Heap Abstractions for Static Analysis

Vini Kanvar and Uday P. Khedker

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Email: {vini, uday}@cse.iitb.ac.in

21 March, 2014

Abstract

Heap data is potentially unbounded and seemingly arbitrary. This survey provides a high level view of the abstraction techniques used for compile time analysis of heap data. We view a heap abstraction as consisting of two features: a heap model to represent the heap memory and a summarization technique for bounding the heap representation. We classify the models as storeless, store based, and hybrid. We describe various summarization techniques based on k-limiting, allocation sites, patterns, variables, and other generic predicates. We describe how these summarization techniques have been adopted in the literature for different heap models.

1 Heap Analysis: Motivation

Heap memory allows on-demand memory allocation during program execution. It facilitates creation of flexible data structures which can outlive the procedures that create them and whose sizes can change during execution. As processors and memories become faster, the role of heap memory in user programs as well as design and implementation of programming languages is increasing. As a consequence, analysis of heap memory is becoming increasingly important. In this paper, we restrict ourselves to static analysis of heap.

At a generic level, heap analysis provides useful information about data in procedural languages whereas in object-oriented and functional languages, it provides useful information about control flow as well. Specific applications which can benefit from such information include program understanding, program refactoring, verification, debugging, enhancing security, improving performance, compile time garbage collection, instruction scheduling, parallelization etc. Appendix A provides an overview of applications of heap analysis.

Heap memory usage involves pointer (or reference) variables and dynamic memory allocation. Both these features are hard to analyse at compile time. We summarize the challenges posed by these two features to a static analysis as follows:

- **Challenges in pointer analysis.** Even without the heap, pointer analysis is inherently difficult because it requires uncovering indirect manipulations of data and control flow. Additionally, modern features such as dynamic typing, field accesses, dynamic field additions and deletions, implicit casting, pointer arithmetic, etc. make pointer analysis even harder.

  Points-to analysis which discovers addresses possibly held by pointers is undecidable [10]. Alias analysis which discovers which pointers hold a common address at a given program point is also undecidable [43]. It is impossible to compute precise compile time alias information for languages with if-statements, loops, dynamic storage, and recursive data structures.
• **Challenges in analysis of dynamically allocated memory.** Dynamically allocated memory poses the following additional challenges due to its temporal and spatial structure.

  – *Unpredictable lifetime.* The lifetime of a heap object is not limited to the scope that creates it. Although the creation of a heap object is easy to discover, the last use of a heap object, and hence the most appropriate point of its deallocation, is not easy to discover.

  – *Unbounded size.* Since heap locations are created on-demand during runtime, size of a heap allocated data structure is unbounded. Therefore, it seems to have an arbitrary structure.

  – *Unnamed locations.* Heap locations cannot be named in programs. A compile time analysis of a heap manipulating program therefore, needs to create appropriate symbolic names for heap memory locations. This is non-trivial because unlike stack and static data, the association between symbolic names and memory locations cannot remain fixed.

These challenges are absent in analysis of stack and static data whose temporal and spatial structures are easier to discover. As a consequence of these challenges, the analysis of heap requires creation of

  – models of heap which represent the structure of heap memory, and

  – summarization techniques to bound the representations.

This paper uses the above theme to survey the heap abstractions found in the literature on program analysis. Section 2 presents the basic concepts which are then used in Section 3 to categorize heap models as storeless, store based, or hybrid and describe various summarization techniques. These are generic ideas which are then used in Sections 4, 5, and 6 to describe the related investigations in the literature in terms of the interactions between the heap models and summarization techniques. Section 7 compares the models and summarization techniques based on various metrics such as efficiency, precision, ease of understanding and reasoning, and generality. Section 8 concludes the paper by observing the overall trend.

Since the focus of this paper is on heap abstraction techniques, we have relegated some information to appendices. Appendix A describes major heap analyses and their applications. In Appendix B, we briefly compare the heap memory view of C/C++ and Java. Finally Appendix C mentions some notable engineering approximations used in heap analysis.

## 2 Basic Concepts

In this section, we build the basic concepts required to explain the heap abstractions in later sections. We assume Java like programs, which use program statements: \( x := \text{new}, \ x.f := \text{new}, \ x := \text{null}, \ x.f := \text{null}, \ x := y, \ x.f := y, \ x := y.f. \) The dot followed by a field represents field dereference by a pointer variable. For easy understanding, we draw our programs as control flow graphs. \( \text{In}_n \) and \( \text{Out}_n \) denote the program point before and after program statement \( n \) respectively.

### 2.1 Examples of Heap Related Information

Some of the heap related questions asked during various applications are listed below. This list is an adaption of the original list by Sagiv, Reps, and Wilhelm [52].

1. **Null pointers:** Does a pointer variable or a pointer component of a heap cell contain null at the entry of a statement that dereferences the pointer or the component?

2. **Memory leak:** Does a procedure or a program leave behind unreachable heap cells when it returns?
3. **Alias and points-to**: Do two pointer expressions reference the same heap cell? Is a node pointed to by a variable or a field of a variable?

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

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

6. **Cyclicity**: Does a node reside on a directed cycle of fields?

7. **Shape**: What is the shape of (some part of) the contents of the heap? Does the heap contain a singly linked list, potentially with (or definitely without) a cycle, a doubly linked list, a binary tree, and so on.

8. **Liveness**: Will the heap cell be used in the future?

Two heap cells or pointers are said to share a property if the answer to a given question for the two cells or pointers is same. For example, all pointers that are null at a given program point share the property of nullity. As we shall see later, sharing of properties provides an important clue for summarization.

Two most important examples of heap information are aliasing and points-to relations because the rest of the questions are often answers using them.

- In **alias analysis**, two pointer expressions are said to be aliased to each other if they evaluate to the same memory location. There are three possible cases of aliases between a given set of pointers:
  - the given pointers cannot alias in any execution instance of the program,
  - the given pointers **must-alias** in every execution instance of the program, or
  - the given pointers **may-alias** in some execution instances but not necessarily in all execution instances.

- A **points-to analysis** attempts to determine the addresses that a pointer holds. A points-to information also has three possible cases: **must-points-to**, **may-points-to**, and cannot points-to.

An analysis is said to perform a **strong update** if it can remove some alias/points-to information on processing an assignment statement involving indirections on the left hand side (e.g. \*x, x.n, or x->n). It is said to perform a **weak update** if it cannot do so. Strong updates require the use of must-alias/must-points-to information whereas weak updates can be performed using may-alias/may-points-to information. Strong update is an expensive operation.

### 2.2 Soundness and Precision of Heap Analysis

A static analysis computes information representing the runtime behaviour of the program being analyzed. Two important considerations in a static analysis of a program are **soundness** and **precision**. Soundness guarantees that the effects of all possible executions of the program have been included in the information computed. Precision is a qualitative measure of the amount of spurious information which is the information that cannot correspond to any execution instance of the program.

When an analysis computes information that must hold for all execution instances of a program, soundness is ensured by under-approximation of the information. When it computes information that may hold in some execution instances, soundness is ensured by over-approximation of the information. Precision is governed by the extent of over- or under-approximation introduced in the process.
Applications involving program transformations or verification require sound analyses because they must hold for all execution instances. On the other hand error detection or validation applications can afford to compromise on soundness and may not cover all possible execution paths.

Consider the program in Figure 1. Assume that an analysis computes the set of pointers that may be possibly null in order to report a null-dereference at statement 7. Assume that we restrict ourselves to the set \{x.f, x.g, y.f, y.g\}. We know that x.f and y.f are both guaranteed to be non-null because of the assignment in statement 6. Such an assertion cannot be made about x.g and y.g.

• Assume that an analysis reports the set \{x.f, x.g, y.f, y.g\} at statement 7 as result. This result is sound (because it does not miss out on any pointers that may be possibly null in some execution reaching statement 7) but not precise (because it includes pointers that are not null in any execution reaching statement 7).

• Assume that an analysis reports the set \{x.g, y.g\} at statement 7 as result. This result is both sound and precise.

• Assume that an analysis reports the set \{y.g\} at statement 7 as result. This result is not sound because it excludes x.g which may be null at statement 7.

3 Heap Abstractions

In this section we define some generic ideas which are then used to describe the work reported in the literature in subsequent sections.

A heap abstraction is the heap modeling and summarization of the runtime data structures at compile time. A heap model is a representation of the dynamically allocated storage of a program at compile time. Since heap allocations are unbounded in size, summarization of heap objects is critical during any heap analysis. Heap summarization is the process of representing an unbounded number of concrete heap objects by a finite set of abstract/summarized objects in order to bound the representation.

We categorize heap models as storeless, store based, and hybrid models. The summarizations are categorized as those that use allocation sites, k-limiting, patterns, variables, or other generic predicates. Their combinations are illustrated in Figure 2.
Heap Abstractions

3.1 Heap Models

Heap objects do not have any fixed name during their lifetime as they are dynamically allocated. Hence, various schemes are used to name them. The choice of naming them gives rise to different views of heap. We define the resulting models and explain them using a running example in Figure 3. Figure 4 associates the models with the figures that illustrate them for our example program.

- **Store based model.** The store based model generally views the heap at a program point as a directed graph called memory graph [7, 8, 11, 20, 25, 34, 41, 44, 51, 56]. It is a collection of heap objects connected together by a selector field. Object nodes are pointed to by root variable(s) and/or a field name through an object node. Let \( V \) be the set of root variables, \( F \) be the set of fields names, and \( O \) be the set of heap objects. Then, formally the memory graph is a collection of two mappings: \( V \to 2^O \) and \( O \times F \to 2^O \). Note that the mappings are to \( 2^O \) rather than \( O \) because a memory graph at a program point is a union of the memory graphs created along all paths reaching the program point.

Figure 3 shows our running example and a snapshot of the runtime heap memory created and accessed by it. This structure is represented using store based model in Figure 5. The root variable \( y \) may point to \( x.f.f \) or \( x.f.f.f.f \) depending on the number of iterations of the loop in program in Figure 3a. We mark allocation sites in the heap nodes—\( x \) is allocated at program statement 1; \( x.f \) and \( x.f.f.f \) are allocated at program statement 3; \( x.(f.f)^* \) are allocated at program statement 5.

- **Storeless model.** The storeless model (originally proposed by Jonkers [27]), views the heap as a collection of access paths [14, 18, 27, 30, 33, 40]. An access path is a root variable followed by a sequence of zero or more field names. The desired properties of heap are stored as relations of access paths. The storeless model does not identify objects corresponding to these access paths. Given \( V \) as the set of root variables and \( F \) as the set of field names, the set of access paths is defined as \( V \times F^* \). In other words, an access path is of the form \( r.e \), where \( r \) is a root variable and \( e \) is a regular expression of field names defined in terms of concatenation (\( . \)), closures (\( * \) and \( + \) used as superscripts), and wild card (\( * \) used inline) operators. For example, access path \( x.f.* \)
3.2 Summarization and Materialization

In the presence of loops and recursion, the size of graphs in the store based representations and the lengths of the access paths in the storeless representations may be unbounded and the number of access paths may be infinite. If an algorithm for the compile time analysis tries to compute these directly, termination cannot be guaranteed. This problem can be solved by using a bounded representation to summarize heap information. In other words, we need to approximate the unbounded heap or dynamic memory into summarized heap locations called summarized objects. A summarized object is a compile time representation of one or more runtime (aka concrete) heap objects.

Accessing the same summarized object does not necessarily mean accessing the same concrete heap object every time. A property discovered for a summarized object is satisfied by some of the represented heap objects and not necessarily by all. For example, when determining which pointer expressions refer to the same heap location, all pointer expressions pointing to the same summarized object will be recognized as possible candidates, even though some of them may have been changed by new assignments to different heap locations. Therefore, a heap analysis using this traditional summarization technique has a serious disadvantage: it can answer only may-pointer questions, that is, whether two variables may potentially refer to the same object. It cannot in general answer must-pointer questions, that is, whether two variables must always refer to the same object. As a
result, traditional summarization technique [11] cannot be of use in some applications requiring must-pointer information. In order to compute precise must-pointer information, Sagiv et al. [49] proposed to materialize (“un-summarize”) summarized objects (explained in Section 5.2). The literature contains many approaches for must-pointer analysis, ranging from relatively simple abstractions such as the recency abstraction [4] to full-fledged shape analysis [49]. This process of materialization is very expensive, making the computation and use of must-pointer analysis prohibitive.

### 3.3 Heap Summarization Techniques

The main distinction between various heap analysis approaches is how they map a heap of potentially unbounded size to a bounded size. We introduce various summarization techniques using our running program in Figure 3a. The figures illustrating these techniques have been listed in Figure 4.

- **k-limiting summarization** records heap information precisely up to a sequence of $k$ dereferences (i.e. paths of length $k$ in the memory graph) and over-approximates beyond $k$.

  $k$-limiting summarization has been performed on store based model [34]. Figure 5b represents a $k$-bounded representation of the store based model in Figure 5. For $k = 2$, heap nodes beyond two dereferences are not stored. A self loop is created on the second dereference (node corresponding to $x.f.f$) to over-approximate this information. This stores spurious aliases for access paths with more than $k = 2$ dereferences (for example, $x.f.f.f$ and $y$ are spuriously marked as aliases at Out$_6$).

  $k$-limiting summarization has also been performed on storeless model [26,33]. This was proposed by Jones et al. [26]. Figure 6b represents a $k$-bounded representation of the storeless model in Figure 6. This also introduces the same spurious alias pairs as in Figure 5b.

- **Summarization using allocation sites** merges heap objects which have been allocated at the same program site. This technique is used for approximating store based heap model [4,34,41]. It gives all the objects allocated in a give program statement, the same name. Figure 5c represents allocation site based approximated heap graph of the store based model in Figure 5. Here all
3.3 Heap Summarization Techniques

(a) Unbounded store based model.

(b) $k$-limiting ($k = 2$) summarization.

(c) Allocation site based summarization.

(d) Variable based summarization.

Figure 5. Store based heap graphs at $Out_6$ for the program in Figure 3a. Figures 5b, 5c, and 5d are bounded representations of heap information in Figure 5a. The numbers inside the graph nodes indicate the object’s allocation site in the program in Figure 3a.

Figure 6. Storeless model of heap graph at $Out_6$ of the program in Figure 3a. Figures 6b and 6c are the bounded representations of heap information in Figure 6a. We store sets of aliased pointer expressions as a set of equivalence classes denoted by $\langle$ and $\rangle$.

the objects allocated at program statements 3 and 5 are respectively clustered together. This summarization on the given example does not introduce any spurious alias pairs. We will show spuriousness introduced due to this summarization in Section 5.6.

- **Summarization using patterns** merges access paths based on the patterns of occurrences of field names in the access paths. Pattern based summarization has been used to bound the heap access paths [14, 30, 40]. Figure 6c represents pattern based summarization of the storeless model in Figure 6. For this example, it precisely marks every second dereference of field $f$ (along the chain rooted by $x$) as aliased with $y$.

- **Summarization using variables** merges those heap objects that are not pointed to by root variables. An object node pointed to by a root variable is represented by a single concrete node. Variable based summarization has been performed on store based heap model [7, 11, 49, 50]. Figure 5d represents variable based summarization of the store based model in Figure 5. After the first iteration of the loop, there are three nodes—the first pointed to by $x$ and the third pointed to by $y$. Therefore, no summary node needs to be created. In the second iteration of the loop of the program in Figure 3a, nodes $x.f$, $x.f.f$, and $x.f.f.f$ are not pointed to by any
variable (as shown in Figure 3b). Therefore, they are merged together as a summary node represented by dashed lines in Figure 5d which shows the graphs after the first and the second iterations of the loop. This also imprecisely marks x.f.f.f and y as aliases at Out<sub>6</sub>.

Figure 7b is a variable based summarized representation of the unbounded hybrid model in Figure 7a. A summary node (shown with dashed lines in the figure) is created on the same logic as in Figure 5d (explained before). Summarized access paths are appropriately marked on the nodes in the hybrid model.

- **Summarization using other generic predicates** merge those heap objects that satisfy a given predicate [4, 8, 18, 20, 25, 44, 46, 51, 52, 61]. Note that the summarization techniques introduced above are also based on a predicate as listed below:

  1. **k-limiting predicate**: Is a heap cell at most k dereferences away from a root variable?
  2. **Allocation site based predicate**: Is a heap cell allocated at a program site?
  3. **Pattern based predicate**: Does the pointer expression to a heap cell have a particular pattern?
  4. **Variable based predicate**: Is a heap cell pointed to by a root variable?

Since these are very commonly used predicates, we have separated them out in our classification. Apart from these common predicates, summarization may be based on other predicates too depending on the requirements of a client analysis. Some examples of these predicates are: is a heap cell part of a cycle, is a heap cell pointed to by more than one fields of objects, is a heap cell allocated most recently. We group such possible predicates under **generic predicates**. A shape analysis framework [51, 52, 61] accepts any predicate as parameter to provide varying degree of efficiency and precision in the summarization technique.

The above mentioned heap summarization techniques can be combined judiciously. Most investigations indeed involve multiple summarization techniques and their variants by using additional ideas. Sections 4, 5, and 6 use the concepts defined in Section 3 to review the heap abstractions proposed in the literature.

## 4 Summarization in Storeless Heap Model

As described in Section 3.1, a storeless heap model views the heap memory as a collection of access paths. By contrast, the store based model views the memory as a graph in which nodes are heap objects and edges are fields containing addresses. The view of storeless model may seem to be a secondary view of memory that builds on a primary view of memory created by the store based model. However,
4.1 k-Limiting Summarization

May-aliases have been represented as sets of pairs of $k$-limited access paths [33]. For a program in Figure 8a with its unbounded memory graph in Figure 8b, information bounded using $k$-limiting summarization of access paths is shown in Figure 9a [33]. The method records alias pairs precisely up to $k$ dereferences and approximates beyond that. Observe that this summarization stores spurious alias relationship $\langle x.f.f.f, g \rangle$.

4.2 Summarization Using Patterns

A common theme in the literature has been to construct expressions consisting of access paths approximated and stored either as a regular expression or a context free language.

- Consider the possibility of representing access paths in terms of regular expressions [40]. For example, let $p$ be the initial access path outside a program loop. After each iteration of the loop, if the value has advanced into the heap relative to its previous value via the field left or right, then the access path can be represented as $p.(\text{left } | \text{ right})^*$. The bounded alias information for the unbounded memory graph of Figure 8b is shown in Figure 9b. The example illustrates that the method is able to identify $(f.f)$ as the repeating sequence of dereferences in the access path rooted at $x$ and $(f)$ as the repeating sequence of dereferences in the access path rooted at $w$. Observe that due to statements 3 and 4, this alias information implies that $x.(f.f)^*.g$ is aliased to $w.(f)^*.g$. However, here the inference that $x.f.f.g$ is aliased to $w.g$ is spurious.
4.2 Summarization Using Patterns

(a) Alias pairs for variables $x$ and $w$ at $Out_8$ for $k = 3$ [33].

(b) Aliases of variables $y$ and $z$ at $Out_8$ [40].

(c) Parameterized alias pairs for variables $x$ and $w$ at $Out_8$ [14].

(d) Live access graph at $In_1$ when variable $t$ is live at $Out_{10}$ [30].

(e) Direction and interference matrices for variables $x$ and $y$ at $Out_8$ [18].

**Figure 9.** Summarization techniques on a storeless model for the program in Figure 8a: $k$-limiting (Figure 9a), pattern based (Figure 9b, 9c, 9d), and other generic predicates based (Figure 9e) summarization techniques are shown. (We store aliased access paths in an equivalence class denoted by ⟨ and ⟩ in Figures 9a, 9b, and 9c.)

The general problem of detecting possible iterative accesses can be undecidable in the worst case [40]. This is because a repeated advance into the heap may arise from an arbitrarily long cycle of pointer relations. Therefore, the focus remains on detecting only consecutive repetitions of the same type of field accesses. For efficiency, finite state automata are used to compactly represent sets of access paths that share common prefixes.

- On similar lines, repetition of field dereferences (for heap liveness analysis) in program loops has been identified more efficiently and precisely as follows. In a program loop, a field reference is likely to advance inside the heap with each iteration of the loop. Therefore, access graphs are summarized by creating summarization cycles on the field nodes referenced by the same statement [30]. In other words, all the objects, dereferenced by the same field in a given program statement, are given the same name. This is shown in Figure 9d. The last field $f$ in each of the access paths $x.f$, $x.f.f.f$, and so on is dereferenced in statement 5. Therefore, all these fields $f$ (dereferenced in statement 5) are represented by the same node $f_5$. Similarly, the last field $f$ in each of the access paths $x.(f.f)^*$ is represented by the same node $f_6$ as each of them is dereferenced in statement 6. Since variable $t$ is live at $Out_{10}$, the location accessed through field $g$ of variable $y$ is live at $In_3$. Thus, the heap graph precisely marks $x.(f.f)^*$.g and $x.g$ as the live access paths at $In_1$ of program in Figure 8a. The heap graph illustrates that the method is able to identify $(f.f)$ as the repeating sequence in the live access path.

This idea has been taken further for functional programs and combined with the idea of using grammars (earlier proposed in [25], see Section 5.3) for creating the notion of a demand that the execution of an expression makes on the heap memory [2]. This is then used to perform liveness based garbage collection for functional programs.

- More precise expressions of access paths compared to those in the above methods are constructed by parameterizing the expressions with a counter to denote the number of unbounded repetitions of the expression [14]. Its algorithm is particularly interesting because, for certain programs
that manipulate lists, it offers a way of representing the exact (infinite set of) may-aliases in a compact way. Right-regular equivalence relation on access paths is used to express aliasing properties of data structures. The precisely bounded information for the unbounded memory graph of Figure 8b is illustrated in Figure 9c. The key idea of the summarization is to represent the position of an element in a recursive structure by counters denoting the number of times each recursive component of the structure has to be unfolded to give access to this element. This records the fact that the object reached after dereferencing $2^i$ number of $f$ fields on access path $x$ is aliased with the object reached after dereferencing $i$ number of $f$ fields on the access path $w$. Due to the parameterization with $2^i$ and $i$ on field $f$ of both aliased access paths which are rooted at variables $x$ and $w$ respectively, the method excludes the spurious alias pairs derived from the alias information in Figure 9b. However, this method records only may-alias information; due to the lack of performing strong updates, it cannot determine circular lists.

### 4.3 Summarization Using Generic Predicates

The study of the structure of heap allocated storage (also known as shape analysis), began by using the following predicates [18] in a storeless model of heap to classify it in terms of known shapes such as tree, DAG, and cyclic graph.\(^1\)

1. Direction relationship, which is true for pointers $x$ and $y$, if $x$ has a path to $y$.
2. Interference relationship, which is true for pointers $x$ and $y$, if a common heap object can be accessed starting from $x$ and $y$.

Direction and interference relationships are stored as direction and interference matrices, respectively (shown in Figure 9e for the program in Figure 8a). Here, the heap has been encoded as access paths in path matrices (direction and interference) at each program statement. Direction relationship between pointers $x$ and $y$ is true (represented by 1 in the direction matrix), since $x$ has a path to $y$ at $Out_{8}$ of the program in Figure 8a. Since $y$ cannot reach a node pointed to by $x$ at $Out_{8}$, 0 is marked in the corresponding entry of the direction matrix. Here, from the direction relationship, we can derive that $x$ and $y$ are not part of a cycle, since $x$ has a path to $y$, but not vice versa. Interference relationship between pointers $x$ and $y$ is true, since a common heap object can be accessed starting from $x$ and $y$.

### 5 Summarization in Store Based Heap Model

It is easier to visualize a memory graph as heap objects connected through fields. This is the view of a store based heap model as introduced in Section 3.1. The following sections summarize this unbounded view using techniques involving a combination of allocation sites, $k$-limiting, variables, and some other generic predicates.

#### 5.1 Summarization Using Allocation Sites and Variables

Chase et al. [11] were the first to summarize heap cells using techniques involving allocation sites and variables. In their method, heap cells with the following properties are summarized:

1. heap cells created at the same program point (i.e. allocation site) and
2. those that cannot be kept apart by the set of pointer variables pointing to them; nodes that are not pointed to by variables are clustered into a single node as a summary node.

---

\(^1\)Eventually, the term shape analysis started representing an analysis paradigm for answering a variety of questions related to the structure of heap in a store based model [52]. See Sections 5.2 and 5.3.
5.2 Summarization Using Variables

The method analyses Lisp-like programs and constructs shape graphs for heap variables. The abstraction is based on the premise that nodes allocated at different allocation sites need to be treated differently, while the ones allocated at a given allocation site would be updated similarly. The technique can determine the shape of the allocated heap as tree, simple cycle, and doubly linked list. In case of lists and trees, if all the nodes are allocated at the same site then the shape graph would contain a single summary node with a self loop, making all the nodes aliases to each other. To avoid this, each node is augmented with a reference count. A reference count of less than two for each node indicates that the data structure is a tree or a list. However, this shape analysis is not able to determine list reversal and list insertion programs. This is because it lacks a way to materialize summary nodes (see Section 5.2). Finally, this method does not perform strong updates for a statement of the form $x.f := null$, except under very limited circumstances.

5.2 Summarization Using Variables

Variable-based summarization technique has been used in 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 [52]. Shape analysis uses a flexible naming system of objects. Rather than giving an object the same name throughout the program, objects can change names depending on the program’s actions. Sometimes, several distinct objects with different names may be summarized (or merged), so that they have the same summarized name. Then when a summarized object is about to be pointed to by a root variable, it can be materialized, i.e. the summarized object is split into two objects with distinct names—one representing a single object and the other representing the remaining summarized objects. The basic heuristic of summarization in shape analysis is that objects that are being pointed to by root variables are represented using unique materialized objects; while a set of consecutive objects not pointed to by a root variable are summarized [60].

- To overcome the problem highlighted in Section 5.1, Sagiv et al. [49, 50] materialize copies of the summary node (representing all heap cells not pointed to by any root variable) as non-summary nodes whenever a root variable points to a previously summarized heap location. This
5.2 Summarization Using Variables

(a) Shape graph at Out₆ [49]. Pointer \( z \) materializes the summary node pointed to by \( y.f \).

(b) Shape graph at Out₆ [52]. The two summary nodes are distinguished based on whether they are reachable from root variable \( x \) or \( y \) (or \( z \)).

(c) Alias graph at Out₄ [4]. With the materialization of the most-recent Site 2, \( \langle x,y \rangle \) are marked as must-aliases.

(d) Alias graph at Out₆ [4]. Node Site 2 is not materialized further. Dashed edges denote may-alias.

Figure 11. Summarization techniques on a store based model for the program in Figure 10a. Variable based and other generic predicates based summarizations are represented in Figures 11a and 11b. Combined allocation sites and some other generic predicates based summarizations are represented in Figures 11c and 11d.

helps in achieving strong updates and helps in distinguishing nodes, which have different pointer properties in the graph. They use the following predicate in order to summarize heap cells with the same properties.

pointed-to-by-\( x \) property denotes whether a heap cell is directly pointed to by a root pointer variable \( x \).

We use the running program in Figure 10a to illustrate various forms of the shape analysis techniques. Unbounded memory graphs of the program are shown in Figure 10b and Figure 10c. Bounded shape graph at Out₆ is shown in Figure 11a. Let us see how this is obtained. Shape graph at Out₄ contains a node pointed to by both \( x \) and \( y \). This node in turn points to a summary node through link \( f \) representing an unbounded number of dereferences of field \( f \). At program statement 5, \( z \) points to a node \( y.f \). A node (pointed to by \( z \)) is created by materializing the summary node \( y.f \). At program statement 6, \( y \) points to this materialized node (pointed to by \( z \)). In the subsequent iteration of the loop, \( z \) and \( y \) point to a subsequent node. The remaining nodes (not pointed to by any of \( x \), \( y \), or \( z \)—those between \( x \) and \( y \) and those beyond \( y \)) get summarized (represented using dashed line). With this we obtain the shape graph of Figure 11a. Node \( x \) either directly points to node \( y \) (or \( z \)) or points to an unbounded number of nodes before pointing to node \( y \) (or \( z \)). Node \( y \) (or \( z \)) is followed by another unbounded number of dereferences of field \( f \). The unbounded number of nodes are represented using summary nodes.

- An imprecision in shape analysis is that its summary nodes do not permit remembering the exact count of the number of concrete nodes represented by the summary node in an abstract heap.
5.3 Summarization Using Generic Predicates

We describe below some other generic predicates based summarization techniques (Section 3.3), including TVLA, separation logic, and context free grammars that have been used for a store based

Figure 12. Variable based summarization technique on store based model. Counter automaton [7] for the program statements 5 to 6 in Figure 10a is shown. States of the automaton denote the abstract heaps at the program points shown. Edges of the automaton denote the condition of transition in the automaton. Counter variables \((i, j, \text{and } k)\) corresponding to each abstract node in the heap are depicted inside the node itself.

graph. An interesting solution to this problem is the use of a counter with every such summary node in the heap graph [7]. A summary node represents a list segment which has no incoming edges. The counter associated with each summary node denotes the number of concrete heap locations in the list segment that the summary node represents. This is used to define a counter automaton abstraction of the state transition behaviour of heap manipulating programs. This is illustrated in Figure 12 for the program in Figure 10a. The automaton starts with a heap graph containing one summary node (with counter \(i\)), pointed to by \(x\) and \(y\) at \(\text{In}_5\). It proceeds to \(\text{Out}_5\) if counter \(i > 1\), and materializes the node into a unique node (with a new counter \(j = 1\)) pointed to by \(x\) and \(y\), and the remaining summary node (with counter \(i\)) pointed to by \(z\). Here counter \(i\) used at \(\text{In}_5\) is decremented at \(\text{Out}_5\). The graph at \(\text{Out}_5\) is then transformed to \(\text{Out}_6\) under the influence of program statement 6. To further transform this graph from \(\text{Out}_6\) to \(\text{Out}_5\) in the loop, if counter \(i > 1\), it materializes the summary node pointed to by \(y\) at \(\text{Out}_6\) into a unique node (with a new counter \(k = 1\)) pointed to \(y\), and the remaining summary node (with counter \(i\)) pointed to by \(z\). Here counter \(i\) used at \(\text{Out}_6\) is decremented by one at \(\text{Out}_5\). To transform from \(\text{Out}_5\) to \(\text{Out}_6\), since \(y\) must start to point to \(z\), the node with counter \(k\) will not be pointed to by any variable. Therefore, nodes with counters \(k\) and \(j\) are merged, and their counter values updated (added up) at \(\text{Out}_6\).
5.3 Summarization Using Generic Predicates

heap model.

- As an improvement over the summarization technique using only variables [49,50], the following predicates are used in order to summarize heap cells more precisely [51,52,61].
  
  - pointed-to-by-x property denotes whether a heap cell is directly pointed to by a root pointer variable x.
  
  - reachable-from-x-via-f property denotes whether a heap cell is transitively reachable from a root pointer variable x via field f.

We use the running program in Figure 10a to illustrate the summarization. Unbounded memory graph at Out6 of the program is shown in Figure 10c. Bounded shape graph using predicates points-to-by-x and reachable-from-x-via-f for summarization at Out6 is shown in Figure 11b. We have already explained the bounded shape graph obtained using only points-to-by-x predicate for summarization at Out6 in Figure 11a (see Section 5.2). Observe that the bounded shape graphs obtained are the same with respect to the nodes pointed by a root pointer variable. However, they differ with respect to the summary nodes (those not pointed to by any root pointer variable) due to the use of the additional predicate (reachable-from-x-via-f); the summary node that follows the node pointed to by the root variable x is kept separate from the summary node that follows the node pointed to by the root variable y (or z). Therefore, the shape graph represents unbounded dereferences of field f following root node x and another sequence of unbounded dereferences of field f following root node y (or z).

Shape analysis techniques are precise for a single field name, but it is not clear if the abstraction is sensitive to multiple field names. 3-valued predicates in TVLA [51,52,61] are used to describe false, true, and don’t know properties of the shape graph, like pointer information (may and must), heap cell being part of a cycle, and others. Predicate logic allows them to use shape analysis for identifying any desired heap property. Shape analysis stores and summarizes heap information precisely, but is very expensive at the same time.

- The efficiency of shape analysis can be boosted by representing independent portions of the heap using formulae in separation logic [45]. Separation logic has primarily revolved around store based modeling. It is particularly useful for interprocedural analysis because the portion of the heap that is not reachable by a called procedure can be easily separated from the rest and later recombined with the modified heap on encountering procedure return [20]. Furthermore, spatial locality of code (i.e. the fact that each program statement accesses only a very limited portion of the concrete state) matches the spatial locality in the representation, dramatically reducing the amount of reasoning that must be performed when summarizing a procedure. An interprocedural analysis exploits the locality by using separation logic’s frame rule [20]:

\[
\frac{\{P\} foo() \{Q\}}{\{P*R\} foo() \{Q*R\}}
\]

This rules says that if the heap before execution of function foo is P and the execution of foo transforms it to Q, then if a disjoint heap region R is added to P before the execution of foo, it remains unchanged by the execution of foo.

This can be used in the following manner to increase the efficiency of analysis: To process a call to foo, it determines the part of the symbolic heap at the call site that needs to be made available to the procedure. Consider the following example [20]. Suppose that before executing the procedure call foo(u) (with actual parameter u and formal parameter x), we have a symbolic heap in separation logic as list(u,v) ∗ list(v,null) ∗ u ⇒ v. The part of the heap reachable from the actual parameter u is list(u,v) ∗ list(v,null). Here v is the cutpoint. Cutpoints are
objects, which separate the local heap (pointed by \( v \)) of the invoked function from the rest of the heap (pointed by \( w \)). Each cutpoint variable (in this case \( v \)) should be given a name, say \( X \), which does not clash with the local variables of the called procedure. Also the actual parameters (in this case \( u \)) are mapped to the formal parameters (in this case \( x \)). Therefore, the symbolic heap
\[
\text{list}(x,X) \ast \text{list}(X,\text{null})
\]
is passed to function \( \text{foo()} \).

Since each cutpoint variable has to be given a unique name, this may become unbounded in the case of recursive procedures. To solve this problem, a \( k \)-bound on the maximum number of cutpoints that can appear in the symbolic heap is used. Consider the above example and suppose that \( k = 0 \). Then the local heap \( \text{list}(x,X) \ast \text{list}(X,\text{null}) \), which can be sent to \( \text{foo()} \) will be \( \text{list}(x,\text{null}) \). In this case, the information that \( w \) points to a node in the list, is lost.

- Assertions expressed in separation logic may produce infinite sets of concrete states. A fixed point computation can be achieved using finitely represented inductive predicate assertions [8, 20] like \( \text{list()} \), \( \text{tree}() \), \( \text{dlist}() \), representing unbounded number of concrete states, shaped like a linked list, tree, doubly linked list, respectively. The abstraction comes from not tracking the precise number of inductive unfoldings from the base case. Let us work out the symbolic heap for our program in Figure 10a. For a pre-condition at \( \text{In}_1 \), denoted as the symbolic heap \( x \rightarrow X \ast x \rightarrow y \rightarrow Y \) (for some variables \( X \) and \( Y \)), the post-condition at \( \text{Out}_4 \) is \( y = z \wedge \text{list}(x,z) \ast \text{list}(z,\text{null}) \) (as also shown using the runtime heap in Figure 10c). Let us see how this is obtained by first synthesizing the post-condition at \( \text{Out}_4 \), i.e. after the first program loop. Just after program statement 2, we obtain the assertion \( x = \text{null} \wedge y \rightarrow \text{null} \) which says that \( x \) holds a null value and \( y \) is an allocated cell pointing to a null value. After statement 3, the heap is expressed using the assertion \( x = \text{null} \wedge y \rightarrow x \), which says that \( x \) holds a null value and \( y \) is an allocated cell pointing to \( x \). After program statement 4, the assertion becomes \( x = y \wedge x \rightarrow \text{null} \). In the second iteration, we obtain the assertion \( x = y \wedge x \rightarrow X' \ast X' \rightarrow \text{null} \), where \( X' \) is a new variable, not used anywhere in the program. In the third iteration, we obtain \( x = y \wedge x \rightarrow X'' \ast X'' \rightarrow X' \ast X' \rightarrow \text{null} \), for another new variable \( X'' \). If we continue in this way, we will get ever longer formulae. This unboundedness is abstracted using a predicate \( \text{list}() \), where \( \text{list}(u,v) \) says that there is a linked list segment of indeterminate length from \( u \) to \( v \). Therefore, we obtain the abstraction by using the following operation in the second iteration at \( \text{Out}_4 \):

\[
\text{replace } x = y \wedge x \rightarrow X' \ast X' \rightarrow \text{null} \text{ with } x = y \wedge \text{list}(x,\text{null})
\]

This is in sync with the runtime heap of Figure 10b. Using a similar way of synthesizing, the assertion at \( \text{Out}_6 \) can be obtained to be \( y = z \wedge \text{list}(x,z) \ast \text{list}(z,\text{null}) \). This also agrees with the runtime heap of Figure 10c.

- Context free grammar has been derived for functional programs consisting of list type primitive functions (\( \text{cons,car,cdr} \)) [25]. We describe it by reproducing the following definition of function \( \text{APPEND} \).

\[
\text{APPEND}(x,y) = \begin{cases} 
\text{if null}(x) \text{ then } y \\
\text{else } \text{cons}(\text{car}(x), \text{APPEND}(\text{cdr}(x), y))
\end{cases}
\]

Each argument of the function is represented by a non-terminal in a context free grammar. Derivation rule for variable \( x_1 \), which is the \( i \)-th argument of function \( F \) is \( F_i \rightarrow s_1 \mid s_2 \mid \cdots \mid s_k \), where \( s_1, s_2, \ldots, s_k \) are all the strings obtained from the function body. These strings consist of \( \text{car}_1, \text{cdr}_1, \text{cons}_1 \), and \( \text{cons}_2 \), and user defined functions. Each function name is used with a subscript indicating the argument position of variable \( x_1 \) in the function. Using the function body of \( \text{APPEND} \), its context free grammar is given below.
APPEND\(_2\) denotes the values, which the second argument of APPEND, \(y\) can take. It would either be null or would be passed as the second argument to cons (denoted by cons\(_2\)). APPEND\(_1\) denotes the values, which the first argument of APPEND, \(x\) can take. The strings generated by APPEND\(_1\) are of the form cons\(_x\).cons\(_1\).car\(_1\).cdr\(_1\). By reading the string in the reverse order, we can see that APPEND decomposes list \(x, k\) number of times by the application of cdr, and then a car selects the element at that position, followed by a cons\(_1\) on the element to make it the left child of a new cell, which itself will be acted on by cons\(_2\) the same \(k\) number of times.

This grammar has been used to detect garbage cells in a functional programming language through compile time analysis [25]. It is based on the idea that the unshared cells that have been created as a result of a function call but not inherited in the result of an outer function call will not be used. Hence, they can be garbage collected after the outer function call. However, it does not perform full sharing analysis.

- Another way of building context free grammar of heap access paths is by posing shape analysis as CFL reachability problem. This has been done for Lisp like languages that do not support strong updates [44]. Equation dependence graph is constructed using the program’s control flow graph. An equation dependence graph is constructed by marking all program variables at each program point. The edges between these variables are labelled with head, tail, and tail\(^{-1}\). For example, for program statement \(x := \text{cons}(y, z)\) at program point \(n\), program variables \(x, y,\) and \(z\) are marked at program points \(n\) and \(n + 1\). The label head is marked on the edge from variable \(y\) at \(n\) to \(x\) at \(n + 1\) and tail on the edge from variable \(z\) at \(n\) to \(x\) at \(n + 1\). This denotes that \(x\) at \(n + 1\) derives its head from \(y\) at \(n\) and tail from \(z\) at \(n\). For program statement \(x := \text{car}(y)\) at program point \(n\), head\(^{-1}\) is labelled on the edge from variable \(y\) at \(n\) to \(x\) at \(n + 1\). This denotes that \(x\) gets its value using head of \(y\). Similarly, tail\(^{-1}\) is labelled for program statement \(x := \text{cdr}(y)\).

Heap language (expressed in terms of access paths) is identified by traversing paths of this equation dependence graph. Each path defines a word, which is obtained by concatenating, in order, the labels of the edges on the path. Shape information can be obtained by solving CFL reachability problem on the equation dependence graph. The following CFL reachability problems have been defined using context free grammars [44]:

- A CFL representing paths in which the number of head\(^{-1}\)(tail\(^{-1}\)) should be balanced by a matching number of head(tail), implying that the heap was used (through head\(^{-1}\)(tail\(^{-1}\))) as much as it was constructed (using head(tail)).

- A CFL representing paths in which the number of head(tail) may be more than the number of head\(^{-1}\)(tail\(^{-1}\)), implying that the amount of heap allocated using head(tail) is more than the amount of heap dereferenced using head\(^{-1}\)(tail\(^{-1}\)).

### 5.4 Summarization Using Allocation Sites and Other Generic Predicates

As an attempt to reduce the cost of shape analysis, recency-abstraction [4] is used as an approximation of heap allocated storage. Here, only the most recently allocated node at an allocation site is kept materialized representing a unique node. Therefore, its precision level is intermediate between (a) one summary node per allocation site and (b) complex shape abstractions. Note that for the program in Figure 10a, summarization based only on allocation sites creates a summary node for objects allocated at site 2. Root variables \(x\) and \(y\) point to a summary node. Therefore, allocation site based summarization cannot derive that \(x\) and \(y\) are must-aliased. Recency-abstraction is illustrated in
Figure 11c for the unbounded graph of Figure 10b. Due to materialization of the most recently allocated node, the method is able to precisely mark $x$ and $y$ as must-aliases. But materializing only once is not enough and introduces imprecision at `out_g`. This is shown in Figure 11d, where $y$ and $z$ are marked as may-aliases (instead of the precise must-alias, as shown by the unbounded runtime memory graph in Figure 10c).

5.5 Summarization Using Allocation Sites

Object sensitivity allows an analysis to distinguish between the receiver objects created by the same allocation site [56]. The context of a method call `obj.f()` consists primarily of the allocation site of the method’s receiver object `obj`, which can then be refined further by the allocation site of object `obj'` that allocated the method’s receiver object `obj`. In principle, the context can be refined further by including the allocation site of object `obj''` that created `obj'` and taking a transitive closure.

Even though object sensitivity has been used for context disambiguation, we have presented it under heap summarizations because it can be used to make allocation site based summarizations more precise by disambiguating object contexts based on all the transitively used allocation sites.

5.6 Summarization Using Allocation Sites and $k$-Limiting

Using the store based model, alias graphs record may-aliases [34]. The bounded representation of the unbounded memory graph of Figure 8b is illustrated in Figure 13a using this technique.\(^2\) The program in Figure 8a shows that memory locations `z.(f)^*` (or `v.(f)^*`) are allocated at program statement 7. A self loop around node (marked with `Site 7`) denotes unbounded dereferences of field `f`. In other words, this abstract node represents multiple concrete nodes. However, this summarization spuriously stores the alias relationship `(x.f.f.g,v.f.f.g)` too. Note that all paths are limited to $k$ fields of dereferences. In Figure 13a, $k$ is set to 4.

In the second approach, nodes allocated by same allocation site are distinguished by labeling a newly allocated node by an aggregate of the labels of the arguments to the allocation function (cons in Lisp). In order to limit the abstract graph to a finite size, summary nodes are created using the concept of $s$-$l$ limiting in which no node has more than $s$ outgoing edges (other than the nodes representing the bottom element), and no node has a label longer than $l$. It uses regular expressions (limited to be no longer than a chosen constant $l$) to label each node of the shape graph. However, as pointed out by Sagiv et al. [50], these shape node labels do not add any information to its representation because the pointer access expressions can always be reconstructed from the graph stripped of node labels.

6 Summarization in Hybrid Heap Model

For some heap applications, the heap memory is best viewed as a hybrid model combining the storeless and the store based heap model. This model can also be summarized using various techniques, like $k$-limiting, variable, and other generic predicates.

6.1 $k$-Limiting Summarization

De et al. [12] highlight an imprecision in saving pointer information as graphs. We illustrate this imprecision using statements 9 and 10 of our running program in Figure 8a on the graph created in Figure 13a. The problem is caused due to the fact that a summarized object node may represent multiple concrete objects; therefore, the analysis cannot perform a strong update on such objects. At `ln_g` of the program, `y` is aliased to the summary node `x.f.f.f(.f)^+` of Figure 13a (`y` has not been shown in Figure 13a but is shown in the unbounded graph in Figure 8b). Therefore, strong update cannot be

\(^2\) Although the technique formulates its analysis for Lisp like programs, we have extended it for our imperative program.
Summarization techniques for the program in Figure 8a: Combined allocation site based approach for $k = 4$ [34].

De et al. [12] believe that this imprecision is caused by storing points-to information as graphs. Therefore, instead of using graphs, they use access paths. Their technique maps $k$-limited access paths (storeless model) to sets of summarized objects (store based model) (represented as $o\langle n \rangle$ in Figure 13b and Figure 13c). For example, $x \Rightarrow \{o1\}$ means that the access path $x$ points to (is mapped to) the object named $o1$. Since the access paths are precise up to $k$ length, it can perform strong updates up to $k$ length. In Figure 13b at $\text{In}_9$, $y$ points to a summarized object $\{o3,o5\}$ (pointed to by $x.f.f$ and $x.f.f.f.f$, respectively). In Figure 13c, at $\text{Out}_{10}$, it is safe to strongly update the access path $y.f$ to
\{o11\} (pointed by u), even though y points to multiple objects \(o3\) and \(o5\) at \(In_9\). Therefore, at \(Out_{10}\), access path v also precisely points to only the new object \{o11\} (pointed by u).

### 6.2 Summarization Using Variables and Other Generic Predicates

We describe below some application specific predicates that have been used in a hybrid model.

- In order to remove unreachable parts of the heap across functions in interprocedural analysis, cutpoints are marked on the heap [46]. Cutpoints are objects, which separate the local heap of the invoked function from the rest of the heap. 3-valued logic shape analysis (earlier classified under the store based model) is used for summarization [51]. A predicate for the cutpoint summarization is “Is the heap cell in the graph a cutpoint?”. Each cutpoint is identified by an access path (a feature of a storeless model) which is not relevant to the function being called. When the function returns, the access path of the cutpoint object is used to update the caller’s local heap with the effect of the call. Therefore, irrelevant parts of abstract states that will not be used during the analysis are removed by modeling the heap using both storeless and store based representations.

For example, an acyclic list pointed to by x is passed to the reverse() function, which reverses the list performing strong updates. Let us say, before the function call, y.g.g and x.f are aliased and y is not in scope of function reverse(). On return of the function, we should be able to derive that y.g.g.f and x are aliased. To capture this kind of a relationship, effect of the function on cutpoints is tracked. In this example, the second node of list x is a cutpoint and in the function reverse() can be identified with a new alias relationship between access paths as \(\langle C, x.f \rangle\), where C is the access path used to label the second node (cutpoint) in the list. On return of the function reverse(), we will derive \(\langle x, C.f \rangle\) as the alias relationship. Thus, we will be able to restore the alias relationship between x and y as \(\langle x, y.g.g.f \rangle\) in the calling function.

- Connection analysis (similar to access paths used in a storeless model) along with store based points-to analysis has been used as an abstraction [19]. This method first resolves all pointer relationships on the stack using a store based points-to analysis, which abstracts all heap locations as a single symbolic location called heap. All pointers reported to be pointing to heap are then further analyzed via a storeless heap analysis, called connection analysis, and shape analysis.

### 7 Comparing Heap Abstraction Techniques

In this section, we compare and contrast the heap models and their summarization techniques.

#### 7.1 Storeless Vs. Store Based Models

The store based models identify possible targets of a given pointer. They generally represent the heap as graphs and have the following advantages over a storeless model.

1. **Efficiency.** Due to their close resemblance with the memory graph, store based models provide an efficient method to store the structure of heap memory; therefore, also an efficient procedure to merge the graphs at control flow graph meet points. However, it may be possible to devise a more efficient way of storing access paths, perhaps using some variant of access graphs [30].

2. **Conceptual simplicity.** A store based model seems more intuitive to visualize and the mapping from the runtime locations to heap graph is easier. The storeless models seem more difficult to conceive. This is because in a storeless model, each runtime location needs to be mapped to an access path. This mapping varies dynamically due to the arbitrariness of heap manipulating...
programs. Also, the length of access paths is unbounded due to the presence of program loops, recursion, and cyclic data structures.

3. **Proving soundness.** Since store based model is a collection of the runtime concrete memory graphs, proving soundness of a store based analysis is more direct as compared to an analysis involving a storeless model.

4. **Applicability.** The literature has an abundance of investigations involving store based models and their uses for numerous applications. A large number of them deal with the problem of determining the shape of a data structure in particular or the structure of heap memory in general. These shape analyses are able to detect whether some data forms a list or not, which is important for some applications such as those which parallelize loops over lists [3]. In contrast, there are only a few storeless modeled analyses (for example, [18]), which deal with the problem of determining the shape of a data structure.

The Storeless models identify relationships between pointers. They generally represent the heap as a collection of access paths and have the following advantages over a store based model.

1. **Ease of reasoning.** Access paths are closer to the static program text whereas the shape graphs used in store based model are closer to the runtime behaviour of the program. Therefore in some heap analysis algorithms, it is easier to reason about a program in terms of access paths than in terms of shape graphs. Ghiya and Hendren [19] demonstrate that store based approaches are more suited to analyze stack directed pointers because they have compile time names. For analyzing heap directed pointers, storeless models are better suited because heap locations do not have compile time names and are anonymous; naming them using access paths seems to be the most natural way.

2. **Precision.** Treating the heap as a set of statically named locations (as done in a store based model) can lead to significant imprecision because one name can represent several heap objects which may be completely unrelated [13]. Storeless heap models, therefore, in general provide more accurate information.

### 7.2 Summarization Techniques

Parameterised shape analysis frameworks [51, 52, 61] summarize the heap data based on any desired client predicates. Thus they are the most general among all summarization techniques and help in achieving varying degrees of precision and efficiency based on the client requirements.

#### 7.2.1 Efficiency

In this section we discuss the impact of various summarization techniques on the efficiency of heap analysis.

- **k-limiting.** The efficiency of the analysis is heavily dependent on the value of $k$; larger values improve the results but may slow down the analysis significantly [3]. The analysis may be extremely costly because the number of possible shape graphs is doubly exponential in $k$ [49]. This is because heap locations beyond $k$ dereferences have to be (conservatively) assumed to be aliased to every other heap location. Hence, $k$-limiting is practically feasible only for small values such as $k \leq 2$ [53]. Observe that it is difficult to know the best value of $k$ that should be chosen for a particular program.

- **Allocation sites.** This technique models the heap as a store based model and names them by the program point of their creation. Therefore it has the advantages of a store based model described
in Section 7.1. As noted by Sagiv et al. [50], the techniques based on allocation sites are more efficient than $k$-limiting summarizations, both from a theoretical perspective [11] and from an implementation perspective [3]. The size of an allocation site based graph is bounded by the number of allocation sites in the program. Therefore, this method is likely to be space efficient for most practical programs.

- **Patterns.** Identifying precise repeating patterns is undecidable in the most general case because a repeated advance into the heap may arise from an arbitrarily long cycle of field dereferences [40]. However, practical observations have found this approach to be more efficient than TVLA based shape analysis techniques [30].

- **Variables.** This summarization technique generally uses a points-to graph (in a store based model). Therefore it has the advantages of a store based model highlighted in Section 7.1. Since each shape graph node is labelled with a set of root variables in this technique, the number of shape nodes is bounded by $2^{\text{Var}}$, where $\text{Var}$ is the number of variables in the program. Unfortunately for some pathological programs the number of shape nodes can actually grow to be this large (although Sagiv et al. [49] believe that it is unlikely to arise in practice).

- **Generic Predicates.** This category of summarization includes multiple predicates. The efficiency of the algorithm depends on the chosen predicate. Implementation of generic predicates using TVLA [51] has potentially exponential runtime in the number of predicates. Therefore, it is not suitable for large programs.

### 7.2.2 Precision

In this section we discuss the impact of various summarization techniques on the precision of the information discovered by a heap analysis.

- **$k$-limiting.** This technique yields poor results for programs that manipulate elements beyond $k$. An imprecision is illustrated in Figures 5b and 6b. Methods using $k$-limiting summarizations merge all the paths that are longer than a fixed constant $k$. The tail of even a linear linked list will be (conservatively) treated as an arbitrary and possibly cyclic data structure. Due to the over-approximation of information beyond $k$ dereferences, these algorithms lack strong nullification operations. Therefore, they cannot infer that the assignment $x.f := \text{null}$ in the program cuts the list pointed to by $x$. Consequently, they cannot determine that either “list-ness” or “circular list-ness” is preserved by a program that inserts an element into a list [49].

- **Allocation sites.** This summarization has some imprecision, since nodes allocated at the same allocation site but called from different contexts, may have different properties. Figure 13a contains an example of imprecision using allocation sites. In fact this technique could be highly imprecise if all the allocations in the program are made through a single function which in turn allocates a location (using for example, `new` or `malloc`). This is because it would summarize all heap nodes to a single graph node. Therefore, allocation site information alone is insufficient to determine interesting facts in many programs. For example, it cannot determine that “list-ness” is preserved for either the insert program or the reverse program on a list [11].

- **Patterns.** Heap manipulations consist of repeating patterns of access paths which bear a close resemblance to the program structure [30]. Storeless model allows an algorithm to find this pattern using access paths. This storeless modeling has reasoning and precision benefits over store based model, as highlighted in Section 7.1. However, finding a precise repeating pattern is undecidable in the most general case. Therefore, generally the focus remains on detecting only consecutive repetitions of the same type of field accesses which may be imprecise. Also, since
the length of access paths is unbounded, it seems difficult for an analysis to precisely determine if an identified repetition will occur an unbounded number of times.

- **Variables.** This method can perform strong nullification: when \( x \) is in the variable-set of a shape node, a strong nullification (and hence a strong update) can be performed on the field edges emanating from the shape node. However, there are situations where variable based summarization can be imprecise. Since it merges nodes not pointed to by any root variable, sometimes nodes are abstracted imprecisely. This imprecision is illustrated in Figure 5d. Contrast this with the summarization of Figure 5c.

- **Generic Predicates.** This category of summarization includes multiple predicates. The precision of the algorithm depends on the chosen predicate. By identifying one or more suitable predicates, the client heap analysis can be made very precise.

8 Conclusions

In this literature survey, we have presented a high level view of the heap abstraction techniques in terms of modeling and summarization. The heap models are classified as storeless, store based, and hybrid. The summarization techniques use \( k \)-limiting, allocation sites, patterns, variables, and other generic predicates. Summarization using allocation sites is a commonly used technique, but is too coarse for some clients. Shape analysis is precise but is inefficient. As program analysts, we are still facing the challenge of creating summarizations that are efficient and yield results that are precise enough to be useful.

Due to a combinatorially large pointer information and its computationally complex analysis, researchers have focussed on engineering techniques for building efficient approximations. Michael Hind observes that although the problem of pointer analysis is undecidable, “fortunately many approximations exists” and goes on to note that “unfortunately too many approximations exist” [22]. We view this trend as unwelcome because a large fraction of pointer analysis community seems to believe that compromising on precision is necessary for scalability and efficiency. Amer Diwan adds, “It is easy to make pointer analysis that is very fast and scales to large programs. But are the results worth anything?” [22].

In our opinion, a more desirable approach is to begin with a careful and precise modeling of the required information even if it is not computable. Then the analysis can be gradually refined into a computable version which can further be refined to make it scalable and efficient to make it practically viable. Tom Reps notes that “There are some interesting precision/efficiency trade-offs: for instance, it can be the case that a more precise pointer analysis runs more quickly than a less precise one” [22]. Various implementations [29, 36, 56] show that this top-down approach does not hinder efficiency. In fact increased precision in pointer information not only causes a subsequent (dependent) analysis to produce more precise results, it also causes the subsequent analysis to run faster [55].

Without undermining the engineering efforts, we believe that a search for approximations should begin only after building clean abstractions and not before it [28].

Acknowledgements

An invigorating discussion on the Dagstuhl Seminar on Pointer Analysis [37] sowed the seeds of this survey paper. We would like to thank Amitabha Sanyal for his comments on this paper as also for valuable discussions related to heap analysis from time to time. Alefiya Lightwala helped in rewording some text and Prakash Agrawal pointed out some errors in the shape analysis examples. Vini Kanvar is partially supported by TCS Fellowship.
References


A Heap Analyses and Their Applications

In this section, we describe major heap analyses and their applications.

A.1 Heap Analyses

A compile-time program analysis that needs to discover and verify properties of heap data could perform one or more of the following analyses.

- **Shape Analysis** [18, 51, 61] also called *storage analysis* discovers invariants that describe the data structures in a program and identifies alias relationships between paths in the heap. Its applications include program understanding and debugging [16], compile time detection of memory and logical errors, establishing shape properties, code optimizations, and others.

- **Liveness Analysis** for scalar variables essentially involves determining whether a variable is used in the future and is relatively much simpler for scalar variables because it does not have to consider pointer dereferencing. Basically, liveness analysis for heap data identifies heap objects that would be used in the future. These are used to identify reachable yet unused heap locations to perform garbage collection of such heap locations [2, 25, 30].

- **Escape Analysis** is a method for determining the dynamic scope or lifetime of pointers including heap pointers. The analysis is used for (a) scalar replacement of fields, (b) synchronization removal, and (c) stack allocation of heap objects [32].

- **Side-Effect Analysis** determines the memory locations that may be read or modified by the execution of a program statement. This analysis can optimize code by eliminating redundant loads and stores [41].

- **Def-Use Analysis** identifies pairs of statements that set the value of a memory location and subsequently use that value. This analysis is used to check for the uses of undefined variables and unused variables [41].

- **Heap Reachability Analysis** finds whether an object can be reached from another variable or object via pointer dereferences. Precise reachability information is useful for a number of clients, including compile time detection of memory leaks [6].
• **Object Sensitive Analysis** was first introduced by Milanova et al. [41]. This is a form of context sensitivity for object-oriented languages. This analysis uses receiver objects of a method call to distinguish contexts. This object sensitive analysis can avoid conflation of analyses performed in methods called by distinct objects.

• **Call Graph Construction** requires analysis of pointers (including heap). In object-oriented languages, virtual calls make building a call graph non-trivial. Instead of relying on a call graph constructed with some less precise points-to analysis (note that points-to analysis and call graph construction are interdependent), the program call graph has also been constructed on-the-fly with pointer analysis [38, 42, 47, 57, 59].

### A.2 Applications of Heap Analyses

We present the applications of pointer analysis (in particular heap analysis) under these three broad categories.

- For understanding heap manipulating programs, software engineering techniques based on heap analysis are used to re-engineer or reverse engineer programs (Section A.2.1).

- Heap analysis is used in software verification to prove soundness of programs (Section A.2.2).

- Finally, modern compilers use heap analysis results to produce code that maximizes performance (Section A.2.3).

Various heap analyses and their applications have been presented by Sagiv et al. [52] and Sridharan [57].

#### A.2.1 Program Understanding

Heap related information like shape, size, connectivity, cyclicity, and others are collected in order to inspect, understand, and debug programs.

Program slicing can help in program understanding by extracting a part of the program that is relevant to the user’s constraints/conditions. For example, the user may want to study the reads and writes on a particular (dynamically allocated) resource in the program. Slicing has been performed for programs that can access and manipulate heap structures [31]. Slicing has found varied applications, including debugging, software testing, software metrics, program understanding, program parallelization, and state-space reduction for efficient verification.

#### A.2.2 Verification and Validation

Usage of pointers is error prone. The following introduce various verification and validations performed on user programs at compile time to prevent memory and logical errors before the code is actually executed.

*Memory Errors.* Heap analysis is used for automatic compile time detection of memory and logical errors (dereferencing null pointers, dereferencing dangling pointers, memory leaks, freeing a block of memory more than once, and premature deallocation) [19, 24, 39, 54].

*Verification of Specifications.* Verification of sorting programs that manipulate linked lists takes into consideration potential use of heap structures [35]. The paper [35] demonstrates how verification can show that a sorting algorithm is sufficiently precise to discover that (correct versions of) bubble sort and insertion sort procedures, do in fact, produce correctly sorted lists as outputs, and that the invariant \( is\text{-}sorted \) is maintained by list manipulation operations such as element insertion, element deletion, and even list reversal using strong updates and merging of two sorted lists. When the algorithm is run on erroneous versions of bubble sort and insertion sort procedures, it is able to discover and sometimes even locate and diagnose the error.
Heap Abstractions

Checking Security Attacks. Pointer analysis is important in security vulnerability checks. Windows operating system kernel has been validated against an important security property [15]. The paper explains this vulnerability as follows: the kernel provides a set of API functions that can be called from a user program, to perform tasks requiring the kernel to read data from, or store data into locations referenced by the pointer values passed in as arguments. One possible source of security attack is that a user program may pass in pointers that point into the reserved section, thereby overwriting system data. A validation tool ensures that all user mode pointers are checked to ensure that they do not point into kernel memory, before they are dereferenced. Following are some other security vulnerabilities, which involve data structures, such as buffers and file handles that are allocated on the heap and manipulated through pointers. (a) Format string vulnerability occurs when untrusted data ends up as part of the format string argument to Standard C Library, such as printf(). (b) Remote access vulnerability: hostile clients can manipulate programs through various program inputs [21].

Verification in Multithreaded Systems. Concurrent software is becoming increasingly important due to the increased use of multi-core computing architectures. 3-valued logic can be used for checking properties of multithreaded systems [62]. Multithreaded systems require the verification of race and deadlock freedom properties while soundly summarizing the dynamic behaviours of the program [17].

A.2.3 Optimizations

The goal of pointer analysis is to disambiguate memory references of the program; thereby simplifying the program’s constructs. Subsequent compiler passes, are therefore, more effective in their optimizations when they work on this disambiguated program. Basically, pointer analysis algorithm has been used to extend several standard scalar compiler optimizations, like dead code elimination of unused pointer variables [19]. Another optimization of heap manipulating programs is the garbage collection of accessible yet non-live objects [2,30].

Parallelization and Locality Optimizations. Due to the presence of pointer variables, automatic parallelization for modern languages like Java, C++, and even C, is not mature enough. To maximize the opportunities for automatic parallelization and instruction scheduling, we need precise program dependence information [52]. Note that program statement $S_1$ depends on program statement $S_2$ if the value written to a resource in $S_1$ is directly used at $S_2$, that is, without intervening writes to this resource. Since this resource may be a heap allocated storage, we need to perform heap analysis in order to determine the program dependences. Sagiv et al. [52] highlight literature that have used shape analysis to construct program dependences. These dependences are identified in sequential programs in order to enable a parallelizing compiler to transform sequential input program for better parallel execution [5]. Pointer analysis also helps in performing data prefetching based on future uses and updates on heap data structures in the program [19]. Data locality of dynamically allocated data has been identified and exploited using heap analysis by Castillo et al. [9].

Optimization of Multithreaded Systems. Elimination of unnecessary synchronizations helps in optimizing multithreaded heap manipulating programs. Thread-state reachability analysis is also used in optimization of multithreaded programs. It asks if a given thread state is reachable by executing a program with some (possibly infinite) number of threads in parallel; here thread-state comprises of the local state of one thread, along with a valuation of the shared program variables. Compilers perform optimizations on thread-state invariant variables in programs [17]. These are invariants that do not refer to the local variables of more than one thread.

B Heap and Stack Memory in C/C++ and Java

In this section, we briefly compare the programming constructs related to pointer variables in C/C++ and Java programs.
Referencing variables on stack and heap. In C/C++, both stack and heap allow pointer variables. Java does not allow stack directed pointers. C/C++ allows pointers to variables on the stack through the use of addressof operator &; Java does not have this operator. Both C/C++ and Java allow pointers/references to objects on the heap using malloc function (in C/C++) and new operator (in C++ and Java).

Dereferencing pointers. Every variable on the stack, whether it contains a reference or a value, always has a name because all the objects allocated on the stack have compile time names associated with them. Heap allocated data items do not possess names and are all anonymous. The only way to access heap items is using pointer dereferences. C/C++ has explicit pointers. Pointer variables in C/C++ are dereferenced using star operator (*), for example, y := *x. Fields of a pointer to an aggregate data type (struct, union, or class) can be accessed using star operator (*) and dot operator (.), for example, (*x).f, or using arrow operator (→), for example, x->f; both are equivalent pointer dereferences of the member field f of pointer variable x. In Java, fields are dereferenced using the dot operator (.), for example, x.f.

Analysis of scalar and aggregate pointers. In Java, a pointer variable cannot point to an object of scalar data type such as integer or floating point number; pointer variables point to an object of only aggregate data types in Java such as structures, classes etc. However, C/C++ allows pointers to both scalars and aggregate structures. In C++, pointer analysis of scalar variables is comparatively straightforward (due to type restrictions) as compared to the pointer analysis of aggregate variables. For example, a program statement x := *x is syntactically invalid—the scalar pointer x cannot advance to a location of a different data type. On the other hand an aggregate pointer can be advanced subject to its type compatibility making it difficult to find properties of such pointers. For example, program statement x := x->f in a loop allows the aggregate pointer x to point to any location after x through field f. Further, cycles in recursive data structures, cause infinite number of paths that refer to the same memory location. This makes the analysis of an aggregate pointer challenging over a scalar pointer.

Mapping C/C++ memory to the Java memory. As explained before, C/C++ heap and stack pointers can point to locations on both stack and heap. On the other hand, Java stack pointers can point only to Java heap locations. In spite of this difference in memory modeling, stack and heap memory in
C/C++ can be modeled like a Java memory. To achieve this, C/C++ memory is viewed as consisting of two partitions of the memory—addresses of variables and the rest of the memory (stack and heap together) [30]. Here, the first partition of the C/C++ memory (i.e. the addresses of variables) works like the Java stack. The second partition of the C/C++ memory consisting of the rest of the memory (stack and heap together) works like the Java heap.

Figure 14 illustrates a C/C++ memory snapshot, which has been modeled as Java memory (in dotted lines). Pointer variables w, x, y, and z are on the C/C++ stack and pointer variables A, B, C, and D are on the Java stack. C/C++ pointers point to stack variables x and z in the figure. The stack and heap of C/C++ are represented as the Java heap. Java stack is the set of addresses of C/C++ locations (viz. w, x, y, and z) stored in A, B, C, and D, respectively. To overcome the difference of pointer dereferences (\*) and addressof (&) operator in C/C++ which are absent in Java, Khedker et al. [30] model these two C/C++ constructs as follows:

- Pointer dereference (*) is considered as a field dereference \texttt{deref}, which has not been used elsewhere in the program. In an example from the paper [30], \texttt{(*x).f} in C/C++ is viewed as \texttt{x.deref.f} in Java.

- The addresses of C/C++ variables are represented by the Java stack (as shown in figure 14, where A denotes &w, B denotes &x, C denotes &y, and D denotes &z). In an example from the paper [30], \texttt{y.f} in Java is modeled as \texttt{&y.deref.f} in C/C++.

C Engineering Approximations for Efficiency

Given the vital importance of pointer analysis and the inherent difficulty of performing precise pointer analysis for practical programs [10, 23, 33, 43], a large number of investigations involve a significant amount of engineering approximations [28]. A detailed description of these is beyond the scope of this paper because its focus is to build concepts of various modeling and summarization techniques. Here we merely list some notable efforts in engineering approximations used in heap analysis.

Since heap data is huge at compile time Calcagno et al. [8] perform compositional/modularized analysis, i.e. using function summaries. Heap data can also be restricted by propagating the part of the heap that is sufficient for a procedure [8, 20, 46]. Amount of heap data collection can be controlled by a demand-driven analysis using client intervention [21, 57]. Rountev et al. [48] limit the program scope where high precision is required. For example, they determine program fragments where accuracy is vital (like certain regions of code, certain pointer variables) and find ways to improve results just for those critical regions. They have also performed safe analysis for incomplete programs. Limiting the analysis to live and defined variables of the program has also helped in achieving scalability without any loss of precision [1, 12, 29]. Incremental computations [58] and efficient encoding of information by using BDDs [59] are amongst other engineering techniques employed for efficient heap analysis.