Interprocedural Shape Analysis for Recursive Programs

Noam Rinetzky*  Mooly Sagiv†

Abstract

A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The results can be used to optimize, understand, debug or verify programs. Existing algorithms are rather imprecise in the presence of recursive procedures. This is unfortunate, since recursion provides a natural way to manipulate linked data structures.

We present a novel shape analysis algorithm. The algorithm analyzes programs which manipulate linked lists by recursive procedures. Our algorithm is significantly more precise than existing algorithms for this type of programs. For example, it can verify the absence of memory leaks in many such programs. This is beyond the scope of existing algorithms. The algorithm is implemented. Preliminary experimental results are reported.

1 Introduction

A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The analysis algorithm is conservative, i.e., the determined information is true for every input but the algorithm may fail to identify some information. The information can be used to understand, verify, optimize [GH98] or parallelize [LHS8, Hen90, AW93, PCK93, Zap99] programs. For example, it is utilized to compile-time check the absence (or warn against the possible existence) of certain types of memory management errors such as memory leakage or dereference of null pointers. [DRS98, DRS00].

This paper addresses the problem of shape analysis in the presence of recursive procedures. This problem is important since recursion provides a natural way to manipulate linked data structures. We present a novel interprocedural shape analysis algorithm for programs manipulating linked lists. Our algorithm analyzes recursive procedures more precisely than any existing algorithm we know of. For example, it is able to verify that all the recursive list manipulating procedures of a small library we experimented with always return a list and never create memory leaks (see Section 5). In fact, not only that our algorithm can verify that correct programs indeed does not produce errors, it also finds interesting bugs in incorrect programs. For instance, it correctly finds that the recursive rev procedure shown in Figure 2, which reverses a list (declared in Figure 1) returns a cyclic linked list and does not create memory leaks. Furthermore, if an error is introduced by removing the statement x->n = NULL, the resultant program creates a cyclic list which leads to an infinite loop on some inputs. Interestingly, our analysis locates this error. Such a precise analysis of the rev procedure is quite a difficult task since (i) rev is recursive, thus there is no bound on the number of activation records that can be created when its executes, (ii) the global store is updated destructively in each invocation, and (iii) this procedure is not tail recursive: It sets the value of the local variable x before the recursive call and uses it as an argument to app after this call ends. No other shape analysis algorithm is capable of producing results with such a high level of precision for programs which invoke this, or similar, procedures.

Shape analysis algorithm, like any other static program analysis algorithm, is forced to represent execution states of potentially unbounded size in a bounded way. This process, often called summarization, naturally entails a loss of information. In the case of interprocedural analyses, it is also necessary to summarize all incarnations of recursive procedures in a bounded way.

*Computer Science Department, Technion, Technion City, Haifa 32000, Israel, Tel: +972-3-6407606, Fax: +972-3-6407601, maen@cs.technion.ac.il
†Computer Sciences Department, School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Tel: +972-3-6407606, Fax: +972-3-6407601, msagiv@acm.org
Shape analysis algorithms can analyze linked lists in a fairly precise way, e.g., see [SRW99]. For an inter- procedural analysis, we therefore follow the approach suggested in [JMS2, Deu90] by summarizing activation records in essentially the same way linked list elements are summarized. By itself, it does not suffice to analyze the values of the local variables after a call in a precise or efficient way: the abstract execution of a procedure call forces the analysis to summarize. The execution of the corresponding return has the problem of recovering the information lost at the call. Due to lack of information beside the potential values of the local variables, the analysis must make overly conservative assumptions. For example, in the rev procedure, if the analysis is not aware of the fact that no list element is pointed to by more than one instance of the variable x, it may fail to verify that rev returns an acyclic list (see Example 4.3).

A leading concept in our algorithm is the identification of certain important global properties of the heap elements pointed to by a local (stack allocated) pointer variable across successive recursive calls. This allows the analysis to handle return statements rather precisely. For example, in the rev procedure shown in Figure 2, our analysis determines that the list element pointed to by x is different from all the list elements reachable from t just before the app procedure is invoked and thus app must return an acyclic linked list. Proving that no memory leaks occur is achieved by determining that if an element of the list (being reversed) is not reachable from t at l1, then it is pointed to by at least one instance of x.

A question that comes to mind is how our analysis determines such global properties in the absence of a specification. Fortunately, we found out that a small set of a so-called predefined library of “local” properties of the stack variables in the analyzed program, can be used to determine many global properties. Furthermore, our analysis does not assume that a local property holds for the analyzed program. Instead, the analysis determines the stack variables which have this property. Of course it can benefit from specification, e.g., [HHN92], which will allow us to look for the special global properties of the specified program.

For example, the property $sh_\prec(v)$ holds for a list element $v$ which is pointed to by two or more invisible instances of the parameter x from previous activation records. When $sh_\prec(v)$ does not hold for any list element, it is assured that no list element is pointed to by more than one instance of the variable x. Therefore, in the rev procedure, the analysis can verify that rev returns an acyclic list.

This simple local property suffices to show that the procedure rev returns a list. Interestingly, this property

Figure 1: A type declaration for singly linked lists.

```c
/* list.h */
typedef struct node {
    int d;
    struct node *n;
} *L;
```

Figure 2: A recursive procedure which reverses a list in two stages: reverse the tail of the original list and store the result in t; then append the original first element at the end of the list pointed to by t. The code for procedure app is given in Appendix A. We also analyzed this procedure with a recursive version of append (see Section 5.)

```c
/* rev.c */
#include "list.h"
L rev(L x)
{
    L xn, t;
    if (x == NULL) return NULL;
    xn = x->n;
    x->n = NULL;
    l1: t = rev(xn);
    return app(t, x);
    l2:

Figure 3: The main procedure creates a list and then reverses it. The code for procedure create is given in Appendix A.

```c
/* main.c */
#include "list.h"
void main()
{
    L hd, z;
    hd = create(8);
    l3: z = rev(hd);
}
also sheds some light on the importance of tracking the sharing properties of stack variables. Existing shape analysis algorithms [JMS81, CWZ90, SRW98, SRW99] only record sharing properties of the heap assuming that the number of stack variables is fixed. However, in the presence of recursive calls different incarnations of a local variable may point to the same heap cell.

The distinction between invisible instances of variables based on their local properties sets the difference in terms of precision between our method and the existing ones mentioned in [LH88, Hen90, LH88, CW99, AW93, GH96, SRW98]. In Section 4 we also present properties that capture relationships between the stack and the heap. In many cases, this distinction also leads to a more efficient analysis. Technically, these properties and the analysis algorithm itself are explained (and implemented) using the 3-valued logic framework developed in [SRW99, SRW98, LAS00]. While our algorithm can be utilized in an independent way, this framework provides a sound theoretical foundation for our ideas and immediately leads to the prototype implementation described in Section 5. Therefore, Section 3 recalls some basic introduction to 3-valued logic.

We also employed the basic ideas underlying our algorithm to develop a cheaper pointer analysis problem: the problem of determining if a stack variable may point to another stack variable. We were able to prove some global properties beyond the ones shown in [EGH94] e.g., a pointer variable must always point to another variable in the same activation record. The application is not discussed further in this submission for reasons of space.

2 Calling conventions

In this section, we define our calling conventions. These conventions are somewhat arbitrary and are only used to establish a common terminology.

Without loss of generality, we assume that all local variables have unique names. Every invoked procedure has an activation record in which its local variables and parameters are stored. An invocation of procedure \( f \) at a call-site label is performed in several steps: (i) store values of actual parameters and label in some designated global variables; (ii) at the entry-point of \( f \) create a new activation record at the top of the stack and copy values of parameters and label into that record; (iii) continue to execute statements in \( f \) until a return statement occurs or \( f \)'s exit-point is reached (We assume that a return statement stores the return value in a designated global variable and transfers the control to \( f \)'s exit-point); (iv) at \( f \)'s exit-point, pop the stack and transfers control back to the matching return-site of label; (v) at the return-site, copy the return value if needed and continue execution.

The activation record at the top of the stack is referred to as the current activation record. Local variables and parameters stored in current activation record and global variables are called visible; local variables and parameters stored in other activation records are invisible.

2.1 The running example

The C program whose main procedure shown in Figure 3 invokes \( \text{rev} \) on a list with eight elements. This program is used throughout the paper as a running example. In procedure \( \text{rev} \), label \( l_1 \) plays the role of the recursive call site, \( l_0 \) that of \( \text{rev} \) entry point, and \( l_2 \) of \( \text{rev} \) exit point.

3 The use of 3-valued logic for program analysis

The algorithm is explained (and implemented) using the 3-valued logic framework developed in [SRW99, SRW98]. In this section, we summarize their framework which shows how 3-valued logic can serve as the basis for program analysis.

3.1 Representing memory states via 2-valued logical structures

A 2-valued logical structure \( S \) is comprised of a set of individuals (nodes) called a universe, denoted by \( U^S \), and an interpretation over that universe for a set of predicate symbols denoted core predicates. The interpretation of a predicate symbol \( p \) in \( S \) is denoted by \( p^S \). For every predicate \( p \) of arity \( k \), \( p^S \) is a function \( (U^S)^k \to \{0, 1\} \).

In this paper, 2-valued logical structures represent memory states. An individual corresponds to a memory element: either a heap cell (a list element) or an activation record. The core predicates describe atomic properties of the program memory state. The properties of each memory element are described by unary core predicates. The relations that hold between two memory elements are described by binary core predicates. The core predicates intended meaning is given in Table 1. This representation intentionally ignores the specific values of variables and locations, and keeps record of certain relationships that hold among variables and memory elements:

- Every individual node \( v \) is either a heap cell having \( \text{heap}^S(v) = 1 \) or an activation record having \( \text{stack}^S(v) = 1 \).
- The unary predicate \( cs_{\text{label}} \) indicates the call-site in which a function is invoked. Its similarities with the call-strings of [SP81] are discussed in Section 6.
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$heap(v)$</td>
<td>$v$ is a heap element</td>
</tr>
<tr>
<td>$stack(v)$</td>
<td>$v$ is an activation record</td>
</tr>
<tr>
<td>$c_{label}(v)$</td>
<td>label is the return-address of the function whose activation record is $v$</td>
</tr>
<tr>
<td>$g(v)$</td>
<td>The list element $v$ is pointed to by a global variable g</td>
</tr>
<tr>
<td>$n(v_1,v_2)$</td>
<td>The n-component of list element $v_1$ points to the list element $v_2$</td>
</tr>
<tr>
<td>$top(v)$</td>
<td>$v$ is the current activation record</td>
</tr>
<tr>
<td>$pr(v_1,v_2)$</td>
<td>The activation record $v_2$ is previous to the activation record $v_1$ in the stack</td>
</tr>
<tr>
<td>$x(v_1,v_2)$</td>
<td>The local (parameter) variable $x$ which is stored in the activation record $v_1$ points to the list element $v_2$</td>
</tr>
</tbody>
</table>

Table 1: The core predicates used in this paper. There is a separate predicate $g$ for every global program variable $g$, $x$ for every local one and parameter $x$, $c_{label}$ for every label immediately following a function call.

- The unary predicate $top$ is true for the current activation record.
- The binary relation $n$ captures the $n$-successor relation between list elements.
- The binary relation $pr$ connects an activation record to the one of the caller.
- For a local variable or parameter named $x$, the binary relation $x$ captures its value in a specific activation record.

2-valued logical structures are depicted as directed graphs. A directed edge between nodes $u_1$ and $u_2$ that is labeled with binary predicate symbol $p$ indicates that $p^S(u_1,u_2) = 1$. Also, for a unary predicate symbol $p$, we draw $p$ inside a node $u$ when $p^S(u) = 1$; conversely, when $p^S(u) = 0$ we do not draw $p$ in $u$. For clarity, we treat the unary predicates $heap$ and $stack$ in a special way and draw nodes $u$ having $heap^S(u) = 1$ as ellipses to indicate heap elements and nodes having $stack^S(u) = 1$ as rectangles to indicate stack elements.$^1$

**Example 3.1** The 2-valued structure $S_3$ shown in Figure 4 corresponds to the memory state at program point $l_3$ in the $rev$ function upon exit of the fourth invocation of the $rev$ function in the running example. The five rectangular nodes correspond to the activation records of the five function invocations; to comply with the usual notations, the stack grows downwards. The current activation record (of $rev$) is drawn at the bottom with $top$ written inside. The three activation records (of $rev$) drawn above it correspond to pending invocations of $rev$.

The three isolated heap nodes on the left side of the figure correspond to the list elements pointed to by $x$ in pending invocations of $rev$. The chain of five heap nodes to the right correspond to the (reversed) part of the original list. The last element in the list corresponds to the list element appended by $app$ invoked just before $l_3$ in the current invocation of $rev$.

Notice that the $n$ predicate is the only one which is specific to the linked list structure declared in Figure 1.

### 3.2 Consistent 2-valued structures

Some 2-valued structures cannot represent memory states, e.g., when a unary predicate $g$ holds at two different nodes for a global variable $g$. A 2-valued structure is consistent if it can represent a memory state. It turns out that the analysis can be more precise by eliminating inconsistent 2-valued structures. Therefore, in Section 4.3 we describe a constructive method to check if a 2-valued structure is inconsistent and thus can be discarded by the analysis.

### 3.3 Kleene's 3-valued logic

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of $1/2$ (unknown) for cases in which predicates could have either value, i.e., 1 (true) or 0 (false). Kleene's interpretation of the propositional operators is given in Table 2. We say that the values 0 and 1 are definite values and that $1/2$ is an indefinite value.

### 3.4 Conservative representation of sets of memory states via 3-valued structures

Like 2-valued structures, a 3-valued logical structure $S$ is also comprised of a universe $U^S$, and an interpretation of the predicate symbols. However, for every predicate

<table>
<thead>
<tr>
<th>$\land$</th>
<th>0</th>
<th>0</th>
<th>1/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\lor$</td>
<td>0</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Table 2: Kleene’s 3-valued interpretation of the propositional operators.
Figure 4: The 2-valued structure $S_4$ which corresponds to the program state at $l_2$ in the \texttt{rev} function upon exit of the fourth recursive invocation of the \texttt{rev} function.

$\phi$ of arity $k$, $\phi^S$ is a function $\phi^S: (U^S)^k \rightarrow \{0, 1, 1/2\}$, where $1/2$ explicitly captures unknown predicate values.

3-valued logical structures are also drawn as directed graphs. Definite values are drawn as in the 2-valued structures. Binary indefinite (1/2) predicate values are drawn as dotted directed edges. Also, we draw $p = 1/2$ inside a node $u$ when $\phi^S(u) = 1/2$.

Let $S^1$ be a 2-valued structure, $S$ be a 3-valued structure, and $f: U^{S^1} \rightarrow U^S$ such that $f$ is surjective. We say that $f$ embeds $S^1$ into $S$ if for every predicate $\phi$ of arity $k$ and $u_1, u_2, \ldots, u_k \in U^{S^1}$, either $\phi^S(u_1, u_2, \ldots, u_k) = \phi^S(f(u_1), f(u_2), \ldots, f(u_k))$ or $\phi^S(f(u_1), f(u_2), \ldots, f(u_k)) = 1/2$. We say that $S$ conservatively represents all the 2-valued structures that can be embedded into it by some function $f$. Thus, $S$ can compactly represent many structures.

Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called \textit{summary nodes}. We use a designated unary predicate $s\phi$ to maintain summary-node information. A summary node $w$ has $s\phi^S(w) = 1/2$, indicating that it may represent more than one node from 2-valued structures. These nodes are depicted graphically as dotted ellipses or rectangles. In contrast, if $s\phi^S(w) = 0$, then $w$ is known to represent a unique node. Only nodes with $s\phi^S(w) = 1/2$ can have more than one node mapped to them by the embedding function.

**Example 3.2** The 3-valued structure $S_5$ shown in Figure 5 represents the 2-valued structure $S_4$ shown in Figure 4. The dotted ellipsis summary node represents all the eight list elements. The indefiniteness of the n-self edge results from the fact that there is a n-component pointer between each two successors and no n-component pointer between non-successors.

The dotted rectangle summary node represents the activation records from the second and third invocation of \texttt{rev}. The unary predicates drawn inside it indicate that it (only) represents activation records of \texttt{rev} which return into $l_1$ (i.e., recursive calls). The dotted x-edge from this summary node indicates that an invisible instance of $x$ from the second or the third call may or may not point to one of the list elements. The rectangle at the top of Figure 5 represents the activation record at the top of Figure 4 which is an invocation of \texttt{main}. The second rectangle from the top in $S_5$ represents the second rectangle from the top in $S_4$ which is an invocation of \texttt{rev} from \texttt{main} (indicated by drawing $c_{s_4}$ inside this node). The bottom rectangle in $S_5$ represents the bottom rectangle in $S_4$, which is the current activation record (indicated by drawing \texttt{top} inside this node). All other activation records are known not to be the current activation record (i.e., the top predicate does not hold for these nodes) since \texttt{top} is not drawn in either of them.

Figure 5: The 3-valued structure $S_5$ which represents the 2-valued structure shown in Figure 4.
3.5 Expressing properties via formulae

Properties of structures can be extracted by evaluating formulae. We use first-order logic with transitive closure and equality, but without function symbols and constant symbols. For example, the formula

\[ \exists v_1, v_2 : \neg top(v_1) \land \neg top(v_2) \land v_1 \neq v_2 \land x(v_1, v) \land x(v_2, v) \]  

expresses the fact that there are two different invisible instances of the parameter variable \( x \) pointing to the same list element \( v \).

The Embedding Theorem (see [SRW99, Theorem 3.7]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all of the 2-valued structures embedded into that structure. The Embedding Theorem is the foundation for the use of 3-valued logic in static-analysis: It ensures that it is sensible to take a formula that—when interpreted in 2-valued logic—defines a property, and reinterprets it on a 3-valued structure \( S \) to get a solution which is conservative w.r.t. evaluating the formula in all the 2-valued structures represented by \( S \).

Example 3.3 Consider the 2-valued structure \( S_3 \) shown in Figure 4. The formula (1) evaluates to 0 at all the list nodes.

In contrast, consider the 3-valued structure \( S_5 \) shown in Figure 5. This formula (1) evaluates to 1/2 at the dotted ellipsis summary heap node. This is in line with the Embedding Theorem since 1/2 is less precise than 0. However, it is not very precise since the fact that different invisible instances of \( x \) are never aliased is lost.

4 A technique for interprocedural shape analysis

In this section, we describe our shape analysis algorithm for recursive programs manipulating linked lists. The algorithm iteratively annotates each program point with a set of 3-valued logical structures in a conservative manner, i.e., when it terminates, every 2-valued structure that can arise at a program point is represented by one of the 3-valued structures computed at this point. However, it may also conservatively include spurious 3-valued structures.

Section 4.1 describes the properties of heap elements and local variables which are observed by the algorithm. For ease of understanding, in Section 4.2, we give a high level description of the iterative analysis algorithm. The actual algorithm is presented in Section 4.3.

4.1 Observing important properties in order to improve the Analysis

In order to overcome the kind of imprecision described in Example 3.3, we now introduce instrumentation predicates. These predicates are stored in the structure just like core predicates. The values of these predicates are derived from the core predicates. Thus, every instrumentation predicate has a formula over the set of core predicates which defines its meaning. The instrumentation predicates that our interprocedural algorithm utilizes are described in Table 3 together with their informal meaning and their defining formulae (other interprocedural instrumentation predicates are defined in [SRW99]).

The instrumentation predicates are divided into four classes, separated by two horizontal lines in Table 3: (i) Properties of heap elements with regard to visible variables, i.e., \( x \) and \( r_{n,x} \). These are the ones originally used in [SRW99]. (ii) Properties of heap elements with regard to invisible variables. These are \( x \) and \( r_{n,x} \) which are variants of \( x \) and \( r_{n,x} \) from the first class but for invisible variables. The \( sh_{\varphi}(v) \) predicate is motivated in Example 3.3. It is similar to the heap sharing predicate used in [JMS81, CWZ90, SRW98, SRW99]. (iii) Generic properties of an individual activation record. For example, \( m_{i,j}^{x}(u) = 1 \) (for \( m \) for not NULL) in a 2-valued structure \( S \) indicates that the invisible instance of \( x \) which is stored in the activation record \( u \) points to some list element. (iv) Properties across successive recursive calls. For example, the predicate \( al_{x,y}^{pr}[y] \) captures aliasing between \( x \) at the callee and \( y \) at the caller. The other properties are similar but also involve the \( n \) component.

Example 4.1 The 3-valued structure \( S_6 \) shown in Figure 6 also represents the 2-valued structure \( S_4 \) shown in Figure 4. In contrast with \( S_4 \) shown in Figure 5 where all the eight list elements are represented by one heap node, in \( S_6 \), they are represented by six heap nodes. The leftmost heap node in \( S_4 \) represents the leftmost list element in \( S_4 \) (which was originally the list element). The fact that \( x \) is drawn inside this node indicates that it represents a list element pointed to by the invisible instance of \( x \). This fact can also be extracted from \( S_4 \) by evaluating the \( x \) defining formula at this node, but this is not always the case, as we show immediately. The second leftmost heap node is a summary node that represents \( S_4 \)'s both second and third list elements from the left. There is an indefinite \( x \)-edge into this summary node. Still, since \( x \) is drawn inside it, every list element it represents must be pointed to by at least one invisible instance of \( x \). Therefore, the analysis can determine that this node does not represent a possible memory leakage.

The other summary heap node (second heap node from the right) represents the second and third (from
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Intended Meaning</th>
<th>Defining Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(v)$</td>
<td>Is the list element $v$ pointed to by the visible instance of $x$?</td>
<td>$\exists v_1 : top(v_1) \land x(v_1, v)$</td>
</tr>
<tr>
<td>$r_{n,x}(v)$</td>
<td>Is the list element $v$ reachable by $n$-component from the visible instance of $x$?</td>
<td>$\exists v_1, v_2 : top(v_1) \land x(v_1, v_2) \land n^*(v_2, v)$</td>
</tr>
<tr>
<td>$\hat{x}(v)$</td>
<td>Is the list element $v$ pointed to by an invisible instance of $x$?</td>
<td>$\exists v_1 : \neg top(v_1) \land x(v_1, v)$</td>
</tr>
<tr>
<td>$r_{n,\hat{x}}(v)$</td>
<td>Is the list element $v$ reachable by $n$-component from an invisible instance of $x$?</td>
<td>$\exists v_1, v_2 : \neg top(v_1) \land x(v_1, v_2) \land n^*(v_2, v)$</td>
</tr>
<tr>
<td>$sh_{\hat{x}}(v)$</td>
<td>Is the list element $v$ pointed to by more than one invisible instance of $x$?</td>
<td>$\exists v_1, v_2 : \neg top(v_1) \land \neg top(v_2) \land v_1 \neq v_2 \land x(v_1, v) \land x(v_2, v)$</td>
</tr>
<tr>
<td>$nn_{\hat{x}}(v)$</td>
<td>Does the invisible instance of $x$ stored in the activation record $v$ point to a list element?</td>
<td>$\exists v_1 : \neg top(v) \land x(v, v_1)$</td>
</tr>
<tr>
<td>$a_{x,y}(v)$</td>
<td>Are the invisible instances of $x$ and $y$ stored in the activation record $v$ aliased?</td>
<td>$\exists v_1 : \neg top(v) \land x(v, v_1) \land y(v, v_1)$</td>
</tr>
<tr>
<td>$a_{x,pr}<a href="v">y</a>$</td>
<td>Is the instance of $x$ stored in the activation record $v$ aliased with the instance of $y$ stored in $v$'s previous activation record?</td>
<td>$\exists v_1, v_2 : pr(v, v_1) \land x(v, v_2) \land y(v, v_1)$</td>
</tr>
<tr>
<td>$a_{x,pr}[y] \rightarrow n(v)$</td>
<td>Is the instance of $x$ stored in the activation record $v$ aliased with $y \rightarrow n$ for the instance of $y$ stored in $v$'s previous activation record?</td>
<td>$\exists v_1, v_2, v_3 : x(v, v_1) \land pr(v, v_2) \land y(v, v_3) \land n(v_3, v_1)$</td>
</tr>
<tr>
<td>$a_{x \rightarrow n, pr}<a href="v">y</a>$</td>
<td>Is $x \rightarrow n$ for the instance of $x$ stored in the activation record $v$ aliased with the instance of $y$ stored in $v$'s previous activation record?</td>
<td>$\exists v_1, v_2, v_3 : x(v, v_2) \land n(v_2, v_1) \land pr(v, v_3) \land y(v, v_1)$</td>
</tr>
</tbody>
</table>

Table 3: The instrumentation predicates used for the interprocedural analysis. Here $x$ and $y$ are generic names for local variables and parameters $x$ and $y$ of an analyzed function. The $n^*$ notation used in the defining formula for $r_{n,x}(v)$ denotes the reflexive transitive closure of $n$.

Note that the predicate $sh_{\hat{x}}$ does not hold for any heap node in $S_0$. Therefore, no list element in any 2-valued structure $S_0$ represents is pointed to by more than one invisible instance of the variable $x$.

The stack elements are represented in the same way they are represented by $S_0$. Since $a_{x,pr}[en]$ is drawn inside the two stack nodes at the bottom, for every activation record $v$ they represent, the instance of $x$ stored in $v$ is aliased with the instance of $en$ stored in the activation record preceding $v$.

4.2 The best iterative algorithm

This section provides a high level description of the algorithm in terms of the general abstract interpretation framework [CC79]. Intuitively speaking, the most precise (also called best) conservative effect of a program statement on a 3-valued logical structure $S$ is defined in three stages shown in Figure 7: (i) finding each consistent 2-valued structure $S'$ represented by $S$ (concretization); (ii) applying the C operational semantics to every such structure $S'$ resulting in a 2-valued structure $S''$ and (iii) finally abstracting each of the 2-valued structures $S''$ by a 3-valued structure of bounded size (abstraction). Thus, the result of the statement is a set of 3-valued structures of bounded size.

4.2.1 The abstraction principle

The abstraction function is defined by a subset of the unary predicates, which are called abstraction properties in [SR99]. The abstraction of a 2-valued structure is defined by mapping all the nodes which have the same values for the abstraction properties into the same abstract node. Thus, the values of abstraction predicates remain the same in the abstracted 3-valued structures. The values of any other predicate $p$ in the abstracted 3-valued structure is determined conservatively to yield...
Figure 6: The 3-valued structure $S_3$ with instrumentation predicates which represents the 2-valued structure shown in Figure 4. For brevity, we do not show $r_{n,x}$ for nodes having the property $x$.

Figure 7: The best abstract semantics of a statement $st$ with respect to 3-valued structures. $[st]$ is the operational semantics of $st$ applied pointwise to every consistent 2-valued structure.

an indefinite value whenever corresponding values of $p$ in the represented concrete 2-valued structure disagree.

**Example 4.2** The structure $S_3$ shown in Figure 5 is an abstraction of $S_4$ shown in Figure 4 for the abstraction properties of $S_4$ being unary core predicates. For example, the activation records of the 2nd and 3rd recursive call to $rev$ are both mapped into the summary stack node since they are both invisible activation records of invocations of $rev$ from the same call-site (i.e., $top$ does not hold for these activation records, but $cs_{i_1}$ does). Also, all the eight heap nodes are mapped to the same summary node having only the heap core predicate hold at them. The $pr$ edge into stack node at the top of the figure is definite since there is only one node abstracted into each of its ends. In contrast, the $hd$ edge emanating from the topmost stack node must be indefinite in order for $S_4$ to conservatively represent $S_4$; in $S_4$, the $hd$ predicate holds at for topmost stack node and the leftmost heap node, but it does not hold for any other heap node and all heap nodes of $S_4$ are summarized into one summary heap node.

The structure $S_3$ shown in Figure 6 is an abstraction of $S_4$ shown in Figure 4 for the abstraction properties of $S_4$ being unary core and instrumentation predicates.

Note that since the set of unary predicates is fixed, there can only be a constant number of nodes in the abstracted structure which guarantees that the analysis always terminates. Also, notice that nodes with different observed properties lead to different instrumentation predicate values and thus are never represented by the same abstract node.

**4.2.2 Analyzing return statement**

Analyzing the return statement is the core of the analysis. Our technique is capable of analyzing it quite precisely, by using the instrumentation predicates. This is demonstrated in the following example.

**Example 4.3** Let us exemplify the application of the return statement to the 3-valued structure $S_3$ shown in Figure 6. following the stages of the best iterative algorithm described in Section 4.2.

**Stage 1-Concretization**

Let $S^1$ be one of the consistent 2-valued structures represented by $S_3$. Let $k \geq 1$ be the number of activation records represented by the summary stack node in $S_3$. Since $S^1$ is a consistent 2-valued structure, the $x$ parameter variable in each of these $k$ activation records must points to one of the isolated list elements represented by the left summary heap node. This can be
inferred by the following series of observations: the fact that the \( x \) variable in each of these activation records points to a list element is indicated by having \( m_X \)-drawn inside the stack summary node. The list elements pointed to by these variables must be represented by the left summary heap node since only one \( x \) edge emanates from the summary stack node, and this edge enters the left summary heap node.

Let \( m \geq 1 \) be the number of list elements represented by the left summary heap node. Since \( \hat{x} \) is drawn inside this node, each of the \( m \) list elements it represents must be pointed to by at least one invisible instance of \( x \). Thus, \( m \leq k \). However since \( s_{\hat{x}} \) is not drawn inside this summary node, none of the \( m \) list elements it represents is pointed to by more than one invisible instance of \( x \). Thus, we conclude that \( m = k \).

Using the fact that \( a_{x,pr[\text{en}]} \) is drawn inside the two stack nodes at the bottom of Figure 6 we conclude that the instance of \( x \) of each recursive invocation of \( \text{rev} \) is aliased with the instance of \( x \) of \( \text{rev} \) previous invocation. Thus, \( S^1 \) looks essentially like the structure shown in Figure 4, but with \( k \) isolated list elements not pointed to by \( \mathcal{X} \), rather than two, and with the length of the list pointed to by \( \mathcal{X} \) being unknown.

**Stage II---Applying the operational semantics:**

Applying the operational semantics of \( \text{return} \) to \( S^1 \) (see Section 2), results in a (consistent) 2-valued structure \( S_1 \). Note that the list element pointed to by the visible instance of \( x \) in \( S^1 \) is not pointed to by any other instance of \( x \), and it is not part of the reversed suffix. Thus \( S^1 \) differs from \( S \) by having the top activation record of \( S \) removed from the stack and the activation record preceding it becoming the new current activation record.

**Stage III---abstraction:**

Abstracting \( S^1 \) into a 3-valued structure may result, depending on \( k \), in one of three possible structures. If \( k > 2 \) then the resulting structure is very similar to \( S_0 \) since the information regarding the number of the remaining isolated list elements and invisible activation record is lost in the summarization. For \( k = 2 \) and \( k = 1 \) we get a consistent 2-valued structure with four and three activation records, respectively. Abstracting these structures, results in no summary stack nodes since the call-site of each non current activation record is different. For \( k = 1 \) only one isolated list elements remains, thus it is not summarized. For \( k = 2 \) the two remaining isolated heap nodes are not merged since different local variables. For example one of them is pointed to by \( \mathcal{X} \) and the other one is not.

Notice that if no instrumentation predicates correlating invisible variables and heap nodes are maintained, a conservative analysis cannot deduce that the list element pointed to by the visible instance of \( x \) in \( S^1 \) is not pointed to by another instance of this variable. Thus, the analysis must conservatively assume that future calls to append may create cycles. However, even when \( a_{x,pr[\text{en}]} \) is not maintained the analysis can still produce fairly accurate results using only the \( s_{\hat{x}} \) and \( \hat{x} \) instrumentation predicates.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Predicate-update formulae</th>
</tr>
</thead>
</table>
| \( \text{label: call f()} \) | \( \begin{align*}
    \text{stack}(v) &= \text{stack}(v) \lor \text{new}(v) \\
    \text{cs}_{\text{label}}(v) &= \text{cs}_{\text{label}}(v) \lor \text{new}(v) \\
    \text{top}(v) &= \text{new}(v) \\
    \text{pp}(v_1, v_2) &= \text{pp}(v_1, v_2) \lor \text{new}(v_1) \land \text{top}(v_1) \\
    \text{pp}^*(v_1, v_2) &= \text{pp}^*(v_1, v_2) \lor \text{new}(v_2) \\
    \text{return} &= \text{stack}(v) \land \text{top}(v) \\
    \text{cs}_i(v) &= \text{cs}_i(v) \land \neg \text{top}(v) \\
    \text{pp}(v) &= 3 \exists v_1 : \text{top}(v_1) \land \text{pp}(v_1, v) \\
    \text{pp}^*(v_1, v_2) &= \text{pp}^*(v_1, v_2) \land \neg \text{top}(v_1) \\
    \text{pp}^*(v_1, v_2) &= x(v_1, v_2) \land \neg \text{top}(v_1)
\end{align*} \) |

Table 4: The predicate-update formulae defining the operational semantics of the call and return statements for the core predicates. The value of each core predicate \( p \) after the statement execution, denoted by \( p' \), is defined in terms of the core predicate values before the statement execution (denoted as unprimed). Core predicates which are not specified are assumed to be unchanged, i.e., \( p'(v_1, \ldots) = p(v_1, \ldots) \). The predicate \( \text{new}(v) \), used in the update formula of the \( \text{cs}_{\text{label}}(v) \) predicate, holds only for the newly allocated activation record.

4.3 Our iterative algorithm

Unlike the best algorithm which explicitly applies the operational semantics to each of the (potentially infinite) structures represented by a three-valued structure \( S \), our algorithm explicitly operates on \( S \) yielding a set of 3-valued structures \( S' \). By employing a set of judgements, similar in spirit to the ones described Example 4.3 our algorithm produces a set which conservatively represents all the structures that could arise after applying the \( \text{return} \) statement to each consistent 2-valued structure \( S \) represents. However it may be overly conservative; \( S' \) may represent more 2-valued structures than those represented by applying the best algorithm. Our current experience reported in Section 5 indicates that it usually gives good results.

Technically, Our algorithm computes the resulting 3-valued structure \( S' \) by evaluating formulae in 3-valued logic. When interpreted in 2-valued logic these formulae
define the operational semantics. Thus, the Embedding Theorem guarantees that the results are conservative w.r.t. the best algorithm. The update formulae for the core-predicates describing the operational semantics for call and return statements are given in Table 4.

Instead of calculating the instrumentation predicate values at the resulting structure by their defining formulae, which may be overly conservative, predicate-update formulae for instrumentation predicates are used. The formulae is omitted here for reasons of space. The reader is referred to [SRW99] for many examples of predicate-update formulae for instrumentation predicates and other operations used by 3-valued logic framework to increase the precision employed by our algorithm.

5 A Prototype implementation

A prototype of the iterative algorithm sketched in Section 4.3. was implemented for a small subset of C. The main goal is to determine if the results of the analysis are useful before scaling the algorithm to handle arbitrary C programs. In theory, the algorithm might be overly conservative and yield many indefinite values. This may lead to many “false alarms”. For example, the algorithm may report that every program point possibly contain a memory leak or a dereference to NULL pointer. Fortunately, in Section 5.1 we show that this is not the case for the C functions analyzed.

The algorithm was implemented using a 3-valued logic analysis system called TVLA [LAS00] (for Three-Valued-Logic Analyzer). The inputs of the system are the function's control flow graph and logical formulae describing the operational semantics of each statement and condition. The system iteratively computes a set of three-valued structures at every program point. It is quite powerful but slow and only supports intraprocedural analysis specified using low level logical formulae. Therefore, we implemented in Java, a frontend that generates a TVLA input from a program in a subset of C. The instrumentation predicates described in Section 4.1 allow our frontend to treat call and return statements in the same way intraprocedural statements are handled, without sacrificing precision in many recursive functions. Our frontend also performs certain minimal optimizations not described here for reasons of space.

5.1 Empirical results

The analyzed C programs together with the running space and running-time on a Pentium II 233 Mhz machine running Windows 95 with JDK 1.2 are listed in Table 5. The analysis verified that in deed these func-

<table>
<thead>
<tr>
<th>Func.</th>
<th>Description</th>
<th># of Structs</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>creates a list.</td>
<td>219</td>
<td>7.31</td>
</tr>
<tr>
<td>delete</td>
<td>frees the entire list</td>
<td>139</td>
<td>12.74</td>
</tr>
<tr>
<td>insert</td>
<td>creates and inserts an element into a sorted list</td>
<td>344</td>
<td>34.61</td>
</tr>
<tr>
<td>delete</td>
<td>deletes an element from a sorted list</td>
<td>423</td>
<td>38.20</td>
</tr>
<tr>
<td>search</td>
<td>searches an element in a sorted list</td>
<td>303</td>
<td>8.07</td>
</tr>
<tr>
<td>app</td>
<td>adds one list to the end of another</td>
<td>326</td>
<td>40.64</td>
</tr>
<tr>
<td>rev</td>
<td>the running example (non recursive append)</td>
<td>797</td>
<td>95.35</td>
</tr>
<tr>
<td>rev.x</td>
<td>the running example (with recursive append)</td>
<td>2285</td>
<td>1204.13</td>
</tr>
<tr>
<td>rev.d</td>
<td>reverses a list with destructive updates</td>
<td>414</td>
<td>47.56</td>
</tr>
</tbody>
</table>

Table 5: The total number of 3-valued structures that arise during analysis and running times for the recursive functions analyzed. The programs are available at “http://www.cs.technion.ac.il/~mao”.

6 Conclusions and related work

In the future, we plan to extend it to a larger subset of C, and to scale it up to programs of realistic size. One possible way involves first running a cheap and imprecise pointer-analysis algorithm, such as the flow-insensitive points-to analysis described in [Ste96], before proceeding to our quite precise but expensive analysis. Another possibility is to analyze a procedure once in all the calling contexts. The underlying generator that we used in our implementation yields rather expensive analyses which need to be improved.

We focused this research on linked lists but plan to investigate tree manipulation programs.

Finally, our analysis is limited by its fixed set of predefined “library” properties. This makes our tool easy to use since it is fully automatic and does not require
References


A Appendix

```c
/* create.c */
#include "list.h"
L create(int s)
{
    L tmp, tl;
    if (s <= 0) return NULL;
    tl = create(s-1);
    tmp = (L) malloc(sizeof(struct node));
    tmp->n = tl;
    tmp->d = s;
    return tmp;
}
```

Figure 8: A recursive function which creates a list.

```c
/* app.c */
#include "list.h"
L app(L p, L q)
{
    L r;
    if (p == NULL) return q;
    r = p;
    while(r->n != NULL)
        r = r->n;
    r->n = q;
    return p;
}
```

Figure 9: A non recursive function which appends the list pointed to by q at the end of the list pointed to by p.