
    pl: [] [S] []
    t2: [] [] [S]
)
HandleAssignment(pr, HandleId(t2))
HSimple(
    root pl pr
    root: [S] [L2] [R1L1]
    pl: [] [S] []
    pr: [] [] [S]
)
================================= (ELSE PART) =================================
HandleAssignment(t1, HandleField(pl, right))
HSimple(
root pr t1
root: [S] [R1] [L1R1]
pr: [] [S] []
t1: [] [] [S]

) HandleAssignment(p1, HandleId(t1))
HSimple(
  root pl pr
root: [S] [L1R1] [R1]
pl: [] [S] []
pr: [] [] [S]
)

) HandleAssignment(t2, HandleField(pr, right))
HSimple(
  root pl t2
root: [S] [L1R1] [R2]
pl: [] [S] []
t2: [] [] [S]
)

) HandleAssignment(pr, HandleId(t2))
HSimple(
  root pl pr
root: [S] [L1R1] [R2]
pl: [] [S] []
pr: [] [] [S]
)

====================== END COND =======================

==================== (ELSE PART) ======================

==================== COND (THEN PART) =================

HandleAssignment(t1, HandleField(p1, left))
HSimple(
  root pl pr t1
root: [S] [L1] [R1] [L2]
pl: [] [S] [] [L1]
pr: [] [] [S] []
t1: [] [] [] [S]

32
HandleAssignment(t2, HandleField(pr, left))
HSimple(
    root pl  pr  t1  t2
root:  [S]  [L1]  [R1]  [L2]  [R1L1]
pl:   []    [S]  []    [L1]  []
pr:   []    []    [S]  []    [L1]
t1:   []    []    []    [S]  []
t2:   []    []    []    []    [S]
)
HandleUpdate(HandleField(pl, left), HandleName(t2, t2@2))
HSimple(
    root pl  pr  t1  t2@2@2
root:  [S]  [L1]  [R1]  []  [R1L1,L2]
pl:   []    [S]  []    []    [L1]
pr:   []    []    [S]  []    [L1]
t1:   []    []    []    [S]  []
t2@2:   []    []    []    []    [S]
)
HandleUpdate(HandleField(pr, left), HandleName(t1, t1@3))
HSimple(
    root pl  pr
root:  [S]  [L1]  [R1]
pl:   []    [S]  []
pr:   []    []    [S]
)
HandleAssignment(t1, HandleField(pl, right))
HSimple(
    root pr  t1
root:  [S]  [R1]  [L1R1]
pr:   []    [S]  []
t1:   []    []    [S]
)
HandleAssignment(pl, HandleId(t1))
HSimple(
    root pl  pr
root:  [S]  [L1R1]  [R1]
pl:   []    [S]  []
pr:   []    []    [S]
)
) HandleAssignment(t2, HandleField(pr, right))
HSimple(
    root pl   t2
root:  [S] [L1R1] [R2]
pl:   []  [S]  []
t2:    []  []  [S]
)
)
HandleAssignment(pr, HandleId(t2))
HSimple(
    root pl   pr
root:  [S] [L1R1] [R2]
pl:   []  [S]  []
pr:    []  []  [S]
)

=============(ELSE PART)======================

) HandleAssignment(t1, HandleField(pl, left))
HSimple(
    root pr   t1
root:  [S] [R1] [L2]
pr:   []  [S]  []
t1:    []  []  [S]
)
)
HandleAssignment(pl, HandleId(t1))
HSimple(
    root pl   pr
root:  [S] [L2] [R1]
pl:   []  [S]  []
pr:    []  []  [S]
)
)
HandleAssignment(t2, HandleField(pr, left))
HSimple(
    root pl   t2
root:  [S] [L2] [R1L1]
pl:   []  [S]  []
t2:    []  []  [S]
)
)
HandleAssignment(pr, HandleId(t2))

34
HSimple(
    root pl pr
root: [S] [L2] [R1L1]
pl: [] [S] []
pr: [] [] [S]
)

---------------------------- END COND ---------------------------

---------------------------- END COND ---------------------------

------- results of iteration #1 -------

IN:
    root pl pr
root: [S] [L1] [R1]
pl: [] [S] []
pr: [] [] [S]

OUT:
    root pl pr
root: [S] [L1D1] [R1D1]
pl: [] [S] []
pr: [] [] [S]

------- results of iteration #2 -------

IN:
    root pl pr
root: [S] [L1D1] [R1D1]
pl: [] [S] []
pr: [] [] [S]

OUT:
    root pl pr
root: [S] [L1D2] [R1D2]
pl: [] [S] []
pr: [] [] [S]

IN merge OUT => next(IN):
    root pl pr
root: [S] [L1D+] [R1D+]
pl: [] [S] []

35
pr:   []   []   [S]

--------- results of iteration #3 ---------

IN:
  root pl   pr
root:  [S] [L1D+] [R1D+]
p1:   []   [S]   []
pr:   []   []   [S]

OUT:
  root pl   pr
root:  [S] [L1D1D+] [R1D1D+]
p1:   []   [S]   []
pr:   []   []   [S]

IN merge OUT => next(IN):
  root pl   pr
root:  [S] [L1D+] [R1D+]
p1:   []   [S]   []
pr:   []   []   [S]

================ END WHILE =================

================ COND (THEN PART) ===============

HandleAssignment(r1, HandleField(root, left))
HWhile(HSimple(
    root r1
root:  [S] [L1]
r1:    []   [S]
),
HSimple(
    root r1
root:  [S] [L1]
r1:    []   [S]
))
HandleAssignment(rr, HandleField(root, right))
HWhile(HSimple(
    root r1   rr
root:  [S] [L1] [R1]

36
rl:  []  [S]  []
rr:  []  []  [S]

HSimple(
  root rl rr
  root:  [S]  [L1]  [R1]
  rl:  []  [S]  []
  rr:  []  []  [S]
)

================ (ELSE PART) =================

================ END COND =================
References


