

12

Shape Analysis and Applications¹

Thomas Reps²

*Computer Sciences Department,
University of Wisconsin-Madison, WI
reps@cs.wisc.edu*

Mooly Sagiv

*Department of Computer Science,
School of Mathematics and Science,
Tel Aviv University, Tel Aviv, Israel
Sagiv@math.tau.ac.il*

Reinhard Wilhelm

*Fachbereich Informatik,
Universitaet des Saarlandes,
Saarbruecken, Germany
Wilhelm@cs.uni-sb.de*

12.1	Introduction	12-2
	Structure of the Chapter	
12.2	Questions about the Heap Contents	12-3
	Traditional Compiler Analyses • Analyzing Programs for Shapes • Answers as Given by Shape Analysis	
12.3	Shape Analysis	12-9
	Summarization • Parametric Shape Analysis • Abstraction Functions • Designing a Shape Abstraction	
12.4	An Overview of a Shape-Analysis Framework	12-13
	Representing Stores via 2-Valued and 3-Valued Logical Structures • Extraction of Store Properties • Expressing the Semantics of Program Statements • Abstraction via Truth-Blurring Embeddings • Conservative Extraction of Store Properties • Abstract Interpretation of Program Statements	
12.5	Applications	12-32
	Identifying May- and Must-Aliases • Constructing Program Dependences • Other Applications	
12.6	Extensions	12-38
	Interprocedural Analysis • Computing Intersections of Abstractions • Efficient Heap Abstractions and Representations • Abstracting Numeric Values • Abstraction Refinement	
12.7	Related Work	12-40
12.8	Conclusions	12-41
	References	12-41

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 understand programs or to verify properties of programs. Shape analysis also recovers information that is valuable for debugging, compile-time garbage collection, instruction scheduling, and parallelization.

¹Portions of this paper were adapted from [65] (© Springer-Verlag) and excerpted from [58] (© ACM).

²Supported in part by NSF Grants CCR-9619219, CCR-9986308, CCF-0540955, and CCF-0524051; by ONR Grants N00014-01-1-0796 and N00014-01-1-0708; by the Alexander von Humboldt Foundation; and by the John Simon Guggenheim Memorial Foundation. Address: Comp. Sci. Dept.; Univ. of Wisconsin; 1210 W. Dayton St.; Madison, WI 53706.

³Address: School of Comp. Sci.; Tel Aviv Univ.; Tel Aviv 69978; Israel.

⁴Address: Fachrichtung Informatik, Univ. des Saarlandes; 66123 Saarbrücken; Germany.

12.1 Introduction

Pointers and heap-allocated storage are features of all modern imperative programming languages. However, they are ignored in most formal treatments of the semantics of imperative programming languages because their inclusion complicates the semantics of assignment statements: an assignment through a pointer variable (or through a pointer-valued component of a record) may have far-reaching side effects. Works that have treated the semantics of pointers include [5, 42, 43, 45].

These far-reaching side effects also make program dependence analysis harder, because they make it difficult to compute the aliasing relationships among different pointer expressions in a program. Having less precise program dependence information decreases the opportunities for automatic parallelization and for instruction scheduling.

The usage of pointers is error prone. Dereferencing NULL pointers and accessing previously deallocated storage are two common programming mistakes. The usage of pointers in programs is thus an obstacle for program understanding, debugging, and optimization. These activities need answers to many questions about the structure of the heap contents and the pointer variables pointing into the heap.

By *shapes*, we mean descriptors of heap contents. *Shape analysis* is a generic term denoting static program-analysis techniques that attempt to determine properties of the heap contents relevant for the applications mentioned above.

12.1.1 Structure of the Chapter

Section 12.2 lists a number of questions about the contents of the heap. Figure 12.1 presents a program that will be used as a running example, which inserts an element into a singly linked list. Section 12.2.3 shows how shape analysis would answer the questions about the heap contents produced by this program. Section 12.3 then informally presents a parametric shape-analysis framework along the lines of [58], which provides a generative way to design and implement shape-analysis algorithms. The “shape semantics” — plus some additional properties that individual storage elements may or may not possess — are specified in logic, and the shape-analysis algorithm is automatically generated from such a specification. Section 12.4 shows how the informal treatment from Section 12.3 can be made precise by basing it on predicate logic. In particular, it is shown how a 2-valued interpretation and a 3-valued interpretation of the same set of

<pre> /* list.h */ typedef struct node { struct node *n; int data; } *List; (a) </pre>	<pre> / * insert.c */ #include 'list.h' void insert (List x, int d) { List y, t, e; assert(acyclic _list (x) && x != NULL); y = x; while (y->n != NULL && ...) { y = y->n; } t = malloc(); t->data = d; e = y->n; t->n = e; y->n = t; } (b) </pre>
--	---

FIGURE 12.1 (a) Declaration of a linked-list data type in C. (b) A C function that searches a list pointed to by parameter *x*, and splices in a new element.

formulas can be used to define the concrete and abstract semantics, respectively, of pointer-manipulating statements. Section 12.5 lists some applications of shape analysis. Section 12.6 briefly describes several extensions of the shape-analysis framework that have been investigated. Section 12.7 discusses related work. Section 12.8 presents some conclusions.

12.2 Questions about the Heap Contents

Shape analysis has a somewhat constrained view of programs. It is not concerned with numeric or string values that programs compute, but exclusively with the linked data structures they build in the heap and the pointers into the heap from the stack, from global memory, or from cells in the heap.⁵ We will therefore use the term *execution state* to mean the set of cells in the heap, the connections between them (via pointer components of heap cells), and the values of pointer variables in the store.

12.2.1 Traditional Compiler Analyses

We list some questions about execution states that a compiler might ask at points in a program, together with (potential) actions enabled by the respective answers:

NULL pointers: Does a pointer variable or a pointer component of a heap cell contain NULL at the entry to a statement that dereferences the pointer or component?

Yes (for every state): Issue an error message.

No (for every state): Eliminate a check for NULL.

Maybe: Warn about the potential NULL dereference.

Alias: Do two pointer expressions reference the same heap cell?

Yes (for every state): Trigger a prefetch to improve cache performance, predict a cache hit to improve cache-behavior prediction, or increase the sets of uses and definitions for an improved liveness analysis.

No (for every state): Disambiguate memory references and improve program dependence information [11, 55].⁶

Sharing: Is a heap cell shared?⁷

Yes (for some state): Warn about explicit deallocation, because the memory manager may run into an inconsistent state.

No (for every state): Explicitly deallocate the heap cell when the last pointer to it ceases to exist.

Reachability: Is a heap cell reachable from a specific variable or from any pointer variable?

Yes (for every state): Use this information for program verification.

No (for every state): Insert code at compile time that collects unreachable cells at runtime.

Disjointness: Do two data structures pointed to by two distinct pointer variables ever have common elements?

No (for every state): Distribute disjoint data structures and their computations to different processors [24].

⁵However, the shape-analysis techniques presented in Sections 12.3 and 12.4 can be extended to account for both numeric values and heap-allocated objects. See Section 12.6.4 and [20, 21, 28].

⁶The answer “yes (for some state)” indicates the case of a may-alias. This answer prevents reordering or parallelizing transformations from being applied.

⁷Later in the chapter, the sharing property that is formalized indicates whether a cell is “heap-shared,” that is, pointed to by two or more pointer components of heap cells. Sharing due to two pointer variables or one pointer variable and one heap cell component pointing to the same heap cell is also deducible from the results of shape analysis.

Cyclicity: Is a heap cell part of a cycle?

No (for every state): Perform garbage collection of data structures by reference counting. Process all elements in an acyclic linked list in a *doall*-parallel fashion.

Memory leak: Does a procedure or a program leave behind unreachable heap cells when it returns?

Yes (in some state): Issue a warning.

The questions in this list are ones for which several traditional compiler analyses have been designed, motivated by the goal of improving optimization and parallelization methods. The may-alias-analysis problem, which seeks to find out whether the answer to the alias question is “yes (in some state)” is of particular importance in compiling. The goal of providing better may-alias information was the motivation for our work that grew into shape analysis.

Alias, sharing, and disjointness properties are related but different. To appreciate the difference, it suffices to see that they are defined on different domains and used in different types of compiler tasks. *Alias* relations concern pairs of pointer expressions; they are relevant for disambiguating memory references. *Sharing* properties concern the organization of neighboring heap cells; they are relevant for compile-time memory management. *Disjointness* relations concern pairs of data structures; they are relevant for determining whether traversals of two data structures can be parallelized. The relations between these properties are as follows:

Disjointness-aliasing: Two data structures D_1 and D_2 are disjoint in every state if there exist no two pointer expressions e_1 , referring to D_1 , and e_2 , referring to D_2 , that may be aliased in any state.

Disjointness-sharing: If two data structures D_1 and D_2 are not disjoint in some state, at least one of the common elements of D_1 and D_2 is shared in this state.

Aliasing-sharing: If two different pointer expressions e_1 and e_2 reference the same heap cell in some state, then this cell or one of its “predecessors” must be shared in this state. However, the opposite need not hold because not all heap cells are necessarily reachable from a variable.

Some of the other questions in the list given earlier concern memory-cleanness properties [14], for example, no NULL-dereferences, no deallocation of shared cells, and no memory leaks.

12.2.1.1 Memory Disambiguation

Many compiler transformations and their enabling analyses are based on information about the independence of program statements. Such information is used extensively in compiler optimizations, automatic program parallelizations, code scheduling for instruction-level parallel machines, and in software-engineering tools such as code slicers. The concept of *program dependence* is based on the notions of *definition* and *use of resources*. Such analyses can be performed at the source-language level, where resources are mostly program variables, as well as at the machine-language level, where resources are registers, memory cells, status flags, and so on. For source-level analysis, these notions have been generalized from scalar variables to array components. Definitions and uses, in the form of indexed array names, now denote resources that are subsections of an array. Definitions and uses, which were uniquely determining resources in the case of scalar variables, turn into *potential* definitions (respectively uses) of sets of resources. Using these sets in the computation of dependences may induce spurious dependences. Many alias tests have been developed to ascertain whether two sets of potentially referenced resources are actually disjoint, that is, whether two given references to the same array never access the same element [66].

The same is overdue for references to the heap through pointer expressions. However, pointer expressions may refer to an unbounded amount of storage that is located in the heap. Appropriate analyses of pointer expressions should find information about:

Must-aliases: Two pointer expressions refer to the same heap cell on all executions that reach a given program point.

May-aliases: Two pointer expressions may refer to the same heap cell on an execution that reaches a given program point.

Approaches that attempt to identify may-aliases and must-aliases have traditionally used path expressions [27]. In Section 12.5.1 we provide a new approach based on shape analysis, which yields very precise results.

12.2.2 Analyzing Programs for Shapes

Several of the properties listed above can be combined to formulate more complex properties of heap contents:

Shape: What is the “shape” of (some part of) the contents of the heap? Shapes (or, more precisely, shape descriptors) characterize data structures. A shape descriptor could indicate whether the heap contains a singly linked list, potentially with (or definitely without) a cycle, a doubly linked list, a binary tree, and so on. The need to track many of the properties listed above, for example, sharing, cyclicity, reachability, and disjointness, is an important aspect of many shape-analysis algorithms. Shape analysis can be understood as an extended type analysis; its results can be used as an aid in program understanding and debugging [13].

Nonstructural properties: In addition to the shape of some portions of the contents of the heap, what properties hold among the value components of a data structure? These combined properties can be used to prove the partial correctness of programs [35].

History properties: These track where a heap cell was allocated and what kinds of operations have been performed on it. This kind of information can be used to identify dependences between points in the program (see Section 12.5.2).

12.2.2.1 Shape Descriptors and Data Structures

We claimed above that shape descriptors can characterize data structures. The constituents of shape descriptors that can be used to characterize a data structure include:

- i. Root pointer variables, that is, information about which pointer variables point from the stack or from the static memory area into a data structure stored in the heap
- ii. The types of the data-structure elements and, in particular, which fields hold pointers
- iii. Connectivity properties, such as:
 - Whether all elements of the data structure are reachable from a root pointer variable
 - Whether any data-structure elements are shared
 - Whether there are cycles in the data structure
 - Whether an element v pointed to by a “forward” pointer of another element v' has its “backward” pointer pointing to v'
- iv. Other properties, for instance, whether an element of an ordered list is in the correct position

Each data structure can be characterized by a certain set of such properties.

Most semantics track the values of pointer variables and pointer-valued fields using a pair of functions, often called the *environment* and the *store*. Constituents *i* and *ii* above are parts of any such semantics; consequently, we refer to them as *core* properties.

Connectivity and other properties, such as those mentioned in *iii* and *iv*, are usually not explicitly part of the semantics of pointers in a language but instead are properties derived from this core semantics. They are essential ingredients in program verification, however, as well as in our approach to shape analysis of programs. Noncore properties will be called *instrumentation* properties (for reasons that will become clear shortly).

Let us start by taking a Platonic view, namely that ideas exist without regard to their physical realization. Concepts such as “is shared,” “lies on a cycle,” and “is reachable” can be defined either in graph-theoretic terms, using properties of paths, or in terms of the programming-language concept of pointers. The definitions of these concepts can be stated in a way that is independent of any particular data structure; for instance:

Example 12.1

A heap cell is *heap-shared* if it is the target of two pointers — either from two different heap cells or from two different pointer components of the same heap cell.

Data structures can now be characterized using sets of such properties, where “data structure” is still independent of a particular implementation; for instance:

Example 12.2

An *acyclic singly linked list* is a set of objects, each with one pointer field. The objects are *reachable from a root pointer* either directly or by following pointer fields. No object *lies on a cycle*, that is, is reachable from itself by following pointer fields.

To address the problem of verifying or analyzing a particular program that uses a certain data structure, we have to leave the Platonic realm and formulate shape invariants in terms of the pointer variables and data-type declarations from that program.

Example 12.3

Figure 12.1a, above, shows the declaration of a linked-list data type in C, and Figure 12.1b shows a C program that searches a list and splices a new element into the list. The characterization of an acyclic singly linked list in terms of the properties “is reachable from a root pointer” and “lies on a cycle” can now be specialized for that data-type declaration and that program as follows:

- “Is reachable from a root pointer” means “is reachable from x , or is reachable from y , or is reachable from t , or is reachable from e .”
- “Lies on a cycle” means “is reachable from itself following one or more n -fields.”

This chapter deals with analyses that attempt to determine the shapes of all data structures in the heap. To obtain shape descriptors, these analyses track many of the properties that have been discussed above. Looking at things in the other direction, however, once such shape descriptors have been obtained, answers to many of the above questions can merely be “read off” of the shape descriptors.

12.2.3 Answers as Given by Shape Analysis

This section discusses the results obtained by analyzing `insert` using a particular shape-analysis algorithm designed to analyze programs that manipulate singly linked lists. In this case, the analysis of `insert` has been carried out under the assumption that the inputs to `insert` are a nonempty, acyclic singly linked list and an integer. The former requirement is captured by the shape descriptors shown in Figure 12.2, which are provided as input to the shape-analysis algorithm.

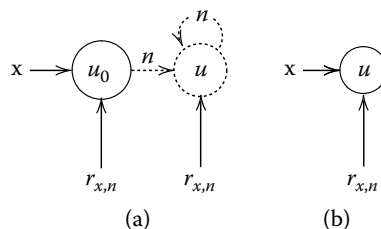


FIGURE 12.2 Shape descriptors that describe the input to `insert`. (a) Represents acyclic lists of length at least 2. (b) Represents acyclic lists of length 1.

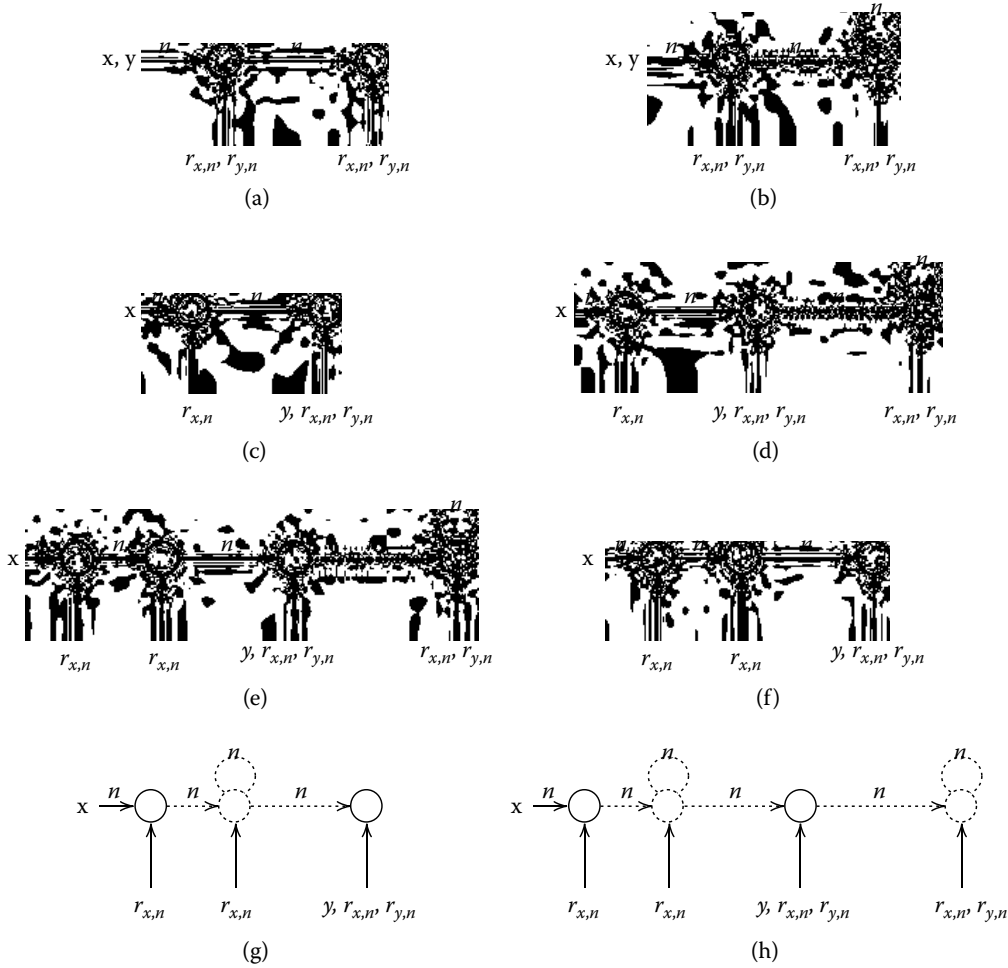


FIGURE 12.3 The eight shape graphs that arise at the beginning of the while-loop body in the program of Figure 12.1.

The shape-analysis algorithm produces information for each program point that describes the lists that can arise there. At the entry to the while-loop body some of the properties are:

- Pointer variables x and y point into the same list: x always points to the head; y points to either the head of the x -list or some tail of the x -list.
- All other pointer variables of the program have the value NULL.
- The list is acyclic.
- No memory leaks occur.

In addition, the information obtained by the shape-analysis algorithm shows that no attempt to dereference a NULL-valued pointer is ever made. Figure 12.3 shows the eight shape graphs produced by the analysis for the program point at the entry to the loop body.

Each shape graph represents a set of concrete memory configurations. In `insert`, the loop body is executed when the argument list is of length 2 or greater, and it advances variable y along the list that is pointed to by x . The shape-analysis algorithm is able to discover eight shape graphs that represent all such memory configurations. The graphs represent lists of various lengths, with various numbers of list cells between the list cells pointed to by x and y : Figure 12.3a and 12.3b represent lists in which x and y point to the same list cell; Figure 12.3c and 12.3d represent lists in which x and y point to list cells that are one apart; Figure 12.3e and 12.3f represent lists in which x and y point to list cells that are two apart; Figure 12.3g and 12.3h represent lists in which x and y point to list cells that are three or more apart.

Heap cells and their properties in the represented heaps can be read off from a shape graph in the following way:

- The name p represents pointer variable p . For instance, two of the pointer variables of program `insert`, namely x and y , appear in the shape graphs in Figures 12.2 and 12.3. The absence of the name p in a shape graph means that, in the stores represented by the shape graph, program variable p definitely has the value NULL. In Figure 12.3a, the absence of the name t means that t definitely has the value NULL in the stores that the shape graph represents.
- Circles stand for abstract nodes. A solid circle stands for an abstract node that represents exactly one heap cell. In Figure 12.2b, the circle u represents the one cell of an input list of length 1. Solid circles could be viewed as abstract nodes with the property “uniquely representing.” (This is the complement of the “summary” property sm that is introduced later on.)
- A dotted circle stands for an abstract node that may represent one or more heap cells; in Figure 12.2a, the dotted circle u represents the cells in the tail of the input list.
- A solid edge labeled c between abstract nodes m and m' represents the fact that the c -field of the heap cell represented by m points to the heap cell represented by m' . Figure 12.3a indicates that the n -field of the first list cell points to the second list cell.
- A dotted edge labeled c between abstract nodes m and m' tells us that the c -field of one of the heap cells represented by m may point to one of the heap cells represented by m' . When m and m' are the same abstract nodes, this edge may or may not represent a cycle. In Figure 12.3b, the dotted self-cycle on the dotted circle represents n -fields of heap cells represented by this abstract node possibly pointing to other heap cells represented by the dotted circle. Additional information about noncyclicity (see below) implies that, in this case, the dotted self-cycle does not represent a cycle in the heap.
- A unary property q that holds for all heap cells represented by an abstract node is represented in the graph by having a solid arrow from the property name q to that node. (These names are typically subscripted, such as $r_{x,n}$ or c_n .) For example, the property “reachable-from- x -via- n ,” denoted in the graph by $r_{x,n}$, means that the heap cells represented by the corresponding abstract nodes are (transitively) reachable from pointer variable x via n -fields. Both nodes in Figure 12.3b are the targets of a solid edge from an instance of property name $r_{x,n}$. This means the concrete cell represented by the first abstract node and all concrete cells represented by the second abstract node are reachable from x via n -fields.
- A dotted arrow from a property name p to an abstract node represents the fact that p may be true for some of the heap cells represented by the abstract node and may be false for others. The absence of an arrow from p to an abstract node means that none of the represented heap cells has property p . (Examples with dotted edges are given in Section 12.3.4.)

In summary, the shape graphs portray information of three kinds:

- **Solid**, meaning “always holds” for properties (including “uniquely representing”)
- **Absent**, meaning “never holds” for properties
- **Dotted**, meaning “don’t know” for properties (including “uniquely representing”)

Shape analysis associates sets of shape graphs with each program point. They describe (a superset of) all the execution states that can occur whenever execution reaches that program point. To determine whether a property always (ever) holds at a given program point, we must check that it holds for all (some) of the shape graphs for that point.

With this interpretation in mind, all of the claims about the properties of the heap contents at the entry to the while-loop body listed at the beginning of this subsection can be checked by verifying that they hold for all of the graphs shown in Figure 12.3.

12.3 Shape Analysis

The example program `insert` works for lists of arbitrary lengths. However, as described in the preceding section (at least for one program point), the description of the lists that occur during execution is finite. As shown in Figure 12.3, eight shape graphs are sufficient to describe all of the execution states that occur at the entry of the loop body in `insert`. This is a general requirement for shape analysis. Although the data structures that a program builds or manipulates are in general of unbounded size, the shape descriptors, manipulated by a shape-analysis algorithm, have to have *bounded size*.

This representation of the heap contents has to be *conservative* in the sense that whoever asks for properties of the heap contents — for example, a compiler, a debugger, or a program-understanding system — receives a reliable answer. The claim that “pointer variable `p` or pointer field `p->c` never has the value `NULL` at this program point” may only be made if this is indeed the case for all executions of the program and all program paths leading to the program point. It may still be the case that in no program execution `p` (respectively `p->c`) will be `NULL` at this point but that the analysis will be unable to derive this information. In the field of program analysis, we say that program analysis is allowed to (only) err on the safe side.

In short, shape analysis computes for a given program and each point in the program:

a finite, conservative representation of the heap-allocated data structures that could arise when a path to this program point is executed.

12.3.1 Summarization

The constraint that we must work with a bounded representation implies a loss of information about the heap contents. Size information, such as the lengths of lists or the depths of trees, will in general be lost. However, structural information may also be lost because of the chosen representation. Thus, a part of the execution state (or some of its properties) is exactly represented, and some part of the execution state (or some of its properties) is only approximately represented. The process leading to the latter is called *summarization*. Summarization intuitively means the following:

- Some heap cells will lose their identity, that is, will be represented together with other heap cells by one abstract node.
- The connectivity among those jointly represented heap cells will be represented conservatively; that is, each pointer in the heap will be represented, but several such pointers (or the absence of such pointers) may be represented jointly.
- Properties of these heap cells will also be represented conservatively. This means the following:
 - A property that holds for all (for none of the) summarized cells will be found to hold (not to hold) for their summary node.
 - A property that holds for some but not all of the summarized cells will have the value “don’t know” for the summary node.

12.3.2 Parametric Shape Analysis

Shape analysis is a generic term representing a whole class of algorithms of varying power and complexity that try to answer questions about the structure of heap-allocated storage. In our setting, a particular shape-analysis algorithm is determined by a set of properties that heap cells may have and by relations that may or may not hold between heap cells.

First, there are the aforementioned *core properties*, for example, the “pointed-to-by-`p`” property for each program pointer variable `p`, and the property “connected-through-`c`,” which pairs of heap cells (l_1, l_2) possess if the `c`-field of l_1 points to l_2 (see Table 12.1). These properties are part of any pointer semantics. The core properties in the particular shape analysis of the `insert` program are

TABLE 12.1 Predicates used for representing the stores manipulated by programs that use the `List` data-type declaration from Figure 12.1(a)

Predicate	Intended Meaning
$q(v)$	Does pointer variable q point to cells v ?
$n(v_1, v_2)$	Does the n -field of v_1 point to v_2 ?

“pointed-to-by- x ,” denoted by x , “pointed-to-by- y ,” denoted by y , “pointed-to-by- t ,” denoted by t , “pointed-to-by- e ,” denoted by e , and “connected-through- n ,” denoted by $n(\cdot, \cdot)$.

The *instrumentation properties* [58], denoted by \mathcal{I} , together with the core properties determine what the analysis is capable of observing. These are expressed in terms of the core properties. Our example analysis is designed to identify properties of programs that manipulate acyclic singly linked lists. Reachability properties from specific pointer variables have the effect of keeping disjoint sublists summarized separately. This is particularly important when analyzing a program in which two pointers are advanced along disjoint sublists.

Therefore, the instrumentation properties in our example analysis are “is-on-an- n -cycle,” denoted by c_n , “reachable-from- x -via- n ,” denoted by $r_{x,n}$, “reachable-from- y -via- n ,” denoted by $r_{y,n}$, and “reachable-from- t -via- n ,” denoted by $r_{t,n}$. For technical reasons, a property that is part of every shape analysis is “summary,” denoted by $sm(\cdot)$.

12.3.3 Abstraction Functions

The abstraction function of a particular shape analysis is determined by a distinguished subset of the set of all unary properties, the so-called *abstraction properties*, \mathcal{A} . Given a set \mathcal{A} of abstraction properties, the corresponding abstraction function will be called *\mathcal{A} -abstraction function* (and the act of applying it, *\mathcal{A} -abstraction*). If the set $\mathcal{W} = \mathcal{I} - \mathcal{A}$ is not empty, that is, if there are instrumentation predicates that are not used as abstraction predicates, we will call the abstraction *\mathcal{A} -abstraction with \mathcal{W}* .

The principle of abstraction is that heap cells that have the same definite values for the abstraction properties are summarized to the same abstract node. Thus, if we view the set of abstraction properties as our means of observing the contents of the heap, the heap cells summarized by one summary node have no observable difference.

All concrete heap cells represented by the same abstract heap cells agree on their abstraction properties; that is, either they all have these abstraction properties, or none of them have them. Thus, summary nodes inherit the values of the abstraction properties from the nodes they represent. For nonabstraction properties, their values are computed in the following way: if all summarized cells agree on this property — that is, they have the same value — the summary node receives this value. If not all summarized cells agree on a property, their summary node will receive the value “don’t know.” The values of binary properties are computed the same way.

From what has been said above, it is clear that there is a need for three values: two definite values, representing 0 (false) and 1 (true), and an additional value, 1/2, representing uncertainty. This abstraction process is called *truth-blurring embedding* (see also Section 12.4.4).

Example 12.4

The shape graphs in Figure 12.2 and the ones in Figure 12.3 are obtained using the $\{x, y, t, e, r_{x,n}, r_{y,n}, r_{t,n}, r_{e,n}, c_n\}$ -abstraction function. In Figure 12.2a, all the cells in the tail of an input list of length at least 2 are summarized by the abstract node u , because they all have the property $r_{x,n}$ and do not have the properties $x, y, t, e, r_{y,n}, r_{t,n}, r_{e,n}$, and c_n . The abstract node u_0 represents exactly one cell — the first cell of the input list. It has the properties x and $r_{x,n}$ and none of the other properties.

Now consider how the value of the property n is computed for the summary node u . The different list cells that are summarized by u do not have the same values for n , because at any one time a pointer field

Name	Graphical Representation
S_0^h	
S_1^h	$x \rightarrow$
S_2^h	$x \rightarrow$
S_3^h	$x \rightarrow$
S_4^h	$x \rightarrow$

FIGURE 12.4 Concrete lists pointed to by x of length ≤ 4 .

may point to at most one heap cell. Thus, the connected-by- n -field properties of the resulting summary nodes have the value $1/2$.

12.3.4 Designing a Shape Abstraction

This section presents a sequence of example shape abstractions to demonstrate how the precision of a shape abstraction can be changed by changing the properties used — both abstraction and nonabstraction properties. Here *precision* refers to the set of concrete heap structures that each abstract shape descriptor represents; a more precise shape descriptor represents a smaller set of concrete structures. One abstraction is more precise than another if it yields more precise shape descriptors. All examples treat singly linked lists of the type declared in Figure 12.1. The core properties are x , later also y , and n .

Example 12.5

Consider the case of $\{x\}$ -abstraction; that is, the only abstraction property is x . Figure 12.4 depicts four lists of length 1 to 4 pointed to by x and the empty list. Figure 12.5 shows the shape graphs obtained by applying $\{x\}$ -abstraction to the concrete lists of Figure 12.4. In addition to the lists of length 3 and 4 from Figure 12.4 (i.e., S_3^h and S_4^h), the shape graph S_3 also represents:

- The acyclic lists of length 5, 6, and so on that are pointed to by x
- The cyclic lists of length 3 or more that are pointed to by x , such that the backpointer is not to the head of the list, but to the second, third, or later element

Thus, S_3 is a finite shape graph that captures an infinite set of (possibly cyclic) concrete lists. The example shows that a “weak” abstraction may lose valuable information: even when only acyclic lists are abstracted, the result of the abstraction is a shape graph that also represents cyclic lists.

Name	Graphical Representation
S_0	
S_1	$x \rightarrow u_1$
S_2	$x \rightarrow u_1 \xrightarrow{n} u$
S_3	$x \rightarrow u_1 \xrightarrow{n} u \xrightarrow{n} u$

FIGURE 12.5 Shape graphs that are obtained by applying $\{x\}$ -abstraction to the concrete lists that appear in Figure 12.4.


Name	Graphical Representation
S_6^l	

FIGURE 12.8 A concrete list pointed to by x , where y points into the middle of the list.

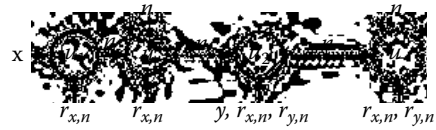

Name	Graphical Representation
S_{reach}	
S_{middle}	

FIGURE 12.9 The shape graphs that are obtained by applying $\{x, y, r_{x,n}, r_{y,n}\}$ -abstraction and $\{x, y\}$ -abstraction, respectively, to the list S_6^l from Figure 12.8.

Note that the situation depicted in Figure 12.8 occurs in `insert` as y is advanced down the list; the reachability abstraction properties play a crucial role in developing a shape-analysis algorithm that is capable of obtaining precise shape information for `insert`.

12.4 An Overview of a Shape-Analysis Framework

This section provides an overview of the formal underpinnings of the shape-analysis framework presented in [58]. The framework is *parametric*; that is, it can be instantiated in different ways to create a variety of specific shape-analysis algorithms. The framework is based on 3-valued logic. In this paper, the presentation is at a semi-technical level; for a more detailed treatment of this material, as well as several elaborations on the ideas covered here, the reader should refer to [58].

To be able to perform shape analysis, the following concepts need to be formalized:

- An encoding (or representation) of stores, so that we can talk precisely about store elements and the relationships among them.
- A language in which to state properties that store elements may or may not possess.
- A way to extract the properties of stores and store elements.
- A definition of the concrete semantics of the programming language, in particular, one that makes it possible to track how properties change as the execution of a program statement changes the store.
- A technique for creating abstractions of stores so that abstract interpretation can be applied.

In our approach, the formalization of each of these concepts is based on predicate logic.

12.4.1 Representing Stores via 2-Valued and 3-Valued Logical Structures

To represent stores, we work with what logicians call *logical structures*. A logical structure is associated with a *vocabulary* of predicate symbols (with given arities). So far we have talked about *properties* of different

classes, that is, core, instrumentation, and abstraction properties. Properties in our specification language, predicate logic, correspond to predicates.

Each logical structure S , denoted by $\langle U^S, \iota^S \rangle$, has a universe of *individuals* U^S . In a 2-valued logical structure, ι^S maps each arity- k predicate symbol p and possible k -tuple of individuals (u_1, \dots, u_k) , where $u_i \in U^S$, to the value 0 or 1 (i.e., *false* and *true*, respectively). In a 3-valued logical structure, ι^S maps p and (u_1, \dots, u_k) to the value 0, 1, or $1/2$ (i.e., *false*, *true*, and *unknown*, respectively).

2-valued logical structures will be used to encode concrete stores; 3-valued logical structures will be used to encode abstract stores; members of these two families of structures will be related by “truth-blurring embeddings” (explained in Section 12.4.4).

2-valued logical structures are used to encode concrete stores as follows: individuals represent memory locations in the heap; pointers from the stack into the heap are represented by unary “pointed-to-by-variable- q ” predicates; and pointer-valued fields of data structures are represented by binary predicates.

Example 12.9

Table 12.1 lists the predicates used for representing the stores manipulated by programs that use the `List` data-type declaration from Figure 12.1a. In the case of `insert`, the unary predicates x , y , t , and e correspond to the program variables x , y , t , and e , respectively. The binary predicate n corresponds to the n -fields of `List` elements.

Figure 12.10 illustrates the 2-valued logical structures that represent lists of length ≤ 4 that are pointed to by program variable x . Column 3 of Figure 12.10 gives a graphical rendering of these 2-valued logical structures; note that these graphs are identical to those depicted in Figure 12.4:

- Individuals of the universe are represented by circles with names inside.
- A unary predicate p is represented in the graph by having a solid arrow from the predicate name p to node u for each individual u for which $\iota(p)(u) = 1$ and no arrow from predicate name p to node u' for each individual u' for which $\iota(p)(u') = 0$. (If $\iota(p)$ is 0 for all individuals, the predicate name p will not be shown.)
- A binary predicate q is represented in the graph by a solid arrow labeled q between each pair of individuals u_i and u_j for which $\iota(q)(u_i, u_j) = 1$ and no arrow between pairs u'_i and u'_j for which $\iota(q)(u'_i, u'_j) = 0$.

Name	Logical Structure	Graphical Representation
S_0^H	unary preds. binary preds. 	
S_1^H	unary preds. binary preds. 	$x \rightarrow u_1$
S_2^H	unary preds. binary preds. 	$x \rightarrow u_1 \xrightarrow{n} u_2$
S_3^H	unary preds. binary preds. 	$x \rightarrow u_1 \xrightarrow{n} u_2 \xrightarrow{n} u_3$
S_4^H	unary preds. binary preds. 	$x \rightarrow u_1 \xrightarrow{n} u_2 \xrightarrow{n} u_3 \xrightarrow{n} u_4$

FIGURE 12.10 The 2-valued logical structures that represent lists of length ≤ 4 .

Thus, in structure S_2^h , pointer variable x points to individual u_1 , whose n -field points to individual u_2 . The n -field of u_2 does not point to any individual (i.e., u_2 represents a heap cell whose n -field has the value NULL).

12.4.2 Extraction of Store Properties

2-valued structures offer a systematic way to answer questions about properties of the concrete stores they encode. For example, consider the formula

$$\varphi_{is}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2 \quad (12.1)$$

which expresses the “is-shared” property. Do two or more different heap cells point to heap cell v via their n -fields? For instance, $\varphi_{is}(v)$ evaluates to 0 in S_2^h for the assignment $[v \mapsto u_2]$, because there is no assignment of the form $[v_1 \mapsto u_i, v_2 \mapsto u_j]$ such that $\iota^{S_2^h}(n)(u_i, u_2)$, $\iota^{S_2^h}(n)(u_j, u_2)$, and $u_i \neq u_j$ all hold.

As a second example, consider the formula

$$\varphi_{c_n}(v) \stackrel{\text{def}}{=} n^+(v, v) \quad (12.2)$$

which expresses the property of whether a heap cell v appears on a directed n -cycle. Here n^+ denotes the transitive closure of the n -relation. Formula $\varphi_{c_n}(v)$ evaluates to 0 in S_2^h for the assignment $[v \mapsto u_2]$, because the transitive closure of the relation $\iota^{S_2^h}(n)$ does not contain the pair (u_2, u_2) .

The preceding discussion can be summarized as the following principle:

Observation 12.1 (Property-Extraction Principle). *By encoding stores as logical structures, questions about properties of stores can be answered by evaluating formulas. The property holds or does not hold, depending on whether the formula evaluates to 1 or 0, respectively, in the logical structure.*

The language in which queries are posed is standard first-order logic with a transitive-closure operator. The notion of evaluating a formula φ in logical structure S with respect to assignment Z (where Z assigns individuals to the free variables of φ) is completely standard (e.g., see [17, 58]). We use the notation $\llbracket \varphi \rrbracket_2^S(Z)$ to denote the value of φ in S with respect to Z .

12.4.3 Expressing the Semantics of Program Statements

Our tool for expressing the semantics of program statements is also based on evaluating formulas:

Observation 12.2 (Expressing the Semantics of Statements via Logical Formulas). *Suppose that σ is a store that arises before statement st , that σ' is the store that arises after st is evaluated on σ , and that S is the logical structure that encodes σ . A collection of predicate-update formulas — one for each predicate p in the vocabulary of S — allows one to obtain the structure S' that encodes σ' . When evaluated in structure S , the predicate-update formula for a predicate p indicates what the value of p should be in S' .*

In other words, the set of predicate-update formulas captures the concrete semantics of st .

This process is illustrated in Figure 12.11 for the statement $y = y \rightarrow n$, where the initial structure S_a^h represents a list of length 4 that is pointed to by both x and y . Figure 12.11 shows the predicate-update formulas for the five predicates of the vocabulary used in conjunction with `insert`: x , y , t , e , and n ; the symbols x' , y' , t' , e' , and n' denote the values of the corresponding predicates in the structure that arises after execution of $y = y \rightarrow n$. Predicates x' , t' , e' , and n' are unchanged in value by $y = y \rightarrow n$. The predicate-update formula $y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$ expresses the advancement of program variable y down the list.

Structure Before	<div><div>unary preds.</div><table><thead><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr></thead><tbody><tr><td>u_1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></tbody></table><div><div>binary preds.</div><table><thead><tr><th>n</th><th>u_1</th><th>u_2</th><th>u_3</th><th>u_4</th></tr></thead><tbody><tr><td>u_1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></tbody></table><div>S_a^H y</div></div></div>	indiv.	x	y	t	e	u_1	0	0	0	0	u_2	0	0	0	0	u_3	0	0	0	0	u_4	0	0	0	0	n	u_1	u_2	u_3	u_4	u_1	0	0	0	0	u_2	0	0	0	0	u_3	0	0	0	0	u_4	0	0	0	0
indiv.	x	y	t	e																																															
u_1	0	0	0	0																																															
u_2	0	0	0	0																																															
u_3	0	0	0	0																																															
u_4	0	0	0	0																																															
n	u_1	u_2	u_3	u_4																																															
u_1	0	0	0	0																																															
u_2	0	0	0	0																																															
u_3	0	0	0	0																																															
u_4	0	0	0	0																																															
Statement	$y = y \rightarrow n$																																																		
Predicate Update Formulae	$x'(v) = x(v)$ $y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$ $t'(v) = t(v)$ $e'(v) = e(v)$ $n'(v_1, v_2) = n(v_1, v_2)$																																																		
Structure After	<div><div>unary preds.</div><table><thead><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr></thead><tbody><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></tbody></table><div><div>binary preds.</div><table><thead><tr><th>n</th><th>u_2</th><th>u_3</th><th>u_4</th></tr></thead><tbody><tr><td>u_2</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td></tr></tbody></table><div>S_b^H y</div></div></div>	indiv.	x	y	t	e	u_2	0	0	0	0	u_3	0	0	0	0	u_4	0	0	0	0	n	u_2	u_3	u_4	u_2	0	0	0	u_3	0	0	0	u_4	0	0	0														
indiv.	x	y	t	e																																															
u_2	0	0	0	0																																															
u_3	0	0	0	0																																															
u_4	0	0	0	0																																															
n	u_2	u_3	u_4																																																
u_2	0	0	0																																																
u_3	0	0	0																																																
u_4	0	0	0																																																

FIGURE 12.11 The given predicate-update formulas express a transformation on logical structures that corresponds to the semantics of $y = y \rightarrow n$.

12.4.4 Abstraction via Truth-Blurring Embeddings

The abstract stores used for shape analysis are 3-valued logical structures that, by the construction discussed below, are a priori of bounded size. In general, each 3-valued logical structure corresponds to a (possibly infinite) set of 2-valued logical structures. Members of these two families of structures are related by *truth-blurring embeddings*.

The principle behind truth-blurring embedding is illustrated in Figure 12.12, which shows how 2-valued structure S_a^H is abstracted to 3-valued structure S_a when we use $\{x, y, t, e\}$ -abstraction. Abstraction is driven by the values of the “vector” of unary predicate values that each individual u has — that is, for S_a^H , by the values $\iota(x)(u)$, $\iota(y)(u)$, $\iota(t)(u)$, and $\iota(e)(u)$ — and, in particular, by the equivalence

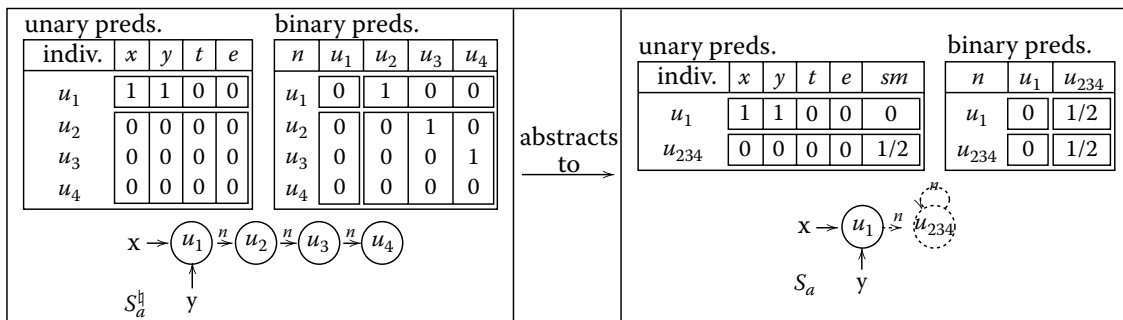


FIGURE 12.12 The abstraction of 2-valued structure S_a^H to 3-valued structure S_a when we use $\{x, y, t, e\}$ -abstraction. The boxes in the tables of unary predicates indicate how individuals are grouped into equivalence classes; the boxes in the tables for predicate n indicate how the quotient of n with respect to these equivalence classes is performed.

TABLE 12.2 Kleene’s 3-valued interpretation of the propositional operators

\wedge	0	1	1/2	\vee	0	1	1/2	\neg
0	0	0	0	0	0	1	1/2	0
1	0	1	1/2	1	1	1	1	1
1/2	0	1/2	1/2	1/2	1/2	1	1/2	1/2

classes formed from the individuals that have the same vector for their unary predicate values. In S_a^\natural , there are two such equivalence classes: (a) $\{u_1\}$, for which x , y , t , and e are 1, 1, 0, and 0, respectively, and (b) $\{u_2, u_3, u_4\}$, for which x , y , t , and e are all 0. (The boxes in the table of unary predicates for S_a^\natural show how individuals of S_a^\natural are grouped into two equivalence classes.)

All members of such equivalence classes are mapped to the same individual of the 3-valued structure. Thus, all members of $\{u_2, u_3, u_4\}$ from S_a^\natural are mapped to the same individual in S_a , called u_{234} ;⁸ similarly, all members of $\{u_1\}$ from S_a^\natural are mapped to the same individual in S_a , called u_1 .

For each non-unary predicate of the 2-valued structure, the corresponding predicate in the 3-valued structure is formed by a *truth-blurring quotient*. For instance:

- In S_a^\natural , $\iota^{S_a^\natural}(n)$ evaluates to 0 for the only pair of individuals in $\{u_1\} \times \{u_1\}$. Therefore, in S_a the value of $\iota^{S_a}(n)(u_1, u_1)$ is 0.
- In S_a^\natural , $\iota^{S_a^\natural}(n)$ evaluates to 0 for all pairs from $\{u_2, u_3, u_4\} \times \{u_1\}$. Therefore, in S_a the value of $\iota^{S_a}(n)(u_{234}, u_1)$ is 0.
- In S_a^\natural , $\iota^{S_a^\natural}(n)$ evaluates to 0 for two of the pairs from $\{u_1\} \times \{u_2, u_3, u_4\}$ (i.e., $\iota^{S_a^\natural}(n)(u_1, u_3) = 0$ and $\iota^{S_a^\natural}(n)(u_1, u_4) = 0$), whereas $\iota^{S_a^\natural}(n)$ evaluates to 1 for the other pair (i.e., $\iota^{S_a^\natural}(n)(u_1, u_2) = 1$); therefore, in S_a the value of $\iota^{S_a}(n)(u_1, u_{234})$ is 1/2.
- In S_a^\natural , $\iota^{S_a^\natural}(n)$ evaluates to 0 for some pairs from $\{u_2, u_3, u_4\} \times \{u_2, u_3, u_4\}$ (e.g., $\iota^{S_a^\natural}(n)(u_2, u_4) = 0$), whereas $\iota^{S_a^\natural}(n)$ evaluates to 1 for other pairs (e.g., $\iota^{S_a^\natural}(n)(u_2, u_3) = 1$); therefore, in S_a the value of $\iota^{S_a}(n)(u_{234}, u_{234})$ is 1/2.

In Figure 12.12, the boxes in the tables for predicate n indicate these four groupings of values.

An additional unary predicate, called *sm* (standing for “summary”), is added to the 3-valued structure to capture whether individuals of the 3-valued structure represent more than one concrete individual. For instance, $\iota^{S_a}(sm)(u_1) = 0$ because u_1 in S_a represents a single individual of S_a^\natural . However, u_{234} represents three individuals of S_a^\natural . For technical reasons, *sm* can be 0 or 1/2, but never 1; therefore, $\iota^{S_a}(sm)(u_{234}) = 1/2$.

12.4.5 Conservative Extraction of Store Properties

Questions about properties of 3-valued structures can be answered by evaluating formulas using Kleene’s semantics of 3-valued logic (see [58]). The value of a formula is obtained in almost exactly the same way that it is obtained in ordinary 2-valued logic, except that the propositional operators are given the interpretations shown in Table 12.2. (The evaluation rules for \exists , \forall , and transitive closure are adjusted accordingly; that is, \exists and \forall are treated as indexed- \vee and indexed- \wedge operators, respectively.) We use the notation $\llbracket \varphi \rrbracket_3^S(Z)$ to denote the value of φ in 3-valued logical structure S with respect to 3-valued assignment Z .

We define a partial order \sqsubseteq on truth values to reflect their degree of definiteness (or *information content*): $l_1 \sqsubseteq l_2$ denotes that l_1 is at least as definite as l_2 .

⁸The reader should bear in mind that the names of individuals are completely arbitrary. u_{234} could have been called u_{17} or u_{99} and so on; in particular, the subscript “234” is used here only to remind the reader that, in this example, u_{234} of S_a is the individual that represents $\{u_2, u_3, u_4\}$ of S_a^\natural . (In many subsequent examples, u_{234} will be named u .)

Name	Logical Structure	Graphical Representation
S_0	<div style="display: flex; justify-content: space-between;"> unary preds. binary preds. </div>	
S_1	<div style="display: flex; justify-content: space-between;"> unary preds. binary preds. </div>	x
S_2	<div style="display: flex; justify-content: space-between;"> unary preds. binary preds. </div>	x
S_3	<div style="display: flex; justify-content: space-between;"> unary preds. binary preds. </div>	x

FIGURE 12.13 The 3-valued logical structures that are obtained by applying truth-blurring embedding to the 2-valued structures that appear in Figure 12.10.

Definition 12.1 (Information Order). For $l_1, l_2 \in \{0, 1/2, 1\}$, we define the **information order** on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$. The symbol \sqcup denotes the least-upper-bound operation with respect to \sqsubseteq :

\sqcup	0	1/2	1
0	0	1/2	1/2
1/2	1/2	1/2	1/2
1	1/2	1/2	1

The 3-valued semantics is monotonic in the information order (see Table 12.2).

In [58] the *embedding theorem* states that the 3-valued Kleene interpretation in S of every formula is consistent with (i.e., \sqsupseteq) the formula's 2-valued interpretation in every concrete store S^\natural that S represents. Consequently, questions about properties of stores can be answered by evaluating formulas using Kleene's semantics of 3-valued logic:

- If a formula evaluates to 1, then the formula holds in every store represented by the 3-valued structure S .
- If a formula evaluates to 0, then the formula does not hold in any store represented by S .
- If a formula evaluates to 1/2, then we do not know if this formula holds in all stores, does not hold in any store, or holds in some stores and does not hold in some other stores represented by S .

Consider the formula $\varphi_{c_n}(v)$ defined in Equation 12.2. (Does heap cell v appear on a directed cycle of n -fields?) Formula $\varphi_{c_n}(v)$ evaluates to 0 in structure S_3 from Figure 12.13 for the assignment $[v \mapsto u_1]$, because $n^+(u_1, u_1)$ evaluates to 0 in Kleene's semantics.

Formula $\varphi_{c_n}(v)$ evaluates to 1/2 in S_3 for the assignment $[v \mapsto u]$, because $\iota^{S_3}(n)(u, u) = 1/2$, and thus $n^+(u, u)$ evaluates to 1/2 in Kleene's semantics. Because of this, we do not know whether S_3 represents a concrete store that has a cycle; this uncertainty implies that (the tail of) the list pointed to by x *might* be cyclic.

In many situations, however, we are interested in analyzing the behavior of a program under the assumption, for example, that the program's input is an acyclic list. If an abstraction is not capable of expressing the distinction between cyclic and acyclic lists, an analysis algorithm based on that abstraction will usually be able to recover only very imprecise information about the actions of the program.

For this reason, we are interested in having our parametric framework support abstractions in which, for instance, the acyclic lists are distinguished from the cyclic lists. Our framework supports such distinctions by using *instrumentation predicates*.

The preceding discussion illustrates the following principle:

Observation 12.3 (Instrumentation Principle). *Suppose that S is a 3-valued structure that represents the 2-valued structure S^\natural . By explicitly “storing” in S the values that a formula φ has in S^\natural , it is sometimes possible to extract more precise information from S than can be obtained just by evaluating φ in S .*

In our experience, we have found three kinds of instrumentation predicates to be useful:

- Nullary predicates record Boolean information (and are similar to the “predicates” in predicate abstraction [3, 22]). For example, to distinguish between cyclic and acyclic lists, we can define an instrumentation predicate c_0 by the formula

$$\varphi_{c_0} \stackrel{\text{def}}{=} \exists v : n^+(v, v) \quad (12.3)$$

which expresses the property that some heap cell v lies on a directed n -cycle. Thus, when $\iota^S(c_0)$ is 0, we know that S does not represent any memory configurations that contain cyclic data structures.

- Unary instrumentation predicates record information for unbounded sets of objects. Examples of some unary instrumentation predicates are given in Section 12.3.4. Notice that the unary cyclicity predicate c_n (defined by an open formula [see Equation 12.2]) allows finer distinctions than are possible with the nullary cyclicity predicate (defined by a closed formula [see Equation 12.3]). Unary cyclicity predicate c_n records information about the cyclicity properties of individual nodes — namely, $c_n(v)$ records whether node v lies on a cycle; nullary cyclicity predicate c_0 records a property of the heap as a whole — namely, whether the heap contains *any* cycle.
- Binary instrumentation predicates record relationships between unbounded sets of objects. For example, the instrumentation predicate $t[n](v_1, v_2) \stackrel{\text{def}}{=} n^+(v_1, v_2)$ records the existence of n -paths from v_1 to v_2 .

Moreover, instrumentation predicates that are unary can also be used as abstraction predicates.

In Section 12.3.4, we saw how it is possible to change the shape abstraction in use by changing the set of instrumentation predicates in use and/or by changing which unary instrumentation predicates are used as abstraction predicates. By using the right collection of instrumentation predicates and abstraction predicates, shape-analysis algorithms can be created that, in many cases, determine precise shape information for programs that manipulate several (possibly cyclic) data structures simultaneously. The information obtained is more precise than that obtained from previous work on shape analysis.

In Section 12.5, several other instrumentation predicates are introduced that augment shape descriptors with auxiliary information that permits flow-dependence information to be read off from the results of shape analysis.

12.4.6 Abstract Interpretation of Program Statements

The goal of a shape-analysis algorithm is to associate with each vertex v of control-flow graph G , a finite set of 3-valued structures that “describes” all of the 2-valued structures that can arise at v (and possibly more). The abstract semantics can be expressed as the least fixed point (in terms of set inclusion) of a system of equations over variables that correspond to vertices in the program. The right-hand side of each equation is a transformer that represents the abstract semantics for an individual statement in the program.

The most complex issue we face is the definition of the abstract semantics of program statements. This abstract semantics has to (a) be conservative, that is, must account for every possible runtime situation, and (b) should not yield too many “unknown” values.

The fact that the concrete semantics of statements can be expressed via logical formulas (Observation 12.2), together with the fact that the evaluation of a formula φ in a 3-valued structure S is guaranteed to be safe with respect to the evaluation of φ in any 2-valued structure that S represents (the embedding theorem), means that one abstract semantics falls out automatically from the concrete semantics. One merely has to evaluate the predicate-update formulas of the concrete semantics on 3-valued structures.

Observation 12.4 (Reinterpretation Principle). *Evaluation of the predicate-update formulas for a statement st in 2-valued logic captures the transfer function for st of the concrete semantics. Evaluation of the same formulas in 3-valued logic captures a sound transfer function for st of the abstract semantics.*

If st is a statement, $\llbracket st \rrbracket_3$ denotes the transformation on 3-valued structures that is defined by evaluating in 3-valued logic the predicate-update formulas that represent the concrete semantics of st .

Figure 12.14 combines Figures 12.11 and 12.12 (see column 2 and row 1, respectively, of Figure 12.14). Column 4 of Figure 12.14 illustrates how the predicate-update formulas that express the concrete semantics for $y = y -> n$ also express a transformation on 3-valued logical structures — that is, an abstract semantics — that is safe with respect to the concrete semantics (cf. $S_a^\natural \rightarrow S_b^\natural$ versus $S_a \rightarrow S_b$).⁹

As we will see, this approach has a number of good properties:

- Because the number of elements in the 3-valued structures that we work with is bounded, the abstract-interpretation process always terminates.
- The embedding theorem implies that the results obtained are conservative.
- By defining appropriate instrumentation predicates, it is possible to emulate some previous shape-analysis algorithms (e.g., [8, 25, 30, 33]).¹⁰

Unfortunately, there is also bad news: the method described above and illustrated in Figure 12.14 can be very imprecise. For instance, the statement $y = y -> n$ illustrated in Figure 12.14 sets y to the value of $y -> n$; that is, it makes y point to the next element in the list. In the abstract semantics, the evaluation in structure S_a of the predicate-update formula $y'(v) = \exists v_1 : y(v_1) \wedge n(v_1, v)$ causes $\iota^{S_b}(y)(u_{234})$ to be set to $1/2$. When $\exists v_1 : y(v_1) \wedge n(v_1, v)$ is evaluated in S_a , we have $\iota^{S_a}(y)(u_1) \wedge \iota^{S_a}(n)(u_1, u_{234}) = 1 \wedge 1/2 = 1/2$.

⁹The abstraction of S_b^\natural , as described in Section 12.4.4, is S_c . Figure 12.14 illustrates that in the abstract semantics we also work with structures that are even further “blurred.” We say that S_c *embeds into* S_b ; u_1 in S_c maps to u_1 in S_b ; u_2 and u_{34} in S_c both map to u_{234} in S_b ; the n predicate of S_b is the truth-blurring quotient of n in S_c under this mapping.

Our notion of the 2-valued structures that a 3-valued structure represents is based on this more general notion of embedding [58]. Note that in Figure 12.13, S_2 can be embedded into S_3 ; thus, structure S_3 also represents the acyclic lists of length 2 that are pointed to by x .

¹⁰The discussion above ignores the fact that for every statement and condition in the program, we also need to define how to update each instrumentation predicate p . That is, if p is defined by φ_p , an update formula is needed for transformation $\llbracket st \rrbracket_3(S)$ to produce an appropriate set of values for predicate p .

The simplest way is to reevaluate φ_p on the core predicates produced by $\llbracket st \rrbracket_3(S)$. In practice, however, this approach does not work very well because information will be lost under abstraction. As observed elsewhere [58], when working in 3-valued logic, Observation 12.3 implies that it is usually possible to retain more precision by defining a special *instrumentation-predicate maintenance formula*, $\mu_{p,st}(v_1, \dots, v_k)$, and evaluating $\mu_{p,st}(v_1, \dots, v_k)$ in structure S .

In [37, 50] algorithms are given that create an alternative predicate-maintenance formula $\mu_{p,st}$ for $p \in \mathcal{I}$ in terms of two *finite-differencing operators*, denoted by $\Delta_{st}^-[\cdot]$ and $\Delta_{st}^+[\cdot]$, which capture the negative and positive changes, respectively, that execution of statement st induces in an instrumentation predicate’s value. The formula $\mu_{p,st}$ is created by combining p with $\Delta_{st}^-[\varphi_p]$ and $\Delta_{st}^+[\varphi_p]$ as follows: $\mu_{p,st} = p \ ? \ \neg \Delta_{st}^-[\varphi_p] : \Delta_{st}^+[\varphi_p]$.

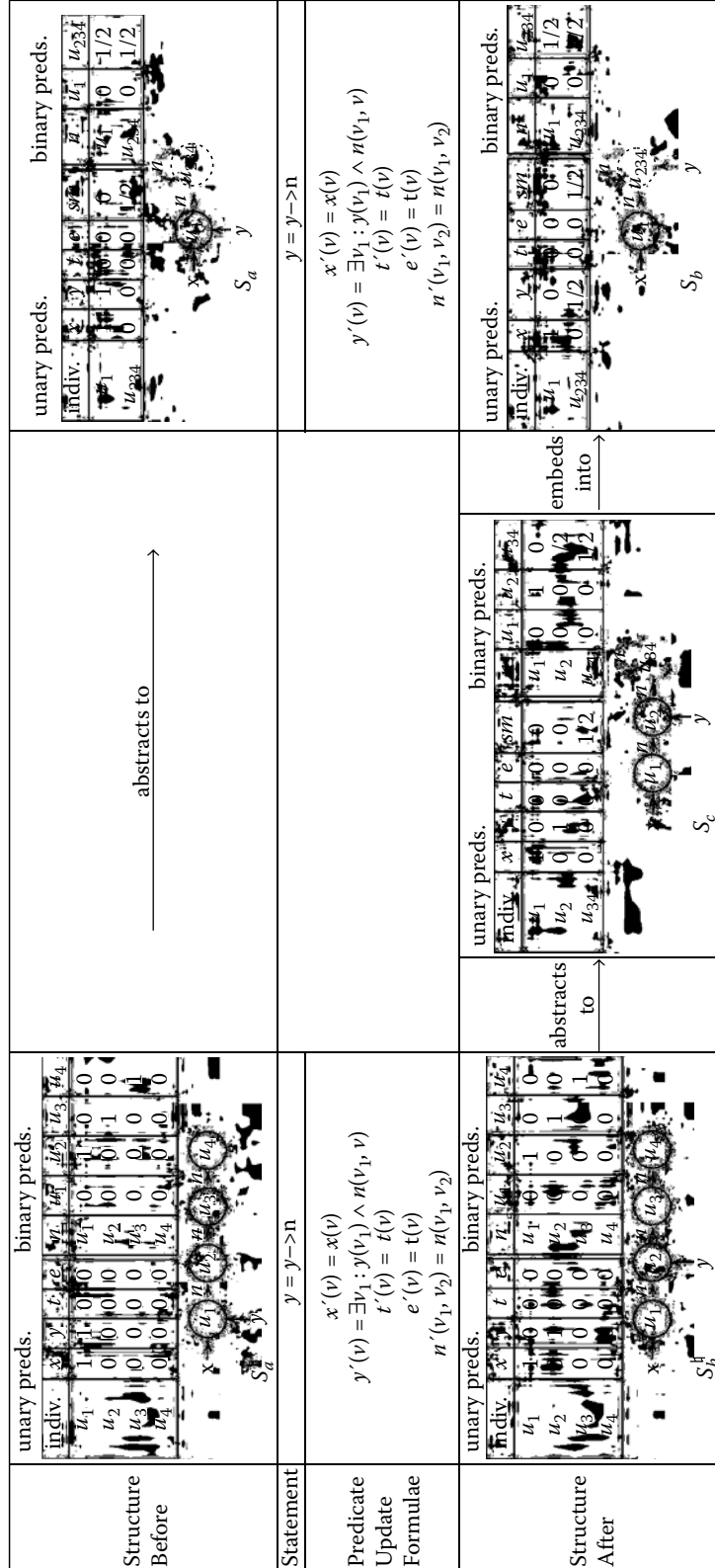


FIGURE 12.14 Commutative diagram that illustrates the relationships among (i) the transformation on 2-valued structures (defined by predicate-update formulas) that represents the concrete semantics for $y = y \rightarrow n$, (ii) abstraction, and (iii) the transformation on 3-valued structures (defined by the same predicate-update formulas) that represents the simple abstract semantics for $y = y \rightarrow n$ obtained via the reinterpretation principle (Observation 12.4). (In this example, $\{x, y, t, e\}$ -abstraction is used.)



Input Structure		
Update Formulas	$\phi_{y^{st_0}}^{st_0}(v)$	$\phi_{r_{n,y}^{st_0}}^{st_0}(v)$
	$\exists v_1 : y(v_1) \wedge n(v_1, v)$	$r_{y,n}(v) \wedge (c_n(v) \vee \neg y(v))$
Output Structure		

FIGURE 12.15 An application of the simplified abstract transformer for statement $st_0: y = y \rightarrow n$ in `insert`.

Consequently, all we can surmise after the execution of $y = y \rightarrow n$ is that y may point to one of the heap cells that summary node u_{234} represents (see S_b).

In contrast, the truth-blurring embedding of S_b^h is S_c ; thus, column 4 and row 4 of Figure 12.14 show that the abstract semantics obtained via Observation 12.4 can lead to a structure that is not as precise as what the abstract domain is capable of representing (cf. structures S_c and S_b).

As mentioned in Example 12.8, the use of reachability information is very important for retaining precision during shape analysis. However, even this mechanism is not sufficiently powerful to fix the problem. The same problem still occurs even if we use $\{x, y, t, e, is, r_{x,n}, r_{y,n}, r_{t,n}, r_{e,n}\}$ -abstraction with $\{c_n\}$. Figure 12.15 shows the result of applying the abstract semantics of the statement $st_0: y = y \rightarrow n$ to structure S_a — one of the 3-valued structures that arises in the analysis of `insert` just before y is advanced down the list by statement st_0 . Similar to what was illustrated in Figure 12.14, the resulting structure S_b shown in Figure 12.15 is not as precise as what the abstract domain is capable of representing. For instance, S_b does not contain a node that is definitely pointed to by y .

This imprecision leads to problems when a destructive update is performed. In particular, the first column in Table 12.3 shows what happens when the abstract transformers for the five statements that follow the search loop in `insert` are applied to S_b . Because $y(v)$ evaluates to $1/2$ for the summary node, we eventually reach the situation shown in the fourth row of structures, in which y, e, r_x, r_y, r_e, r_t , and is are all $1/2$ for the summary node. As a result, with the approach that has been described thus far, the abstract transformer for $y \rightarrow n = \tau$ sets the value of c_n for the summary node to $1/2$. Consequently, the analysis fails to determine that the structure returned by `insert` is an acyclic list.

In contrast, the analysis that uses the techniques described in the remainder of this section is able to determine that at the end of `insert` the following properties always hold: (a) x points to an acyclic list that has no shared elements, (b) y points into the tail of the x -list, and (c) the value of e and $y \rightarrow n$ are equal.

It is worthwhile to note that the precision problem becomes even more acute for shape-analysis algorithms that, like [8], do not explicitly track reachability properties. The reason is that, without reachability, S_b represents situations in which y points to an element that is not even part of the x -list.

12.4.6.1 Mechanisms for an Improved Abstract Semantics

The remainder of this section describes the main ideas behind two mechanisms that provide a more precise way of defining the abstract semantics of program statements. In particular, these mechanisms are able to “materialize” new nonsummary nodes from summary nodes as data structures are traversed. As we will see, these improvements allow us to determine more precise shape descriptors for the data structures that arise in the `insert` program.

In formulating an improved approach, our goal is to retain the property that the transformer for a program statement falls out automatically from the predicate-update formulas of the concrete semantics and the predicate-update formulas supplied for the instrumentation predicates. Thus, the main idea

TABLE 12.3 Selective applications of the abstract transformers using the one-stage and the multi-stage approaches, for the statements in `insert` that come after the search loop. (For brevity, r_z is used in place of $r_{z,n}$ for all variables z , and node names are not shown.)

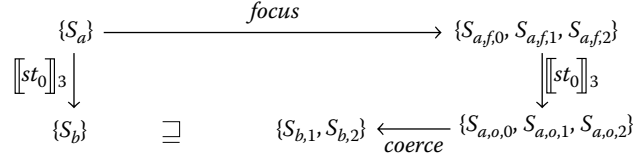
One-Stage		Multi-Stage	
$t = \text{malloc}(); t \rightarrow \text{data} = d;$			
$e = y \rightarrow n$			
$t \rightarrow n = e;$			
$y \rightarrow n = t;$			

behind the improved approach is to decompose the transformer for st into a composition of several functions, as depicted in Figure 12.16 and explained below, each of which falls out automatically from the predicate-update formulas of the concrete semantics and the predicate-update formulas supplied for the instrumentation predicates:

- The operation *focus* refines 3-valued structures so that the formulas that define the meaning of st evaluate to definite values. The *focus* operation thus brings these formulas “into focus.”
- The simple abstract meaning function for statement st , $\llbracket st \rrbracket_3$, is then applied.
- The operation *coerce* converts a 3-valued structure into a more precise 3-valued structure by removing certain kinds of inconsistencies.

(The 10 structures referred to in Figure 12.16 are depicted in Figure 12.17. Figure 12.17 will be used to explain the improved mechanisms that are presented in Sections 12.4.6.2 and 12.4.6.3.)

It is worth noting that both *focus* and *coerce* are *semantic-reduction* operations (a concept originally introduced in [12]). That is, they convert a set of 3-valued structures into a more precise set of 3-valued structures that describe the same set of stores. This property, together with the correctness of the structure transformer $\llbracket st \rrbracket_3$, guarantees that the overall multi-stage semantics is correct. In the context of a

FIGURE 12.16 One-stage vs. multi-stage abstract semantics for statement $st_0: y = y -> n$.

parametric framework for abstract interpretation, semantic reductions are valuable because they allow the transformers of the abstract semantics to be defined in the modular fashion shown in Figure 12.16.

12.4.6.2 The Focus Operation

The operation focus_F generates a set of structures on which a given set of formulas F have definite values for all assignments. (This operation will be denoted by focus when F is clear from the context or when we are referring to a focus operation for F in the generic sense.) The focus formulas used in shape analysis are determined from the left-hand side (as an L-value) and right-hand side (as an R-value) of each kind of statement in the programming language. These are illustrated in the following example.

Example 12.10

For the statement $st_0: y = y -> n$ in procedure `insert`, we focus on the formula

$$\varphi_0(v) \stackrel{\text{def}}{=} \exists v_1 : y(v_1) \wedge n(v_1, v) \quad (12.4)$$

Input Structure			
Focus Formulas	$\{\varphi_0(v)\}$, where $\varphi_0(v) \stackrel{\text{def}}{=} \exists v_1 : y(v_1) \wedge n(v_1, v)$		
Focused Structures			
Update Formulas		$\varphi_y^{st_0}(v)$ $\exists v_1 : y(v_1) \wedge n(v_1, v)$	$\varphi_{r,y}^{st_0}(v)$ $r_{y,n}(v) \wedge (c_n(v) \vee \neg y(v))$
Output Structures			
Coerced Structures			

FIGURE 12.17 The first application of the improved transformer for statement $st_0: y = y -> n$ in `insert`.

which corresponds to the R-value of the right-hand side of st_0 (the heap cell pointed to by $y \rightarrow n$). The upper part of Figure 12.17 illustrates the application of $focus_{\{\varphi_0\}}(S_a)$, where S_a is the structure shown in Figure 12.15 that occurs in `insert` just before the first application of statement $st_0: y = y \rightarrow n$. This results in three structures: $S_{a,f,0}$, $S_{a,f,1}$, and $S_{a,f,2}$:

- In $S_{a,f,0}$, $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,0}}([v \mapsto u])$ equals 0. This structure represents a situation in which the concrete list that x and y point to has only one element, but the store also contains garbage cells, represented by summary node u . (As we will see later, this structure is inconsistent because of the values of the $r_{x,n}$ and $r_{y,n}$ instrumentation predicates and will be eliminated from consideration by *coerce*.)
- In $S_{a,f,1}$, $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,1}}([v \mapsto u])$ equals 1. This covers the case where the list that x and y point to has exactly two elements. For all of the concrete cells that summary node u represents, φ_0 must evaluate to 1, so u must represent just a single list node.
- In $S_{a,f,2}$, $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,2}}([v \mapsto u.0])$ equals 0 and $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,2}}([v \mapsto u.1])$ equals 1. This covers the case where the list that x and y point to is a list of three or more elements. For all of the concrete cells that $u.0$ represents, φ_0 must evaluate to 0, and for all of the cells that $u.1$ represents, φ_0 must evaluate to 1. This case captures the essence of node materialization as described in [57]: individual u is bifurcated into two individuals.

The structures shown in Figure 12.17 are constructed by $focus_{\{\varphi_0\}}(S_a)$ by considering the reasons why $\llbracket \varphi_0 \rrbracket_3^{S_a}(Z)$ evaluates to $1/2$ for various assignments Z . In some cases, $\llbracket \varphi_0 \rrbracket_3^{S_a}(Z)$ already has a definite value; for instance, $\llbracket \varphi_0 \rrbracket_3^{S_a}([v \mapsto u_1])$ equals 0, and therefore φ_0 is already in focus at u_1 . In contrast, $\llbracket \varphi_0 \rrbracket_3^{S_a}([v \mapsto u])$ equals $1/2$. We can construct three (maximal) structures S from S_a in which $\llbracket \varphi_0 \rrbracket_3^S([v \mapsto u])$ has a definite value:

- $S_{a,f,0}$, in which $\iota^{S_{a,f,0}}(n)(u_1, u)$ is set to 0, and thus $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,0}}([v \mapsto u])$ equals 0.
- $S_{a,f,1}$, in which $\iota^{S_{a,f,1}}(n)(u_1, u)$ is set to 1, and thus $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,1}}([v \mapsto u])$ equals 1.
- $S_{a,f,2}$, in which u has been bifurcated into two different individuals, $u.0$ and $u.1$. In $S_{a,f,2}$, $\iota^{S_{a,f,2}}(n)(u_1, u.0)$ is set to 0, and thus $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,2}}([v \mapsto u.0])$ equals 0, whereas $\iota^{S_{a,f,2}}(n)(u_1, u.1)$ is set to 1, and thus $\llbracket \varphi_0 \rrbracket_3^{S_{a,f,2}}([v \mapsto u.1])$ equals 1.

An algorithm for *focus* that is based on these ideas is given in [58].

The greater the number of formulas on which we focus, the greater the number of distinctions that the shape-analysis algorithm can make, leading to improved precision. However, using a larger number of focus formulas can increase the number of structures that arise, thereby increasing the cost of analysis. Our preliminary experience indicates that in shape analysis there is a simple way to define the formulas on which to focus that guarantees that the number of structures generated grows only by a constant factor. The main idea is that in a statement of the form $lhs = rhs$, we only focus on formulas that define the heap cells for the L-value of lhs and the R-value of rhs . Focusing on L-values and R-values ensures that the application of the abstract transformer does not set to $1/2$ the entries of core predicates that correspond to pointer variables and fields that are updated by the statement. This approach extends naturally to program conditions and to statements that manipulate multiple L-values and R-values.

For our simplified language and type `List`, the target formulas on which to focus can be defined as shown in Table 12.4. Let us examine a few of the cases from Table 12.4:

- For the statement $x = \text{NULL}$, the set of target formulas is the empty set because neither the *lhs* L-value nor the *rhs* R-value is a heap cell.
- For the statement $x = t \rightarrow n$, the set of target formulas is the singleton set $\{\exists v_1 : t(v_1) \wedge n(v_1, v)\}$ because the *lhs* L-value cannot be a heap cell, and the *rhs* R-value is the cell pointed to by $t \rightarrow n$.
- For the statement $x \rightarrow n = t$, the set of target formulas is the set $\{x(v), t(v)\}$ because the *lhs* L-value is the heap cell pointed to by x , and the *rhs* R-value is the heap cell pointed to by t .
- For the condition $x == t$, the set of target formulas is the set $\{x(v), t(v)\}$; the R-values of the two sides of the conditional expression are the heap cells pointed to by x and t .

TABLE 12.4 The target formulas for *focus*, for statements and conditions of a program that uses type `List`

st	Focus Formulae
$x = \text{NULL}$	\emptyset
$x = t$	$\{t(v)\}$
$x = t \rightarrow n$	$\{\exists v_1 : t(v_1) \wedge n(v_1, v)\}$
$x \rightarrow n = t$	$\{x(v), t(v)\}$
$x = \text{malloc}()$	\emptyset
$x == \text{NULL}$	$\{x(v)\}$
$x != \text{NULL}$	$\{x(v)\}$
$x == t$	$\{x(v), t(v)\}$
$x != t$	$\{x(v), t(v)\}$
UninterpretedCondition	\emptyset

12.4.6.3 The Coerce Operation

The operation *coerce* converts a 3-valued structure into a more precise 3-valued structure by removing certain kinds of inconsistencies. The need for *coerce* can be motivated by the following example:

Example 12.11

After *focus*, the simple transformer $\llbracket st \rrbracket_3$ is applied to each of the structures produced. For instance, in Example 12.10, $\llbracket st_0 \rrbracket_3$ is applied to structures $S_{a,f,0}$, $S_{a,f,1}$, and $S_{a,f,2}$ to obtain structures $S_{a,o,0}$, $S_{a,o,1}$, and $S_{a,o,2}$, respectively (see Figure 12.17).

However, this process can produce structures that are not as precise as we would like. The intuitive reason for this state of affairs is that there can be interdependences between different properties stored in a structure, and these interdependences are not necessarily incorporated in the definitions of the predicate-update formulas. In particular, consider structure $S_{a,o,2}$. In this structure, the n -field of $u.0$ can point to $u.1$, which suggests that y may be pointing to a heap-shared cell. However, this is incompatible with the fact that $\iota(is)(u.1) = 0$ (i.e., $u.1$ cannot represent a heap-shared cell) and the fact that $\iota(n)(u_1, u.1) = 1$ (i.e., it is known that $u.1$ definitely has an incoming n -edge from a cell other than $u.0$).

Also, the structure $S_{a,o,0}$ describes an impossible situation: $\iota(r_{y,n})(u) = 1$ and yet u is not reachable — or even potentially reachable — from a heap cell that is pointed to by y .

The *coerce* mechanism is a systematic method that captures interdependences among the properties stored in 3-valued structures; *coerce* removes indefinite values that violate certain consistency rules, thereby “sharpening” the structures that arise during shape analysis. This remedies the imprecision illustrated in Example 12.11. In particular, when the sharpening process is applied to structure $S_{a,o,2}$ from Figure 12.17, the structure that results is $S_{b,2}$. In this case, the sharpening process discovers that (a) two of the n -edges with value $1/2$ can be removed from $S_{a,o,2}$ and (b) individual $u.1$ can only ever represent a single individual in each of the structures that $S_{a,o,2}$ represents, and hence $u.1$ should not be labeled as a summary node. These facts are not something that the mechanisms that have been described in earlier sections are capable of discovering. Also, the structure $S_{a,o,0}$ is discarded by the sharpening process.

The sharpening mechanism that *coerce* provides is crucial to the success of the improved shape-analysis framework because it allows a more accurate job of materialization to be performed than would otherwise be possible. For instance, note how the sharpened structure, $S_{b,2}$, clearly represents an unshared list of length 3 or more that is pointed to by x and whose second element is pointed to by y . In fact, in the domain of $\{x, y, t, e, is, r_{x,n}, r_{y,n}, r_{t,n}, r_{e,n}\}$ -abstraction with $\{c_n\}$, $S_{b,2}$ is the most precise representation possible for the family of unshared lists of length 3 or more that are pointed to by x and whose second element is pointed to by y . Without the sharpening mechanism, instantiations of the framework would rarely be

able to determine such things as “The data structure being manipulated by a certain list-manipulation program is actually a list.”

The *coerce* operation is based on the observation that 3-valued structures obey certain consistency rules that are a consequence of truth-blurring embedding. These consistency rules can be formalized as a system of “compatibility constraints.” Moreover, the constraint system can be obtained automatically from formulas that express certain global invariants on concrete stores.

Example 12.12

Consider a 2-valued structure S^\sharp that can be embedded in a 3-valued structure S , and suppose that the formula φ_{is} for “inferring” whether an individual u is shared evaluates to 1 in S (i.e., $\llbracket \varphi_{is}(v) \rrbracket_3^S([v \mapsto u]) = 1$). By the embedding theorem, $\iota^{S^\sharp}(is)(u^\sharp)$ must be 1 for any individual $u^\sharp \in U^{S^\sharp}$ that the embedding function maps to u .

Now consider a structure S' that is equal to S except that $\iota^{S'}(is)(u)$ is $1/2$. S^\sharp can also be embedded in S' . However, the embedding of S^\sharp in S is a “better” embedding — one that preserves more definite values. This has operational significance: it is needlessly imprecise to work with structure S' in which $\iota^{S'}(is)(u)$ has the value $1/2$; instead, we should discard S' and work with S . In general, the “stored predicate” is should be at least as precise as its inferred value; consequently, if it happens that φ_{is} evaluates to a definite value (1 or 0) in a 3-valued structure, we can sharpen the stored predicate is .

Similar reasoning allows us to determine, in some cases, that a structure is inconsistent. In $S_{a,o,0}$, for instance, $\varphi_{r_{y,n}}(u) = 0$, whereas the value stored in S for $r_{y,n}$, namely $\iota^{S_{a,o,0}}(r_{y,n})(u)$, is 1; consequently, $S_{a,o,0}$ is a 3-valued structure that does not represent any concrete structures at all. Structure $S_{a,o,0}$ can therefore be eliminated from further consideration by the shape-analysis algorithm.

This reasoning applies to all instrumentation predicates, not just is and $r_{y,n}$, and to both of the definite values, 0 and 1.

The reasoning used in Example 12.12 can be summarized as the following principle:

Observation 12.5 (The Sharpening Principle). *In any structure S , the value stored for $\iota^S(p)(u_1, \dots, u_k)$ should be at least as precise as the value of p ’s defining formula, φ_p , evaluated at u_1, \dots, u_k (i.e., $\llbracket \varphi_p \rrbracket_3^S([v_1 \mapsto u_1, \dots, v_k \mapsto u_k])$). Furthermore, if $\iota^S(p)(u_1, \dots, u_k)$ has a definite value and φ_p evaluates to an incompatible definite value, then S is a 3-valued structure that does not represent any concrete structures at all.*

This observation can be formalized in terms of *compatibility constraints*, defined as follows:

Definition 12.2 A **compatibility constraint** is a term of the form $\varphi_1 \triangleright \varphi_2$, where φ_1 is an arbitrary 3-valued formula, and φ_2 is either an atomic formula or the negation of an atomic formula over distinct logical variables. We say that a 3-valued structure S and an assignment Z **satisfy** $\varphi_1 \triangleright \varphi_2$ if, whenever Z is an assignment such that $\llbracket \varphi_1 \rrbracket_3^S(Z) = 1$, we also have $\llbracket \varphi_2 \rrbracket_3^S(Z) = 1$. (If $\llbracket \varphi_1 \rrbracket_3^S(Z)$ equals 0 or $1/2$, S and Z satisfy $\varphi_1 \triangleright \varphi_2$, regardless of the value of $\llbracket \varphi_2 \rrbracket_3^S(Z)$.)

The compatibility constraint that captures the reasoning used in Example 12.12 is $\varphi_{is}(v) \triangleright is(v)$. That is, when φ_{is} evaluates to 1 at u , then is must evaluate to 1 at u to satisfy the constraint. The compatibility constraint used to capture the similar case of sharpening $\iota(is)(u)$ from $1/2$ to 0 is $\neg \varphi_{is}(v) \triangleright \neg is(v)$.

Compatibility constraints can be generated automatically from formulas that express certain global invariants on concrete stores. We call such formulas *compatibility formulas*. There are two sources of compatibility formulas:

- The formulas that define the instrumentation predicates
- Additional formulas that formalize the properties of stores that are compatible with the semantics of C (i.e., with our encoding of C stores as 2-valued logical structures)

The following definition supplies a way to convert formulas into compatibility constraints:

Definition 12.3 *Let φ be a closed formula and a be an atomic formula such that (a) a contains no repetitions of logical variables, and (b) $a \not\equiv \text{sm}(v)$. Then the **compatibility constraint generated from φ** is defined as follows:*

$$\varphi_1 \triangleright a \quad \text{if } \varphi \equiv \forall v_1, \dots, v_k : (\varphi_1 \Rightarrow a) \quad (12.5)$$

$$\varphi_1 \triangleright \neg a \quad \text{if } \varphi \equiv \forall v_1, \dots, v_k : (\varphi_1 \Rightarrow \neg a) \quad (12.6)$$

The intuition behind Equations 12.5 and 12.6 is that for an atomic predicate, a truth-blurring embedding is forced to yield 1/2 only in cases in which a evaluates to 1 on one tuple of values for v_1, \dots, v_k but evaluates to 0 on a different tuple of values. In this case, the left-hand side will evaluate to 1/2 as well.

Our first source of compatibility formulas is the set of formulas that define the instrumentation predicates. For every instrumentation predicate $p \in \mathcal{I}$ defined by a formula $\varphi_p(v_1, \dots, v_k)$, we generate a compatibility formula of the following form:

$$\forall v_1, \dots, v_k : \varphi_p(v_1, \dots, v_k) \Leftrightarrow p(v_1, \dots, v_k) \quad (12.7)$$

So that we can apply Definition 12.3, this is then broken into two implications:

$$\forall v_1, \dots, v_k : \varphi_p(v_1, \dots, v_k) \Rightarrow p(v_1, \dots, v_k) \quad (12.8)$$

$$\forall v_1, \dots, v_k : \neg \varphi_p(v_1, \dots, v_k) \Rightarrow \neg p(v_1, \dots, v_k) \quad (12.9)$$

For instance, for each program variable x , we have the defining formula of instrumentation predicate $r_{x,n}$:

$$\varphi_{r_{x,n}}(v) \stackrel{\text{def}}{=} x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) \quad (12.10)$$

and thus

$$\forall v : x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) \Leftrightarrow r_{x,n}(v) \quad (12.11)$$

which is then broken into

$$\forall v : x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) \Rightarrow r_{x,n}(v) \quad (12.12)$$

$$\forall v : \neg(x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) \Rightarrow \neg r_{x,n}(v) \quad (12.13)$$

We then use Definition 12.3 to generate the following compatibility constraints:

$$x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v) \triangleright r_{x,n}(v) \quad (12.14)$$

$$\neg(x(v) \vee \exists v_1 : x(v_1) \wedge n^+(v_1, v)) \triangleright \neg r_{x,n}(v) \quad (12.15)$$

The constraint-generation rules defined in Definition 12.3 generate interesting constraints only for certain specific syntactic forms, namely implications with exactly one (possibly negated) predicate symbol on the right-hand side. Thus, when we generate compatibility constraints from formulas written as implications (such as Equations 12.12 and 12.13 and those in Table 12.5), the set of constraints generated depends on the form in which the compatibility formulas are written. However, not all of the many equivalent forms possible for a given compatibility formula lead to useful constraints. For instance, when Definition 12.3 is applied to the formula $\forall v_1, \dots, v_k : (\varphi_1 \Rightarrow a)$, it generates the constraint $\varphi_1 \triangleright a$; however, Definition 12.3 does not generate a constraint for the equivalent formula $\forall v_1, \dots, v_k : (\neg \varphi_1 \vee a)$.

This phenomenon can prevent an instantiation of the shape-analysis framework from having a suitable compatibility constraint at its disposal that would otherwise allow it to sharpen or discard a structure that arises during the analysis — and hence can lead to a shape-analysis algorithm that is more conservative than we would like.

TABLE 12.5 The formulas listed above the line are compatibility formulas for structures that represent a store of a C program that operates on values of the type `List` defined in Figure 12.1(a). The corresponding compatibility constraints are listed below the line.

	$\neg \exists v : sm(v)$	(12.22)
for each $x \in PVar, \forall v_1, v_2 : x(v_1) \wedge x(v_2)$	$\Rightarrow v_1 = v_2$	(12.23)
$\forall v_1, v_2 : (\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2))$	$\Rightarrow v_1 = v_2$	(12.24)
$(\exists v : sm(v))$	$\triangleright 0$	(12.25)
for each $x \in PVar, x(v_1) \wedge x(v_2)$	$\triangleright v_1 = v_2$	(12.26)
$(\exists v_3 : n(v_3, v_1) \wedge n(v_3, v_2))$	$\triangleright v_1 = v_2$	(12.27)

The way around this difficulty is to augment the constraint-generation process to generate constraints for some of the logical consequences of each compatibility formula:

Example 12.13

The defining formula for instrumentation predicate *is* is

$$\varphi_{is}(v) \stackrel{\text{def}}{=} \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2 \quad (12.16)$$

We obtain the following formula from Equation 12.16:

$$\forall v : (\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Leftrightarrow is(v) \quad (12.17)$$

which is broken into the two formulas

$$\forall v : (\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Rightarrow is(v) \quad (12.18)$$

$$\forall v : \neg(\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \Rightarrow \neg is(v) \quad (12.19)$$

By rewriting the implication in Equation 12.18 as a disjunction and then applying De Morgan's laws, we have

$$\forall v, v_1, v_2 : \neg n(v_1, v) \vee \neg n(v_2, v) \vee v_1 = v_2 \vee is(v) \quad (12.20)$$

One of the logical consequences of Equation 12.20 is

$$\forall v, v_2 : (\exists v_1 : n(v_1, v) \wedge v_1 \neq v_2 \wedge \neg is(v)) \Rightarrow \neg n(v_2, v) \quad (12.21)$$

from which we obtain the following compatibility constraint:

$$(\exists v_1 : n(v_1, v) \wedge v_1 \neq v_2 \wedge \neg is(v)) \triangleright \neg n(v_2, v) \quad (12.22)$$

(In addition to Equation 12.22, we obtain a number of other compatibility constraints from other logical consequences of Equation 12.20 [58].)

As we will see shortly, Equation 12.22 allows a more accurate job of materialization to be performed than would otherwise be possible: When $is(u)$ is 0 and one incoming n -edge to u is 1, to satisfy Equation 12.22 a second incoming n -edge to u cannot have the value 1/2. It must have the value 0; that is, the latter edge cannot exist (cf. Examples 12.11 and 12.15). This allows edges to be removed (safely) that a more naive materialization process would retain (cf. structures $S_{a,o,2}$ and $S_{b,2}$ in Figure 12.17), and permits the improved shape-analysis algorithm to generate more precise structures for `insert` than the ones generated by the simple shape-analysis algorithm sketched at the beginning of Section 12.4.6.

12.4.6.3.1 Compatibility Constraints from Hygiene Conditions

Our second source of compatibility formulas stems from the fact that not all structures $S^\natural \in \text{STRUCT}[\mathcal{P}]$ represent stores that are compatible with the semantics of C. For example, stores have the property that each pointer variable points to at most one element in heap-allocated storage.

Example 12.14

The set of formulas listed above the line in Table 12.5 is a set of compatibility formulas that must be satisfied for a structure to represent a store of a C program that operates on values of the type `List` defined in Figure 12.1a. Equation 12.23 captures the condition that concrete stores never contain any summary nodes. Equation 12.24 captures the fact that every program variable points to at most one list element. Equation 12.25 captures a similar property of the `n`-fields of `List` structures: whenever the `n`-field of a list element is non-`NULL`, it points to at most one list element. The corresponding compatibility constraints generated according to Definition 12.3 are listed below the line.

12.4.6.3.2 An Example of Coerce in Action

We are now ready to show how the *coerce* operation uses these compatibility constraints to either sharpen or discard a 3-valued logical structure. The *coerce* operation is a constraint-satisfaction procedure that repeatedly searches a structure S for assignments Z that fail to satisfy $\varphi_1 \triangleright \varphi_2$ (i.e., $\llbracket \varphi_1 \rrbracket_3^S(Z) = 1$ but $\llbracket \varphi_2 \rrbracket_3^S(Z) \neq 1$). This is used to improve the precision of shape analysis by (a) sharpening the values of predicates stored in S when the constraint violation is repairable, and (b) eliminating S from further consideration when the constraint violation is irreparable. (An algorithm for this process is given in [58].)

Example 12.15

The application of *coerce* to the structures $S_{a,o,0}$, $S_{a,o,1}$, and $S_{a,o,2}$ yields $S_{b,1}$ and $S_{b,2}$, as shown in the bottom block of Figure 12.17:

- The structure $S_{a,o,0}$ is discarded because the violation of Equation 12.15 is irreparable.
- The structure $S_{b,1}$ was obtained from $S_{a,o,1}$ by removing incompatibilities as follows:
 - Consider the assignment $[v \mapsto u, v_1 \mapsto u_1, v_2 \mapsto u]$. Because $\iota(n)(u_1, u) = 1$, $u_1 \neq u$, and $\iota(is)(u) = 0$, Equation 12.22 implies that $\iota(n)(u, u)$ must equal 0. Thus, in $S_{b,1}$ the (indefinite) `n`-edge from u to u has been removed.
 - Consider the assignment $[v_1 \mapsto u, v_2 \mapsto u]$. Because $\iota(y)(u) = 1$, Equation 12.28 implies that $\llbracket v_1 = v_2 \rrbracket_3^{S_{b,1}}([v_1 \mapsto u, v_2 \mapsto u])$ must equal 1, which in turn means that $\iota^{S_{b,1}}(sm)(u)$ must equal 0. Thus, in $S_{b,1}$ u is no longer a summary node.
- The structure $S_{b,2}$ was obtained from $S_{a,o,2}$ by removing incompatibilities as follows:
 - Consider the assignment $[v \mapsto u.1, v_1 \mapsto u_1, v_2 \mapsto u.0]$. Because $\iota(n)(u_1, u.1) = 1$, $u_1 \neq u.0$, and $\iota(is)(u.1) = 0$, Equation 12.22 implies that $\iota^{S_{b,2}}(n)(u.0, u.1)$ must equal 0. Thus, in $S_{b,2}$ the (indefinite) `n`-edge from $u.0$ to $u.1$ has been removed.
 - Consider the assignment $[v \mapsto u.1, v_1 \mapsto u_1, v_2 \mapsto u.1]$. Because $\iota(n)(u_1, u.1) = 1$, $u_1 \neq u.1$, and $\iota(is)(u.1) = 0$, Equation 12.22 implies that $\iota^{S_{b,2}}(n)(u.1, u.1)$ must equal 0. Thus, in $S_{b,2}$ the (indefinite) `n`-edge from $u.1$ to $u.1$ has been removed.
 - Consider the assignment $[v_1 \mapsto u.1, v_2 \mapsto u.1]$. Because $\iota(y)(u.1) = 1$, Equation 12.28 implies that $\llbracket v_1 = v_2 \rrbracket_3^{S_{b,2}}([v_1 \mapsto u.1, v_2 \mapsto u.1])$ must equal 1, which in turn means that $\iota^{S_{b,2}}(sm)(u.1)$ must equal 0. Thus, in $S_{b,2}$ $u.1$ is no longer a summary node.

Important differences between the structures $S_{b,1}$ and $S_{b,2}$ result from applying the multi-stage abstract transformer for statement $st_0 : y = y \rightarrow n$, compared with the structure S_b that results from applying the one-stage abstract transformer (see Figure 12.15). For instance, y points to a summary node in S_b , whereas y does not point to a summary node in either $S_{b,1}$ or $S_{b,2}$; as noted earlier, in the domain of $\{x, y, t, e, is, r_{x,n}, r_{y,n}, r_{t,n}, r_{e,n}\}$ -abstraction with $\{c_n\}$, $S_{b,2}$ is the most precise representation possible for

TABLE 12.6 The structures that occur before and after successive applications of the multi-stage abstract transformer for the statement $y = y - n$ during the abstract interpretation of `insert`. (For brevity, node names are shown.)

The image is a 10x2 grid of 20 black and white photographs, likely from a film strip, showing the progression of a forest fire. The photographs are arranged in two columns and ten rows. The left column shows the fire's initial spread and the dense smoke it produces. The right column shows the fire's progression across different areas of the landscape, including the burning of trees and the formation of large smoke plumes. The images are high-contrast, with the fire appearing as bright white and yellow against the dark background of the forest and smoke.

the family of unshared lists of length 3 or more that are pointed to by x and whose second element is pointed to by y .

Example 12.16

Table 12.6 shows the 3-valued structures that occur before and after applications of the abstract transformer for the statement $y = y \rightarrow n$ during the abstract interpretation of `insert`.

The material in Table 12.3 that appears under the heading “Multi-Stage” shows the application of the abstract transformers for the five statements that follow the search loop in `insert` to $S_{b,1}$ and $S_{b,2}$. For

space reasons, we do not show the abstract execution of these statements on the other structures shown in Table 12.6; however, the analysis is able to determine that at the end of `insert` the following properties always hold: (a) `x` points to an acyclic list that has no shared elements, (b) `y` points into the tail of the `x`-list, and (c) the value of `e` and `y->n` are equal. The identification of the latter condition is rather remarkable: the analysis is capable of showing that `e` and `y->n` are must-aliases at the end of `insert` (see also Section 12.5.1).

12.5 Applications

The algorithm sketched in Section 12.4 produces a set of 3-valued structures for each program point pt . This set provides a conservative representation, that is, it describes a superset of the set of concrete stores that can possibly occur in any execution of the program that ends at pt . Therefore, questions about the stores at pt can be answered (conservatively) by posing queries against the set of 3-valued structures that the shape-analysis algorithm associates with pt . The answers to these questions can be utilized in an optimizing compiler, as explained in Section 12.2. Furthermore, the fact that the shape-analysis framework is based on logic allows queries to be specified in a uniform way using logical formulas.

In this section, we discuss several kinds of questions. Section 12.5.1 discusses how instantiations of the parametric shape-analysis framework that have been described in previous sections can be applied to the problem of identifying may- and must-aliases. Section 12.5.2 shows that the shape-analysis framework can be instantiated to produce flow-dependence information for programs that manipulate linked data structures. Finally, Section 12.5.3 sketches some other applications for the results of shape analysis.

12.5.1 Identifying May- and Must-Aliases

We say that two pointer access paths, e_1 and e_2 , are *may-aliases* at a program point pt if there exists an execution sequence ending at pt that produces a store in which both e_1 and e_2 point to the same heap cell. We say that e_1 and e_2 are *must-aliases* at pt if, for every execution sequence ending at pt , e_1 and e_2 point to the same heap cell.¹¹

Consider the access paths $e_1 \equiv x \rightarrow f_1 \rightarrow \dots \rightarrow f_n$ and $e_2 \equiv x \rightarrow g_1 \rightarrow \dots \rightarrow g_m$. To extract aliasing information, we use the formula

$$al[e_1, e_2] \stackrel{\text{def}}{=} \exists v_0, \dots, v_n, w_0, \dots, w_m : \begin{aligned} & x(v_0) \wedge f_1(v_0, v_1) \wedge \dots \wedge f_n(v_{n-1}, v_n) \\ & \wedge y(w_0) \wedge g_1(w_0, w_1) \wedge \dots \wedge g_m(w_{m-1}, w_m) \\ & \wedge v_n = w_m \end{aligned} \quad (12.23)$$

If Equation 12.23 evaluates to 0 in every 3-valued structure that the shape-analysis algorithm associates with program point pt , we know that e_1 and e_2 are not may-aliases at pt . Similarly, when $al[e_1, e_2]$ evaluates to 1 in every such structure, we know that e_1 and e_2 are must-aliases at pt . In all other cases, e_1 and e_2 are considered may-aliases.

Note that in some cases, $al[e_1, e_2]$ may evaluate to 1/2, in which case e_1 and e_2 are considered may-aliases; this is a conservative result.

The answer can sometimes be improved by first applying *focus* with Equation 12.23. This will produce a set of structures in which $al[e_1, e_2]$ does not evaluate to an indefinite value. Finally, one can run *coerce* on the 3-valued structures produced by *focus* to eliminate infeasible 3-valued structures.

Example 12.17

Consider the 3-valued structure at the bottom right corner of Table 12.3. The formula $al[y \rightarrow n \rightarrow n, e]$ evaluates to 1 in this structure and in all of the other structures arising after $y \rightarrow n = t$; thus, $y \rightarrow n \rightarrow n$

¹¹Variants of these definitions can be defined that account for the case when e_1 or e_2 has the value NULL.

<pre> int y; List p, q; q = (List) malloc(); p = q; l₁: p->data = 5; l_{1.5}: l₂: y = q->data; </pre>	<pre> int y; List p, q, t; q = (List) malloc(); p = q; t = p; l₁: p->data = 5; l_{1.5}: t->data = 7; l₂: y = q->data; </pre>	<pre> int y; List p, q; q = (List) malloc(); p = q; l₁: p->data = 5; l_{1.5}: p = (List) malloc(); l₂: y = q->data; </pre>
(a)	(b)	(c)

FIGURE 12.18 A motivating example to demonstrate the differences between may-aliases and flow dependences.

and e are must-aliases at this point. Also, $al[e \rightarrow n, y \rightarrow n]$ evaluates to 0 in this structure and in all of the other structures; thus, $e \rightarrow n$ and $y \rightarrow n$ are not may-aliases.

However, the formula $al[e \rightarrow n \rightarrow n, e \rightarrow n]$ evaluates to 1/2 in this structure. If we focus on the $al[e \rightarrow n \rightarrow n, e \rightarrow n]$ formula, we obtain several structures; in one of them, $e \rightarrow n$ points to a nonsummary node that has a definite n -edge to itself. This structure is eliminated by *coerce*. In all of the remaining structures, the formula $al[e \rightarrow n \rightarrow n, e \rightarrow n]$ evaluates to 0, which shows that $e \rightarrow n \rightarrow n$ and $e \rightarrow n$ are not may-aliases.

12.5.2 Constructing Program Dependences

This section shows how to use information obtained from shape analysis to construct program dependence graphs [19, 31, 46]. To see why the problem of computing flow dependences is nontrivial, consider the example program fragments shown in Figure 12.18. A formal definition of flow dependence is given in Definition 12.4; for the purposes of this discussion, a statement l_b *depends on* l_a if the value written to a resource in l_a is directly used at l_b , that is, without intervening writes to this resource.

A naive (and unsafe) criterion that one might use to identify flow dependences in Figure 12.18a would be to say that l_2 depends on l_1 if p and q can refer to the same location at l_2 (i.e., if p and q are may-aliases at l_2). In Figure 12.18a, this would correctly identify the flow dependence from l_1 to l_2 . The naive criterion sometimes identifies more flow dependences than we might like. In Figure 12.18b it would say there is a flow dependence from l_1 to l_2 , even though $l_{1.5}$ overwrites the location that p points to. The naive criterion is unsafe because it may miss dependences. In Figure 12.18c, there is a flow dependence from l_1 to l_2 ; this would be missed because statement $l_{1.5}$ overwrites p , and thus p and q are never may-aliases at l_2 .

One safe way to identify dependences in a program that uses heap-allocated storage is to introduce an abstract variable for each allocation site, use the results of a flow-insensitive points-to analysis [1, 18, 23, 61, 62] to determine a safe approximation of the variables that are possibly defined and possibly used at each program point, and then use a traditional algorithm for reaching definitions (where each allocation site is treated as a use of its associated variable).

In this section, we utilize the parametric shape-analysis framework to define an alternative, and much more precise, algorithm. This algorithm is based on an idea developed by Horwitz et al. [25].¹² They introduced an augmented semantics for the programming language; in addition to all of the normal

¹²An alternative approach would have been to use the Ross–Sagiv construction [56], which reduces the problem of computing program dependences to the problem of computing may-aliases, and then to apply the method of Section 12.5.1. The method presented in this section is a more direct construction for identifying program dependences and thereby provides a better demonstration of the utility of the parametric shape-analysis framework for this problem.

```

void Append()
List head, tail, temp;
l1: head = (List) malloc();
l2: scanf("%c", &head->data);
l3: head->n = NULL;
l4: tail = head;
l5: if (tail->data == 'x') goto l12;
l6: temp = (List) malloc();
l7: scanf("%c", &temp->data);
l8: temp->n = NULL;
l9: tail->n = temp;
l10: tail = tail->n;
l11: goto l5;
l12: printf("%c", head->data);
l13: printf("%c", tail->data);

```



FIGURE 12.19 A program that builds a list by appending elements to `tail`, and its flow-dependence graph.

aspects of the language's semantics, the augmented semantics also records information about the history of resource usage — in this case, “last-write” information — for each location in the store. As we will see, it is natural to record this extra information using additional core predicates. As discussed in more detail below, this instantiation of the shape-analysis framework creates an algorithm from which conservative dependence information can then be extracted.

The resulting algorithm is the most precise algorithm known for identifying the data dependences of programs that manipulate heap-allocated storage. In addition, it does not need the artificial concept of introducing an abstract variable for each allocation site. For example, Figure 12.19 shows a program that builds a list by destructively appending elements to `tail`, together with a graph that shows the flow dependences that the algorithm identifies.

The rest of this subsection is organized as follows: Dependences are discussed in Section 12.5.2.1. Predicates for recording history information are introduced in Section 12.5.2.2, which also illustrates the results obtained via this dependence-analysis method.

12.5.2.1 Program Dependences

Program dependences can be grouped into flow dependences (def-use), output dependences (def-def), and anti-dependences (use-def) [19, 31]. In this section, we focus on flow dependences between program statements. Other types can be handled in a similar fashion.

We allow programs to explicitly modify the store via assignments through pointers. Because of this, we phrase the definition of flow dependence in terms of memory locations rather than program variables [25].

Definition 12.4 (Flow Dependence). Consider labeled statements $l_i : st_i$ and $l_j : st_j$. We say that l_i *has a flow dependence on* l_j if there is an execution path along which st_j writes into a memory location, loc , that st_i reads, and there is no intervening write into loc .

Example 12.18

In the program fragment shown in Figure 12.18b, statement l_2 does not depend on l_1 because statement l_2 reads from a location that is last written at statement $l_{1.5}$. In Figure 12.18c, $l_{1.5}$ does not interrupt the dependence between l_2 and l_1 because it does not write into a location that is read by l_2 .

TABLE 12.7 Predicates for recording history information

Predicate	Intended Meaning
$lst_w_v[l, z]$	Program variable z was last written into by the statement at label l .
$lst_w_f[l, n](v)$	The n -field of list element v was last written into by the statement at label l .
$lst_w_f[l, d](v)$	The $data$ -field of list element v was last written into by the statement at label l .

Example 12.19

Consider the program and graph of flow dependences shown in Figure 12.19. Notice that l_{12} is flow dependent only on l_1 and l_2 , while l_{13} is flow dependent on l_2 , l_4 , l_7 , and l_{10} . This information could be used by a slicing tool to determine that the loop need not be executed in order to print `head->data` in l_{12} , or by an instruction scheduler to reschedule l_{12} to be executed any time after l_2 . Also, l_3 , l_8 , and l_{11} have no statements that are dependent on them, making them candidates for elimination.

Thus, even in this simple example, knowing the flow dependences would allow several code transformations.

12.5.2.2 Recording History Using Predicates

Table 12.7 shows the predicates that are introduced to implement the augmented semantics à la Horwitz et al. [25]. As indicated in the column labeled “Intended Meaning,” the intention is that these predicates will record the label of the program statement that last writes into a given memory location.

The predicate $lst_w_v[l, z]$ is similar to the one used in reaching-definitions analysis; it records that program variable z was last written into by the statement at label l . The other two predicates record, for each field of each list element, which statement last wrote into that location.

Table 12.8 shows the predicate-update formulas for recording which statement last wrote into a location. The definitions given in Table 12.8 would be used to augment the instantiation of the shape-analysis

TABLE 12.8 Predicate-update formulae for recording last-write information. Here rhs denotes an arbitrary expression.

Statement	Cond.	Predicate
$l_1: x = rhs$		$\varphi_{lst_w_v[l, x]}^{st} = 1$
	$l \neq l_1$	$\varphi_{lst_w_v[l, x]}^{st} = 0$
	$l \neq l_1, z \neq x$	$\varphi_{lst_w_v[l, z]}^{st} = lst_w_v[l, z]$
		$\varphi_{lst_w_f[l, n]}^{st}(v) = lst_w_f[l, n](v)$
		$\varphi_{lst_w_f[l, d]}^{st}(v) = lst_w_f[l, d](v)$
$l_1: x \rightarrow n = rhs$		$\varphi_{lst_w_v[l, z]}^{st} = lst_w_v[l, z]$
		$\varphi_{lst_w_f[l, n]}^{st}(v) = (lst_w_f[l, n](v) \wedge \neg x(v)) \vee x(v)$
	$l \neq l_1$	$\varphi_{lst_w_f[l, n]}^{st}(v) = lst_w_f[l, n](v) \wedge \neg x(v)$
		$\varphi_{lst_w_f[l, d]}^{st}(v) = lst_w_f[l, d](v)$
$l_1: x \rightarrow data = rhs$		$\varphi_{lst_w_v[l, z]}^{st} = lst_w_v[l, z]$
		$\varphi_{lst_w_f[l, n]}^{st}(v) = lst_w_f[l, n](v)$
		$\varphi_{lst_w_f[l, d]}^{st}(v) = lst_w_f[l, d](v) \wedge \neg x(v) \vee x(v)$
	$l \neq l_1$	$\varphi_{lst_w_f[l, d]}^{st}(v) = lst_w_f[l, d](v) \wedge \neg x(v)$

TABLE 12.9 Formulas that use the last-write information in the structures associated with statement at l_2 to identify flow dependences from statement l_1 . (In this table, c stands for any constant.)

Statement	Formula
$l_2: x = \text{NULL}$	0
$l_2: x = \text{malloc}()$	0
$l_2: x = y$	$\text{lst_w_v}[l_1, y]$
$l_2: x = y \rightarrow n$	$\text{lst_w_v}[l_1, y] \vee \exists v : y(v) \wedge \text{lst_w_f}[l_1, n](v)$
$l_2: x = y \rightarrow \text{data}$	$\text{lst_w_v}[l_1, y] \vee \exists v : y(v) \wedge \text{lst_w_f}[l_1, d](v)$
$l_2: x \rightarrow f = \text{NULL}$	$\text{lst_w_v}[l_1, x]$
$l_2: x \rightarrow f = y$	$\text{lst_w_v}[l_1, x] \vee \text{lst_w_v}[l_1, y]$
$l_2: x \rightarrow \text{data} = c$	$\text{lst_w_v}[l_1, x]$

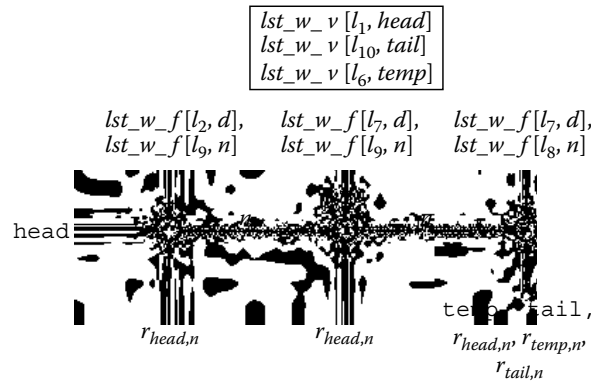


FIGURE 12.20 The most complex structure that the analysis yields at l_{12} .

framework that was described in Section 12.4. When the shape-analysis algorithm that we obtain in this way is applied to a program, it produces a set of 3-valued structures for each program point. Then, for each statement l_2 , to determine whether there is a flow dependence from l_1 to l_2 , each of the structures associated with l_2 is checked by evaluating the formulas from the appropriate line of Table 12.9.¹³ These formulas use the last-write information in the structures associated with l_2 to determine whether there is flow dependence from a statement l_1 . The idea behind Table 12.9 is that for each location accessed in the evaluation of l_2 's left-hand side (as an L-value) and l_2 's right-hand side (as an R-value), we need to check which statement last wrote into that location. If the formula is potentially satisfied by some 3-valued structure at l_2 , there is a flow dependence from l_1 to l_2 .

Example 12.20

Figure 12.20 shows one of the 3-valued structures that occurs at l_{12} when the program shown in Figure 12.19 is analyzed. (Three other structures arise at l_{12} ; these correspond to simpler configurations of memory than the one depicted.) The formula $\text{lst_w_v}[l_2, \text{head}] \vee \exists v : \text{head}(v) \wedge \text{lst_w_f}[l_2, d](v)$ evaluates to 1, which indicates that l_{12} depends on l_2 .

In contrast, the formula $\text{lst_w_v}[l_7, \text{head}] \vee \exists v : \text{head}(v) \wedge \text{lst_w_f}[l_7, d](v)$ evaluates to 0 (in this structure and in all of the other structures that arise at l_{12}). This allows us to conclude that l_{12} does not depend on l_7 .

¹³It is straightforward to provide similar formulas to extract flow dependences from the structures that the shape-analysis algorithm associates with program conditions.

12.5.3 Other Applications

12.5.3.1 Cleanness of Programs That Manipulate Linked Lists

Java programs are more predictable than C programs because the language assures that certain types of error conditions will be trapped, such as NULL-dereference errors. However, most Java compilers and Java Virtual Machine (JVM) implementations check these conditions at runtime, which slows the program down and does not provide any feedback to the programmer at compile time. In [14], it is shown how shape analysis can be used to detect the absence (and presence) of NULL dereferences and memory leaks in many C programs. It is also shown there that shape analysis yields much more precise information than what is obtained by a flow-sensitive points-to analysis, such as the one developed in [15]. If such methods could be implemented in Java compilers and/or a JVM, some runtime checks could be avoided.

12.5.3.2 Correctness of Sorting Implementations

In [35] a shape-analysis abstraction is developed to analyze programs that sort linked lists. It is shown that the analysis is precise enough to discover that (correct versions of) bubble-sort and insertion-sort procedures always produce correctly sorted lists as outputs and that the invariant “is-sorted” is maintained by list-manipulation operations such as merge. In addition, it is shown that when the analysis is applied to erroneous versions of bubble-sort and insertion-sort procedures, it is able to discover the error. In [37, 38] a novel abstraction-refinement method is defined, based on inductive logic programming, and successfully used to derive abstractions *automatically* that establish the partial correctness of several sorting algorithms. The derived abstractions are also used to establish that the algorithms possess additional properties, such as stability and antistability.

12.5.3.3 Conformance to API Specifications

[48] shows how to verify that client programs using a library conform to the library’s API specifications. In particular, an analysis is provided for verifying the absence of concurrent-modification exceptions in Java programs that use Java collections and iterators. In [68] separation and heterogeneous abstraction are used to scale the verification algorithms and to allow verification of larger programs that use libraries such as the Java Database Connectivity (JDBC) API.

12.5.3.4 Checking Multithreaded Systems

In [67] it is shown how to apply 3-valued logic to the problem of checking properties of multithreaded systems. In particular, [67] addresses the problem of state-space exploration for languages, such as Java, that allow (a) dynamic creation and destruction of an unbounded number of threads, (b) dynamic allocation and freeing of an unbounded number of storage cells from the heap, and (c) destructive updating of structure fields. This combination of features creates considerable difficulties for any method that tries to check program properties.

In this chapter, the problem of program analysis is expressed as a problem of annotating a control-flow graph with sets of 3-valued structures; in contrast, the analysis algorithm given in [67] builds and explores a 3-valued transition system on-the-fly.

In [67] problems (a) and (b) are handled essentially via the techniques developed in this chapter; problem (c) is addressed by reducing it to problem (b). Threads are modeled by individuals, which are abstracted using truth-blurring embedding — in this case, with respect to the collection of unary thread properties that hold for a given thread. This naming scheme automatically discovers commonalities in the state space, but without relying on explicitly supplied symmetry properties, as in, for example, [10, 16].

Unary core predicates are used to represent the program counter of each thread object; *focus* implements the interleaving of threads. The analysis described in [67] is capable of proving the absence of deadlock in a dining-philosophers program that permits there to be an unbounded number of philosophers.

In [70] this approach was applied to verify partial correctness of concurrent-queue implementations.

In [69] the above approach was extended to provide a general framework for proving temporal properties of programs by representing program traces as logical structures. A more efficient technique for proving

local temporal properties is presented in [60] and applied to compile-time garbage collection in Javacard programs. (While the aforementioned technique was developed as a verification technique, it can also be utilized to reduce synchronization overhead, e.g., [64].)

12.6 Extensions

The approach to shape analysis presented in this chapter has been implemented by T. Lev-Ami in a system called *TVLA* (three-valued-logic analyzer) [34, 36]. TVLA provides a language in which the user can specify (a) an operational semantics (via predicates and predicate-update formulas), (b) a control-flow graph for a program, and (c) a set of 3-valued structures that describe the program's input. Using this specification, TVLA builds the corresponding equation system and finds its least fixed point.

The experience gained from building TVLA led to a number of improvements to, and extensions of, the methods described in this chapter (and in [58]). The enhancements that TVLA incorporates include:

- The ability to declare that certain binary predicates specify *functional properties*.
- The ability to specify that structures should be stored only at nodes of the control-flow graph that are targets of backedges.
- An enhanced version of *coerce* that incorporates some methods that are similar in flavor to relational-database query-optimization techniques (cf. [63]).
- An enhanced *focus* algorithm that generalizes the methods of Section 12.4.6.2 to handle focusing on arbitrary formulas.¹⁴ In addition, this version of *focus* takes advantage of the properties of predicates that are specified to be functions.
- The ability to specify criteria for merging together structures associated with a program point. This feature is motivated by the idea that when the number of structures that arise at a given program point is too large, it may be better to create a smaller number of structures that represent at least the same set of 2-valued structures. In particular, nullary predicates (i.e., predicates of 0-arity) are used to specify which structures are to be merged together. For example, for linked lists, the “x-is-not-null” predicate, defined by the formula $nn[x]() = \exists v : x(v)$, discriminates between structures in which x points to a list element, and structures in which it does not. By using $nn[x]()$ as the criterion for whether to merge structures, the structures in which x is `NULL` are kept separate from those in which x points to an allocated memory cell.

Further details about these features can be found in [34].

The remainder of this section describes several other extensions of our parametric logic-based analysis framework that have been investigated; many of these extensions have also been incorporated into TVLA.

12.6.1 Interprocedural Analysis

Several papers have investigated interprocedural shape analysis. In [53] procedures are handled by explicitly representing stacks of activation records as linked lists, allowing rather precise analysis of (possibly recursive) procedures. In [29] procedures are handled by automatically creating summaries of their behavior. Abstractions of *two-vocabulary structures* are used to capture an over-approximation of the relation that describes the transformation effected by a procedure. In [52] a new concrete semantics for programs that manipulate heap-allocated storage is presented, which only passes “local” heaps to procedures. A simplified version of this semantics is used in [54] to perform more modular summarization by only representing reachable parts of the heap.

¹⁴The enhanced *focus* algorithm may not always succeed.

12.6.2 Computing Intersections of Abstractions

Arnold et al. [2] considers the problem of computing the intersection (meet) of heap abstractions, namely the greatest lower bound of two sets of 3-valued structures. This problem turns out to have many applications in program analysis, such as interpreting program conditions, refining abstract configurations, reasoning about procedures [29], and proving temporal properties of heap-manipulating programs, either via greatest-fixed-point approximation over trace semantics or in a staged manner over the collecting semantics. [2] describes a constructive formulation of meet that is based on finding certain relations between abstract heap objects. The enumeration of those relations is reduced to finding constrained matchings over bipartite graphs.

12.6.3 Efficient Heap Abstractions and Representations

Manevich et al. [39] addresses the problem of space consumption in first-order state representations by describing and evaluating two new representation techniques for logical structures. One technique uses ordered binary decision diagrams (OBDDs) [7]; the other uses a variant of a functional map data structure [44, 51]. The results show that both the OBDD and functional implementations reduce space consumption in TVLA by a factor of 4 to 10 relative to the original TVLA state representation, without compromising analysis time.

Manevich et al. [40] present a new heap abstraction that works by merging shape descriptors according to a partial isomorphism similarity criterion, resulting in a partially disjunctive abstraction. There it is also shown that on the existing TVLA examples, the abstract interpretation using this abstraction is drastically faster than the powerset heap abstraction, practically without a significant loss of precision.

Manevich et al. [41] provide a family of simple abstractions for potentially cyclic linked lists. In particular, it provides a relatively efficient predicate abstraction that allows verification of programs that manipulate potentially cyclic linked lists.

12.6.4 Abstracting Numeric Values

In this chapter, we ignore numeric values in programs, so the analysis would be imprecise for programs that perform numeric computations. [20] presents a generic solution for combining abstractions of numeric values and heap-allocated storage. This solution has been integrated into a version of TVLA. In [21] a new abstraction of numeric values is presented, which like canonical abstraction, tracks correlations between aggregates and not just indices. For example, it can identify loops that perform array-kills (i.e., assign values to an entire array). In [28] this approach has been generalized to define a family of abstractions (for relations as well as numeric quantities) that is more precise than pure canonical abstraction and allows the basic idea from [20] to be applied more widely.

12.6.5 Abstraction Refinement

The model-checking community has had much success with the notion of *automatic abstraction refinement*, in which an analyzer is started with a crude abstraction, and the results of analysis runs that fail to establish a definite answer (about whether the property of interest does or does not hold) are used as feedback about how the abstraction should be refined [4, 9, 32]. However, the abstract domains used in shape analysis are based on first-order logic, whereas model-checking tools that use abstraction refinement are based on predicate-abstraction domains [3, 22].

Abstraction-refinement methods suitable for use with shape-analysis domains were investigated in [37, 38]. The methods are based on inductive logic programming [47], which is a machine-learning technique for identifying general rules from a set of observed instances. In [37, 38] this was used to identify appropriate formulas that define new instrumentation relations (and thereby change the abstraction in use).

12.7 Related Work

The shape-analysis problem was first investigated by Reynolds [49], who studied it in the context of a Lisp-like language with no destructive updating. Reynolds treated the problem as one of simplifying a collection of set equations. A similar shape-analysis problem, but for an imperative language supporting nondestructive manipulation of heap-allocated objects, was formulated independently by Jones and Muchnick, who treated the problem as one of solving (i.e., finding the least fixed point of) a collection of equations using regular tree grammars [30].

Jones and Muchnick [30] also began the study of shape analysis for languages *with* destructive updating. To handle such languages, they formulated an analysis method that associates program points with sets of finite shape-graphs.¹⁵ To guarantee that the analysis terminates for programs containing loops, the Jones–Muchnick approach limits the length of acyclic selector paths by some chosen parameter k . All nodes beyond the “ k -horizon” are clustered into a summary node. The Jones–Muchnick formulation has two drawbacks:

- The analysis yields poor results for programs that manipulate cons-cells beyond the k -horizon. For example, in the list-reversal program of Figure 12.1, little useful information is obtained. The analysis algorithm must model what happens when the program is applied to lists of length greater than k . However, the tail of such a list is treated conservatively, as an arbitrary, and possibly cyclic, data structure.
- The analysis may be extremely costly because the number of possible shape-graphs is doubly exponential in k .

In addition to Jones and Muchnick’s work, k -limiting has also been used in a number of subsequent papers (e.g., Horwitz et al. [25]).

Another well-known shape-analysis algorithm, developed by Chase et al. [8], is based on the following ideas:

- Sharing information, in the form of abstract heap reference counts (0, 1, and ∞), is used to characterize shape-graphs that represent list structures.¹⁶
- Several heuristics are introduced to allow several shape-nodes to be maintained for each allocation site.
- For an assignment to $x \rightarrow n$, when the shape-node that x points to represents only concrete elements that will definitely be overwritten, the n -field of the shape-node that x points to can be overwritten (a so-called strong update).

The Chase–Wegman–Zadeck algorithm is able to identify list-preservation properties in some cases; for instance, it can determine that a program that appends a list to a list preserves “listness.” However, as noted in [8], allocation-site information alone is insufficient to determine interesting facts in many programs. For example, it cannot determine that “listness” is preserved for either the list-insert program or a list-reversal program that uses destructive-update operations. In particular, in the list-reversal program, the Chase–Wegman–Zadeck algorithm reports that a possibly cyclic structure may arise, and that the two lists used by the program might share cells in common (when in fact the two lists are always disjoint).

The parametric framework presented in this paper can be instantiated to implement the Chase–Wegman–Zadeck algorithm, as well as other shape-analysis algorithms. Furthermore, in Section 12.5.2, we presented a new algorithm for computing flow dependences using our parametric approach.

For additional discussion of related work, the reader is referred to [57, 58].

¹⁵In this section, we use the term *shape-graph* in the generic sense, meaning any finite graph structure used to approximate the shapes of runtime data structures.

¹⁶The idea of augmenting shape-graphs with sharing information also appears in the earlier work of Jones and Muchnick [30].

12.8 Conclusions

Many of the classical data-flow-analysis algorithms use bit vectors to represent the characteristic functions of set-valued data-flow values. This corresponds to a logical interpretation (in the abstract semantics) that uses two values. It is *definite* on one of the bit values and *conservative* on the other. That is, either “false” means “false” and “true” means “may be true/may be false,” or “true” means “true” and “false” means “may be true/may be false.” Many other static-analysis algorithms have a similar character.

Most static analyses have such a one-sided bias; exceptions include data-flow analyses that simultaneously track “may” and “must” information, for example, [6, 59].

The material presented in this chapter shows that while *indefiniteness* is inherent (i.e., a static analysis is unable, in general, to give a definite answer), one-sidedness is not. By basing the abstract semantics on 3-valued logic, definite truth and definite falseness can both be tracked, with the third value, $1/2$, capturing indefiniteness.

This outlook provides some insight into the true nature of the values that arise in other work on static analysis:

- A one-sided analysis that is precise with respect to “false” and conservative with respect to “true” is really a 3-valued analysis over 0, 1, and $1/2$ that conflates 1 and $1/2$ (and uses “true” in place of $1/2$).
- Likewise, an analysis that is precise with respect to “true” and conservative with respect to “false” is really a 3-valued analysis over 0, 1, and $1/2$ that conflates 0 and $1/2$ (and uses “false” in place of $1/2$).

In contrast, the analyses developed in this chapter are unbiased: They are precise with respect to both 0 and 1 and use $1/2$ to capture indefiniteness. Other work that uses 3-valued logic to develop unbiased analyses includes [26].

We hope the ideas presented in this chapter (and the TVLA system, which embodies these ideas) will help readers implement new static-analysis algorithms that identify interesting properties of programs that make use of heap-allocated data structures.

References

1. L. O. Andersen. 1994. Program analysis and specialization for the C programming language. Ph.D. dissertation, DIKU, University of Copenhagen (DIKU report 94/19).
2. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. 2006. Combining shape analyses by intersecting abstractions. In *Verification, Model Checking, and Abstract Interpretation*.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. 1998. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*.
4. T. Ball and S. K. Rajamani. 2000. Bebop: A symbolic model checker for Boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, 113–30. London: Springer-Verlag.
5. A. Bijlsma. 1999. A semantics for pointers. Talk at the Dagstuhl-Seminar on Program Analysis.
6. R. Bodik, R. Gupta, and M. L. Soffa. 1998. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*.
7. R. E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. In *IEEE Trans. Comput.* 6:677–91.
8. D. R. Chase, M. Wegman, and F. Zadeck. 1990. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Language Design and Implementation*.
9. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification*.
10. E. M. Clarke and S. Jha. 1995. Symmetry and induction in model checking. In *Computer Science Today: Recent Trends and Developments*, New York: Springer-Verlag.

11. F. Corbera, R. Asenjo, and E. L. Zapata. 1999. New shape analysis techniques for automatic parallelization of C codes. In *ACM International Conference on Supercomputing*, 220–27.
12. P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*.
13. N. Dor, M. Rodeh, and M. Sagiv. 1998. Detecting memory errors via static pointer analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*.
14. N. Dor, M. Rodeh, and M. Sagiv. 2000. Checking cleanness in linked lists. In *Static Analysis Symposium*.
15. M. Emami, R. Ghiya, and L. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*.
16. E. Emerson and A. P. Sistla. 1993. Symmetry and model checking. In *Computer Aided Verification*.
17. H. B. Enderton. 1972. *A mathematical introduction to logic*. New York: Academic Press.
18. M. Fähndrich, J. Foster, Z. Su, and A. Aiken. 1998. Partial online cycle elimination in inclusion constraint graphs. In *SIGPLAN Conference on Programming Language Design and Implementation*.
19. J. Ferrante, K. Ottenstein, and J. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 3(9):319–49.
20. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. 2004. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*.
21. D. Gopan, T. Reps, and M. Sagiv. 2005. A framework for numeric analysis of array operations. In *Symposium on Principles of Programming Language*.
22. S. Graf and H. Saïdi. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification*.
23. N. Heintze and O. Tardieu. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*.
24. L. Hendren, J. Hummel, and A. Nicolau. 1992. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conference on Programming Language Design and Implementation*.
25. S. Horwitz, P. Pfeiffer, and T. Reps. 1989. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Language Design and Implementation*.
26. M. Huth, R. Jagadeesan, and D. A. Schmidt. 2001. Modal transition systems: A foundation for three-valued program analysis. In *European Symposium on Programming*.
27. Y. S. Hwang and J. Saltz. 1997. Identifying def/use information of statements that construct and traverse dynamic recursive data structures. In *Languages and Compilers for Parallel Computing*, 131–45.
28. B. Jeannet, D. Gopan, and T. Reps. 2005. A relational abstraction for functions. In *Static Analysis Symposium*.
29. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. 2004. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium*.
30. N. D. Jones and S. S. Muchnick. 1981. Flow analysis and optimization of Lisp-like structures. In *Program flow analysis: Theory and applications*, ed. S. S. Muchnick and N. D. Jones, 102–31. Englewood Cliffs, NJ: Prentice-Hall.
31. D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. 1981. Dependence graphs and compiler optimizations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*.
32. R. Kurshan. 1994. *Computer-Aided Verification of Coordinating Processes*. Princeton, NJ: Princeton University Press.
33. J. R. Larus and P. N. Hilfinger. 1988. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Language Design and Implementation*.
34. T. Lev-Ami. 2000. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel.

35. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. 2000. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 26–38.
36. T. Lev-Ami and M. Sagiv. 2000. TVLA: A system for implementing static analyses. In *Static Analysis Symposium*, 280–301.
37. A. Loginov. 2006. Refinement-based program verification via three-valued-logic analysis. Ph.D. dissertation and Tech. Rep. TR-1574, Computer Science Department, University of Wisconsin, Madison.
38. A. Loginov, T. Reps, and M. Sagiv. 2005. Abstraction refinement via inductive learning. In *Computer Aided Verification*.
39. R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. 2002. Compactly representing first-order structures for static analysis. In *Static Analysis Symposium*.
40. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. 2004. Partially disjunctive heap abstraction. In *Static Analysis Symposium*.
41. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. 2005. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation*.
42. B. Möller. 1999. Calculating with acyclic and cyclic lists. *Inf. Sci.* 119(3-4):135–54.
43. J. M. Morris. 1982. Assignment and linked data structures. In *Theoretical foundations of programming methodology*, ed. M. Broy and G. Schmidt, 35–41. Boston: D. Reidel.
44. E. W. Myers. 1984. Efficient applicative data types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, 66–75.
45. G. Nelson. 1983. Verifying reachability invariants of linked structures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*.
46. K. J. Ottenstein and L. M. Ottenstein. 1984. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*.
47. J. R. Quinlan. 1990. Learning logical definitions from relations. *Machine Learn.* 5:239–66.
48. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. 2002. Deriving specialized program analyses for certifying component-client conformance. In *SIGPLAN Conference on Programming Language Design and Implementation*.
49. J. C. Reynolds. 1968. Automatic computation of data set definitions. In *Information Processing 68: Proceedings of the IFIP Congress*, 456–61. New York: North-Holland.
50. T. Reps, M. Sagiv, and A. Loginov. 2003. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming*.
51. T. Reps, T. Teitelbaum, and A. Demers. 1983. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program Lang. Syst.* 5(3):449–77.
52. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. 2005. A semantics for procedure local heaps and its abstractions. In *Symposium on Principles of Programming Language*.
53. N. Rinetzky and M. Sagiv. 2001. Interprocedural shape analysis for recursive programs. In *Proceedings of the 10th International Conference on Compiler Construction*, 133–49. London: Springer-Verlag.
54. N. Rinetzky, M. Sagiv, and E. Yahav. 2005. Interprocedural shape analysis for cutpoint-free programs. In *Static Analysis Symposium*.
55. J. L. Ross and M. Sagiv. 1998. Building a bridge between pointer aliases and program dependences. In *European Symposium on Programming*.
56. J. L. Ross and M. Sagiv. 1998. Building a bridge between pointer aliases and program dependences. *Nordic J. Comput.* 8:361–86.
57. M. Sagiv, T. Reps, and R. Wilhelm. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program Lang. Syst.* 20(1):1–50.
58. M. Sagiv, T. Reps, and R. Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program Lang. Syst.* 24(3):217–98.
59. S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. 1998. A logic-based approach to data flow analysis problems. *Acta Inf.* 35(6):457–504.

60. R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. 2003. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Static Analysis Symposium*.
61. M. Shapiro and S. Horwitz. 1997. Fast and accurate flow-insensitive points-to analysis. In *ACM SIGPLAN-SIGACT Symposium Principles of Programming Language*, 1–14.
62. B. Steensgaard. 1996. Points-to analysis in almost-linear time. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, 32–41.
63. J. D. Ullman. 1989. *Principles of database and knowledge-base systems*, Vol. II, *The new technologies*. Rockville, MD: Computer Science Press.
64. C. Ungureanu and S. Jagannathan. 2000. Concurrency analysis for Java. In *Static Analysis Symposium*.
65. R. Wilhelm, M. Sagiv, and T. Reps. 2000. Shape analysis. In *Compiler Construction*.
66. M. Wolfe and U. Banerjee. 1987. Data dependence and its application to parallel processing. *Int. J. Parallel Program.* 16(2):137–78.
67. E. Yahav. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symposium on Principles of Programming Language*.
68. E. Yahav and G. Ramalingam. 2004. Verifying safety properties using separation and heterogeneous abstractions. In *SIGPLAN Conference on Programming Language Design and Implementation*.
69. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. 2003. Verifying temporal heap properties specified via evolution logic. In *European Symposium on Programming*.
70. E. Yahav and M. Sagiv. 2003. Automatically verifying concurrent queue algorithms. In *Workshop on Software Model Checking*.