Shape Analysis

Reinhard Wilhelm[1], Mooly Sagiv[2], and Thomas Reps[3]

[1] Fachbereich Informatik, Universität des Saarlandes
[2] Department of Computer Science, Tel-Aviv University
[3] Computer Science Department, University of Wisconsin at Madison

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 or verify programs. They also contain information valuable for debugging, compile-time garbage collection, instruction scheduling, and parallelization.

1 Introduction

Pointers and anonymous objects stored in the heap seem to be the dark corner of imperative programming languages. There are only a few treatments of the semantics of pointers. Most semantic descriptions of imperative programming languages even assume the nonexistence of pointers, since otherwise the semantics of an assignment statement becomes much more complex. The reason is that an assignment through a pointer variable or pointer component may have far reaching side-effects.

These far reaching side-effects also make program dependence analysis harder, since 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.

Shape analysis is a generic term denoting static program-analysis techniques attempting to determine properties of the heap contents relevant for the applications mentioned above.

1.1 Structure of the Paper

The structure of the paper is as follows: In Section 2, a number of questions about the contents of the heap are listed. Subsection 2.1 presents a program that will be used as a running example, a program that destructively reverses a singly linked list. Subsection 2.2 shows how shape analysis would answer the questions about the heap contents produced by this program. Section 3 then introduces a parametric shape analysis along the lines of [23], which provides for 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. Throughout the paper, abstraction functions and transfer functions are given intuitively, by means of examples. Section 4 presents experience obtained via an implementation done in Java. Section 5 briefly discusses related work. Section 6 presents some conclusions.

2 Questions about the Heap Contents

Shape analysis has a somewhat constrained view of programs. It is not interested in numerical or string values that programs compute, but exclusively in the linked structures they build in the heap and in the pointers into the heap from the stack, from global memory, or from cells in the heap. We will therefore use the term *execution state* to mean the set of cells in the heap and their connectivity by pointer components of heap cells and the values of pointer variables in the store.

Among the questions about execution states that we might wish to pose at points in a program are:

NULL-pointers: May a pointer variable or a pointer component of a heap cell contain NULL? This is valuable debugging information at the entry of a statement attempting to dereference this pointer.

May-Alias: May two pointer expressions reference the same heap cell? The absence of may-aliasing can be used for improving program dependence information [3, 20].

Must-Alias: Will two pointer expressions always denote the same heap cell? This may be used to predict a cache hit or to trigger a prefetch.

Sharing: May a heap cell be shared?[1] There are many uses for such information. Explicit deallocation of a shared node may leave the storage manager in an inconsistent state (e.g., if the heap cell is deallocated twice – by calls on the deallocator at each predecessor). A nonshared cell may be explicitly deallocated when the last pointer to it ceases to exist. This again is valuable for debugging.

Reachability: Is a heap cell reachable from a specific variable or from any pointer variable? Unreachable cells are certainly garbage.

Disjointness: Will two data structures pointed to by two distinct pointer variables ever have common elements? Disjoint data structures may be processed in parallel by different processors and may be stored in different memories [8].

[1] Later on in the paper the term "shared" means "heap-shared" i.e., 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: May a heap cell be part of a cycle? If not, garbage collection could be done by reference counting.

Shape: What will be the "shape" of (some part of) the heap contents? Shapes (or, more precisely, shape descriptors) characterize data structures. A shape descriptor could indicate whether the heap contains a singly linked list, potentially with/definitely without a cycle, a doubly linked list, a binary tree, etc. Many of the properties listed above are ingredients of shape analyses, e.g., sharing, cyclicity, reachability, disjointness.
Shape analysis can be understood as an extended type analysis; its results can be used as an aid in program understanding, debugging [6], and verification.

2.1 An Example: Destructive Reverse

The following program, `reverse`, is used as a running example. It destructively reverses a list pointed to by `x` into one pointed to by `y`.

```
/* reverse.c */
#include "list.h"
List reverse(List x) {
  List y, t;
  y = NULL;
  while (x != NULL) {
    t = y;
    y = x;
    x = x->n;
    y->n = NULL;
    y->n = t;
  }
  return y;
}
```

It assumes the declaration of a `List` data type

```
/* list.h */
typedef struct node {
  struct node *n;
  int data;
} *List
```

The control flow graph of `reverse` is reproduced in Fig. 1. The program points are labeled for ease of reference. The body of the loop does the following: Pointer variable `t` holds on to the current `y`-list (n_3), while `y`

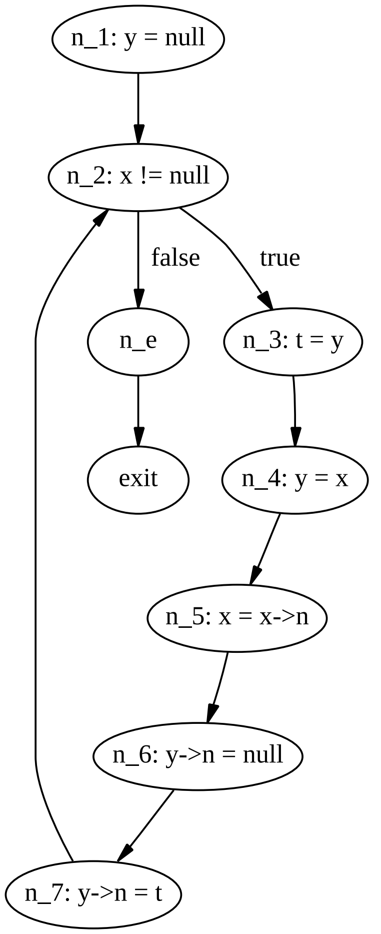


Fig. 1. Flow graph of a list-reversal program

grabs the head of the current x-list (n_4). x is then moved to the tail of the current x-list (n_5). y->n is first nulllified (n_6) and then made to connect the new head to the old y-list (n_7) [2].

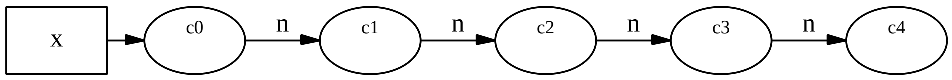


Fig. 2. An input list for `reverse`

The workings of `reverse` is exemplified on the input list shown in Fig. 2. Table 1 shows the execution state after n_3 during the first four iterations of the loop.

Let us assume that the input to this program will be a non-empty acyclic singly linked list made up of unshared elements like the one shown in Fig. 2. Some correct answers to the questions listed in Section 2 would be:

after n_3:
- x and y point to acyclic singly linked lists; the y-list may be empty,
- t and y are may- and must-aliases or both point to NULL.
- x never points to NULL,
- the lists to which x and y point are disjoint,
- no list element is heap-shared.

after n_4: x and y are must-aliases, and therefore also may-aliases.
after n_5: x and y are not may-aliases, and therefore not must-aliases. (Note that this would not be the case if the initial x-list contained a cycle).
everywhere: there are no garbage cells.

2.2 Answers as Given by Shape Analysis

Shape analysis will be started under the aforementioned assumption that the input to the program will be a non-empty acyclic singly linked list. This is expressed by the so-called shape graphs shown in Fig. 3.

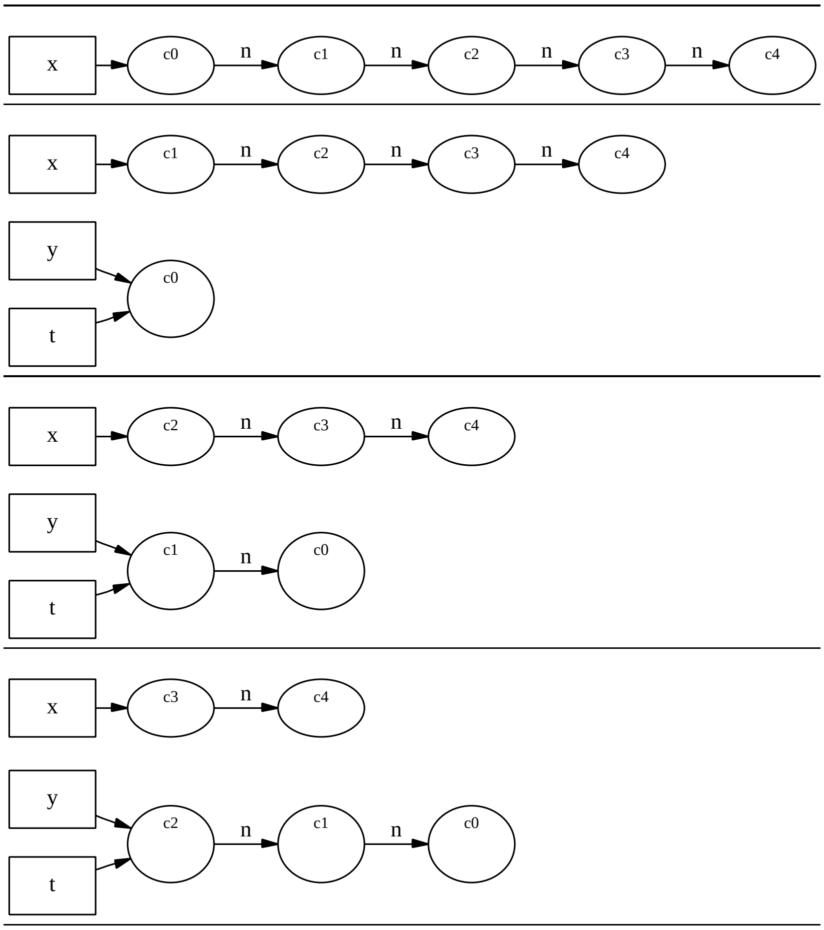
Shape analysis of `reverse` will produce information for each program point about the lists that can arise there. For program point n_3, the results would be given as the shape graphs shown in Fig. 4.

These graphs should be interpreted as follows:

- A rectangular box containing the name p represents pointer variable p. The pointer variables of program `reverse`, namely x, y, and t, appear boxed in the shape graphs in Figures 3 and 4.

[2] Splitting this assignment into two steps is done to simplify the definition of the semantics.

Table 1. The first four iterations of the loop in `reverse` after n_3



– Ovals stand for abstract locations. A solid oval stands for an abstract location that represents exactly one heap cell. In Fig. 3(b), the oval u represents the one cell of an input list of length 1.

  A dotted oval box stands for an abstract location that may represent one or more heap cells; in Fig. 3(a) the dotted oval u represents the cells in the tail of the input list. In the example input list shown in Fig. 2, these are the cells $c1, c2, c3$, and $c4$.

– A solid edge labeled c between abstract locations m and m' represents the fact that the c-component of the heap cell represented by m will point to the heap cell represented by m'. Fig. 4(e) indicates that the `n`-component of the heap cell represented by u will point to the heap cell represented by $u0$.

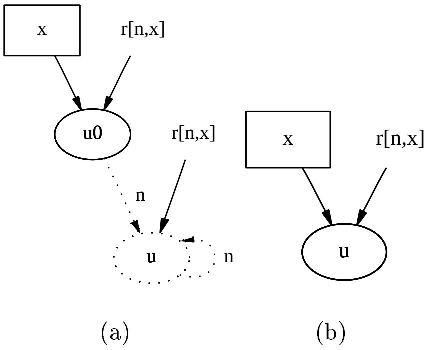


Fig. 3. A description of the input to `reverse`. (a) represents acyclic lists of length at least 2, (b) lists of length 1.

This edge represents the `n`-component of $c1$ pointing to $c0$ in the third figure in Table 1.

- A dotted edge labeled c between abstract locations m and m' tells us that the c-component of one of the heap cells represented by m may point to one of the heap cells represented by m'. In the case when m and m' are the same abstract location, this edge may or may not represent a cycle. In Fig. 4(d), the dotted self-cycle on location $m0$ represents `n`-components of heap cells represented by $m0$ possibly pointing to other heap cells represented by $m0$. Additional information about non-heap-sharing (see below) implies that, in this case, the dotted self-cycle does not represent a cycle in the heap.
  Fig. 4(d) represents the execution state of the second figure in Table 1. The dotted location $m0$ represents the heap cells $c2, c3, c4$; the dotted `n`-back-edge represents the two pointers from $c2$ to $c3$ and from $c3$ to $c4$, respectively.
- Other solid edges to an abstract location m represent properties that are definitely possessed by the heap cells represented by m. For example, a solid edge from a pointer variable p to m represents the fact that p will point to the heap cell represented by m, i.e., this heap cell will have the property "pointed-to-by-`p`". Another property in the example is "reachable-from-`x`-through-`n`", denoted in the graph by `r[n,x]`. It means that the heap cells represented by the corresponding abstract location are (transitively) reachable from pointer variable `x` through `n`-components.
  Solid ovals could be viewed as abstract locations having the property "uniquely representing", later on called "not-summarized".
- A dotted edge from a property to an abstract location indicates that the heap cells represented by that location may or may not have that property.
- The absence of an edge from a property to an abstract location m means that the heap cells represented by m definitely do not have this property.
  In Fig. 4(a), the absence of an edge from `y` to the location $u0$ means that `y` will definitely not point to the cell represented by $u0$. The absence of a

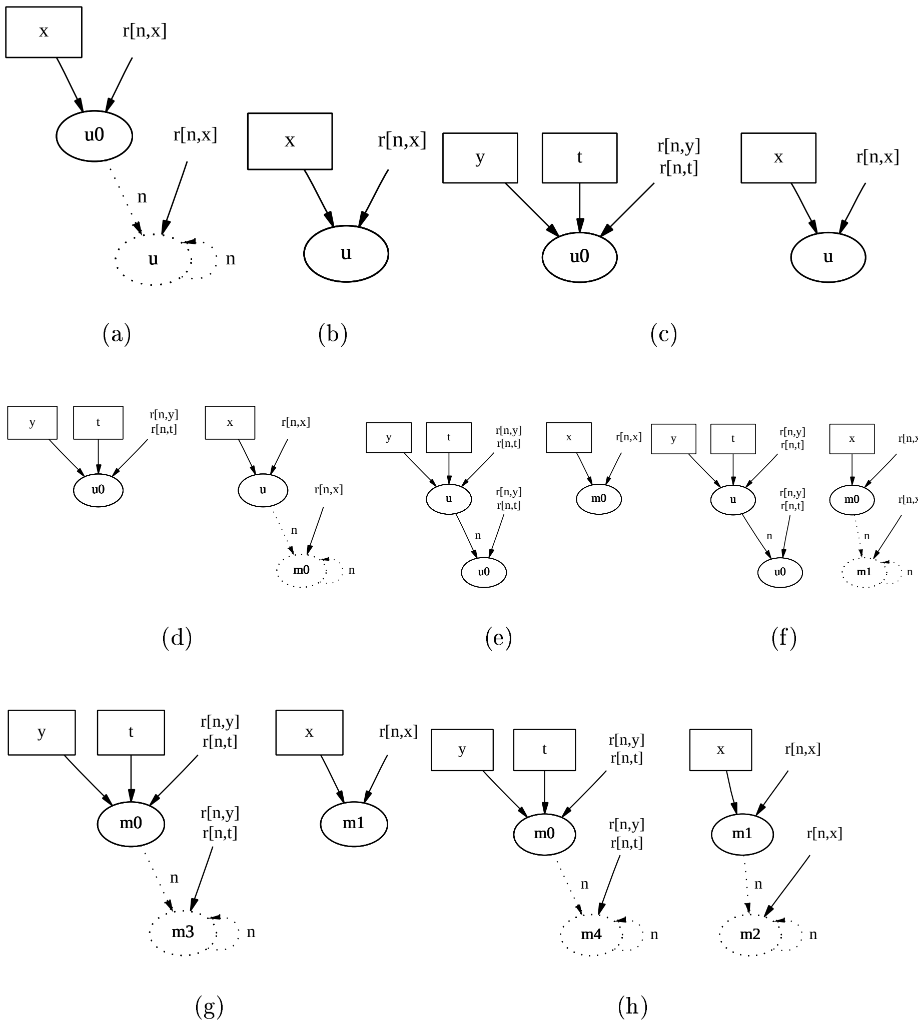


Fig. 4. Shape graphs describing all execution states that can arise after n_3

pointer variable p in a shape graph means that in the stores represented by this graph p points to NULL.

In the analysis of `reverse`, a very important heap cell property is `is` which means "is (heap-)shared". It does not show up in any of the graphs in Fig. 4, signifying that no heap cell is be shared in any of the execution states that can arise after n_3.

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, and
dotted meaning "don't know" for properties (including "uniquely representing"),

In the following, "always holds" and "never holds" will be called *definite* values of properties, while "don't know" will be called the *indefinite* value.

Each shape graph produced at some program point describes execution states that could occur when execution reaches that program point. The set of all graphs produced at some program point describes (a superset of) all the execution states that can occur whenever execution reaches that program point.

With this interpretation in mind, all the claims about the properties of the heap contents after n_3 can be checked by verifying that they hold on all of the graphs shown in Fig. 4.

3 Shape Analysis

The example program `reverse` works for lists of arbitrary lengths. However, as was described in the preceding section (at least for one program point), the description of the lists occurring during execution is finite: there are 8 graphs describing all `x`- and `y`-lists arising after n_3. This is a general requirement for shape analysis. While the data structures that a program builds or manipulates are in general of unbounded size, their shape descriptor has to have a *bounded size.*

This representation of the heap contents has to be *conservative* in the sense that whoever asks for properties of the heap contents—e.g., a compiler, a debugger, or a program understanding system—receives a reliable answer. The claim that "pointer variable p or pointer component p->c never has the value NULL at this program point" may only be made if indeed this is 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 (resp. 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.

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 get lost due to the chosen representation. Thus, there is a part of the execution state (or some of its properties) that is exactly represented, and some part of the execution state (or some of its properties) that is only approximatively represented. The process leading to the latter is called *summarization*. Summarization intuitively means the following:

- Some heap cells will "lose their identity", i.e., will be represented together with other heap cells by one abstract location.
- The connectivity among those jointly represented heap cells will be represented conservatively, i.e., 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 location,
  - a property that holds for some but not all of the summarized cells will have the value "don't know" for the summary location.

As will be seen in Subsection 3.5, one of the main problems is how to extract an abstract location from a summary location when (in a concrete execution of the program) a pointer is advanced to one of the heap cells represented by this summary location. This process will be called "materializing a new abstract location out of a summary location".

3.2 Static Program Analysis

Shape analysis is a *static program analysis* technique. It can thus be couched in terms of the theory of *Abstract Interpretation*, cf. [4, 17]. The most important ingredient of any static program analysis is an *abstraction function*. The abstraction function relates two worlds: in the case of shape analysis, the *concrete world* of program execution states and the *abstract world* of shape graphs. It maps an execution state to its bounded, conservative representation. The abstraction function does what is described as summarization above.

The *concrete semantics* of the programming language is given by a set of functions f_{st} for each statement *st* of the language. They describe the effect of the statements on an execution state. The *abstract semantics* describes the effect of the statements on the representation of execution states. It is given in the form of an *abstract transfer function* $f^{\#}_{st}$ for each statement *st*. Applying $f^{\#}_{st}$ will be called *abstractly executing st*.

3.3 Parametric Shape Analysis

"Shape analysis" is a generic term standing for a whole class of algorithms of different power and complexity trying to answer questions about structures in the heap. In our setting, a particular shape-analysis algorithm is determined by a set of properties that heap cells may have (and whose values will be tracked by the shape-analysis algorithm).

First, there are the *core properties*, e.g., 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-component of l_1 points to l_2. These properties are part of any pointer semantics. The core properties in the particular shape analysis of the `reverse` program are "pointed-to-by-`x`", denoted just by `x`, "pointed-to-by-`y`", denoted by `y`, and "pointed-to-by-`t`", denoted by `t`, and "connected-through-`n`" denoted by `n`.

Further properties are called *instrumentation properties* [23]. They are determined by what the analysis is meant to observe. They are expressed in terms of the core properties. Our example analysis should find out properties of programs manipulating acyclic singly linked lists. Sharing properties are important to detect acyclicity. Reachability properties from specific pointer variables are important to keep disjoint sublists summarized separately when a pointer moves through a list.

Therefore, the instrumentation properties in our example analysis are "is-heap-shared", denoted by `is`, "reachable-from-`x`-through-`n`", denoted by `r[n,x]`, "reachable-from-`y`-through-`n`", denoted by `r[n,y]`, and "reachable-from-`t`-through-`n`", denoted by `r[n,t]`. A property existing in every shape analysis is "not-summarized".

3.4 Abstraction Functions

The abstraction function of a particular shape analysis is determined by a subset of the set of all properties, the so-called *abstraction properties*.[3] The principle is that heap cells that have the same definite values for the abstraction properties are summarized to the same abstract location. 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 location have no observable difference.

In Fig. 3(a), all the cells in the tail of an input list of length at least two are summarized by the abstract location u, since they all have the property `r[n,x]` and do not have the properties `x`, `y`, `t`, `r[n,y]`, `r[n,t]`, `is`. The abstract location $u0$ represents exactly the first cell of the input list. It has the properties `x` and `r[n,x]` and none of the other properties.

3.5 Analyzing a Statement

An abstract semantics of the pointer statements available in the language has to be specified in some formalism. It describes how the core and the instrumentation properties change when a statement is executed.

[3] [23] uses only unary properties.

The most complex statement involving pointers is the one that moves a pointer along a pointer component. In the `reverse` program there is one such statement, `n_5: x=x->n`. Its semantics will be described by means of an example.

The shape graph in Fig. 5(a) describes some of the lists that can arise before program point n_5.

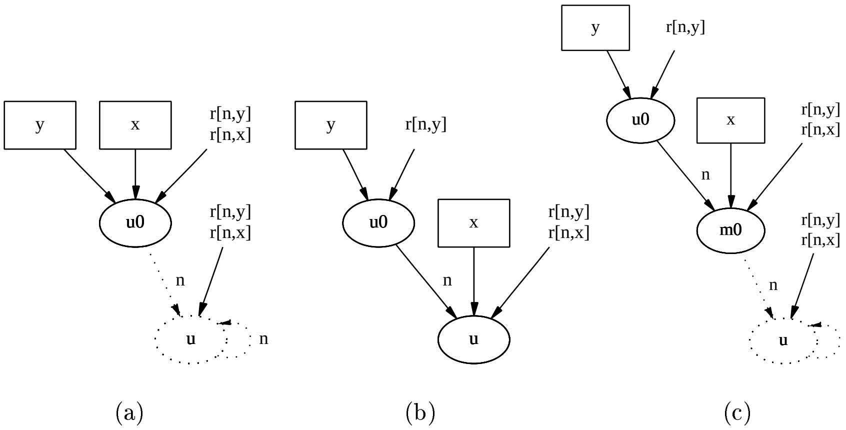


Fig. 5. (a): A shape graph that arises before n_5; (b) and (c): the two shape graphs resulting from (abstractly) executing `x=x->n` on shape graph (a).

At n_5 the statement `x=x->n` is (abstractly) executed. One candidate result might be the shape graph in which `x` has moved along the indefinite edge labeled `n` to point to the abstract location u. However, this would not be an acceptable shape graph, since it is not in the image of the abstraction function. To see this, note that a summarized abstract location representing one or more heap cells cannot have the definite property "pointed-to-by-`x`". The semantics of C (as well as of other languages with pointers) implies that any pointer variable or pointer component can only point to at most one cell at any one time. Hence, only two cases are possible: (i) `x` points to NULL, or (ii) `x` points to exactly one cell.

A better approach to creating the shape graph that results from (abstractly) executing `x=x->n` is to do a case analysis of the reasons why u's "not-summarized" property and the two `n`-edges are indefinite.

- The first case assumes that a definite `n`-edge exists starting at $u0$, and that u represents exactly one heap cell. What about the indefinite trivial cycle labeled `n` on u? The absence of an `is`-edge to u indicates that the cell represented by u is not shared. Thus, the back-edge from u to u in Fig. 5(a) cannot represent a concrete back pointer, since there is already one solid edge to u representing a concrete pointer. Hence, the indefinite back-edge from u to u has to be eliminated. The abstract execution of the statement `x=x->n` results in the graph shown in Fig. 5(b).

– The second case also assumes that a definite `n`-edge exists starting at $u0$. However, we now assume that u represents more than one heap cell. A new definite abstract location, called $m0$, is then "materialized" out of u. It represents the first of the cells in the tail of the `x/y`-list. The definite edge starting at $u0$ is made to point to it. The argumentation about a potential back-edge to it is the same as in the first case: the newly materialized location does not have the property `is` and thus cannot be the target of a back-edge. The execution of the statement `x=x->n` results in the graph shown in Fig. 5(c).
– A third case would have the indefinite `n`-edge representing the absence of a pointer from the cell represented by $u0$ to any of the cells represented by u. However, this is incompatible with the fact that the `r[n,x]` and `r[n,y]` properties of u have the (definite) value 1. Hence, this case is impossible.

The less powerful a shape analysis is, the more information about the heap contents will in general be summarized. Most shape-analysis algorithms are exponential or even doubly exponential in some parameters derived from the program and some parameters determined by the set of properties to be observed.

4 Implementation and Experience

The three-valued-logics approach described in this paper has been implemented in the TVLA-engine (Three-Valued Logic Analysis) by Tal Lev-Ami at Tel-Aviv University [16]. This prototype implementation in Java is good for experimenting with analysis designs. Small examples like `reverse` can be analyzed in a few seconds on a Pentium 166 MHz PC. Bigger examples, e.g., Mobile Ambients [18], may take several hours.

A surprising experience was the following: Although one expects that a more precise analysis is more costly in terms of time and space, this is not necessarily true in three-valued logic based analyses: A more precise analysis may create fewer unneeded structures and thus run faster. For instance, Lev-Ami analyzed a program to sort the elements of a linked list. When a list-merge function was analyzed without reachability properties, the analysis created tens of thousands of graphs such that the machine ran out of space. Adding reachability properties reduced the number of graphs to 327 and the analysis time to 8 seconds.

A number of issues need to be addressed for the TVLA-engine to become a useful tool for large programs.

5 Related Work

Only a very brief account of related work can be given in this paper. A more detailed treatment is given in [22].

5.1 Types and Allocation Sites

One of the earliest properties of heap cells used for abstraction has been the point in the program at which the cells were allocated, i.e., their *allocation* *site* [12, 2, 24, 1, 19]. Cells allocated at different program points would never be represented jointly by one node. The motivation behind this approach was that "nodes allocated in different places probably are going to be treated differently, while all nodes allocated at a given place will probably be updated similarly" [2].

This is sometimes true and sometimes not true. For example, allocation-site information fails to keep abstract locations separate when all cells of a data structure are allocated at the same program point.

The methods described above use additional information such as sharing or heap reference counts to improve the precision of the results.

Types have also been used to partition the set of heap cells. In a typed language, a typed pointer may only point to cells of the appropriate type. They could be used if nothing else is available. However, they will lead only to very coarse representations because often all cells of a data structure have the same type.

Both allocation-site information and type information induce a static partition on the heap. This is in contrast with the approach described above, which induces a dynamic partition on the heap.

5.2 Pointer Variables

[25, 21, 22] used only the "pointed-to-by-p" properties to define the abstraction function. Additional information such as non-sharing information was used to improve precision. The algorithm given in [21, 22] was the first to achieve *strong nullification* and *strong update* of pointer values for statements that modified pointer values. This means, the algorithm could delete or overwrite existing pointers in a conservative way. As a result it was able to materialize new nodes out of summary nodes as described in Section 3.5.

For efficiency reasons, the method presented in [21, 22] (as well as the methods described in [12, 15, 14, 2, 24]) merged all the information available at a program point into one shape graph. This made the semantics hard to understand and implied a loss in precision. A more precise variant of this method (which is also easier to understand) is described in [17]. It uses sets of shape graphs at each program point.

5.3 k-Bounded Approaches

Another approach to represent unbounded structures in a bounded fashion is to choose a constant k and to represent everything "within a diameter k" precisely and to summarize heap cells outside of this diameter. Applied to our problem of shape analysis, this would mean representing lists precisely up to a length of k, trees up to a depth of k, and general graphs up to a diameter of k, and then summarizing the rest of the data structure. The approach of [11] corresponds to using the "reachable-from-p-via-access-path-α"-property with $|\alpha| \leq k$.

The Jones-Muchnick formulation has two drawbacks:

– The analysis yields poor results for programs that manipulate cons-cells beyond the k-horizon. For example, for the list-reversal program `reverse`, little useful information is obtained. The analysis algorithm must model what happens when the program is applied to lists of lengths greater than k. However, the tail of such a list is treated conservatively, i.e., 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., [10]).

5.4 Methods Not Based on Shape Graphs

There are also several algorithms for finding may-alias information for pointer variables that are not based on shape-graphs. The most sophisticated ones are those of Landi and Ryder [13] and Deutsch [5]. For certain programs that manipulate lists, Deutsch's algorithm offers a way of representing the exact (infinite set of) may aliases in a compact way.

A different approach was taken by Hendren and Nicolau, who designed an algorithm that handles only acyclic data structures [9, 7].

6 Conclusions

We conclude with a few general observations.

6.1 A Parametric Framework for Shape Analysis

The method that has been surveyed in this paper provides the basis for a *parametric* framework for shape analysis. Such a framework has two parts: (i) a language for specifying various properties that a heap cell may or may not possess, and how these properties are affected by the execution of the different kinds of statements in the programming language, and (ii) a method for generating a shape-analysis algorithm from such a description. The first is an issue having to do with *specification*; the specified set of properties determines the characteristics of the data-structure elements that the static analysis can distinguish. The second is an issue of how to generate an appropriate algorithm from the specification. The ideal is to have a fully automatic method—a `yacc` for shape analysis, so to speak: The "designer" of a shape-analysis algorithm supplies *only* the specification, and the shape-analysis algorithm is created automatically from this specification. The TVLA system discussed in Section 4 implements such a parametric framework.

Different instantiations of the framework create analyses that use different classes of shape graphs, and hence are prepared to identify different classes of store properties that hold at the different points in a program. Different classes of shape graphs may be needed, depending on the kinds of linked data structures used in a program and on the link-rearrangement operations performed by the program's statements. (In general, an instantiation of the framework will handle every program, but may produce conservative results due to the use of a class of shape graphs that does not make appropriate distinctions.) The essence of the particular shape-analysis methods discussed in Section 5 can be captured via different instantiations of the framework.

The instantiation of the analysis framework described in Subsection 3.3 is sufficiently powerful to successfully analyze the `reverse` program, which works on singly linked lists. It would also produce precise information on many other programs that deal with singly linked lists. However, it would fail on programs that operate on doubly linked lists, since the `is`-property would hold on all but the first and the last elements of such lists. An instantiation for doubly linked lists would need the following two properties:

forward-backward-pairing: If the forward-pointer of a heap cell $c1$ points to a heap cell $c2$, then the backward-pointer of $c2$ must point to $c1$.

backward-forward-pairing: If the backward-pointer of a heap cell $c1$ points to a heap cell $c2$, then the forward-pointer of $c2$ must point to $c1$.

Different versions of doubly linked lists would need some additional properties for the first and last elements, e.g., requiring that the "unused" pointer has the value NULL.

6.2 Biased Versus Unbiased Static Program Analysis

Many of the classical dataflow-analysis algorithms use bit vectors to represent the characteristic functions of set-valued dataflow 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.

Conventional wisdom holds that static analysis must inherently have such a one-sided bias. However, the material developed in [23] 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 capturing indefiniteness.

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

- A one-sided analysis that is precise with respect to "false" and conservative with respect to "true" is really a 3-valued analysis over false, true, and "don't know" that conflates true and don't know (and uses "true" in place of don't know).

– Likewise, an analysis that is precise with respect to "true" and conservative with respect to "false" is really a 3-valued analysis over false, true, and "don't know" that conflates false and don't know (and uses "false" in place of don't know).

In contrast, the shape-analysis work has shown how to create analyses that are unbiased: They are precise with respect to both false and true, and use a separate "don't know" value to capture indefiniteness.

Acknowledgements Tal Lev-Ami carried out the implementation of TVLA. One instantiation of TVLA was used to analyze the example program and generate the figures used in the paper. Hanne Riis Nielson provided very helpful comments.

References

1. U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington, DC, September 1993. IEEE Press.
2. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.
3. F. Corbera, R. Asenjo, and E.L. Zapata. New shape analysis techniques for automatic parallelization of C code. In *International Computing Symposium*, 1999.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.
6. N. Dor, M. Rodeh, and M. Sagiv. Detecting memory errors via static pointer analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 27–34, June 1998. Available at "http://www.math.tau.ac.il/∼ nurr/paste98.ps.gz".
7. L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
8. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.
9. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Par. and Dist. Syst.*, 1(1):35–47, January 1990.
10. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.
11. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

12. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.
13. W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *Symp. on Princ. of Prog. Lang.*, pages 93–103, New York, NY, January 1991. ACM Press.
14. J.R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, Univ. of Calif., Berkeley, CA, May 1989.
15. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, New York, NY, 1988. ACM Press.
16. T. Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, 2000.
17. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.
18. F. Nielson, H. Riis Nielson, and M. Sagiv. A Kleene analysis of mobile ambients. In *Proceedings of ESOP'2000*, 2000.
19. J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lec. Notes in Comp. Sci.*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.
20. J.L. Ross and M. Sagiv. Building a bridge between pointer aliases and program dependences. In *Proceedings of the 1998 European Symposium On Programming*, pages 221–235, March 1998. Available at "http://www.math.tau.ac.il/∼ sagiv".
21. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1996. ACM Press.
22. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.
23. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999. Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".
24. J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. and Comp.*, 101(1):70–102, Nov. 1992.
25. E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language.* PhD thesis, Univ. of Calif., Berkeley, CA, 1994.