Interprocedural Shape Analysis for Recursive Programs

Noam Rinetzky *[1] and Mooly Sagiv[2]

[1] Computer Science Department, Technion, Technion City, Haifa 32000, Israel
maon@cs.technion.ac.il
[2] Computer Sciences Department, Tel-Aviv University, Tel-Aviv 69978, Israel
msagiv@acm.org

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 quite imprecise in the presence of recursive procedure calls. This is unfortunate, since recursion provides a natural way to manipulate linked data structures.
We present a novel technique for shape analysis of recursive programs. An algorithm based on our technique has been implemented. It handles programs manipulating linked lists written in a subset of C. The algorithm is significantly more precise than existing algorithms. For example, it can verify the absence of memory leaks in many recursive programs; this is beyond the capabilities of existing algorithms.

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 discovered information is true for every input. The information can be used to understand, verify, optimize [6], or parallelize [1, 8, 12] programs. For example, it can be utilized to check at compile-time for the absence of certain types of memory management errors, such as memory leakage or dereference of null pointers [5].

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 existing algorithms. 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 Sect. 5). In fact, not only can our algorithm verify that correct programs do not produce errors, it can also find interesting bugs in incorrect programs. For instance, it correctly finds that the recursive procedure `rev` shown

* Partially supported by the Technion and the Israeli Academy of Science.

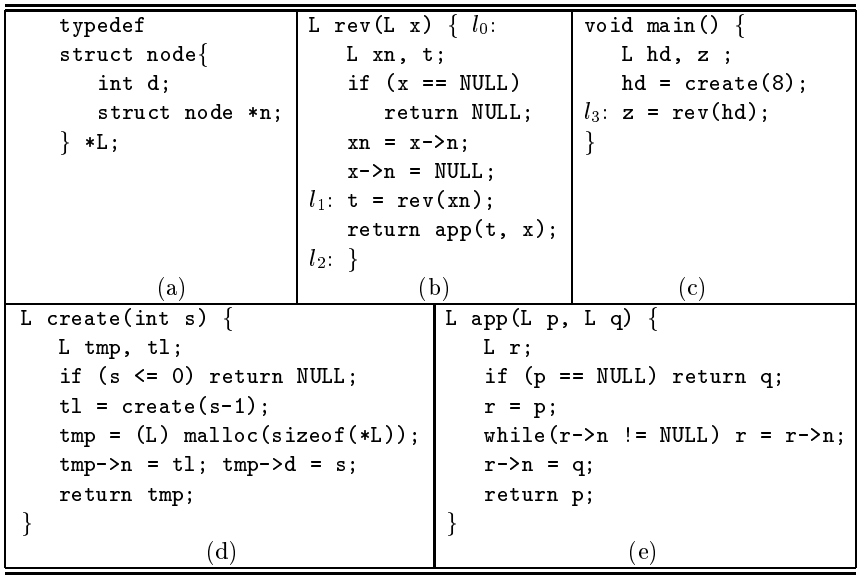

Fig. 1. (a) A type declaration for singly linked lists. (b) A recursive procedure which reverses a list in two stages: reverse the tail of the original list and store the result in `t`; then append the original first element at the end of the list pointed to by `t`. (c) The `main` procedure creates a list and then reverses it. We also analyzed this procedure with a recursive version of `app` (see Sect. 5.) (d) A recursive procedure that creates a list. (e) A non-recursive procedure that appends the list pointed to by `q` to the end of the list pointed to by `p`

in Fig. 1(b), which reverses a list (declared in Fig. 1(a)) returns an acyclic 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 procedure `rev` is quite a difficult task since (i) `rev` is recursive, and 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) the 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 the call ends. No other shape-analysis algorithm we know of is capable of producing results with such a high level of precision for programs that invoke this, or similar, procedures.

A 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.

Shape-analysis algorithms can analyze linked lists in a fairly precise way, e.g., see [15]. For an interprocedural analysis, we therefore follow the approach suggested in [4, 11] by summarizing activation records in essentially the same way linked list elements are summarized. By itself, this technique does not retain the precision we would like. The problem is with the (abstract) values obtained for local variables after a call. The abstract execution of a procedure call forces the analysis to summarize, and the execution of the corresponding return has the problem of recovering the information lost at the call. Due to the lack of enough information about 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 each list element is pointed to by no more than one instance of the variable `x`, it may fail to verify that `rev` returns an acyclic list (see Example 4.3).

An important concept in our algorithm is the identification of certain global properties of the heap elements pointed to by a local (stack-allocated) pointer variable. These properties describe potential and definite aliases between pointer access paths. This allows the analysis to handle return statements rather precisely. For example, in the `rev` procedure shown in Fig. 1(b), the 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, which can be used to conclude that `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 l_1, 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 that a small set 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 that have a given property. Of course, it can benefit from the presence of a specification, e.g., [9], which would allow us to look for the special global properties of the specified program.

For example, the property $sh^{\widehat{}}_{x}(v)$ holds for a list element v that is pointed to by two or more invisible instances of the parameter `x` from previous activation records. When $sh^{\widehat{}}_{x}(v)$ does not hold for any list element, we have a guarantee that no list element is pointed to by more than one instance of the variable `x`. This simple local property plays a vital rule in verifying that the procedure `rev` returns an acyclic list (see Example 4.3). Interestingly, this property also sheds some light on the importance of tracking the sharing properties of stack variables. Existing intraprocedural shape-analysis algorithms [2, 10, 14, 15] only record sharing properties of the heap since the number of variables is fixed in the intraprocedural setting. However, in the presence of recursive calls, different incarnations of a local variable may point to the same heap cell.

The ability to have distinctions between invisible instances of variables based on their local properties is the reason for the difference in precision between our method and the methods described in [1, 2, 7, 8, 12, 14]. In Sect. 4, we also exploit

properties that capture relationships between the stack and the heap. In many cases, the ability to have these distinctions 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 [13, 15]. While our algorithm can be presented in an independent way, this framework provides a sound theoretical foundation for our ideas and immediately leads to the prototype implementation described in Sect. 5. Therefore, Sect. 3 presents a basic introduction to the use of 3-valued logic for program analysis.

2 Calling Conventions

In this section, we define our assumption about the programming language calling conventions. These conventions are somewhat arbitrary; in principle, different ones could be used with little effect on the capabilities of the program analyzer. Our analysis is not effected by the value of non-pointer variables. Thus, we do not represent scalars (conservatively assuming that any value is possible, if necessary), and in the sequel, restrict our attention to pointer variables.

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 the 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) execute the 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 transfer control back to the matching *return-site* of *label*; (v) at the return-site, copy the return value if needed and resume execution in the caller.

The activation record at the top of the stack is referred to as the *current activation record*. Local variables and parameters stored in the 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 Fig. 1 (c) invokes `rev` on a list with eight elements. This program is used throughout the paper as a running example. In procedure `rev`, label l_1 plays the role of the recursive call site, l_0 that of `rev`'s entry point, and l_2 of `rev`'s 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 [13, 15]. In this section, we summarize that framework, which shows how 3-valued logic can serve as the basis for program analysis.

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 variable or parameter `x`, and cs_{label} for every label *label* immediately preceding a procedure call

Predicate	Intended Meaning
$heap(v)$	v is a heap element.
$stack(v)$	v is an activation record.
$cs_{label}(v)$	*label* is the call-site of the procedure whose activation record is v.
$g(v)$	The heap element v is pointed to by a global variable `g`.
$n(v_1, v_2)$	The `n`-component of list element v_1 points to the list element v_2.
$top(v)$	v is the current activation record.
$pr(v_1, v_2)$	The activation record v_2 is the immediate previous activation record of v_1 in the stack.
$x(v_1, v_2)$	The local (parameter) variable `x`, which is stored in activation record v_1, points to the list element v_2.

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 called the *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 $p^S\colon (U^S)^k \to \{0,1\}$.

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 pointer variables (i.e., the specific memory addresses that they contain), and record only certain relationships that hold among the variables and memory elements:

- Every individual v represents either a heap cell in which case $heap^S(v) = 1$, or an activation record, in which case $stack^S(v) = 1$.
- The unary predicate cs_{label} indicates the call-site at which a procedure is invoked. Its similarities with the call-strings of [16] are discussed in Sect. 6.
- 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 activation record 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; we draw nodes u having $heap^S(u) = 1$ as ellipses to indicate heap elements; and we draw nodes having $stack^S(u) = 1$ as rectangles to indicate stack elements.[1]

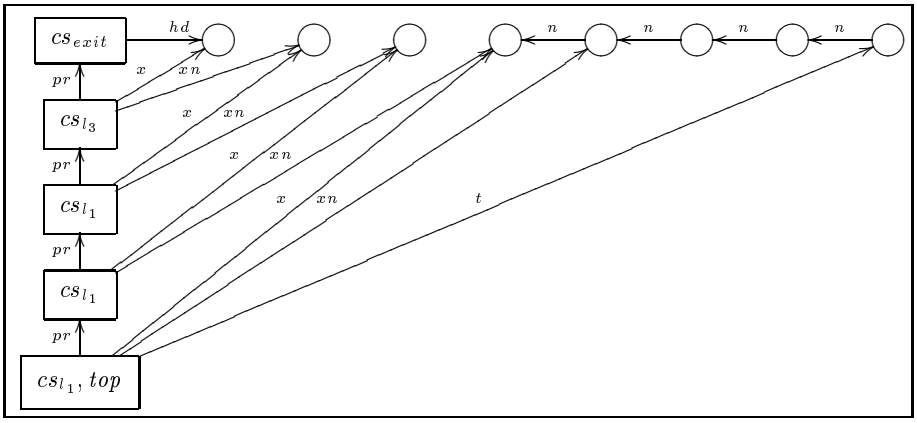

Fig. 2. The 2-valued structure S_2, which corresponds to the program state at l_2 in the `rev` procedure upon exit of the fourth recursive invocation of the `rev` procedure

Example 3.1 The 2-valued structure S_2 shown in Fig. 2 corresponds to the memory state at program point l_2 in the `rev` procedure upon exit from the fourth invocation of the `rev` procedure in the running example. The five rectangular nodes correspond to the activation records of the five procedure invocations. Note that our convention is that a 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_2 in the current invocation of `rev`.

Notice that the `n` predicate is the only one that is specific to the linked list structure declared in Fig. 1(a). The remaining predicates would play a rule in the analysis of any data structure.

[1] This can be formalized alternatively using many sorted logics. We avoided that for the sake of simplicity, and for similarity with [15].

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 Sect. 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 $\frac{1}{2}$ (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 Fig. 3. We say that the values 0 and 1 are *definite values* and that $\frac{1}{2}$ is an *indefinite value*.

$\wedge$	0	1	$\frac{1}{2}$
0	0	0	0
1	0	1	$\frac{1}{2}$
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$

$\vee$	0	1	$\frac{1}{2}$
0	0	1	$\frac{1}{2}$
1	1	1	1
$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$

$\neg$	
0	1
1	0
$\frac{1}{2}$	$\frac{1}{2}$

Fig. 3. Kleene's 3-valued interpretation of the propositional operators

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 p of arity k, p^S is a function $p^S\colon (U^S)^k \to \{0, 1, \frac{1}{2}\}$, where $\frac{1}{2}$ explicitly captures unknown predicate values.

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

Let $S^\natural$ be a 2-valued structure, S be a 3-valued structure, and $f\colon U^{S^\natural} \to U^S$ be a surjective function. We say that f *embeds* $S^\natural$ *into* S if for every predicate p of arity k and $u_1, \ldots, u_k \in U^{S^\natural}$, either $p^{S^\natural}(u_1, \ldots, u_k) = p^S(f(u_1), \ldots, f(u_k))$ or $p^S(f(u_1), \ldots, f(u_k)) = \frac{1}{2}$. 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 *summary nodes*. We use a designated unary predicate sm to maintain summary-node information. A summary node w has $sm^S(w) = \frac{1}{2}$, 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 $sm^S(w) = 0$, then w is known to represent a unique node. We impose an additional restriction on embedding functions: only nodes with $sm^S(w) = \frac{1}{2}$ can have more than one node mapped to them by an embedding function.

Example 3.2 The 3-valued structure S_4 shown in Fig. 4 represents the 2-valued structure S_2 shown in Fig. 2. The dotted ellipse summary node represents all the eight list elements. The indefiniteness of the self n-edge results from the fact that there is an 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 `rev`. The unary predicate cs_{l_1} drawn inside it indicates that it (only) represents activation records of `rev` that return to 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 Fig. 4 represents the activation record at the top of Fig. 2, which is the invocation of `main`. The second rectangle from the top in S_4 represents the second rectangle from the top in S_2 which is an invocation of `rev` from `main` (indicated by the occurrence of cs_{l_3} inside this node). The bottom rectangle in S_4 represents the bottom rectangle in S_2, which is the current activation record (indicated by the occurrence of *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 *top* does not occur in either of them.

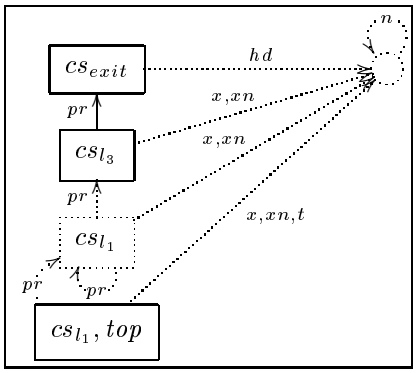

Fig. 4. The 3-valued structure S_4 which represents the 2-valued structure shown in Fig. 2

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.[2] For example, the formula

$$\exists v_1, v_2 : \neg top(v_1) \wedge \neg top(v_2) \wedge v_1 \neq v_2 \wedge x(v_1, v) \wedge x(v_2, v) \qquad (1)$$

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 [15, 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

[2] There is one non-standard aspect in our logic; $v_1 = v_2$ and $v_1 \neq v_2$ are indefinite in case v_1 and v_2 are the same summary node. The reason for this is seen shortly.

logic—defines a property, and reinterpret it on a 3-valued structure S: The Embedding Theorem ensures that one must obtain a value that is conservative with regard to the value of the formula any 2-valued structure represented by S.

Example 3.3 Consider the 2-valued structure S_2 shown in Fig. 2. The formula (1) evaluates to 0 at all of the list nodes. In contrast, consider the 3-valued structure S_4 shown in Fig. 4. This formula (1) evaluates to $\frac{1}{2}$ at the dotted ellipse summary heap node. This is in line with the Embedding Theorem since $\frac{1}{2}$ 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 The Algorithm

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 superfluous 3-valued structures.

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

4.1 Observing Selected Properties

To overcome the kind of imprecision described in Example 3.3, we introduce *instrumentation predicates*. These predicates are stored in each structure, just like the core predicates. The values of these predicates are derived from the core predicates, that is, every instrumentation predicate has a formula over the set of core predicates that defines its meaning. The instrumentation predicates that our interprocedural algorithm utilizes are described in Table 2, together with their informal meaning and their defining formula (other intraprocedural instrumentation predicates are defined in [15]).

The instrumentation predicates are divided into four classes, separated by double horizontal lines in Table 2: (i) Properties of heap elements with respect to visible variables, i.e., x and $r_{n,x}$. These are the ones originally used in [15]. (ii) Properties of heap elements with respect to invisible variables. These are $\widehat{x}$ and $r_{n,\widehat{x}}$, which are variants of x and $r_{n,x}$ from the first class, but involve the invisible variables. The $sh^{\widehat{}}_{x}(v)$ predicate is motivated by Example 3.3. It is similar to the heap-sharing predicate used in [2, 10, 14, 15]. (iii) Generic properties of an individual activation record. For example, $nn^{S}_{\widehat{x}}(u) = 1$ (nn for not NULL) in a 2-valued structure S indicates that the invisible instance of `x` that is stored in activation record u points to some list element. (iv) Properties across successive recursive calls. For example, the predicate $al_{x,pr[y]}$ captures aliasing between `x`

Table 2. 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

Predicate	**Intended Meaning**	**Defining Formula**
$x(v)$	The list element v is pointed to by the visible instance of `x`.	$\exists v_1 : top(v_1) \wedge x(v_1, v)$
$r_{n,x}(v)$	The list element v is reachable by following `n`-components from the visible instance of `x`.	$\exists v_1, v_2 : top(v_1) \wedge x(v_1, v_2) \wedge n^*(v_2, v)$
$\widehat{x}(v)$	The list element v is pointed to by an invisible instance of `x`.	$\exists v_1 : \neg top(v_1) \wedge x(v_1, v)$
$r_{n,\widehat{x}}(v)$	The list element v is reachable by following `n`-component from an invisible instance of `x`.	$\exists v_1, v_2 : \neg top(v_1) \wedge x(v_1, v_2) \wedge n^*(v_2, v)$
$sh_{\widehat{x}}(v)$	The list element v is pointed to by more than one invisible instance of `x`.	$\exists v_1, v_2 : v_1 \neq v_2 \wedge \neg top(v_1) \wedge \neg top(v_2) \wedge x(v_1, v) \wedge x(v_2, v)$
$nn_{\widehat{x}}(v)$	The invisible instance of `x` stored in the activation record v points to some list element.	$\exists v_1 : \neg top(v) \wedge x(v, v_1)$
$al_{x,y}(v)$	The invisible instances of `x` and `y` stored in the activation record v are aliased.	$\exists v_1 : \neg top(v) \wedge x(v, v_1) \wedge y(v, v_1)$
$al_{x,pr[y]}(v)$	The instance of `x` stored in the activation record v is aliased with the instance of `y` stored in v's previous activation record.	$\exists v_1, v_2 : pr(v, v_1) \wedge x(v, v_2) \wedge y(v_1, v_2)$
$al_{x,pr[y]\to n}(v)$	The instance of `x` stored in the activation record v is aliased with `y->n` for the instance of `y` stored in v's previous activation record.	$\exists v_1, v_2, v_3 : x(v, v_1) \wedge pr(v, v_2) \wedge y(v_2, v_3) \wedge n(v_3, v_1)$
$al_{x\to n,pr[y]}(v)$	`x->n` for the instance of `x` stored in the activation record v is aliased with the instance of `y` stored in v's previous activation record.	$\exists v_1, v_2, v_3 : x(v, v_2) \wedge n(v_2, v_1) \wedge pr(v, v_3) \wedge y(v_3, v_1)$

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_5 shown in Fig. 5 also represents the 2-valued structure S_2 shown in Fig. 2. In contrast with S_4 shown in Fig. 4, in which all eight list elements are represented by one heap node, in S_5, they are represented by six heap nodes. The leftmost heap node in S_5 represents the leftmost list element in S_2 (which was originally the first list element). The fact that $\widehat{hd}$ is drawn inside this node indicates that it represents a list element pointed to by an invisible instance of `hd`. This fact can also be extracted from S_5 by evaluating the $\widehat{hd}$ defining formula at this node, but this is not always the case, as we will now see: The second leftmost heap node is a summary node that represents both the second and third list elements from the left in S_2. There is an

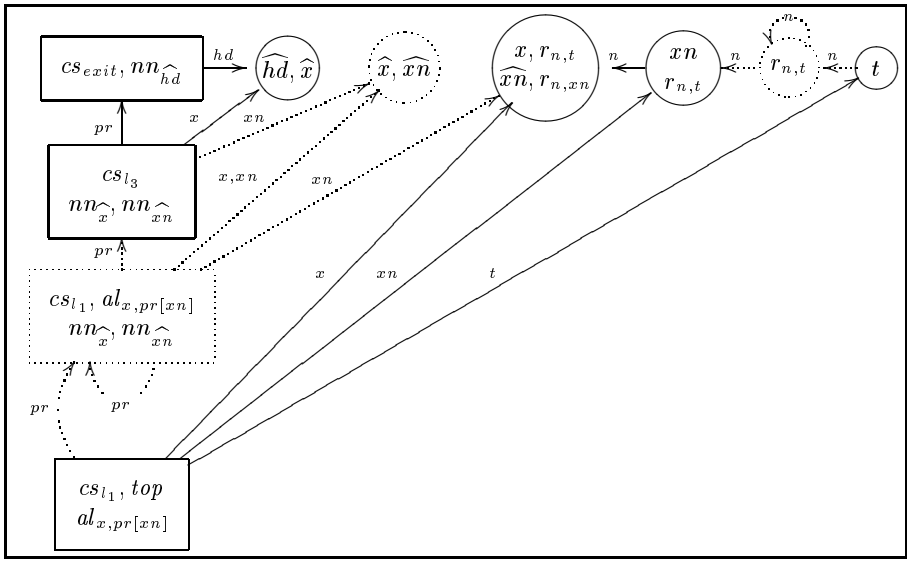

Fig. 5. The 3-valued structure S_5 represents the 2-valued structure shown in Fig. 2. For clarity, we do not show $r_{n,x}$ for nodes having the property x

indefinite x-edge into this summary node. Still, since $\widehat{x}$ 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 storage locations that have been leaked by the program.

The other summary heap node (the second heap node from the right) represents the second and third (from the right) list elements of S_2. Its incoming n edge is indefinite. Still, since $r_{n,t}$ occurs inside this node, we know that all the list elements it represents are reachable from `t`.

Note that the predicate $sh_{\widehat{x}}$ does not hold for any heap node in S_5. Therefore, no list element in any 2-valued structure that S_5 represents is pointed to by more than one invisible instance of the variable `x`. Note that the combination of $sh_{\widehat{x}}(u) = 0$ (pointed to by at most one invisible instance of `x`) and $\widehat{x}() = 1$ (pointed to by at least one invisible instance of `x`) allows determining that each node represented by a summary node u is pointed to by exactly one invisible instance of `x` (cf. the second leftmost heap node in S_5).

The stack elements are depicted in the same way as they are depict by S_4 (see Example 3.2). Since $al_{x,pr[xn]}$ occurs 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 `xn` stored in the activation record preceding v.

Notice that the $r_{n,x}$, $r_{n,\widehat{x}}$, $al_{x,pr[y]\to n}$, $al_{x\to n,pr[y]}$ predicates are the only instrumentation predicates specific to the linked list structure declared in Fig. 1 (a). The remaining predicates would play a role in any analysis that would attempt to analyze the runtime stack.

4.2 The Best Abstract Transformer

This section provides a high level description of the algorithm in terms of the general abstract interpretation framework [3]. Conceptually, 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 Fig. 6: (i) find each consistent 2-valued structure $S^\natural$ represented by S (*concretization*); (ii) apply the C operational semantics to every such structure $S^\natural$ resulting in a 2-valued structure $S^{\natural\prime}$ and (iii) finally abstract each of the 2-valued structures $S^{\natural\prime}$ 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.

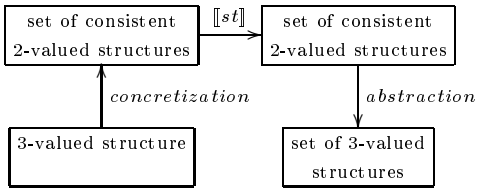

Fig. 6. The best abstract semantics of a statement `st` with respect to 3-valued structures. $[\![st]\!]$ is the operational semantics of `st` applied pointwise to every consistent 2-valued structure

The Abstraction Principle The abstraction function is defined by a subset of the unary predicates, that are called *abstraction properties* in [15]. 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 every other predicate p in the abstracted 3-valued structure are determined conservatively to yield an indefinite value whenever corresponding values of p in the represented concrete 2-valued structure disagree.

Example 4.2 The structure S_4 shown in Fig. 4 is an abstraction of S_2 shown in Fig. 2 when all of the unary core predicates are used as abstraction properties. For example, the activation records of the second and third 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_{l_1} does). Also, all of the eight heap nodes are mapped to the same summary-node for which only the *heap* core predicate holds. The *pr*-edge into the stack node at the top of the figure is definite since there is only one node mapped to each of the edge's endpoints. In contrast, the *hd*-edge emanating from the uppermost stack node must be indefinite in order for S_4 to conservatively represent S_2: In S_2 the *hd* predicate holds for the uppermost stack node and the leftmost heap node, but it does not hold for any other heap node, and all heap nodes of S_2 are summarized into one summary heap node.

The structure S_5 shown in Fig. 5 is an abstraction of S_2 shown in Fig. 2 when the abstraction properties are all of the unary core and instrumentation predicates. Notice that nodes with different observed properties lead to different

instrumentation predicate values and thus are never represented by the same abstract node. Because the set of unary predicates is fixed, there can only be a constant number of nodes in an abstracted structure which guarantees that the analysis always terminates.

Analyzing Return Statements How to retain precision when the analysis performs its abstract execution across a return statement is the key problem that we face. By exploiting the instrumentation predicates, our technique is capable of handling return statements quite precisely. This is demonstrated in the following example. For expository purposes we will explain the abstract execution of return statement in term of the best abstract transformer, described in Sect. 4.2. The actual method our analysis uses is discussed in Sect. 4.3.

Example 4.3 Let us exemplify the application of the return statement to the 3-valued structure S_5 shown in Fig. 5. following the stages of the *best* iterative algorithm described in Sect. 4.2.

Stage I–Concretization : Let $S^\natural$ be one of the consistent 2-valued structures represented by S_5. Let $k \geq 1$ be the number of activation records represented by the summary stack node in S_5. Since $S^\natural$ is a consistent 2-valued structure, the `x` parameter variable in each of these k activation records must point 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 the presence of $nn^{\widehat{}}_x$ 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 $\widehat{x}$ occurs 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 $sh^{\widehat{}}_x$ does not occur 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 $al_{x,pr[xn]}$ is drawn inside the two stack nodes at the bottom of Fig. 5, we conclude that the instance of `x` in each recursive invocation of `rev` is aliased with the instance of `xn` of `rev`'s previous invocation. Thus, each acceptable $S^\natural$ looks essentially like the structure shown in Fig. 2, but with k isolated list elements not pointed to by `hd`, rather than two, and with some number of elements in the list pointed to by `x`.

Stage II–Applying the Operational Semantics: Applying the operational semantics of `return` to $S^\natural$ (see Sect. 2) results in a (consistent) 2-valued structure $S^{\natural\prime}$. Note that the list element pointed to by the visible instance of `x` in $S^{\natural\prime}$ is not pointed to by any other instance of `x`, and it is not part of the reversed suffix. Thus $S^{\natural\prime}$ differs from $S^\natural$ by having the top activation record of $S^\natural$ removed from the stack and by having the activation record preceding it be the new *current* activation record.

Stage III–Abstraction: Abstracting $S^{\sharp'}$ 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_5, since the information regarding the number of remaining isolated list elements and invisible activation record is lost in the summarization. For $k = 1$ and $k = 2$ we have a consistent 2-valued structures 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 they are pointed to by different (invisible) local variables. For example, one of them is pointed to by `hd` 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 $S^{\sharp'}$ is not pointed to by another instance of this variable. Thus, the analysis must conservatively assume that future calls to `app` may create cycles. However, even when $al_{x,pr[xn]}$ is not maintained, the analysis can still produce fairly accurate results using only the $sh_{\widehat{x}}$ and $\widehat{x}$ instrumentation predicates.

4.3 Our Iterative Algorithm

Unlike a hypothetical algorithm based on the best abstract transformer 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 `return` statement to each consistent 2-valued structure S represents. However, in general, the transformers used are conservative approximations of the best abstract transformer; the set of structures obtained may represent more 2-valued structures than those represented by applying the best abstract transformer. Our experience to date, reported in Sect. 5, indicates that it usually gives good results.

Technically, the 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 a hypothetical algorithm based on the best abstract transformer. The update formulae for the core-predicates describing the operational semantics for call and return statements are given in Table 3.

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 are omitted here for lack of space. The reader is referred to [15] for many examples of predicate-update formulae for instrumentation predicates and other operations used by the 3-valued logic framework to increase precision.

Table 3. 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 executes, denoted by p', is defined in terms of the core predicate values before the statement executes (denoted without primes). Core predicates that are not specified above are assumed to be unchanged, i.e., $p'(v_1, \ldots) = p(v_1, \ldots)$. There is a separate update formula for every local variable or parameter `x`, and every label lb immediately preceding a procedure call. The predicate $new(v)$, used in the update formula of the $cs_{label}(v)$ predicates, holds only for the newly allocated activation record

`label: call f()`
$stack'(v) = stack(v) \vee new(v)$
$cs'_{label}(v) = cs_{label}(v) \vee new(v)$
$top'(v) = new(v)$
$pr'(v_1, v_2) = pr(v_1, v_2) \vee (new(v_1) \wedge top(v_2))$

`return`
$stack'(v) = stack(v) \wedge \neg top(v)$
$cs'_{lb}(v) = cs_{lb}(v) \wedge \neg top(v)$
$top'(v) = \exists v_1 : top(v_1) \wedge pr(v_1, v)$
$pr'(v_1, v_2) = pr(v_1, v_2) \wedge \neg top(v_1)$
$x'(v_1, v_2) = x(v_1, v_2) \wedge \neg top(v_1)$

5 A Prototype Implementation

A prototype of the iterative algorithm sketched in Sect. 4.3. was implemented for a small subset of C, in particular we do not support mutual recursion. The main goal has been to determine if the results of the analysis are useful before trying to scale 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 might have reported that every program point possibly leaked memory, performed a NULL-pointer dereference, etc. Fortunately, in Sect. 5.1 we show that this is not the case for the C procedures analyzed.

The algorithm was implemented using TVLA, a **T**hree-**V**alued-**L**ogic **A**nalyzer which is implemented in Java [13]. TVLA is quite powerful but slow, and only supports intraprocedural analysis specified using low level logical formulae. Therefore, we implemented a frontend that generates TVLA input from a program in a subset of C. The instrumentation predicates described in Sect. 4.1 allow our frontend to treat call and return statements in the same way that intraprocedural statements are handled, without sacrificing precision in many recursive procedures. Our frontend also performs certain minimal optimizations not described here for lack of space.

5.1 Empirical Results

The analyzed C procedures together with the space used and the running time on a Pentium II 233 Mhz machine running Windows 2000 with JDK 1.2.2 are listed in Table 4. The analysis verified that indeed these procedures always return a linked list and contain no memory leaks and NULL-pointer dereferences. Verifying the absence of memory leaks is quite challenging for these procedures since it requires information about invisible variables as described in Sect. 4.1.

Table 4. The total number of 3-valued structures that arise during analysis and running times for the recursive procedures analyzed. The procedures are available at "http://www.cs.technion.ac.il/~maon"

Proc.	Description	# of Structs	Time (secs)
create	creates a list.	219	5.91
delall	frees the entire list	139	13.10
insert	creates and inserts an element into a sorted list	344	38.33
delete	deletes an element from a sorted list	423	41.69
search	searches an element in a sorted list	303	8.44
app	adds one list to the end of another	326	42.81
rev	the running example (non recursive append)	829	105.78
rev_r	the running example (with recursive append)	2285	1028.80
rev_d	reverses a list with destructive updates	429	45.99

6 Conclusions, Limitations and Future Work

In this paper, we present a novel interprocedural shape analysis algorithm for programs that manipulate linked lists. The algorithm is more precise than existing shape analysis algorithms described in [2, 10, 14] for recursive programs that destructively update the program store. The precision of our algorithm can be attributed to the properties of invisible instances of local variables that it tracks. Particularly important seems to be the sharing properties of stack variables. Previous algorithms [2, 10] either did not handle the case where multiple instances of the same local variable exist simultaneously, or only represented their potential values [14]. As we have demonstrated, in the absence enough information about the values of local variables, an analysis must make very conservative assumptions. These assumptions lead to imprecise results and performance deteriorates as well, since every potential value of a local variable must be considered.

We follow the approach suggested in [4, 11] and summarize activation records in essentially the same way that linked list elements are summarized. By representing the call site in each activation record the analysis algorithm is capable of encoding the calling context, too. This approach bears some similarity to the *call-string* approach of [16], since it avoids propagating information to return sites that do not match the call site of the current activation record. In our case there is no need to put an arbitrary bound on the "length" of the call-string, the bounded representation is achieved indirectly by the summarization of activation records.

So far, our technique (and our implementation) analyzes small programs in a "friendly" subset of C. We plan to extend it to a larger subset of C, and to experiment with scaling 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 [17], before proceeding to our quite precise but expensive analysis. We focused this research on linked lists, but, plan to also 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 any user intervention, e.g., a specification of the program. However, this is a limitation because the analyzer produce poor results for program in which other properties are the important distinctions to track.

Acknowledgments We are grateful for the helpful comments and the contributions of N. Dor, O. Grumberg, T. Lev-Ami, R. Wilhelm, T. Reps and E. Yahav.

References

1. U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, September 1993.
2. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, 1990.
3. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
4. A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Symp. on Princ. of Prog. Lang.*, 1990.
5. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS'00, Static Analysis Symposium*. Springer, 2000.
6. R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1998. ACM Press.
7. R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1996. ACM Press.
8. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
9. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, June 1992.
10. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4. Prentice-Hall, Englewood Cliffs, NJ, 1981.
11. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.
12. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, 1988.
13. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000.
14. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, Jan 1998.
15. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.
16. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.
17. B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996.