Shape Analysis

Bor-Yuh Evan Chang[1], Cezara Drăgoi[2], Roman Manevich[3], Noam Rinetzky[4] and Xavier Rival[5]

[1] *University of Colorado, USA; evan.chang@colorado.edu*
[2] *INRIA, France and CNRS, PSL University, France; cezarad@di.ens.fr*
[3] *Ben-Gurion University of the Negev, Israel; romanm@cs.bgu.ac.il*
[4] *Tel Aviv University, Israel; maon@cs.tau.ac.il*
[5] *INRIA, France and CNRS, PSL University, France; rival@di.ens.fr*

ABSTRACT

The computation of semantic information about the behavior of pointer-manipulating programs has been a long standing issue, attacked with diverse and numerous techniques and tools for over 50 years. As usual in automatic verification of infinite-state programs, properties of interest are not computable. Thus, static analyses can only be conservative, leading different analyses to make different tradeoffs between the intricacies of the properties they detect, the precision of their inference procedure and analysis, and the scalability of the analysis.

In this context, *shape analyses* focus on inferring highly complex properties of heap-manipulating programs. These programs utilize data structures which are implemented using an unbounded number of dynamically- (heap-) allocated memory cells interconnected via mutable pointer-links. Because shape analyses have to reason about data structures whose size is not bounded by a fixed, known value, they cannot track explicitly the particular properties of every concrete memory cell which the program uses, as done,

Bor-Yuh Evan Chang, Cezara Drăgoi, Roman Manevich, Noam Rinetzky and Xavier Rival (2020), "Shape Analysis", Foundations and Trends® in Programming Languages: Vol. 6, No. 1–2, pp 1–158. DOI: 10.1561/2500000037.

e.g., by analysis of variable-manipulating non-recursive programs. Instead, shape analyses *summarize* memory regions by letting one piece of abstract information, called *summary predicate*, describe several concrete cells. The need to cope with data structures of unbounded sizes is a challenge shape analyses share with static analyzers of array-manipulating programs. However, while the size of an array may change in different executions, its layout (i.e., its dimensions and the way its contents are spread over the memory) is fixed. In contrast, the layout of a pointer-linked data structure, colloquially referred to as its *shape*, may evolve dynamically during the program execution and a memory cell can be part of different data structures at different points in time. As a result, shape analyses need to let the denotation of summary predicates in terms of the constituents and layouts of the memory regions which they represent evolve during the analysis as well.

In this survey, we consider that shape analyses are characterized and defined by the presence of summary predicates describing a set of concrete memory cells that varies during the course of the analysis. We use this characterization as a means for distinguishing shape analyses as a particular class of pointer analyses. We show that many "standard" pointer analyses do not fit the aforementioned description, while many analyses relying on very different mathematical foundations, e.g., shape graphs, three-valued logic, and separation logic, do.

The ambition of this survey is to provide a comprehensive introduction to the field of shape analysis, and to present the foundation of this topic, in a single document that is accessible to readers who are not familiar with it. To do so, we characterize the essence of shape analysis compared to more classical pointer analyses. We supply the intuition underlying the abstractions commonly used in shape analysis and the algorithms that allow to statically compute intricate

semantic properties. Then, we cover the main families of shape analysis abstraction and algorithms, highlight the similarities between them, and also characterize the main differences between the most common approaches. Last, we review a few other static analysis works (such as array abstractions, dictionary abstractions and interprocedural analyses) that were influenced by the ideas of shape analysis, so as to demonstrate the impact of the field.

1

Introduction

1.1 Verifying Pointer-Manipulating Programs

Pointers and dynamic memory allocation are present in one form or another in many modern programming languages and significantly contribute to their expressiveness. For instance, they enable maintaining mutable data structures such as lists, trees, and graphs. The size of such structures may vary during the execution, as cells can be dynamically allocated in the heap when the program needs them in order to store new data. Moreover, the links between elements may be modified locally without changing the whole structure, e.g., to insert a new element into its proper location inside a sorted list. Similarly, common implementations of functional or object oriented languages also make great use of both pointers and dynamic memory allocation so as to represent the call stack, closures, and objects.

On the other hand, these features make reasoning over programs very difficult since the layout of the memory states heavily depends on the program executions. As a consequence, using such features is a notoriously hard task for programmers, and bugs related to them are both common and challenging to diagnose. Depending on the programming language, pointer manipulation errors may cause abrupt crashes due to

runtime errors (as the dereference of a null pointer), memory leakage, i.e., make memory blocks unreachable, and thus impossible to ever deallocate, cause pointers to become dangling, i.e., point to (manually) deallocated memory regions, which may lead to further pointer related errors, e.g., memory corruptions (a write through a dangling pointer, that happens to refer to a memory area that has been freed and then allocated again to store other, unrelated, data).

On top of that, the preservation of structural invariants of pointer-linked data structures is often non-trivial, as a pointer manipulation error might create a cycle in a structure that is supposed to be acyclic and/or leak a large part of it. As an example, Figure 1.1 displays several common examples of dynamic data structures, with very different properties:

- *singly-linked lists* consist of acyclic chains of elements ending with a special element, and where the link from one element to the next usually boils down to a pointer field embedded in every element;
- *doubly-linked lists* augment the singly-linked list structure with backward pointers from each element to its predecessor;
- *circular lists* have the same local structure as the singly-linked lists, but form a loop, so that it is always possible to access the successor of any element;
- *binary trees* are also chained structures, but are such that each non-leaf node has a left and a right successor (a slightly different definition of binary trees accepts structures where some nodes may have no left child or no right child);
- *binary trees with parent pointers* augment binary trees with backward links from every node to its predecessor, Similarly to the way doubly-linked lists augments singly-linked lists with back-pointers;
- *connected graphs* consist of sets of elements, such that each element has a number of successors who are also elements of the structure; in particular, they may contain cycles, elements with no successors, etc.

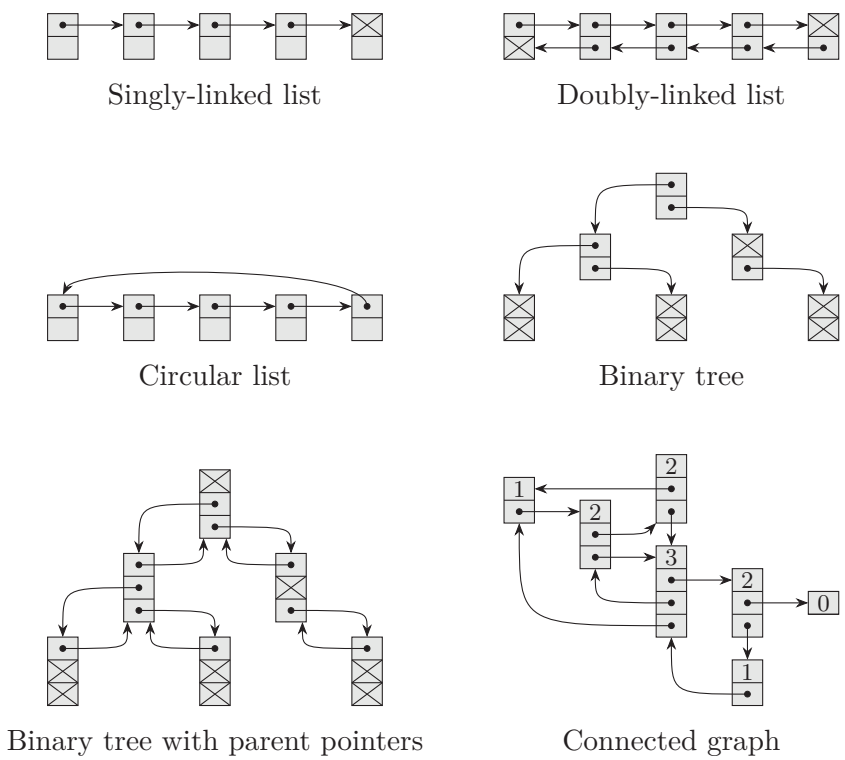


Figure 1.1: A few unbounded and dynamic data structures.

This defines just a small sample of the structures one can imagine, and it is possible to combine these patterns or invent others, e.g., a list of trees or a tree the nodes of which are also connected by a list. Each structure comes with a set of properties (existence of chains of links to next elements, reachability, absence or existence of cycles, existence of a linear order or not...). Furthermore, the correct utilization of each structure relies on the preservation of its *shape invariant*—a combination of global properties pertaining to the layout of its elements—which is generally hard to establish.

Due to these difficulties, a large number of works have searched for techniques to reason about pointer-manipulating programs automatically so as to verify the aforementioned properties. In general, *static*

analysis aims at computing automatically semantic properties of programs, namely properties that are satisfied by every program execution, such as the absence of some classes of errors, or the preservation of some invariants. Broadly speaking, there are two (somewhat overlapping) categories of static analysis of heap-manipulating pointer programs: *pointer analyses* and *shape analyses*, as we discuss next.

1.2 Pointer Analysis

Pointer analyses (see Smaragdakis and Balatsouras, 2015 for a recent survey) attempt to determine properties of pointer values and of the structures they refer to. A first useful property is the validity of pointer values, which expresses that they are neither dangling nor null. While it is useful in order to prove that some errors such as a null/dangling pointer dereference or the corruption of an unknown memory location cannot occur, this property is often too weak to fully understand what a program does. A second useful semantic property focuses on the resolution of pointers so as to determine to which address a pointer may refer, or what pairs of pointers may be equal (alias). This property is extremely useful to resolve memory accesses, and help basically any kind of program reasoning technique when considering a program that manipulates pointers. *Points-to analyses* such as Andersen (1994) or Steensgaard (1996) compute a super-set of the addresses each pointer variable may refer to. Essentially, each memory cell with a pointer type is mapped into a set of symbolic addresses it may point to, and this set can be used so as to resolve memory accesses. *Alias analyses* such as Cooper and Kennedy (1989) compute a super-set of the aliasing relation between pointers, which is another way to describe the topology of pointers (see, e.g., Jonkers and Jonkers, 1981).

1.3 Limitations of Pointer Analyses and Need for More Expressive Abstractions

Points-to and alias analyses rely on basic and generally cheap abstractions of program states, and can often be carried out in a fully flow-insensitive manner for better performance, relying on field-, object

creation site, or context-sensitivity to improve precision. On the other hand, the range of properties they may infer is typically quite limited. In general, when the size of data structures or the numbers of allocated memory blocks are unbounded, many important properties fall beyond the scope of these analyses. As an example, the *reachability* of a cell that is allocated dynamically becomes hard to establish since the chains of pointers from program variables to it may be arbitrarily long. This property is important in order to verify the absence of memory leaks in languages where deallocation is manual. Similarly, the *acyclicity* of a data structure expresses the absence of certain patterns in pointer paths, can only be established by reasoning over arbitrarily long paths. This property is important in order to verify structural preservation or termination of loops. The key issue is that these properties are not local, and can only be justified by global arguments. In fact, it is not rare that even the verification of a local property, e.g., pointer validity, requires establishing a global property, e.g., reachability.

There exist techniques to make pointer analyses less local and extend their expressiveness. As an example, Deutsch (1994) infers aliasing relations over access paths that are of unbounded length, and that can be tied together by the means of numeric relations: this analysis can express that some pointer stores the address of an element that lies somewhere in the middle of a list-like structure. However, such techniques remain limited, and cannot express that a list (or an instance of some other dynamic structure) is well-formed.

1.4 Shape Analysis

Shape analyses, in contrast to pointer analyses, aim at computing global structural properties of unbounded sets of memory cells and pointers, such as the shape invariants of the data structures depicted in Figure 1.1. An example of shape property is the well-formedness of a singly linked list or that of a binary tree without sharing. Such properties concern an unbounded number of memory cells, and tightly constrain correlations between an unbounded number of pointers fields. This allows them to convey, for instance, the absence of cycles over arbitrarily long link

chains. Such relations are intrinsically harder to define and reason about than relations over finite sets of pointers or of regions.

Shape analyses have in common a much higher level of expressiveness than the aforementioned pointer analysis and they rely on very different basic logical predicates. In particular, each of them features some kinds of basic predicates that are able to *summarize* memory regions of unbounded size and in a compact manner while retaining some global information about the shape properties of the summarized region. This is absolutely required to express shape properties over unbounded data structures such as lists, trees and graphs: indeed, abstractions that lack the ability to summarize are either limited to keeping precision on finite sets of memory cells, while losing precision on the rest, or require to resort to a possibly unbounded number of disjuncts.

In addition to summarization, shape analyses need to calculate precisely how program statements transform summaries. In practice, they often need to temporarily refine summaries in order to reason precisely over program statements that impact them. This process, often called *materialization* or *focus*, allows the analysis to apply case analysis regarding the layout of the heap part represented by a summary predicate. Materialization allows to perform strong updates of heap cells located deep in the heap as it enables the analysis to dynamically refine its view of the parts of the heap that pointer variables refer to when analyzing, e.g., the traversal of unbounded data structures.

The use of materialization implies that the analysis also needs to be able to introduce summaries by a generalization process, from more precise predicates. As a consequence, the analysis needs to go back and forth between its base view of data structures and a more refined one, that makes reasoning over local read and destructive update (field mutation) operations possible.

Materializing and Non-Materializing Shape Analyses. In the following, we distinguish between two families of shape analyses: the first category is unable to do materialization at any time and thus can perform strong updates only when certain favorable conditions hold, and the second category that is able to perform dynamic materialization

(at any time during the analysis) and thus is able to perform strong updates in more cases.

Non-Materializing Shape Analyses. As an example for the latter kind of analyses, Ghiya and Hendren (1996) uses global predicates that state that some structures are "tree like", that is, acyclic and without sharing, or simply "DAG like", that is, acyclic, but possibly with some amount of internal sharing. Unlike the pointer analyses mentioned above, this analysis actually captures properties related to the shape of heap data structures that are manipulated by programs.

Materializing Shape Analyses. Two notable examples for the kind of shape analyses which use materialization are the three-valued logic framework for shape analysis of Sagiv *et al.* (1999, 2002), and analyses based on separation logic which was introduced by Reynolds (2002) and Ishtiaq and O'Hearn (2001).

Three-valued logic relies on basic user-defined shape predicates (such as local points-to predicates, global reachability predicates expresses by transitive closure over the points-to predicates, and acyclicity predicates) and summary nodes that stand for unbounded numbers of concrete memory cells or addresses in order to describe large families of shape properties of heap data structures. TVLA (Lev-Ami and Sagiv, 2000) is a parametric system which can very precisely capture structures such as lists or graphs, and it was applied to a wide range of shape analysis problems.

Separation logic was proposed as a language to tie logical properties to heap regions. As an example, it can naturally convey, thanks to the so-called *separating conjunction*, that a memory region can be divided into a finite set of pairwise disjoint regions that store specific data structures, and that can be reasoned about in a separate manner. This is the basis of *local reasoning*, which simplifies the analysis of atomic program statements by letting it focus on the memory cells that they may read or update. Coupled with inductive predicates, separation logic can describe many interesting data structures of unbounded size, and assert that a region stores, e.g., a well-formed singly linked list

or a well-formed binary tree with no sharing. It has served as a basis for several static analyses including those described in Distefano *et al.* (2006), Berdine *et al.* (2007), Chang *et al.* (2007), Dudka *et al.* (2011), or Holík *et al.* (2013).

Applications of Shape Analysis. Besides memory safety and the verification of correctness properties for sequential programs as outlined above, we can cite many applications for shape analysis techniques. An important example is the case of parallel programs, where several threads may concurrently access and modify shared data-structures. Among the many works that have attacked this problem, we can cite Berdine *et al.* (2008), Manevich *et al.* (2008), and Vafeiadis (2010). In general, the works rely on shape abstractions that are rather similar to those used in the sequential case and compute information about the thread interaction in terms of heap abstraction.

More surprisingly, shape analysis abstraction also have applications far outside the world of program analysis. For instance, Srivastava *et al.* (2011) reduces the search of solutions for planning problems to shape analysis problems.

1.5 Summary and Survey Outline

The goal of this survey is to survey the main shape analysis techniques and to convey a general understanding of the main characteristics of these static analyses. As it is not possible to provide an exhaustive recollection of all the works carried out on this topic, we adopt a more modest approach and focus on the main principles related to abstraction (namely, the relation between concrete stores and abstract predicates), to the computation of post-conditions for atomic operations and to the generalization of abstract predicates to enforce termination of analyses. In this process, we intend to highlight similarities and differences among the main approaches. Moreover, the principles underlying shape analysis also inspired other static analyses aimed at programs manipulating other classes of data structures such as arrays or dictionaries. Thus, we also show the link between shape analyses and other families of abstractions and static analysis.

This survey has the following structure. Section 2 presents an intuitive overview of the main principles of shape analysis, without adopting one specific formalism. In fact, it mostly only relies on a graphical presentation. Section 3 formalizes a concrete model of program states and executions to be used in the rest of the survey. As often, the choice of the concrete model of programs deeply influences the ensuing definition of abstractions and static analysis algorithms. Section 4 integrates some of the main approaches to shape analysis into this framework. This is the core part of this survey, since it defines and formalizes the main abstractions and analysis algorithms. Section 5 presents important extensions of shape analysis, so as to describe not only the shape of memory, but also the content and the low level layout of data structures and to analyze programs with functions and procedures. Section 6 describes a few abstractions and static analyses that rely on principles that are similar to the main foundational techniques of shape analysis abstractions and algorithms. Finally, Section 7 draws the main conclusions of our study.

2

Shape Analysis in a Nutshell

In this section, we discuss the main principles of shape analysis at a high level, and mostly based on graphical and intuitive descriptions of abstract states and analysis algorithms. These descriptions are not tied to a specific set of logical predicates. In fact, we discuss the logical predicates and their representation last and defer to Section 4 for an in-depth discussion about them. Moreover, we focus on basic structures such as acyclic singly-linked lists, for the sake of simplicity and readability.

2.1 Running Example

An acyclic singly-linked list is a chained structure, with a single pointer field linking one element to the next. The last element of a singly-linked list is marked with a special flag, or a special value stored in its pointer field (typically null). Furthermore, each element contains a fixed set of additional fields (these fields play no important role in this section, thus we ignore them). Note that as we assume all lists are acyclic, no element should have a link to one of its predecessors.

Figure 2.1 depicts three structures. It adopts the following graphical conventions: a memory block is shown as a rectangle with a gray background, and may be divided into different memory cells (e.g., in

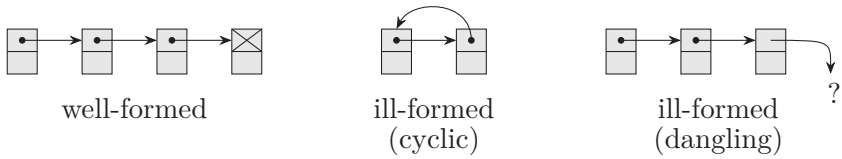


Figure 2.1: Well-formed singly-linked list (left) and ill-formed structures (middle and right).

Figure 2.1, each block is divided into two cells that respectively store a pointer and a value of some unspecified base type); a pointer to a regular address (that is allocated or not) is drawn as an arrow; last, a null-valued pointer field is marked as a crossed out cell. The leftmost structure is a well-formed singly-linked list. The middle structure contains a cycle, and is thus ill-formed. The next link from the last element of the rightmost structure is a *dangling pointer* (i.e., its value is not null, but it does not point to any allocated cell). In the following, we are interested in programs that manipulate singly-linked lists (like the leftmost structure), and we look for analyses that are able to prove that such programs neither break the structural invariants of lists (i.e., produce ill-formed lists) nor cause any memory runtime error.

As an example for a well-behaved list-manipulating program, Figure 2.2 shows a C procedure which inserts a new data element into a singly linked-list. The procedure `insert` takes as arguments a value and a pointer to a list (that is assumed to be well-formed); it then traverses the list, searching for a node after which the new element can be inserted; allocates a new list node; and, finally, it performs the insertion via a pointer surgery which restores the singly-linked list structure.[1]

Several semantic properties are of great interest here. First, *memory safety* properties include the absence of crashes due to a null-valued pointer dereference (null dereference, for short) and the absence of memory leaks. Since, the program of Figure 2.2 dereferences and mutates pointers, it may violate memory safety in several ways. For instance,

[1]Note that the insertion is not be performed if no insertion position is found, and that the insertion cannot take place before the first element. These two assumptions make the program simpler, for the sake of the example.

```
1  typedef struct list {
2    struct list *n;
3    int d;
4  } list;
5
6  void insert(list *l, int d){
7    list *t = l;
8    while( t != NULL ){
9      if( test( t→d, d ) ){
10       list *fresh = (list *) malloc(sizeof(list));
11       fresh→n = t→n;
12       fresh→d = d;
13       t→n = fresh;
14       break;
15     }
16     else
17       t = t→n;
18   }
19 }
```

Figure 2.2: A list insertion method.

if `l` is dangling, the program may crash. Moreover, when the program destructively updates the link-field of the list element at the insertion point (Line 13), the tail of the list may become unreachable, which would mean some memory cells are leaked. Note that we consider a rather simple instance of a C program here, and that we make no use of pointer arithmetic. In the case where pointer arithmetic would be used, additional errors may occur, for instance due to pointer cast issues. In this section, we focus on a rather simple and well-behaved fragment of the C language as it is sufficient to illustrate all the shape analysis concepts that we intend to introduce. Second, *structural preservation* states that the method `insert` does not return a structure that is not a well-formed acyclic singly-linked list. At first, these two properties do not seem strongly related, and memory safety seems simpler, as it does not explicitly refer to a global invariant. In fact, this intuition is incorrect. Indeed, to establish memory safety, one needs to observe that the parameter `l` of `insert` initially points to a well-formed structure, in

particular, that no pointer field is dangling. In other words, establishing memory safety requires to reason over the assumption of a structural invariant. In turn, this means that establishing memory safety for two successive calls to `insert` requires to prove that the first call to it returns a well-formed structure. Thus, we see that proving memory safety requires to also prove structural preservation. In a wider set-up, structural preservation and memory safety would also be required to prove other properties, such as liveness (i.e., to establish that a call to `insert` terminates, one needs to assume that the structure it is applied to is acyclic) or security (i.e., memory safety is often required to establish the absence of memory safety violations or information leakage).

As we have observed here, structural preservation is the key property. Therefore, in the rest of the subsection we sketch an analysis that can verify preservation of structural invariants as well as memory safety.

2.2 Shape Abstraction for Lists

Before we dive into the specific aspects of the abstraction and analysis algorithms that are required to reason over programs such as the code in Figure 2.2, we briefly list a few important requirements. First, the argument of the method is required to be a list, but the size of this list is unknown. In fact, the list could be of any length. This means that the analysis should be powerful enough to reason over the list whatever its size is, and to represent properties such as "`l` points to a well-formed list" without making any assumption related to the size of that structure. Additionally, it should accurately reflect the effect of read and update operations, otherwise the shape properties of the list will be lost. As a consequence, the analysis should include a materialization mechanism that lets it switch between a global view of the list and a local one, so that it can both represent the list as a whole, independently from its size, and accurately reflect basic operations on small fragments of it. These observations significantly impact the abstraction and analysis algorithms we now present.

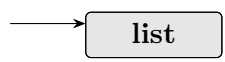
2.2.1 Abstraction and Summarization of Unbounded Regions

Based on the aforementioned observations, we set up an *abstraction* for a static analysis able to reason over list data structures and programs manipulating them.

To describe this abstraction, we still use the graphical representation that we have introduced for concrete states. Indeed, we have noticed the analysis sometimes needs to reason over operations that affect an individual memory cell, thus we let it manipulate concrete predicates, with the usual representation. Besides such very precisely described regions, our abstraction should also feature predicates to represent unbounded regions. Arguably, the most intuitive way to account for unbounded regions is to let the analysis feature predicates that describe any number of memory cells. We call such predicates *summarizing predicates* (or, for short, *summary predicates*). In this section, we focus on the shape analysis of programs that operate on singly-linked lists, thus we consider summary predicates which describe fragments of singly-linked lists, but we could imagine summary predicates that represent other kinds of data structures. Note that we do not fix the specific representation of summary predicates in any way. Instances of shape analyses differ greatly in the way they represent summary predicates. Their actual representations are discussed thoroughly in the next sections.

In the following, we are using the graphical representation below to depict a memory region that stores a well-formed singly-linked list, defined by a chain of elements without cycles or invalid pointers and such that the last element is a null pointer:

Additionally, such a region together with a pointer to the first element of the list will be represented as follows:



This graphical representation is called a *list summary*. Essentially, this predicate describes well-formed lists of any length, as shown in

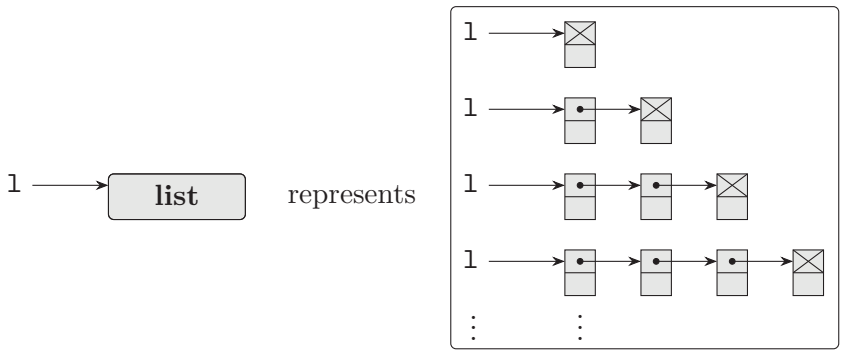


Figure 2.3: A list summary predicate and its meaning.

Figure 2.3: the left hand side of the picture describes a single abstract state with a variable `l` and the right hand side depicts a few example concrete states.

The list summary predicate that we have introduced above can summarize a complete list, but for some programs this is not enough. Indeed, it is common to compute or maintain pointers inside a given singly-linked list. This is the case in Figure 2.2: `t` acts as a cursor in the list in method `insert`. The list summary predicate cannot express the relation between `l` and `t`. Therefore, our analysis will sometimes also need to talk about *list segments*, that is series of list elements that do not necessarily end with a last list element (marked by a null pointer value in the `n` field). To this extent, we define the *list segment summary* predicate that is represented by:

The list segment predicate represents a memory region that stores a well-formed singly-linked list, with the incoming arrow depicting the address of its first element and the out-going arrow the (non-NULL) value stored in the pointer-field going out of the last node in the list. For example, Figure 2.4 depicts an abstract state, that includes two summary predicates. Intuitively, this state describes memory states where `l` points to a list, and such that `t` points to some element of that

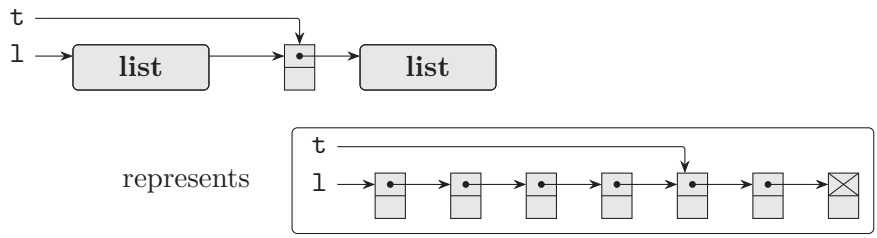


Figure 2.4: Example abstract state and a concrete state it represents.

list (that is neither the first nor last). The concrete state in the right hand side is one of the memory states that this abstract state describes.

2.2.2 From Abstract States to Concrete States and Back

Figure 2.3 describes informally the meaning of an abstract summary predicate: the summary in the left stands for all the fully expanded lists in the right hand side of the figure. Conversely, this remark also means that when we know a memory state can be described by any of the figures in the right side, it can also be described by the one in the left. We can thus go from summaries to concrete memories and back, while still preserving the same meaning. This remark is crucial for the definition of analysis algorithms in the following paragraphs.

From this general observation, we can draw a more incremental technique to refine a summary predicate into a more precise description, or to generalize an abstract state into a summary predicate, as depicted in Figure 2.5. Intuitively, this picture states that a list inductive predicate consists either of a single list element, or of a first element, with a link

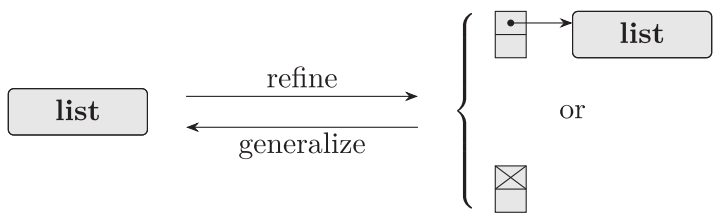


Figure 2.5: Refinement and generalization of inductive predicates.

that points to another list summary. In fact, this correspondence fully defines list predicates by induction. Moreover, a similar consideration would apply to segment summary predicates.

Materialization and Generalization. While Figure 2.3 describes the meaning of a summary predicate in a very eager way, as an infinite set of concrete memories, this new representation based on induction provides a description of the meaning of the summary predicates that is both local and algorithmic: it is possible to transform the abstract state shown in the left into either of the abstract states shown in the right in a few basic computation steps, and the converse transformation is equally easy to implement. This remark is fundamental for the definition of the static analysis algorithms in the next paragraphs. Essentially, the *refinement* transformation allows to make abstract states easier to reason about, and the *generalization* transformation is essential to let the analysis compute invariants for loops or other iterative constructions.

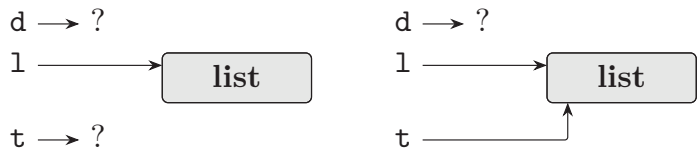
In turn, the exhaustive description of Figure 2.3 can be replicated by iteratively applying the local refinement principle that is shown above, and can be used in order to prove the correctness of the analysis.

2.3 Shape Analysis for Lists

We now look at the computation of shape invariants using the informal abstraction described in the previous subsection.

2.3.1 Abstract Post-Conditions

The analysis that we study here proceeds by forward abstract interpretation (Cousot and Cousot, 1977): it inputs an abstract pre-condition and computes a conservative abstract post-condition from it; moreover, when a program contains a loop, it computes a loop invariant by iterative analysis of the body, and using a widening operator in order to enforce the convergence of the iterates. In the following, we illustrate the main operations of this analysis based on the case of the program of Figure 2.2. Intuitively, the analysis of this program starts with a pre-condition that states that the parameter `l` points to a well-formed and



(a) Abstract pre-condition (b) Abstract post-condition

Figure 2.6: Analysis of the assignment `*t = l`.

non-empty list (described by a summary predicate), and that nothing is known about the other parameter `d` or the local variable `t`. This state is shown in Figure 2.6(a).

We start the description of this analysis by explaining the computation of abstract post-conditions for basic program statements, such as a pointer assignment.

Copy Assignments

We first discuss the analysis of the first statement of the `insert` method, `t = l`. At the concrete level, the effect of that operation boils down to the update of the memory cell corresponding to variable `t`. In our graphical abstract states, this boils down to a modification on the outgoing edge of `t`: after the assignment, instead of pointing to an indeterminate value, it points to the same list as `l`. The effect of that operation is shown in Figure 2.6(b).

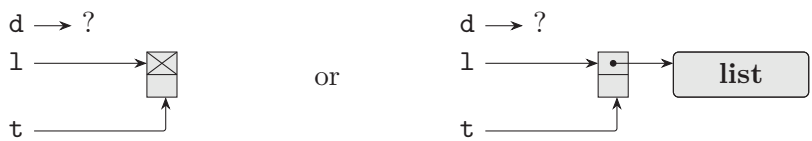
Pointer Dereferences

Not all assignment statements are so simple to analyze. As an example, let us consider the analysis of the first iteration of the loop, starting from the abstract pre-condition is shown in Figure 2.6(b). The first operation is the condition test, which lets the execution proceed into the loop when `t` is not null. In the abstract, we can deduce that the condition is true because `t`, like `l`, points to the first element of a well-formed singly-linked list. Then, there are two execution branches inside the loop, depending on the random choice at the condition statement inside the loop. We consider the second branch, which performs the assignment

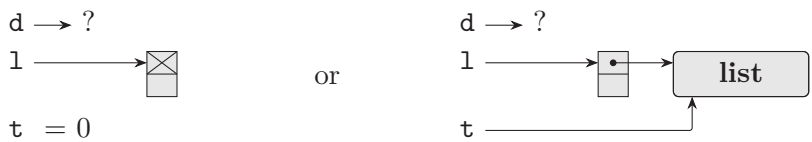
`t = t→n`. This statement reads the `n` field of `t` and then updates `t` to this value. However, this field is not visible in the abstract state shown in Figure 2.6(b): indeed, it is part of the summary predicate that describes the memory region occupied by the list.

Materialization in Action: Adding Precision via Case Analysis

While the analysis cannot perform on that abstract state an update similar to that shown in Figure 2.6, it can apply the refinement principle presented in Figure 2.5 so as to substitute the summary predicate with a more precise description of the list, and perform a simple update on it. This process is shown in Figure 2.7. First, Figure 2.7(a) presents the result of the refinement step, before the computation of a post-condition for the assignment statement. We observe that the abstract pre-condition of Figure 2.6(b) is replaced with a *disjunction* of two abstract states, which are both more precise than the initial abstract state (the one in the left accounts for cases where the list has length one, and the one in the right accounts for cases where the list has length strictly greater than one), and that account for all the memory states represented by Figure 2.6(b) when put together. We remark that these two disjuncts allow to evaluate precisely both sides of the assignment `t = t→n`, since the field read by `t→n` is not impaired by



(a) Refinement into a disjunctive abstract state



(b) Abstract post-condition

Figure 2.7: Analysis of the assignment `t = t→n`.

the summary predicate anymore. Second, Figure 2.7(b) describes the abstract post-condition of the statement `t = t→n`: it also consists of two disjuncts, that are computed from those in Figure 2.7(a), and by a straightforward update, which proceeds as in the case of the trivial assignment shown in Figure 2.6.

The analysis of the first branch of the condition actually follows a similar sequence of steps. Indeed, the assignment at line 11 requires to access to the memory cell `t→n`, even though `t` is only described as a pointer to a singly-linked list which is represented by a summary predicate. This implies that the summary predicate should also be materialized before the assignments at lines 11 and 13 can be analyzed precisely. As above, the materialization process returns a disjunction of two cases, and in each case, the memory cell `t` points to is described precisely. Therefore, the assignments that read `t→n` (at line 11) and modify it (at line 13) can be analyzed precisely in each case. Note that the assignment at line 13 also modifies the structure pointed to by `t`, yet the analysis successfully produces a post-condition. The result is shown in Figure 2.8 (for the sake of clarity, we omit `d`).

More generally, the above approach allows to compute abstract post-conditions for basic statements, such as assignment, condition test, and memory allocation: first, the analysis needs to refine the abstract pre-condition so that it can evaluate completely and accurately the l-values contained in the statement to analyze; in general this phase, produces a disjunction of abstract states; second, the analysis needs to perform the operation on each case of the abstract disjunction.

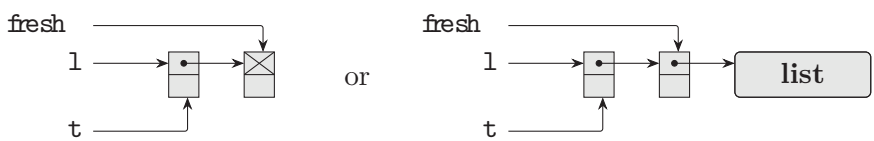


Figure 2.8: Analysis of the first branch inside the loop.

2.3.2 The Need for Generalization: Ensuring Termination

We now consider the analysis of loops and other iterative control structures. As an example, the `insert` method shown in Figure 2.2 consists of a loop that makes an unbounded number of steps before actually performing the insertion. Repeatedly applying the above refinement technique for the computation of abstract post-conditions would not let the analysis terminate. Indeed, we show a few iterates on Figure 2.9. To simplify the presentation, we only consider the executions that go through the second branch of the condition statement, which means they only iterate the statement `t = t→n`. We also omit variable `d` from the representation, and we represent only the cases where the list tail (pointed to by `t`) contains more than a single element. We first represent the abstract state before the loop (iteration 0). Below, we represent abstract states observed after one, two and three iterations, respectively. We observe that the list elements from the beginning of the list to the position pointed to by `t` are kept "concrete", which means that the abstract states in Figure 2.9 describe this portion in the structure very precisely, and account for each element individually. Subsequent iterates would show a similar structure with ever-growing fully-expanded initial

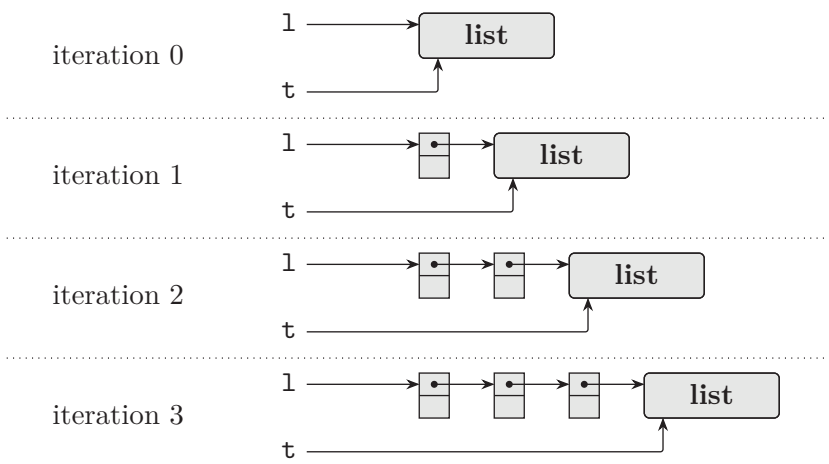


Figure 2.9: A few abstract iterates.

part. As a consequence, this naive analysis approach will keep exploring reachable abstract states forever.

Furthermore, the information that it accumulates is not really useful for the sake of verifying the property of interest, namely memory safety and structural invariants preservation. Indeed, we only need to know that the beginning of the structure pointed to by `l` is a well-formed singly-linked list region that ends with a pointer to the same address as `t` (which is the tail of the structure).

Therefore, the analysis should perform some kind of generalization over the abstract states observed across the first iterates over the loop. Generalization over loop iterates is commonly associated with the computation of loop invariants (e.g., using abstract union or widening operators introduced by Cousot and Cousot, 1977). The generalization transformation shown in Figure 2.5 allows to achieve just this: indeed, it forms summary predicates from collections of concrete predicates, provided they describe an instance of the summary property. In the case of Figure 2.9, this generalization scheme should simply introduce a segment summary predicate. The result is shown in Figure 2.10. We can see that this abstract state captures all states where (1) `l` points to a well formed list, and (2) `t` points to an element of that list.

Although we do not show all the cases here, this generalization step allows to account for all the branches in the loop, and produces a general loop invariant describing all states that can be observed at its head. Following this generalization, the analysis of the loop can produce both sound intermediate invariants inside the loop, and a sound loop abstract post-condition.

The way of computing this generalization transformation depends on the analysis, and we will discuss several approaches in the subsequent sections.

Figure 2.10: Generalization over abstract iterates.

2.4 Instances of Shape Analyses

So far, we have presented a shape analysis based on generic graphical predicates describing sets of memory states. We now briefly discuss the definitions of a few salient families of memory abstract predicates so as to highlight the main characteristics of each abstraction, and to tie them with the graphical representation that we have used so far. In this section, we keep the discussion rather informal (more thorough definitions are given in Section 4). We restrict our discussion to a few important families of abstractions that describe shape properties.

Abstractions of Shape Graphs

A first example is the abstraction of Ghiya and Hendren (1996), which uses basic predicates to describe the points-to graphs in memory states. This abstraction relies on three families of basic predicates:

- *Shape attributes* describe regions using high-level graph properties. For instance the predicate `Tree(p)` states that `p` points to a structure where each location is reachable by exactly one access path from `p`. Thus, it captures linear structures like lists and trees. Other similar predicates capture DAG structures or cyclic structures.

- *Direction predicates* express that there may exist (resp., must not exist) an access path from a pointer variable `p` to another pointer variable `q`.

- *Interference predicates* express that a pair of pointer variables `p` and `q` may (resp., must not) have access paths to a common cell.

These predicates encode global shape properties using basic logical predicates over pointer access paths in a storeless model, which express whether a property *does not hold* or it *may hold*. Each basic predicate describes graph properties of concrete memories. The encoding used by Ghiya and Hendren (1996) is based on truth tables and truth matrixes (i.e., that are indexed over pointer variables and store tables of 0 and

1 values). Other works, e.g., Jones and Muchnick (1979), Chase *et al.* (1990), utilize labeled graphs.

We discuss this family of abstractions in Subsection 4.2.

Three-Valued Logic Predicates

A second example is the abstraction based on *three-valued logic* predicates (Sagiv *et al.*, 1999, 2002). This abstraction also exploits the fact that concrete memories can be viewed as graphs, to describe their properties. Abstract states consist of predicates over sets of nodes, where each node stands for either a single memory location or a set of concrete memory locations, and where the predicates describe pointer properties over these locations. A node that represents a set of concrete memory locations is called a *summary node*.

Three-valued logic analysis is parameterized by a set of predicates. The predicates evaluate to truth values, which can be one of "true", "false", and "maybe". Typical predicates are as follows:

- basic *points-to* predicates (does a field `n` of a memory location `p` point to another location `q`?);
- the *summary status* of nodes (is a node summary or does it describe a single concrete cell?);
- global properties the definition of which relies on basic predicates and transitive closure (to describe chains of pointers of unbounded length), including *reachability* and *acyclicity*.

Summary nodes together with global properties allow to express summary predicates such as the list summaries of Figure 2.3. Indeed, a list boils down to a set of memory blocks that are chained together, and without a cycle, therefore, if `l` points to a list, it is possible to describe the elements of this list with a summary node, together with predicates that assert that they are all reachable from `l`, and that they contain no cycle. The refinement transformation of Figure 2.5 replaces a summary node with a finer representation of a memory region, whereas the generalization operation tends to introduce summary nodes.

The representation of abstract states is based on truth tables, in Kleene's three-valued logic (that is, with truth values that may be either "true", or "false", or "maybe").

We describe this family of shape abstractions in detail in Subsection 4.3.

Separation Logic and Inductive Predicates

A third family of shape analysis abstraction is based on fragments of *separation logic* (O'Hearn *et al.*, 2001; Reynolds, 2002). Separation logic defines a set of logical predicates and connectors specifically tailored to express properties of memory states. The most significant connector is the *separating conjunction*, which is denoted as $*$ and ties together two properties attached to disjoint sub-memories: a memory state satisfies the separating conjunction of two formulas F_0 and F_1 if and only if it can be partitioned into two memory regions that respectively satisfy F_0 and F_1. Common atomic separation logic predicates include:

- basic *points-to* predicates, which describe a single memory cell, with its symbolic address and contents;
- *summary* predicates, which represent unbounded memory regions, generally specified via inductive formulas in separation logic.

The refinement operation shown in Figure 2.5 replaces an inductive summary predicate with a separating conjunction of other (points-to or summary) atomic predicates that describe sub-regions, which means that it partitions abstract memory states into abstract states describing smaller memory regions. As in Figure 2.5, generalization performs the converse action, and introduces inductive summary predicates that replace separating conjunctions of basic memory predicates.

There exist several machine representations for abstract states based on fragments of separation logic. Several works such as Distefano *et al.* (2006), Berdine *et al.* (2007) use logical formulas. Others such as Chang *et al.* (2007), Dudka *et al.* (2011) use graph structures encoding the formulas, where each edge denotes an atomic predicate and a set of edges stands for their separating conjunction. Finally, Holík *et al.* (2013)

relies on specific forms of automata the semantics of which captures disjoint heap regions. While it uses a formalism that is far from that of separation logic, it inherits some of its principles and, in particular, that of local reasoning.

We describe in detail abstractions based on separation logic in Subsection 4.4 and abstractions based on automata in Subsection 4.5.

2.5 Summary: The Essence of Shape Analysis

We can now give an overall view of a shape analysis that is able to compute invariants similar to the abstract state depicted in Figure 2.4. It performs a forward abstract interpretation (Cousot and Cousot, 1977), which means that it incrementally computes an over-approximation for the behaviors of programs, ensuring global soundness of the results, by making sure that each concrete execution step is over-approximated by the analysis computation on abstract states. More precisely, it proceeds as follows:

1. It inputs an abstract pre-condition that accounts for all the possible input states for a given procedure (in the case of a complete program, this initial abstract state should simply account for the empty memory state, where no variable or memory region has been allocated yet).

2. It computes sound abstract post-conditions using basic operations that account for atomic program execution steps; while doing so, it *refines* abstract states when needed, so as to account precisely for the memory locations that are read from or written to by basic statements (assignments, tests, etc.).

3. It *generalizes* abstract states observed on execution paths that may be iterated an unbounded number of times (such as loops, goto edges or recursive function calls), so as to ensure the termination of the analysis, and to make sure that general descriptions of concrete states will be computed.

The version of record is available at: http://dx.doi.org/10.1561/2500000037

3

Generic Shape Analysis

This section describes the general structure of a shape analysis tool based on a forward abstract interpretation of programs, and using a generic memory abstraction. We do not fully define the abstractions at this stage, and defer their study to Section 4, as each of them will fit as an instance of the generic abstraction considered in this section. We first define the semantics of a simple language in Subsection 3.1. We then give the general form of a shape abstraction in Subsection 3.2. Last, we set up a generic abstract interpreter in Subsection 3.3.

3.1 Programs and Semantics

In this survey, we focus on a toy language inspired by a subset of C, even though the principles that we present would apply to other languages as well, such as C++, Java, and Scala. For the sake of simplicity, we consider a drastic simplification of the semantics of these languages. For instance, we allow neither pointer arithmetic nor manual memory reclamation, and we focus on the core pointer operations. On the other hand, pointers may be null or invalid as in C, as newly created variables or freshly allocated cells may initially contain any value. We also do not introduce functions. We make these choices to keep the language

e	::=		expressions
	|	c	constants
	|	x → f	access to a field
	|	e ⊙ e	binary operation ⊙
s	::=		statements
	|	x = new( )	memory allocation (malloc)
	|	x → f = e	field update
	|	s; s	sequence
	|	**if**(e){s}	condition
	|	**while**(e){s}	loop

Figure 3.1: Grammar of a subset of C.

syntax and semantics simple, and focus on the core aspects of shape analysis. In this subsection, we formalize the language used throughout the rest of the survey.

Syntax. The syntax of the language we use is shown in Figure 3.1. Statements include the allocation of a new object, field updates, and classic control structures such as conditional statements and loop statements. Expressions are constants, accesses to object fields and binary operations (such as arithmetic or comparison operators). We let variables be *constant pointers to structures*. Thus, a base type variable would be described by a pointer to a structure with a single field $\underline{\emptyset}$. We let $\mathbb{F}$ denote the set of object fields (so that $\underline{\emptyset} \in \mathbb{F}$). We write $\mathbb{V}$ for the set of values, and $\mathbb{A}$ for the set of addresses (note that we keep addresses symbolic since we do not deal with pointer arithmetics). We assume $\mathbb{A} \subseteq \mathbb{V}$. Last, we let $\mathbb{X}$ stand for the set of program variables. We do not explicitly formalize scoping (it would be easy but would make the formalization heavier), thus we implicitly assume that all the variables are globally accessible. In the following, programs are always assumed to be well-typed even though we do not make the type system explicit here.

Memory States. Before we can define the semantics of programs, we fix the notations and definitions for *memory states*. A memory state

formalizes the status of the memory at a given time during program execution. It defines the object pointed to by each variable, and the value of each object field. Thus, it boils down to the combination of a function from variables to symbolic addresses, and a partial function from pairs made of a symbolic address and a field name into values. In order to keep notations light, and given a memory state m, we write $m(\mathtt{x})$ for the contents of variable $\mathtt{x}$, and $m(a, \mathtt{f})$ for the contents of the field $\mathtt{f}$ of the structure stored at symbolic address a. For instance, the reading of $\mathtt{x}{\to}\mathtt{f}$ boils writes down $m(m(\mathtt{x}), \mathtt{f})$. Similarly, we write $m[\mathtt{x} \mapsto v]$ (resp., $m[(a, \mathtt{f}) \mapsto v]$) for the update of the value of the variable $\mathtt{x}$ (resp., of the field $\mathtt{f}$ of the object stored at address a) in the memory state m. We write $\mathbb{M}$ for the set of memory states.

Semantics. We now fix the definition of the semantics of expressions and statements, which will be used as a reference point for the construction of a parametric shape analysis. For the sake of simplicity, we elect to use a form of denotational semantics, which boils down to a compositional input–output relation.

The evaluation of an expression in a given memory state simply returns a value (namely, a symbolic address or a base value such as a numeric value). Therefore, we let the semantics $[\![\mathtt{e}]\!]$ of an expression $\mathtt{e}$ be a partial function from memory states to values. Indeed, it is not defined when it encounters an error (e.g., due to a null or invalid address dereference). It proceeds by induction over the syntax of expressions and is shown in Figure 3.2(a). In the case of binary operations, we assume that, for each operator $\odot$, there exists a function $f_\odot$ that defines its semantics.

The semantics of a statement $\mathtt{s}$ boils down to a function that inputs a pre-condition and returns a post-condition. Pre- and post-conditions consist of sets of memory states. The semantics of statements is shown in Figure 3.2(b). Its definition proceeds by induction over the syntax of statements and is standard. Freshly allocated cells may contain any possible value. Similarly, we make no assumption about the initial value of variables. Note that the case of loops relies on a fixpoint iteration to describe all states that can be observed after any number of iterations. This semantics can easily be extended to a semantics that collects all

$$
\begin{array}{rcll}
\multicolumn{4}{l}{[\![\mathtt{e}]\!] : \mathbb{M} \longrightarrow \mathbb{V}} \\
[\![\mathtt{c}]\!](m) & = & \mathtt{c} & \text{(constants)} \\
[\![\mathtt{x} \to \mathtt{f}]\!](m) & = & m(m(\mathtt{x}), \mathtt{f}) & \text{(access to a field)} \\
[\![\mathtt{e0} \odot e1]\!](m) & = & f_{\odot}([\![\mathtt{e0}]\!](m), [\![\mathtt{e1}]\!](m)) & \text{(binary operation)}
\end{array}
$$

(a) Semantics of expressions

$$
\begin{array}{rcl}
\multicolumn{3}{l}{[\![\mathtt{s}]\!] : \mathcal{P}(\mathbb{M}) \longrightarrow \mathcal{P}(\mathbb{M})} \\
[\![\mathtt{x} = \mathtt{new}(\ )]\!](M) & = & \{m[\mathtt{x} \mapsto a, (a, \vec{\mathtt{f}}) \mapsto \vec{v}] \mid m \in M \\
 & & \qquad \wedge\ a \text{ fresh in } m \wedge \vec{v} \text{ are any values}\} \\
[\![\mathtt{x} \to \mathtt{f} = \mathtt{e}]\!](M) & = & \{m[(m(\mathtt{x}), \mathtt{f}) \mapsto v] \mid m \in M \wedge [\![\mathtt{e}]\!](m) = v\} \\
[\![\mathtt{s0};\ \mathtt{s1}]\!](M) & = & [\![\mathtt{s1}]\!] \circ [\![\mathtt{s0}]\!](M) \\
[\![\mathbf{if}(\mathtt{e})\{\mathtt{s}\}]\!](M) & = & [\![\mathtt{s}]\!](\{m \in M \mid [\![\mathtt{e}]\!](m) = \mathbf{true}\}) \\
 & & \cup \{m \in M \mid [\![\mathtt{e}]\!](m) = \mathbf{false}\} \\
[\![\mathbf{while}(\mathtt{e})\{\mathtt{s}\}]\!](M) & = & \{m \in M_{\mathbf{fp}} \mid [\![\mathtt{e}]\!](m) = \mathbf{false}\} \\
\multicolumn{3}{l}{\text{where } M_{\mathbf{fp}} = \mathbf{lfp}\,[\lambda M' \cdot (M \cup [\![\mathtt{s}]\!](\{m \in M' \mid [\![\mathtt{e}]\!](m) = \mathbf{true}\}))]}
\end{array}
$$

(b) Semantics of statements

Figure 3.2: Semantics.

the reachable states of a program, though its definition would become slightly more complex; for this reason, we only study this compositional semantics instead.

3.2 Shape Abstraction

We now set up a general shape abstraction. In this section, we do not formalize a specific abstraction. Instead, we simply give the format of a generic one and defer the definition of instances to Section 4.

An abstraction is defined by a set of logical predicates, with a machine implementation, and a logical interpretation. In the following, we formalize these as follows:

Definition 3.1 (Shape Abstraction)**.** A *shape abstraction* is defined via the following components:

- a *set of abstract shapes* $\mathbb{D}$, and
- a *concretization function* γ: $\mathbb{D} \longrightarrow \mathcal{P}(\mathbb{M})$.

An element $\mathbb{d} \in \mathbb{D}$ is called *abstract shape*.

Intuitively, if m is a memory and $\mathbb{d} \in \mathbb{D}$ is an abstract shape, then $m \in \gamma(\mathbb{d})$ means that "m satisfies the abstract shape property expressed by $\mathbb{d}$", which is sometimes noted as $m \vDash \mathbb{d}$.

An abstract shape $\mathbb{d}$ should simultaneously be viewed in two different ways. First, it defines a logical formula that captures the properties of the set of memory states $\gamma(\mathbb{d})$, and that forms the logical interpretation of the abstract shape. This view should be used for reasoning over shape analyses, for assessing whether they are expressive enough, and for proving their soundness. Second, it defines a machine representation that should be efficient to manipulate. Indeed, manipulating logical formulas directly is often not the best solution to derive fast algorithms to compute analysis operations. Instead, $\mathbb{D}$ generally defines a machine representation that is adapted to such computations. Therefore, a shape abstraction is close to the implementation of a program logic that provides predicates to reason over shape properties.

As an example, Section 2 discussed shape analysis in general terms using a graphical shape abstraction depicted in Figure 2.3. In this setup, abstract shapes are graphical objects using basic predicates and summary predicates. Moreover, the concretization relation maps one such drawing into the set of memory states that it represents, following the high level definition of Figure 2.3. In the rest of that section, we have shown how to compute shape abstract predicates that cover all reachable states of a given program.

Later in Section 2, we have also alluded to several instances of abstract shapes, based on shape graphs, on three-valued logic predicates and on separation logic augmented with inductive predicates. Each of these defines an instance of $\mathbb{D}$ and γ. In Section 4, we shall define these more formally and describe their properties more in detail.

3.3 Abstract Interpretation

In this subsection, we sketch the definition of a shape analysis that is based on the semantics presented in Figure 3.2 (Subsection 3.1), and on a shape abstraction described as in Definition 3.1 (Subsection 3.2). This analysis should return an over-approximation of the semantics of a program, that is expressed using the shape abstraction. To do that,

we proceed by abstract interpretation (Cousot and Cousot, 1977) of the semantics of programs, and seek for an abstract semantics that is defined following a structure very similar to the concrete semantics shown in Figure 3.2, and that is computable.

At this stage, we have not fully fixed the shape abstraction, since we left it as a parameter of the analysis. As a consequence, we will not detail each of the operations that are required on shape predicates. Instead, we will simply list the abstract operations that are required to handle each program construction. These operations are fully defined in Section 4.

Since the concrete semantics was modeled as a function that takes as input sets of memory states and returns sets of memory states, the abstract semantics of a statement $\mathtt{s}$, denoted as $[\![\mathtt{s}]\!]^\sharp$, adopts a similar structure, and takes the form of a function that maps a shape abstraction into another shape abstraction. Moreover, it should be *sound* with respect to the concrete semantics, which means that it should not "forget" any concrete behavior described by the concrete semantics as conveyed in the inclusion below.

$$\forall \mathbb{d} \in \mathbb{D},\ [\![\mathtt{s}]\!] \circ \gamma(\mathbb{d}) \subseteq \gamma \circ [\![\mathtt{s}]\!]^\sharp(\mathbb{d})$$

As the structure of $[\![\cdot]\!]^\sharp$ closely follows that of $[\![\cdot]\!]$, the underlying soundness proof also proceeds by induction over the syntax. Therefore, as we consider statement kinds one by one, we will also sketch the soundness proof itself. As an example, the case of sequential composition is trivial, and the analysis of a sequential composition should simply compose the abstract meaning of the constituents statements.

$$[\![\mathtt{s0;\ s1}]\!]^\sharp(\mathbb{d}) = [\![\mathtt{s1}]\!]^\sharp \circ [\![\mathtt{s0}]\!]^\sharp(\mathbb{d})$$

Analysis of Fresh Object Creation Statements

To compute a sound abstract post-condition for memory allocation, the analysis should provide a function $\mathbf{new}_\mathtt{x} \colon \mathbb{D} \to \mathbb{D}$ that satisfies the inclusion below.

$$\begin{array}{l}\forall \mathbb{d} \in \mathbb{D},\ \forall m \in \gamma(\mathbb{d}),\\ \{m[\mathtt{x} \mapsto a, (a, \vec{\mathtt{f}}) \mapsto \vec{v}] \mid a \text{ fresh in } m \wedge \vec{v} \text{ are values}\} \subseteq \gamma(\mathbf{new}_\mathtt{x}(\mathbb{d}))\end{array}$$

Typically, such an operator should synthesize a representation for the newly allocated block, and accurately take into account that the address of the newly allocated memory block is unknown as well as its contents.

Once such an operator is defined, we may simply define $[\![\texttt{x = new( )}]\!]^\sharp(\mathbb{d}) = \mathbf{new}_{\texttt{x}}(\mathbb{d})$.

Analysis of Assignments

In the same way as for memory allocation statements, the analysis of assignment statements is carried out by a shape abstraction specific abstract operation $\mathbf{assign}_{\texttt{x.f}\leftarrow\texttt{e}}\colon \mathbb{D} \to \mathbb{D}$, which should meet the following soundness property.

$$\forall \mathbb{d} \in \mathbb{D},\ \forall m \in \gamma(\mathbb{d}),\ m[(m(\texttt{x}), \texttt{f}) \mapsto [\![\texttt{e}]\!](m)] \in \gamma(\mathbf{assign}_{\texttt{x.f}\leftarrow\texttt{e}}(\mathbb{d}))$$

Intuitively, $\mathbf{assign}_{\texttt{x.f}\leftarrow\texttt{e}}$ should account for the destructive update of the field `x.f` with the value obtained when evaluating `e`. Therefore, its full definition depends on the abstract shape predicates, and we discuss several kinds of such operators in Section 4. Recall that in Section 2 we have informally described such an operation for a simple graphical shape abstraction. In fact, we encountered two cases: first, when the read and updated cells are immediately visible, the analysis of assignment boils down to an abstract pointer switch, as shown in Figure 2.6; second, when either the read or the updated cell is summarized, the analysis needs to refine abstract summaries before proceeding with the regular assignment analysis.

Based on this abstract operator, we can simply define $[\![\texttt{x.f=e}]\!]^\sharp(\mathbb{d}) = \mathbf{assign}_{\texttt{x.f}\leftarrow\texttt{e}}(\mathbb{d})$.

Analysis of Condition Tests

The analysis of a condition test requires two additional abstract operations. First, it requires an operation that accounts for the condition itself, which should compute an over-approximation of the set of memory states such that a Boolean expression evaluates to true. We write $\mathbf{test}_{\texttt{e}}\colon \mathbb{D} \to \mathbb{D}$ for this operation. Furthermore, its soundness condition boils down to the following inclusion.

$$\forall \mathbb{d} \in \mathbb{D},\ \forall m \in \gamma(\mathbb{d}),\ [\![\texttt{e}]\!](m) = \mathbf{true} \implies m \in \gamma(\mathbf{test}_{\texttt{e}}(\mathbb{d}))$$

Second, the semantics of a condition test comprises the union of the behaviors from two branches, thus its analysis should compute an over-approximation of this union. We write **join**: $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$ for this abstract operation, which over-approximates concrete unions, as expressed by the following inclusion.

$$\forall \mathbb{d}_0, \mathbb{d}_1 \in \mathbb{D},\ \gamma(\mathbb{d}_0) \cup \gamma(\mathbb{d}_1) \subseteq \gamma(\mathbf{join}(\mathbb{d}_0, \mathbb{d}_1))$$

Intuitively, **join** returns an abstract shape that is weaker (less precise) than both of its inputs. There exist several techniques to obtain such a weakening. The most immediate one is to simply produce the symbolic disjunction of the two arguments, if $\mathbb{D}$ can represent it. It is the most precise answer, but it might be overly expensive, in terms of computational resources. A second technique proceeds by generalization over the properties expressed by both arguments. In Section 2, we have observed that such a generalization could be performed by generating summary predicates, as shown in Figures 2.5 and 2.9. Based on these two abstract operators, we can simply let the analysis of a condition test statement be defined by $[\![\texttt{if(e)\{s\}}]\!]^\sharp(\mathbb{d}) = \mathbf{join}([\![\texttt{s}]\!]^\sharp(\mathbf{test}_{\texttt{e}}(\mathbb{d})), \mathbf{test}_{\texttt{!e}}(\mathbb{d}))$ (where `!e` denotes the negation of `e`).

Analysis of Loops

We now consider the analysis of a loop `while(e){s}`, given an abstract pre-condition $\mathbb{d}$. The concrete semantics collects all the states observed by iterating the composition of the effects of the condition and of the loop body, using a least-fixpoint definition shown in Figure 3.2. The states observed after exactly one iteration of the loop are over-approximated by $[\![\texttt{s}]\!]^\sharp(\mathbf{test}_{\texttt{e}}(\mathbb{d}))$, since $\mathbf{test}_{\texttt{e}}$ accounts for the effect of the condition and $[\![\texttt{s}]\!]^\sharp$ for that of the loop body. Thus, the abstract semantics should also collect all the iterates of this function. In general, a naive iteration sequence may not converge, thus the analysis should resort to a fixpoint approximation technique, using some kind of widening operation (Cousot and Cousot, 1977). Intuitively, all these fixpoint approximation techniques need to *generalize* over the abstract iterates, in the same sense as in Figure 2.10. Such a generalization typically

introduces summary predicates. Works in shape analysis rely on different techniques in order to achieve such a fixpoint approximation:

- Some works rely on a finite height lattice, which ensures the convergence of naive iteration with an abstract union **join**.
- Some works use an infinite height lattice, but collapse abstract states into a finite height lattice just for loop iteration (at the expense of some precision).
- Finally, some works employ classical widening techniques.

To accommodate for this range of solutions, we adopt a rather general description here, based on an **extrapol** operator, which takes as arguments a function $f\colon \mathbb{D} \to \mathbb{D}$ and an argument $\mathbb{d}$, and that returns a sound over-approximation of all the iterates $f^n(\mathbb{d})$ (this computation typically involves the abstract join or the widening of the iterates of the analysis of the loop body). Using such an operator, the analysis of a loop statement may simply be defined as $[\![\texttt{while(e)\{s\}}]\!]^\sharp(\mathbb{d}) = \mathbf{test}_{!\texttt{e}}(\mathbf{extrapol}([\![\texttt{s}]\!]^\sharp \circ \mathbf{test}_{\texttt{e}}, \mathbb{d}))$.

3.4 Summary

The complete definition of the shape analysis is summarized in Figure 3.3. It consists of a function $[\![\cdot]\!]^\sharp\colon \mathbb{D} \to \mathbb{D}$, which maps an abstract pre-condition to an abstract post-condition, assuming the shape abstract domain is as defined in Definition 3.1. It is sound with respect to the concrete semantics of Figure 3.2 in the following sense:

$$
\begin{array}{rcl}
[\![\texttt{x = new( )}]\!]^\sharp(\mathbb{d}) & = & \mathbf{new}_{\texttt{x}}(\mathbb{d}) \\
[\![\texttt{x.f = e}]\!]^\sharp(\mathbb{d}) & = & \mathbf{assign}_{\texttt{x.f}\leftarrow\texttt{e}}(\mathbb{d}) \\
[\![\texttt{s0; s1}]\!]^\sharp(\mathbb{d}) & = & [\![\texttt{s1}]\!]^\sharp \circ [\![\texttt{s0}]\!]^\sharp(\mathbb{d}) \\
[\![\mathbf{if}\texttt{(e)\{s\}}]\!]^\sharp(\mathbb{d}) & = & \mathbf{join}([\![\texttt{s}]\!]^\sharp(\mathbf{test}_{\texttt{e}}(\mathbb{d})), \mathbf{test}_{!\texttt{e}}(\mathbb{d})) \\
[\![\mathbf{while}\texttt{(e)\{s\}}]\!]^\sharp(\mathbb{d}) & = & \mathbf{test}_{!\texttt{e}}(\mathbf{extrapol}([\![\texttt{s}]\!]^\sharp \circ \mathbf{test}_{\texttt{e}}, \mathbb{d}))
\end{array}
$$

Figure 3.3: Shape analysis.

Theorem 3.1 (Soundness)**.** For any program $\mathtt{s}$, the abstract semantics $[\![\mathtt{s}]\!]^\sharp$ satisfies the soundness property below.

$$\forall \mathbb{d} \in \mathbb{D},\ [\![\mathtt{s}]\!] \circ \gamma(\mathbb{d}) \subseteq \gamma \circ [\![\mathtt{s}]\!]^\sharp(\mathbb{d})$$

This soundness result follows from a straightforward induction over the syntax of programs, and from the soundness of each of the basic operations.

While the definition of actual shape analysis tools varies slightly, this abstract semantics summarizes their overall structure. As an example, the actual implementation of **extrapol** may boil down to a simple iteration over a finite height lattice, or it may require a more complex abstract iteration technique using a widening operator.

To conclude, the definition of a shape analysis for our example language essentially relies on the following components:

- a set of abstract shapes $\mathbb{D}$, with a machine representation and a concretization function $\gamma\colon \mathbb{D} \to \mathcal{P}(\mathbb{M})$;
- an operation $\mathbf{new}_{\mathtt{x}}\colon \mathbb{D} \to \mathbb{D}$, which over-approximates the effect of an object allocation;
- an operation $\mathbf{assign}_{\mathtt{x.f} \leftarrow \mathtt{e}}\colon \mathbb{D} \to \mathbb{D}$, which over-approximates the effect of an assignment;
- an operation $\mathbf{test}_{\mathtt{e}}\colon \mathbb{D} \to \mathbb{D}$, which over-approximates the effect of a condition test;
- an operation **join**: $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$, which over-approximates the union of concrete set;
- an extrapolation operation **extrapol**: $(\mathbb{D} \to \mathbb{D}) \times \mathbb{D} \to \mathbb{D}$; and
- proofs of the soundness of $\mathbf{new}_{\mathtt{x}}$, $\mathbf{assign}_{\mathtt{x.f} \leftarrow \mathtt{e}}$, $\mathbf{test}_{\mathtt{e}}$, **join**, and **extrapol**, and of the termination property of **extrapol**.

4

Memory Layout Abstractions

This section presents the main families of shape analyses, and the underlying abstractions. For each of them, it describes the form of the shape abstract states with their concretization and machine representation, and it shows the main analysis operations so as to discuss the implementation of the operators listed in the end of Section 3. First, Subsection 4.1 identifies important features that shape abstractions should provide. Subsection 4.2 reviews the classical shape analyses based on graph abstractions (which are not necessarily able to dynamically materialize or generalize summaries). The following three subsections discuss shape analyses which are able to handle summaries in a dynamic way, and decide at analysis time when to refine or generalize them. Subsection 4.3 presents the shape analyses based on three-valued logic. Subsection 4.4 discuss the shape analyses based on separation logic. Subsection 4.5 focus on shape analyses that rely on grammar-based abstractions.

4.1 Dividing Lines

Before we dive into specific shape abstractions, we attempt to draw lines between those that try to preserve shape information and those that may fully abstract it away. This is not by any measure an absolute

rating of the strength of an analysis; it simply aims at recognizing how information about shape can be inferred. Moreover, depending on the situation, shape information may be useful to infer other properties of interest, or completely irrelevant, thus we are not either trying to rate analysis usefulness here.

Generally, shape information is about data structures the *size* and the *form* of which are not statically known. It is thus up to the analysis to characterize the size and the form of these data structures, and possibly to abstract them.

Size of Data Structures. When the number of memory cells that the analyzed program may use is fixed, finite, and small, the problem of choosing an abstraction for memory states becomes much simpler, even if the analysis should preserve accurate information. Indeed, it is then possible to either enumerate all reachable configurations, or to use abstractions that rely on the finiteness of the set of memory cells (e.g., by keeping exact information about a well identified subset of the memory cells).

On the other hand, when this number is unbounded or unknown statically, all such finite abstractions are doomed to fail. Since an abstract state may only consist of finitely many abstract predicates, describing such memory states accurately requires using predicates that can account for properties of unbounded numbers of concrete memory cells. As observed in the previous sections, these correspond to *summary* predicates. Note that such predicates may be required even when analyzing programs that allocate a fixed, bounded number of memory cells: indeed, even then, enumerating all possible configurations may be prohibitively costly, and all abstractions based on the finiteness of the memory layout may not apply.

Therefore the main point of this criterion here is to distinguish the following two kinds of analyses:

- analyses that **do not attempt to summarize memory regions** and that can exploit abstractions based on a finite set of memory cells;

- analyses that **attempt to summarize memory regions** using *global* predicates.

By essence, shape analyses fit in the second category, and try to summarize at least some kinds of unbounded data structures, whenever this is possible. Pointer analyses which handle heap-manipulating programs also fit the second category. For example, the popular allocation-site-based abstraction summarizes the properties of all the objects allocated at the same program point using a single abstract object, which, using our nomenclature, is a summary predicate of a particular form. In contrast, numerical analyses of variable-manipulating programs fit the first category. For example, in the interval domain, the possible values of every variable are captured by a unique interval.

Summarization Process. We now focus on static analyses that rely on summary predicates, and further classify them. The next relevant classification criteria should convey how powerful the summarization process is, and how close it is to the semantic definition of the data structures that are considered. As we have observed in Section 2, once summary predicates are present, their *refinement* is generally rather systematic, whereas their introduction is based on a *generalization* step, which is often conceptually harder, since it requires to select pieces of abstract information that should be retained whereas others get discarded. When the definition of the summaries is trivial and fully guided by the syntax of programs, the analysis process is simpler, as it does not need to select partitions.

However, in languages such as C or Java, no information purely based on the types, or more generally on the program text can distinguish between structures that have very different properties, such as doubly-linked lists, binary trees, or data structures that involve sharing like graphs or DAGs. Indeed, for a given type definition where an object has two pointer fields to elements of the same type, one may imagine either that these pointers describe a local relation over a linear structure (as in a doubly linked list), or refer to disjoint memory areas (as in a binary tree). Intuitively, if such a structure comes with no comment (or other implicit information such as well-chosen type or field names), a

human reading the code would have to interpret carefully the whole program so as to understand what kind of structure is actually built or manipulated. The same goes for a static analysis. More generally, the shape properties of such complex data structures are not correlated with allocation sites or other such information. This means that a simple and static definition of summaries should not be expected to work in such cases.

To sum up, we can distinguish two important categories of analyses that infer memory properties and rely on summary predicates:

- Analyses that can **decide based on fully static criteria where and how to introduce summary predicates**; such analyses typically handle summary predicates based on syntactic information present in the program text, or using the results of simple pre-analyses.
- Analyses that **need to decide where and how to introduce summary predicates dynamically** (that is during the analysis) and **based on semantic properties** (that is, on information about the data structures, that are computed by the analysis itself); such analyses cannot rely on a simple pre-analysis; instead, they let the analysis intermediate results guide the choices of the summary predicate introduction points, and of how they should be synthesized.

In the following, while we also report on works that utilize other abstractions, we restrict the use of the term "shape analysis" to analyses of the second kind, that is, analyses which determine where and how to introduce summary predicates dynamically, during the analysis process. This implies that shape analyses need to come up with some semantic criteria in order to trigger the refinement and the generalization algorithms attached to summary predicates, as shown in Figure 2.5.

Division. To recapitulate the discussion so far, we identify *shape analyses* as analyses that are able to describe some families of unbounded

memory regions using *summary predicates*, and that can do so in a dynamic manner, namely at analysis time, and based on intermediate analysis results. The ability to summarize memory regions of unbounded size without enumerating their elements is fundamental here, and requires the use of high level shape predicates. The dynamic summarization is also crucial, as it means that the analysis can handle some cases where the shape information is not readily available in the program text.

As a consequence, we expect shape analyses to provide algorithms for refining and generalizing abstract shapes involving summary predicates over the course of the analysis. However, this definition does not make any assumption regarding to the way these refinement and generalization operations are carried out. For instance, we remarked in Subsection 3.3 that generalization is quite a challenging step and that it may be handled in several rather different ways: a possible technique lossily collapses abstract states into a finite height lattice (either incrementally, or at specific points during the analysis—typically at loop heads) so as to enforce convergence of abstract iterates, whereas another approach relies on a classical widening operation applied to abstract iterates. In the following of this section, we show shape analyses that rely on each of these approaches.

4.2 Graph-Based Shape Abstractions

This subsection introduces abstractions based on shape-graphs and which are able to summarize memory regions of unbound size, but which do not support dynamic materialization during the course of the analysis.

A Brief History of Shape Analysis. To the best of our knowledge, Reynolds (1968) was the first to address the problem of shape analysis. He considered a functional programming language (i.e., one without destructive updates) and formalized the problem as solving a collection of set equations. Jones and Muchnick (1979) describe two kinds of shape analyses: One analysis also considers a Lisp-like language without destructive updates and uses tree grammars to abstract the shape of the heap. The second analysis supports destructive updates and utilizes

shape-graphs with an a priori fixed bound on the length of paths. (The analysis is discussed in depth in Subsection 4.2.1.)

Jones and Muchnick (1982) revisited the problem of shape analysis for languages without destructive updates and improved upon their earlier work by providing a more economical abstraction: they associated a grammar rule for every program point corresponding to a list construction operation instead of associating one with every program point. The idea of utilizing allocation sites as means for summarizing multiple objects was later used for languages with destructive updates in Chase *et al.* (1990) and Stransky (1992). Interestingly, the analysis of Chase *et al.* (1990) can maintain multiple nodes corresponding to objects allocated at the same site. For example, an object allocated at allocation site A is not merged into the summary nodes corresponding to the objects allocated at A if it can be determined that it is definitely pointed to by some variable. This refinement, together with an explicit record of heap-sharing information, allows the analysis to perform strong-updates in certain cases as well as to infer the preservation of "listness" and "treeness" by certain procedures, e.g., top-down or bottom-up creation of a list or a tree. However, the analysis, lacking materialization, often fails to prove the preservation of these properties when a destructive update occurs "deep in the heap", e.g., the preservation of acyclicity after inserting a node into the middle of an acyclic list.

Another innovation of Jones and Muchnick (1982) is the use of a non-disjunctive domain. Roughly speaking, this amount to using a single shape graph at every program point instead of utilizing a set of such graphs. Other classical shape analyses which utilize a non-disjunctive domain include (Chase *et al.*, 1990; Larus and Hilfinger, 1988; Stransky, 1992). The latter, however, are applicable to languages with destructive updates. The use of non-disjunctive domains leads to an interesting phenomenon where the shape graph may contain nodes representing memory objects which appear in one memory state but not in another and that different nodes represent an object allocated at the same address but coming from two different memory states.

These works were the precursors to the work of Sagiv *et al.* (1998) which was the first to introduce the concept of materialization into shape

analysis. The following two subsections discuss two modern techniques for shape analysis.

In the rest of this subsection, we review one of the seminal works in the field: The k-limiting based shape analysis of Jones and Muchnick (1979).

Note 1. In the rest of this subsection, we assume that the analyzed program P uses fixed arbitrary finite sets $\mathcal{X} \subset \mathbb{X}$ and $\mathcal{F} \subset \mathbb{F}$ of variables and fields, respectively.

4.2.1 K-Limited Abstractions

Jones and Muchnick (1979) pioneered the use of *shape graphs* as means for representing concrete memory states of heap-manipulating programs.[1] Similarly to the informal graphical abstraction used in Section 2, a shape graph conservatively represents an unbounded heap in a bounded way by collapsing sub-heaps comprised of "similar" heap-allocated objects into *summary nodes*. The analysis was designed to aid programmers choose an effective memory management technique. Specifically, it was intended to determine whether (i) a program can use reference-counting (by discovering that there are no cyclic data structures), and (ii) an element can be deallocated once the program redirects a pointer-field pointing to it (by establishing that the program does not manipulate heap-shared data structures). Thus, in addition to the maintaining a *summary* predicate distinguishing summary nodes from *regular* (non-summary) nodes, the analysis records two graph-theoretical properties pertaining to the layouts of the collapsed subheaps: heap-sharing and cyclicity.

More technically, a shape graph is a sextuple

$$\mathbb{sg} = (N, SM, S, C, \rho, E)$$

where N is a set of *nodes*, $SM \subseteq N$ records the set of *summary nodes*, and $S \subseteq SM$ and $C \subseteq SM$ register which summary nodes may correspond to subheaps containing a heap-shared node or a cycle, respectively. The function ρ: $\mathcal{X} \to N \cup \{NULL\}$ is an *abstract environment* mapping

[1] Chase *et al.* (1990) coined the term *(storage) shape graphs*.

each pointer variable either to the node that represents the object that it points to, or to the special value *NULL*. The set $E \subseteq N \times \mathcal{F} \times N$ collects directed *edges* abstracting the inter-object pointer-linked layout.

A shape graph $\mathbb{sg} = (N, SM, S, C, N, \rho, E)$ represents a memory state m, i.e., $m \in \gamma(\mathbb{sg})$, if there is a surjective function η from the symbolic addresses of the *reachable* dynamically-allocated objects in m to N such that the following holds:[2]

1. If two distinct objects are represented by the same node then this node must be a summary node, i.e., if $a \neq a'$ and $\eta(a) = \eta(a')$ then $\eta(a) \in SM$.

2. If objects a, a_1, and a_2 are part of the same collapsed subheap, i.e., $\eta(a) = n$, $\eta(a_1) = n$, and $\eta(a_2) = n$ for some $n \in SM$, and a is pointed to by fields $\mathtt{f}_1$ and $\mathtt{f}_2$ of a_1 and a_2, respectively, i.e., $m(a_1, \mathtt{f}_1) = a$, and $m(a_2, \mathtt{f}_2) = a$, and either $a_1 \neq a_2$ or $\mathtt{f}_1 \neq \mathtt{f}_2$ then $\eta(a) \in S$.

3. If a collapsed subheap contains a cycle of pointer-linked objects fields then the summary node representing it is marked as possibly cyclic, i.e., if there are locations $a_0, \ldots, a_k$ and fields $\mathtt{f}_0, \ldots, \mathtt{f}_k$ and a summary node $n \in SM$ such that for any $i = 0 \ldots k$, $\eta(a_i) = n$ and, $m(a_i, \mathtt{f}_i) = a_{(i+1) \bmod k+1}$ then $n \in C$.

4. For any variable $\mathtt{x} \in \mathcal{X}$,[3]

   (a) if $m(\mathtt{x}) = a$ then $\rho(\mathtt{x}) = \eta(a)$, and, conversely,

   (b) if $\rho(\mathtt{x}) = \eta(a)$ and $\eta(a) \notin SM$ then $m(\mathtt{x}) = a$.

5. For any field $\mathtt{f} \in \mathbb{F}$ and locations a and a',

   (a) if $m(a, \mathtt{f}) = a'$, then there is an $\mathtt{f}$-labeled edge from the node representing a to the node representing a', i.e., $(\eta(a), \mathtt{f}, \eta(a')) \in E$, and, conversely,

[2] An object at location a is *reachable* if it is accessible via a path of pointer fields starting at an object pointed to by a variable.

[3] Recall that in our language, variables are in fact constants referring to structures with a single field $\underline{\emptyset}$. For clarity, we write $m(\mathtt{x})$ instead of $m(\mathtt{x}.\underline{\emptyset})$.

(b) if there is an edge $(\eta(a), \mathtt{f}, \eta(a')) \in E$ and neither $\eta(a)$ nor $\eta(a')$ are summary nodes then $m(a, \mathtt{f}) = a'$.

For example, the shape graphs shown in Figures 4.1(a)–(d) represent the concrete memory state depicted in Figure 2.4 with a decreasing degree of accuracy (i.e., any memory state represented by Figure 4.1(a) is also represented by Figure 4.1(b), etc.). We draw nodes of shape graphs as rectangles with rounded corners and mark summary nodes using double lines. We write *shared* resp. *cyclic* below summary nodes representing possibly shared resp. possibly cyclic subheaps. We draw edges emanating from non-summary nodes using solid arrows (indicating they correspond to must-information) whereas those leaving a summary node are dashed (indicating they correspond to may-information).[4] We depict the abstract environment by drawing edges from variable names to the nodes they point to, using the same convention as above for dashed and solid arrows. (NULL-valued variables and fields are not shown in the diagram.) Each node is labeled by an identifier. These are merely used for ease of reference, and take no part in the abstraction.

Note that any memory represented by the shape graph depicted in Figure 4.1(a) must contain at least six elements and `c` must point to a list containing exactly three elements. However, the shape graph

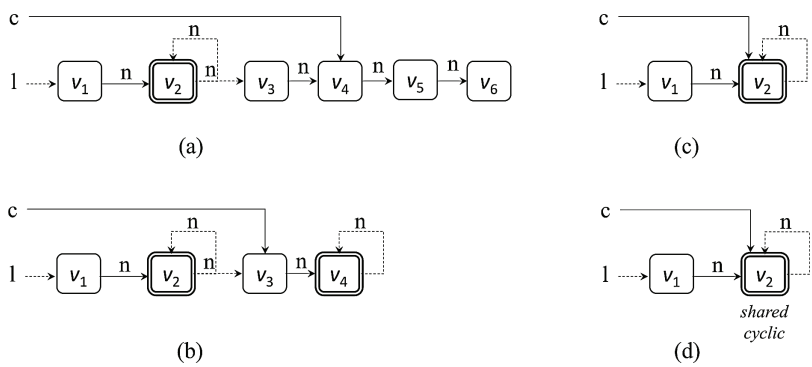


Figure 4.1: Shape graphs.

[4]The must information comes from conditions 4b and 5b of the abstraction whereas the may information is due to conditions 4a and 5a.

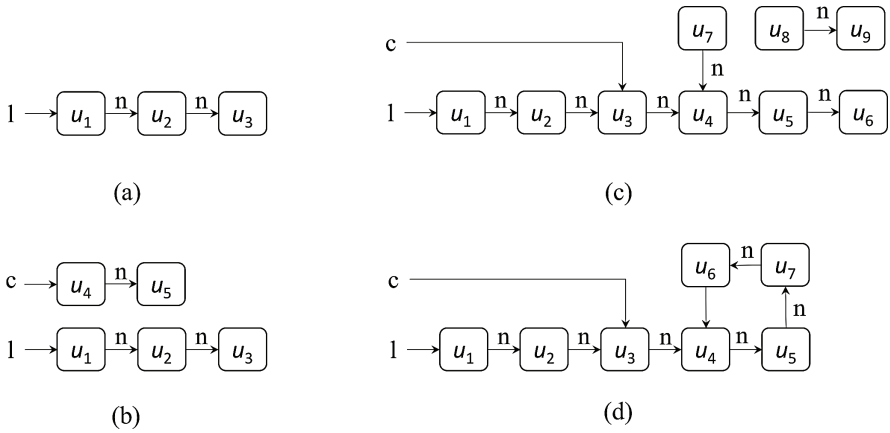


Figure 4.2: Possible concretizations of the shape graph shown in Figure 4.1(b).

depicted in Figure 4.1(c) may represent memory states with two or more allocated objects and variable `c` may point to any element in a list except the one pointed to by `l`.

As was the case in Section 2, these shape graphs also represent memory states which are, intuitively, undesirable. Consider, for example, the shape graph depicted in Figure 4.1(b). Its concretization contain the memory states depicted in Figures 4.2(a)–(c):

- Figure 4.2(a) depicts a memory state in which the value of `c` is *NULL*. This memory state is in the concretization because the shape graph abstraction allows the value of any variable annotating a summary node to be *NULL*.

- Figure 4.2(b) depicts a memory state in which the lists pointed to by `l` and `c` are disconnected. Indeed, as the shape graph does not record reachability information, a summary node may represent an unbounded number of disconnected subheaps.

- Figure 4.2(c) depicts the memory state shown in Figure 2.4, but with some unreachable (garbage) memory cells. Recall that the nodes of a shape graph record explicitly only the reachable objects. Thus, any memory state represented by a shape graph may contain an unbounded number of garbage (i.e., unreachable objects).

Figure 4.2(d) depicts a memory state in which the list ends with a cycle. Thus, out of the four shape graphs depicted in Figure 4.1, only shape graph (d) represents it. Figure 4.1(b) would have represented this memory state too had its rightmost summary node been marked as cyclic. Note that the latter does not need to be marked as shared because every object in the subheap it corresponds to, namely the ones participating in the cycle, is pointed to by exactly one field of another object in that subheap. Interestingly, the summary node does not need to be marked as shared because every object in the subheap it corresponds to, namely the ones participating in the cycle, is pointed to by exactly one field of another object in that subheap.

K-Limited Shape Graphs. The analysis ensures that the number of possible shape graphs is bounded by employing k-limiting, for a given arbitrary positive integer k: a shape graph is *k-limited* if every node it contains can be reached by a field-labeled path of length at most k starting at node pointed to by a variable. For example, the shape graphs shown in Figures 4.1(c), (d) are 0-limited, but not 1-limited, the shape graph shown in Figure 4.1(b) is 1-limited, but not 0-limited, and the shape graph shown in Figure 4.1(a) is 2-limited, but not 1-limited.

K-Bounding. As we discuss shortly, the analysis may temporarily produce non k-bounded shape graphs. To bound the resulting shape graphs, Jones and Muchnick (1979) defines a *clean* function over shape graphs which generalizes shape graphs by merging together nodes that violate the k-limit restriction. Roughly speaking, the analysis utilizes the following process to decide which nodes need to be merged together: It removes from the shape graph all the nodes which are at distance $k - 1$ or less from a node pointed to by a variable, partitions the remaining nodes into weakly connected components, and merges every component into a single summary node. The latter may be declared shared (resp., cyclic) if one of the nodes merged into it is shared (resp., cyclic) or if this is indicated by the pointer layout connecting the merged nodes. For example, applying 1-bounding to the shape graph depicted in Figure 4.1(a) would generate the shape graph depicted in Figure 4.1(b). The removal of nodes v_1 and v_4 results in two weakly

connected components: one is made of nodes v_2 and v_3 (which are merged into node v_2) and nodes v_5 and v_6 (which are merged into node v_4). Note that subgraphs of shape graph (d) made of nodes v_1 and v_4 and of v_2 and v_3 do not contain a cycle nor a shared node. Thus, nodes v_2 and v_4 in Figure 4.1(b) are not marked as shared or as cyclic.

Note 2. The abstraction used by Jones and Muchnick (1979) is slightly coarser than the one we present here as they do not label the edges that stem from summary nodes. To simplify the presentation, we chose to maintain the labels.

Shape Graph Abstract Domain. Since the number of k-bounded shape graphs is finite (for given finite sets of variable names $\mathcal{X} \subset \mathbb{X}$ and fields $\mathbb{F} \subset \mathbb{F}$), the analysis defines $\mathbb{D}$ to be the powerset of all bounded structures for P, i.e., every abstract element $\mathbb{SG} \in \mathbb{D}$ is a set of k-limited shape graphs, **join** is defined as set union, and extrapolation is defined as identity.

Note 3. The above arguments holds only if we assume that node names are irrelevant. Indeed, two shape graphs are considered equal if they are homomorphic.

4.2.2 Computation of Post-Conditions

The abstract transformers, described below, are rather straightforward with one exception: When a pointer-field emanating from a summary node is traversed, the analysis considers all possible options according to the outgoing (dashed) edges. After the transformer is executed, the analysis removes any node which is not reachable from a variable and bounds the resulting shape graphs using the *clean* function.

Here, and in the rest of this subsection, we use $\mathbb{SG} \in \mathbb{D}$ to denote a set of shape graphs and treat $\mathbb{sg}$ as a shorthand for (N, SM, S, C, ρ, E).

Post-Condition for Memory Allocation. The abstract transformer for the memory allocation command adds a new node to each input graph

and labels it with $\mathtt{x}$.

$$\mathbf{new}_{\mathtt{x}}(\mathbb{SG}) \in \\ \{(N \cup \{n\}, SM, S, C, N, \rho[\mathtt{x} \mapsto n], E) \mid \mathtt{sg} \in \mathbb{SG}, n = \mathsf{alloc}(N)\}$$

We assume that $\mathsf{alloc}(N)$ acts as an abstract memory allocating function which returns a node $n \notin N$.

Post-Condition for Pointer Assignments. We consider four kinds of prototypical assignment operations:

Nullification and Copy Assignments. Setting a variable to *NULL* and setting a pointer variable to the value of another variable merely modifies the environment.

$$\mathbf{assign}_{\mathtt{x} \leftarrow \mathtt{NULL}}(\mathbb{SG}) = \{(N, SM, S, C, N, \rho[\mathtt{x} \mapsto \mathit{NULL}], E) \mid \mathtt{sg} \in \mathbb{SG}\}$$
$$\mathbf{assign}_{\mathtt{x} \leftarrow \mathtt{y}}(\mathbb{SG}) = \{(N, SM, S, C, N, \rho[\mathtt{x} \mapsto \rho(\mathtt{y})], E) \mid \mathtt{sg} \in \mathbb{SG}\}$$

Destructive Updates. Directing the $\mathtt{f}$-field of the object pointed by variable $\mathtt{x}$ to the object pointed to by $\mathtt{y}$ adds an appropriate edge to the shape graph, provided that $\mathtt{x}$ points to some node in the shape graph. In case $\mathtt{x}$ points to a non-summary node the analysis removes the previous $\mathtt{f}$-labeled edge, and thus performs *strong-update*. However, this is not done in case $\mathtt{x}$ points to a summary node.

$$\begin{aligned} \mathbf{assign}_{\mathtt{x.f} \leftarrow \mathtt{y}}(\mathbb{SG}) = \{(N, SM, S, C, N, \rho, (E \cup E^+) \setminus E^-) \mid \mathtt{sg} \in \mathbb{SG}\} \\ \text{where } E^+ = \{(\rho(x), f, \rho(x)) \mid \rho(x) \in N\} \\ E^- = \{(\rho(x), f, n) \in E \mid \rho(x) \in N \setminus SM\} \end{aligned}$$

Recall, that a variable pointing to a summary node may still have a *NULL* value. The analysis ignores this possibility, and thus it is not sound to verify the absence of NULL-valued pointer dereferences using this analysis.

Pointer Dereferences. Traversing the $\mathtt{f}$-field of the object pointed by variable $\mathtt{x}$ may lead to more than one node in case $\mathtt{x}$ points to a summary node. Thus, the analysis performs a rather naive case analysis based on the possible targets of the relevant $\mathtt{f}$-labeled edges. This

ensures that in the resulting graphs every variable labels at most one node. Note that the analysis blocks if $\mathtt{x}$ does not point to any node in the shape graph, however it does not raise an alarm.

$$\mathbf{assign}_{\mathtt{y}\leftarrow\mathtt{x.f}}(\mathbb{SG}) = \\ \{(N, SM, S, C, N, \rho[\mathtt{y} \mapsto n], E) \mid \mathtt{sg} \in \mathbb{SG} \wedge (\rho(x), \mathtt{f}, n) \in E\}$$

Post-Condition for Condition Tests. We consider two prototypical pointer-related condition tests $\mathtt{x} \bowtie \mathtt{y}$: checking whether the values of variables $\mathtt{x}$ and $\mathtt{y}$ are equal ($\bowtie$ is $=$) or not ($\bowtie$ is $\neq$). If both variables point to non-summary nodes then the test can be done by checking whether they label the same node or not. Otherwise, the tests are done rather conservatively because the value of the variable labeling a summary node may be *NULL*.

$$\mathbf{test}_{\mathtt{x}\bowtie\mathtt{y}}(\mathbb{SG}) = \\ \{\mathtt{sg} \in \mathbb{SG} \mid \rho(x) \bowtie \rho(y) \vee \\ (\rho(\mathtt{x}) = \bot \vee \rho(\mathtt{x}) \in SM) \wedge (\rho(\mathtt{y}) = \bot \vee \rho(\mathtt{y}) \in SM)\}$$

4.3 Three-Valued Logic Shape Abstraction

Three-valued shape analysis (Sagiv *et al.*, 2002) (3VSA for short) utilizes logic in two important ways: (1) to represent concrete heaps and abstract heaps, and (2) to express semantic operations.

This subsection is organized as follows: (i) We explain how logical structures are used to represent concrete heaps, (ii) We explain how 3-valued logical structures are used to represent abstract heaps and define a parametric abstract domain; (iii) We define a concrete semantics using the concept of predicate updates; (iv) We define an instrumented semantics as an intermediate step between the concrete semantics and the abstract semantics. The instrumented semantics extends the concrete semantics with information used to improve the precision of the abstract semantics; (v) We define a vanilla abstract semantics, which is sound but rather imprecise; and finally (vi) We define techniques for improving the precision of the abstract semantics.

First-Order Predicates. In 3VSA, we use the term *predicate* to refer to a first-order relation symbol p, and write $p^{(k)}$ to explicitly indicate that the rank of p is k. Nullary predicates (predicate of rank 0) are an important special case of first-order predicates. Those are often referred to as *state predicates* and are employed by model checkers and static analyses based on *predicate abstraction* (Graf and Saïdi, 1997). We use the term *vocabulary* to mean a non-empty set of first-order predicates.

4.3.1 A Logical Representation of Concrete Memory States

In 3VSA, concrete memory states are represented by 2-valued structures, which we define next.

Definition 4.1. A 2-*valued logical structure* over a vocabulary $\mathcal{V}$ is a pair $S = \langle U, \iota \rangle$ where U is the *universe* (a finite set of *individuals*) of the 2-valued structure, and ι is the *interpretation function* mapping predicates to their truth-value in the structure: for every predicate $p^{(k)} \in \mathcal{V}$, $\iota(p^{(k)})$: $U^k \to \{0, 1\}$ maps k-tuples of individuals to either 0 (false) or 1 (true).

We will use the notation U^S and ι^S to refer to the universe and interpretation of a structure S, respectively.

The main reason for using 2-valued structures is to provide a freedom in choosing the desired implementation details in order to suit the programming language and analysis goals. The analysis designer can choose a set of individuals to represent, for example: objects, threads, arrays, or stack frames. The analysis designer can also choose predicates to represent properties of the concrete semantics, for example: pointer values, less-than relation over the values of numerical fields, or order of stack frames. Another reason for doing so is that it allows defining the abstract semantics (analysis) using similar means, which greatly simplifies the technical development and the proof of soundness, as we show in the next subsections.

Typically, a 2-valued structure $S = \langle U, \iota \rangle$ represents the set of allocated objects in the heap by U and uses a vocabulary of *core predicates* $\mathcal{V}^{\text{core}}$, consisting of unary predicates and binary predicates.[5]

[5] The reason for using the term core predicates will become clearer later.

Specifically, unary predicates are used to represent the values of pointer variables and binary predicates are used to capture the values of pointer fields and for the equality predicate, $eq^{(2)}$:

Predicate	**Meaning**
$x(u)$	The pointer variable $\mathtt{x} \in \mathbb{X}$ points to object u
$f(u, v)$	The value of the pointer field $\mathtt{f} \in \mathbb{F}$ of object u is v
$eq(u, v)$	The individuals u and v are equal

Example 4.1. To represent the concrete heap shown in Figure 2.4, we use the vocabulary $\{t^{(1)}, l^{(1)}, \mathit{fresh}^{(1)}, n^{(2)}, eq^{(2)}\}$ where t and l represent the variables `t` and `l`, respectively, and n represents the list field. The universe consists of the individuals $u_{1..7}$ for each of the list cells in the order of appearance in the list. The interpretation ι_0 function is as follows:

$$\iota_0(l) = \lambda u \in u_{1..7}. \begin{cases} 1, & \text{if } u = u_1; \\ 0, & \text{otherwise.} \end{cases}$$

$$\iota_0(t) = \lambda u \in u_{1..7}. \begin{cases} 1, & \text{if } u = u_5; \\ 0, & \text{otherwise.} \end{cases}$$

$$\iota_0(n) = \lambda (u_i, u_j) \in u_{1..7} \times u_{1..7}. \begin{cases} 1, & \text{if } j = i + 1; \\ 0, & \text{otherwise.} \end{cases}$$

$$\iota_0(eq) = \lambda (u_i, u_j) \in u_{1..7} \times u_{1..7}. \begin{cases} 1, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases}$$

The resulting structure, $S_0 = \langle u_{1..7}, \iota_0 \rangle$ is shown in Figure 4.3; for now ignore the labels appearing under the nodes, they will be explained in Subsection 4.3.2. We depict unary predicates that correspond to pointer variables by an arrow from the variable name to the individual for which they hold. We depict binary predicates by edges labelled by the name of the predicate, except for eq, which is understood from the set of individuals.

Notice that the null value is represented implicitly by evaluating predicates to 0. For example, $\iota_0(n)(u_7, u) = 0$ for each $u \in u_{1..7}$, which means that the value of n of the object represented by u_7 is null.

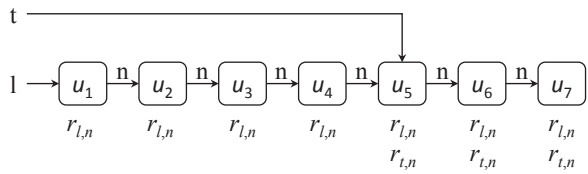


Figure 4.3: The 2-valued structure S_0. A label appearing under a node indicates that the corresponding unary instrumentation predicate holds for that node.

4.3.2 A Logical Representation of Abstract Heaps

To be able to represent infinite sets of 2-valued structures in a finite way, we now define the concept of abstract states used by 3VSA.

Consider the abstract state shown in Figure 2.4. How would we represent it using a logical structure? One idea is to allow abstract individuals (*summary individuals*) to represent more than a single concrete individual. Specifically, our "abstract logical structure", $S_0^\sharp = \langle U_0^\sharp, \iota_0^\sharp \rangle$, would have three summary individuals $U_0^\sharp = \{v_1, v_2, v_3\}$ where v_1 represents $u_{1..4}$, v_2 represents u_5, and v_3 represents $u_{6..7}$.

The question is how would we interpret the predicates over the abstract individuals? For t, the answer is obvious, since v_2 corresponds exactly to u_5:

$$\iota_0^\sharp(t) = \lambda v \in v_{1..3}. \begin{cases} 1, & \text{if } v = v_2; \\ 0, & \text{otherwise.} \end{cases}$$

For l, the answer is obvious for v_2 and v_3, since `l` does not point to any of $u_{5..7}$, but for v_1, which value should we choose as k_1?

$$\iota_0^\sharp(l) = \lambda v \in v_{1..3}. \begin{cases} k_1, & \text{if } v = v_1; \\ 0, & \text{otherwise.} \end{cases}$$

If we want the value of k_1 to represent all the values $\{\iota_0(l)(u_i)\}_{i=1}^4$ then we have a problem, since $\iota_0(l)(u_1) = 1$ and $\iota_0(l)(u_2) = \iota_0(l)(u_3) = \iota_0(l)(u_4) = 0$. We need a way to represent the set of values $\{0, 1\}$.

To achieve this, we utilize Kleene's 3-valued logic (Kleene, 1952), which uses the values 0, 1, and $\frac{1}{2}$ to represent sets of Boolean values:

Kleene value	Boolean values	Description
0	$\{0\}$	True
1	$\{1\}$	False
$\frac{1}{2}$	$\{0,1\}$	Unknown

We equip the set of Kleene values $\{0, 1, \frac{1}{2}\}$ with the partial order $0 \sqsubseteq \frac{1}{2}$ and $0 \sqsubseteq \frac{1}{2}$ and the join operation defined as follows:

$$v_1 \sqcup v_2 = \begin{cases} v_1, & \text{if } v_1 = v_2; \\ \frac{1}{2}, & \text{otherwise.} \end{cases}$$

For example, $\iota_0(l)(u_1) \sqcup \iota_0(l)(u_2) \sqcup \iota_0(l)(u_3) \sqcup \iota_0(l)(u_4) = 1 \sqcup 0 \sqcup 0 \sqcup 0 = \frac{1}{2}$.

To include Kleene values in logical structures, we generalize 2-valued structures by 3-valued structures.

Definition 4.2. A 3-*valued logical structure*, also called an *abstract structure*, over a finite set of predicates $\mathcal{V}$ is a pair $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ where $U^\sharp$ is the universe of the 3-valued structure, and $\iota^\sharp$ is the interpretation function mapping predicates to their truth-values in the structure: for every predicate $p \in \mathcal{V}$ of rank k, $\iota^\sharp(p)$: $U^k \to \{0, 1, \frac{1}{2}\}$.

Technically, the only difference between 2-valued and 3-valued structures is that the interpretation function in the latter might result in an unknown ($\frac{1}{2}$) value while the former always return a *definite* value (i.e., either true or false). Intuitively, however, the main difference between 2-valued and 3-valued structures in our context is that every individual of the former is used to represent a single element of the concrete semantics, e.g., an heap-allocated object or an activation record, where an individual of the latter might represent multiple concrete elements.

Example 4.2. We can represent the abstract state in Figure 2.4 by the 3-valued structure $S_0^\sharp = \langle v_{1..3}, \iota_0^\sharp \rangle$ where $\iota_0^\sharp$ is given as follows:

$$\iota_0^\sharp(l) = \lambda v \in v_{1..3}. \begin{cases} \frac{1}{2}, & \text{if } v = v_1; \\ 0, & \text{otherwise.} \end{cases}$$

$$\iota_0^\sharp(t) = \lambda v \in v_{1..3}. \begin{cases} 1, & \text{if } v = v_2; \\ 0, & \text{otherwise.} \end{cases}$$

$$\iota_0^\sharp(n) = \lambda (v_i, v_j) \in v_{1..3} \times v_{1..3}. \begin{cases} \frac{1}{2}, & \text{if } i = 1, j = 1; \\ \frac{1}{2}, & \text{if } i = 1, j = 2; \\ \frac{1}{2}, & \text{if } i = 2, j = 3; \\ \frac{1}{2}, & \text{if } i = 3, j = 3; \\ 0, & \text{otherwise.} \end{cases}$$

$$\iota_0^\sharp(eq) = \lambda (v_i, v_j) \in v_{1..3} \times v_{1..3}. \begin{cases} \frac{1}{2}, & \text{if } i = 1, j = 1; \\ 1, & \text{if } i = 2, j = 2; \\ \frac{1}{2}, & \text{if } i = 3, j = 3; \\ 0, & \text{otherwise.} \end{cases}$$

The 3-valued structure $S_0^\sharp$ is shown in Figure 4.4. We depict $\frac{1}{2}$ values by dashed edges. A double-lined individual v means that $\iota_0^\sharp(eq)(v, v) = \frac{1}{2}$, which means that v may represent more than one concrete element. In other words, v is a *summary individual*.

Since 2-valued structures are a special case of 3-valued structure, we will often define operations directly over 3-valued structures, instead of repeating the definition once for each type of structures.

We say that a 3-valued structure $S^\sharp$ *represents* a 2-valued structure S when S is *embedded* in $S^\sharp$, as we define next.

Definition 4.3 (Embedding). Let $S = \langle U, \iota \rangle$ and $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ be 3-valued structures. Further, let $\eta\colon U \to U^\sharp$ be a surjective function.

We say that S is η-*embedded* in $S^\sharp$, denoted by $S \sqsubseteq^\eta S^\sharp$, if the following condition holds for every k-arity predicate $p \in \mathcal{V}$ and k-tuple $(u_1, \ldots, u_k) \in U^k$:

$$\iota(p)(u_1, \ldots, u_k) \sqsubseteq \iota^\sharp(p)(\eta(u_1), \ldots, \eta(u_k)). \tag{4.1}$$

We say that S is *embedded* in $S^\sharp$, denoted $S \sqsubseteq S^\sharp$, if there exists an embedding function η such that $S \sqsubseteq^\eta S^\sharp$.

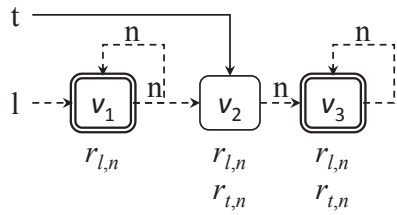


Figure 4.4: The 3-valued structure $S_0^\sharp$.

Example 4.3. We can see that $S_0 \sqsubseteq^\eta S_0^\sharp$ holds where η is defined as follows:[6]

$$\eta = \lambda u \in u_{1..7}. \begin{cases} v_1, & \text{if } u \in u_{1..4}; \\ v_2, & \text{if } u = u_5; \\ v_3, & \text{if } y \in u_{6..7}. \end{cases}$$

Let *2-Struct*$[\mathcal{V}]$ and *3-Struct*$[\mathcal{V}]$ denote the set of 2-valued structures and the set of 3-valued structures over a parametric vocabulary $\mathcal{V}$, respectively. The embedding relation induces the concretization function γ: *3-Struct*$[\mathcal{V}^{\text{core}}] \to \mathcal{P}($*2-Struct*$[\mathcal{V}^{\text{core}}])$, defined as follows:

$$\gamma(S^\sharp) = \{S \in \textit{2-Struct}[\mathcal{V}^{\text{core}}] \mid S \sqsubseteq S^\sharp\}.$$

In our example, $\gamma(S_0^\sharp)$ includes S_0 and, undesirably, the 2-valued structures S_1, S_2, and S_3 shown in Figure 4.5: S_1 does not represent a valid heap, since pointer variables and fields cannot point to more than one object (note that in $S_0^\sharp$, there is a dashed line from `l` to v_1, indicating that the unary predicate l may hold at none, some, or all the concrete node mapped to v_1 by the embedding function); S_2 represents a disconnected list; and S_3 represents a list ending with a cycle.

These examples describe two general problems:

Invalid Structures. Abstract structures may represent concrete structures that do not correspond to any concrete state defined by the operational semantics. To address this problem, 3VSA uses the concept of *integrity constraints*.

Imprecise Structures. Abstract structures may represent concrete structures that are unwanted from the perspective of analysis goals.

[6] The readers may want to check for themselves that (4.1) holds.

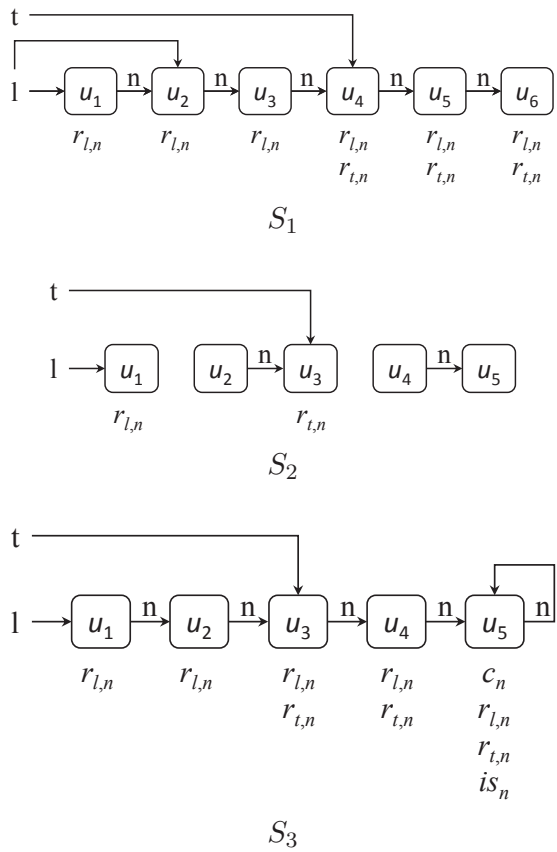


Figure 4.5: $\{S_1, S_2, S_3\} \subset \gamma(S_0^\sharp)$.

To address this problem, 3VSA uses the concept of *instrumentation predicates*.

In both cases mentioned above, 3VSA refines the concretization function by employing formulas in first-order logic with transitive closure, which we now discuss.

First-Order Logic with Transitive Closure. The syntax of formulas in first-order logic with transitive closure, or $\mathsf{FO}^{\mathsf{TC}}$ for short, is as follows:[7]

$$\begin{array}{rcllllll} A & \to & 0 \mid 1 & \mid & p(v_{1..k}) & \mid & u = v \\ & \mid & A \wedge A & \mid & A \vee A & \mid & \neg\, A \\ & \mid & \forall v.\ A & \mid & \exists v.\ A & \mid & p^{+}(u,v) \\ & \mid & p^{*}(u,v) & \mid & A\ ?\ A\colon A & \mid & \end{array}$$

Let *Node* be the set of all individuals from which the universe of each structure draws its elements from and let *Var* be an enumerable set of logical variables. An *assignment* μ is a partial function $\mathit{Var} \rightharpoonup \mathit{Node}$.

The meaning of a formula $\varphi \in A$ with free variables $FV(\varphi) = v_{1..k}$ is given by the function $[\![\varphi]\!]\colon \mathit{2\text{-}Struct}[\mathcal{V}] \to (FV(\varphi) \to \mathit{Node}) \to \{0,1\}$, which is defined by structural induction as follows:[8]

$$\begin{aligned} [\![0]\!](S)(\mu) &\equiv 0 \\ [\![1]\!](S)(\mu) &\equiv 1 \\ [\![p(v_{1..k})]\!](S)(\mu) &\equiv \iota(p)(\mu(v_1),\ldots,\mu(v_1)) \\ [\![A_1 \wedge A_2]\!](S)(\mu) &\equiv [\![A_1]\!](S)(\mu) \wedge [\![A_2]\!](S)(\mu) \\ [\![A_1 \vee A_2]\!](S)(\mu) &\equiv [\![A_1]\!](S)(\mu) \vee [\![A_2]\!](S)(\mu) \\ [\![\neg A]\!](S)(\mu) &\equiv \neg [\![A]\!](S)(\mu) \\ [\![\exists v.\ A]\!](S)(\mu) &\equiv \bigvee_{n \in U} [\![A]\!](S)(\mu[v \mapsto n]) \\ [\![\forall v.\ A]\!](S)(\mu) &\equiv \bigwedge_{n \in U} [\![A]\!](S)(\mu[v \mapsto n]) \\ [\![p^{+}(u,v)]\!](S)(\mu) &\equiv \bigvee_{k=1..|U|} [\![\exists v_{1..k}.\ u = v_1 \wedge \\ &\qquad v = v_k \bigwedge_{i=1..k} p(v_i, v_{i+1})]\!](S)(\mu) \\ [\![p^{*}(u,v]\!](S)(\mu) &\equiv [\![p^{+}(u,v) \vee u = v]\!](S)(\mu) \\ [\![A_1 ? A_2 \colon A_3]\!](S)(\mu) &\equiv [\![A_1 \wedge A_2 \vee \neg A_1 \wedge A_3]\!](S)(\mu) \end{aligned}$$

[7] 3VSA supports a more general form of transitive closure, which considers arbitrary formulas with two free variables.

[8] The Boolean operations $\wedge, \vee, \neg$ are, respectively, min, max, and $\lambda b \in \{0,1\}.\ 1-b$.

When φ is a closed formula, i.e., $FV(\varphi) = \emptyset$, we can omit the variable-to-node assignment and resort to the simpler meaning function $[\![\varphi]\!]$: *2-Struct*$[\mathcal{V}] \to \{0, 1\}$.

Example 4.4. The following formulas check whether there exist cells that are unreachable from any program variable and whether there exists a cycle, respectively:

$$\begin{aligned} \mathit{exists\_leak} &\equiv \neg\Big(\forall v.\ \bigvee_{z \in \{l,t\}} \exists u.\ z(u) \wedge n^*(u, v)\Big) \\ \mathit{exists\_cycle} &\equiv \exists v.\ n^+(v, v). \end{aligned}$$

Evaluating the formulas above on the previously shown structures, yields the following results:

$$\begin{aligned} [\![\mathit{exists\_leak}]\!](S_0) &= 0 \\ [\![\mathit{exists\_leak}]\!](S_2) &= 1 \\ [\![\mathit{exists\_cycle}]\!](S_0) &= 0 \\ [\![\mathit{exists\_cycle}]\!](S_3) &= 1 \end{aligned}$$

Ruling Out Invalid Structures via Integrity Constraints

An *integrity constraint* is a formula $\psi \in \mathsf{FO}^{\mathsf{TC}}$ that must hold for any concrete structure that matches the operational semantics. For example, to ensure that pointer variables reference at most one object, we use the following set of constraints:

$$\mathbb{X}_{cons} \equiv \{\forall u, v.\ x(u) \wedge x(v) \implies u = v \mid x \in \mathbb{X}\}$$

and to ensure that any pointer field ($f \in \mathbb{F}$) of any given object (captured by u below) references at most one object, we use the following set of constraints:

$$\mathbb{F}_{cons} \equiv \{\forall u, v, w.\ f(u, v) \wedge f(u, w) \implies v = w \mid f \in \mathbb{F}\}.$$

Let *cons* be the set of all constraints ($cons = \mathbb{X}_{cons} \cup \mathbb{F}_{cons}$ in our example). We utilize the concretization function γ_{cons}: *3-Struct*$[\mathcal{V}^{\text{core}}] \to \mathcal{P}($*2-Struct*$[\mathcal{V}^{\text{core}}])$, which filters out 2-valued structures that do not

correspond to valid heaps, defined as follows:

$$\gamma_{cons}(S^\sharp) \equiv \gamma(S^\sharp) \cap \left\{ S \in \textit{2-Struct}[\mathcal{V}^{\text{core}}] \mid S \models \bigwedge_{\psi \in cons} \psi \right\}.$$

With the pointer constraints defined above, $S_1 \notin \gamma_{cons}(S_0^\sharp)$ holds.

Ruling Out Imprecise Structures via Instrumentation

How would we rule out S_2 and S_3 from $\gamma_{cons}(S_0^\sharp)$? Intuitively, we need a means to convey the fact that the set of individuals represented by each of the summary individuals v_1 and v_3 are not connected by arbitrary `n`-links but rather that the links start from the node referenced by `l` and then continue to reach each node, without closing a cycle.

We can convey this information by extending the initial set of predicates used to define heaps, which we refer to as the *core predicates* and denote as $\mathcal{V}^{\text{core}}$, by another set of predicates that convey properties deemed as important for the purpose of the analysis. The added predicates are defined by $\mathsf{FO}^{\mathsf{TC}}$ formulas over the core predicates and are therefore dubbed *derived predicates* (a synonym is *instrumentation predicates*) and denoted as $\mathcal{V}^{\text{inst}}$.

In the sequel, we write $\mathcal{V}$ for $\mathcal{V}^{\text{core}} \uplus \mathcal{V}^{\text{inst}}$ and denote the formula defining an instrumentation predicate p as φ_p.

In our example, we are interested in proving the following properties: (i) the list starts out as acyclic and remains acyclic, (ii) list cells are not shared, (iii) cells are not taken out of the list, (iv) and the newly allocated cell is part of the list upon termination. For this purpose, we use the following instrumentation predicates:

Reachability from a Variable. We express that a cell v is reachable from a variable z via a sequence of f-links by the following instrumentation predicate:

$$r_{z,f}(v) \equiv \exists u.\ z(u) \wedge f^*(u, v).$$

In our example, we use $r_{l,n}(v)$, $r_{t,n}(v)$, and $r_{\textit{fresh},n}(v)$.

Cycles. We express that a cell v resides on a cycle of f-links by the following instrumentation predicate:

$$c_f(v) \equiv f^+(v, v).$$

In our example, we use $c_n(v)$.

Sharing. We express that a cell v is shared by two or more f-links by the following instrumentation predicate:

$$is_f(v) \equiv \neg(\forall u_1, u_2.\ f(u_1, v) \wedge f(u_2, v) \implies u_1 = u_2).$$

In our example, we use $is_n(v)$.

Definition 4.4 (Instrumentation). Let $\mu_{\overline{v},\overline{o}} \equiv \{v_i \mapsto o_i \mid i = 1..k\}$ denote the assignment defined via a tuple of logical variables $v_{1..k}$ and a corresponding tuple of nodes $o_{1..k}$.

The *instrumentation function instrument*: $\mathit{2\text{-}Struct}[\mathcal{V}^{\text{core}}] \to \mathit{2\text{-}Struct}[\mathcal{V}^{\text{core}} \cup \mathcal{V}^{\text{inst}}]$ takes a structure $\langle S, \iota \rangle$ over the core vocabulary and extends it into a structure $\langle S, \iota' \rangle$ over both the core predicates and instrumentation predicates:

$$\iota'(p^{(k)}) \equiv \lambda \overline{o} \in \mathit{Node}^k.\ \begin{cases} [\![\varphi_p]\!](S)(\mu_{\overline{v},\overline{o}}), & p \in \mathcal{V}^{\text{inst}}, \overline{v} = FV(\varphi_p); \\ \iota(p)(\overline{o}), & p \in \mathcal{V}^{\text{core}}. \end{cases}$$

Example 4.5. The concrete structures $\{S_0, \ldots, S_4\}$ shown in Figures 4.3 and 4.5 depict the values of the unary instrumentation predicates defined above as labels attached to the corresponding nodes. That is, a label missing from a node indicates that the predicate does not hold for that node. The abstract structure $S^\sharp$ shown in Figure 4.4 similarly depicts the values of instrumentation predicates. Notice that neither of v_1, v_2, v_3 is labelled by c_n or is_n, which indicates that $S^\sharp$ represents an acyclic and unshared list. Also, since all nodes are labeled by $r_{l,n}$, we know that $S^\sharp$ represents concrete states where all cells are on the list.

To formalize how instrumentation filters out undesirable structures, we define the concretization function γ_{inst}: $\mathit{3\text{-}Struct}[\mathcal{V}^{\text{core}} \cup \mathcal{V}^{\text{inst}}] \to \mathcal{P}(\mathit{2\text{-}Struct}[\mathcal{V}^{\text{core}}])$ as follows:

$$\gamma_{\text{inst}}(S^\sharp) \equiv \gamma(S) \cap \{S \mid \mathit{instrument}(S) \sqsubseteq S^\sharp\}.$$

With the definition above, we have that $S_0 \in \gamma_{\text{inst}}(S_0^\sharp)$ while $\{S_1, S_2, S_3, S_4\} \not\subseteq \gamma_{\text{inst}}(S_0^\sharp)$.

Finally, we combine both improvements to precision (due to constraints and due to instrumentation predicates) and define the concretization $\gamma_{precise}$: $3\text{-}Struct[\mathcal{V}^{\text{core}} \uplus \mathcal{V}^{\text{inst}}] \to \mathcal{P}(2\text{-}Struct[\mathcal{V}^{\text{core}}])$ as follows:

$$\gamma_{precise}(S^\sharp) \equiv \gamma_{cons}(S) \cap \gamma_{\text{inst}}(S).$$

Note 4. The predicates used to capture sharing and cyclicity have a similar role to the properties tracked in the shape graph-based abstraction of Jones and Muchnick (1979) (see Subsection 4.2). However, there is a subtle difference: In Jones and Muchnick (1979), a node was marked as shared (resp., cyclic) if this property was true in the subheap comprised of the objects that node represents. Here, in contrast, the predicates have a global interpretation, for example, if the property c_n holds for an abstract individual then this means that one of the concrete individuals that individual represents resides on a cycle, regardless of the way the embedding function treats the other concrete individuals comprising the cycle.

4.3.3 The Abstract Domain

Since the set of 3-valued structures is infinite, we now describe how to abstract (possibly-infinite) sets of (2-valued or 3-valued) structures into finite sets of 3-valued structures.

Bounded Structures

Definition 4.5. Let $S^\sharp\langle U^\sharp, \iota^\sharp\rangle$ be a 3-valued structure. The *canonical name* of a node $u \in U^\sharp$, denoted $cname(u)$ is defined as $cname(u) \equiv \lambda p^{(1)} \in \mathcal{V}.\ \iota^\sharp(p)(u)$.

Definition 4.6. Let $S^\sharp\langle U^\sharp, \iota^\sharp\rangle$ be a 3-valued structure. We say that $S^\sharp$ is *bounded* if a canonical name uniquely identifies a node: $\forall u_1, u_2 \in U^\sharp.\ cname(u_1) = cname(u_2) \implies u_1 = u_2$.

Example 4.6. The structure $S_0^\sharp$ from Figure 4.4 is bounded. To see this notice that nodes v_1 and v_2 are distinguished by the value of $r_{t,n}(v)$ (0

for v_1 and 1 for v_2), nodes v_1 and v_3 are similarly distinguished by the value of $r_{t,n}(v)$, and nodes v_2 and v_3 are distinguished by the value of t (1 for v_2 and 0 for v_3).

We denote the set of all bounded structures over as $BStruct[\mathcal{V}]$.

Corollary 4.1. The size of the set of bounded structures is asymptotically bounded as follows: $|BStruct[\mathcal{V}]| \in O(2^{3^{|\mathcal{V}|}})$.

The operation $\mathsf{blur}[\mathcal{V}]$: $3\text{-}Struct[\mathcal{V}] \to BStruct[\mathcal{V}]$, which is parameterized by the vocabulary $\mathcal{V}$, maps a 3-valued structure $\langle U, \iota\rangle$ into a bounded structure $\langle U^\sharp, \iota^\sharp\rangle$ by conflating individuals with the same canonical name and over-approximating the interpretation function using the Kleene join operation:

$$\begin{aligned}
\mathsf{blur}[\mathcal{V}](\langle U, \iota\rangle) &\equiv \langle U^\sharp, \iota^\sharp\rangle \\
U^\sharp &\equiv \{cname(u) \mid u \in U\} \\
\iota^\sharp &\equiv \lambda p^{(k)} \in \mathcal{V}.\ u^\sharp_{1..k} \in {U^\sharp}^k. \\
&\qquad \sqcup \{\iota(p)(u_{1..k}) \mid cname(u_i) = u^\sharp_i, i = 1..k\}.
\end{aligned}$$

In the sequel, we write blur instead $\mathsf{blur}[\mathcal{V}]$, when confusion is likely.

Example 4.7. Considering S_0 in Figure 4.3 and $S_0^\sharp$ in Figure 4.4, we have the following: $\mathsf{blur}(S_0) = S_0^\sharp$.

We define the 3VSA abstract domain over the vocabulary $\mathcal{V} \equiv \mathcal{V}^{\text{core}} \uplus \mathcal{V}^{\text{inst}}$ as follows:

$$\mathbb{D} \equiv BStruct[\mathcal{V}] \cup \bot \cup \top.$$

The value $\bot$ is somewhat artificial, and used to denote the result of partial functions on structures (for example, checking whether a condition holds). The value $\top$ is intuitively used to signal that a possible error in the program has been detected.

We extend concretization to operate over $\bot, \top$ as follows:

$$\begin{aligned}
\gamma_{precise}(\bot) &\equiv \emptyset \\
\gamma_{precise}(\top) &\equiv BStruct[\mathcal{V}].
\end{aligned}$$

We define the join operation as union **join** $\equiv \cup$. Since the abstract domain is finite, there is no need for extrapolation, which we define as identity.[9]

4.3.4 Concrete Semantics

We now describe how the concrete semantics of basic statements is implemented for 2-valued structures using a *predicate update* mechanism. This semantics forms the basis for the abstract semantics.

The semantics is defined for individual statements over individual elements of the abstract domain, $d \in \mathbb{D}$. More specifically, we will define the semantics for individual structures as $[\![st]\!](S) = S'$ where st is a basic statement, S is a 2-valued structure or $\bot$, and S' is either a 2-valued structure, $\bot$, or $\top$. Then, we extend the semantics to be strict in $\bot$ and $\top$, as follows:

$$[\![st]\!](d) \equiv \begin{cases} \bot, & \text{if } d = \bot; \\ \top, & \text{if } d = \top; \\ [\![st]\!](S) & \text{if } d = S \in \textit{2-Struct}[\mathcal{V}]. \end{cases}$$

Since the concrete domain consists of sets of abstract elements, we lift the semantics to sets of structures as follows:

$$[\![st]\!](X) = \{d \mid [\![st]\!](d), d \in X\} \setminus \{\bot\}.$$

We remove all occurrences of $\bot$, since it is used to signify a missing structure.

A Simplifying Transformation. We employ the following source-to-source semantics-preserving transformations to simplify our semantics:

- We use the transformation `x->f=y->g` $\rightsquigarrow$ `t=y->g; x->f=t`, where `t` is a fresh temporary variable. In our example, the statement at line 11 is rewritten to `tn=t->n; fresh->n=tn` where `tn` is the newly added temporary variable.

[9] Technically, it is possible to apply the blur operation only at loop cut-points, thereby trading efficiency for increased precision.

- We separate memory safety checks from statements that dereference memory. Specifically, we use the following transformations: `x->f=y` $\rightsquigarrow$ `assert x!=null; x->f=y`, and `x=y->f` $\rightsquigarrow$ `assert y!=null; x=y->f`.

Predicate Updates

An update formula for a predicate $p^{(k)}$ has the form $p'(\overline{v}) \leftrightarrow \varphi(\overline{v})$, where $\overline{v}$ is a k-tuple of (distinct) variables and $\varphi(\overline{v})$ is an $\mathsf{FO}^{\mathsf{TC}}$ formula such that whose set of free variables is exactly $\overline{v}$.

The semantic effect of an update is defined as follows:

$$[\![p'(\overline{v}) \leftrightarrow \varphi(\overline{v})]\!](S) \equiv \langle U, \iota[p \mapsto \lambda \overline{o}.\ [\![\varphi]\!](S)(\mu_{\overline{v},\overline{o}})]\rangle.$$

That is, the values of the predicate p in the resulting structure are computed, for each tuple of individuals $\overline{o}$ (matching the variables $\overline{v}$), based on the value of the formula φ, as it is evaluated on the input structure S. The values of the other predicates remain unchanged.

We can extend updates to a set $\{p'_i(\overline{v_i}) \leftrightarrow \varphi_i(\overline{v_i})\}_{i=1}^{k}$ as follows:

$$[\![\{p'_i(\overline{v_i}) \leftrightarrow \varphi_i(\overline{v_i})\}_{i=1}^{k}]\!](S) \equiv$$
$$\langle U, \iota[p_1 \mapsto \lambda \overline{o_1}.\ [\![\varphi_1]\!](S)(\mu_{\overline{v_1},\overline{o_1}}), \ldots, p_k \mapsto \lambda \overline{o_k}.\ [\![\varphi_k]\!](S)(\mu_{\overline{v_k},\overline{o_k}})]\rangle.$$

Modeling Assertions and Conditionals

The semantics of an assertion of the form `assert` e and a conditional expression e is obtained by first representing e by a corresponding *expression formula* $\hat{e} \in \mathsf{FO}^{\mathsf{TC}}$. For example, we employ the following expression formulas:

$$\widehat{\texttt{x==y}} \equiv \forall v.\ x(v) \leftrightarrow y(v)$$
$$\widehat{\texttt{x!=y}} \equiv \neg\forall v.\ x(v) \leftrightarrow y(v)$$
$$\widehat{\texttt{x==null}} \equiv \neg\exists v.\ x(v)$$
$$\widehat{\texttt{x!=null}} \equiv \exists v.\ x(v).$$

We model an assertion as follows:

$$[\![\texttt{assert}\ \varphi]\!](S) \equiv \begin{cases} S, & \text{if } [\![\varphi]\!](S) = 1; \\ \top, & \text{otherwise.} \end{cases}$$

We model a conditional as follows:

$$[\![e]\!](S) \equiv \begin{cases} S, & \text{if } [\![e]\!] = 1; \\ \bot, & \text{otherwise.} \end{cases}$$

Modeling Allocation: `x=malloc()`

To model allocation, we add a fresh individual to the input structure and mark it with the dedicated predicate isNew. Technically, we define the operation *alloc*: *3-Struct*$[\mathcal{V}] \to$ *3-Struct*$[\mathcal{V} \cup \{\mathsf{isNew}\}]$ as follows:

$$alloc(\langle U, \iota \rangle) = \left\langle U \cup \{u\}, \iota \left[\mathsf{isNew} \mapsto \lambda v. \begin{cases} 1, & \text{if } v = u; \\ 0, & \text{otherwise.} \end{cases} \right] \right\rangle$$

where $u \notin U$.

The isNew predicate temporarily extends the vocabulary to enable an update formula to reference the newly-allocated individual:

$$x'(v) \leftrightarrow \mathsf{isNew}(v).$$

The predicate isNew is dropped immediately following the update. That is, isNew is not taken into consideration by the abstraction function. Let *drop*(p): *3-Struct*$[\mathcal{V}] \to$ *3-Struct*$[\mathcal{V} \setminus \{p\}]$ be the operation that drops the predicate p from the interpretation function of the input structure.

The overall allocation is then given by the composition of the operations defined above:

$$[\![\texttt{x=malloc()}]\!] = alloc \circ [\![x'(v) \leftrightarrow \mathsf{isNew}(v)]\!] \circ drop(\mathsf{isNew}).$$

Example 4.8. We now demonstrate the application of $[\![\texttt{fresh=malloc()}]\!]$ to S_0, which models the allocation statement in Figure 2.2. The reader may ignore the application of *instrument* and the values of the instrumentation predicates, which will be explained when describing the instrumented semantics. The analysis is shown in Figure 4.6.

Modeling Simple Reference Assignment: `x=y` *and* `x=NULL`

$$[\![\texttt{x=y}]\!] \equiv [\![x'(v) \leftrightarrow y(v)]\!]$$
$$[\![\texttt{x=NULL}]\!] \equiv [\![x'(v) \leftrightarrow 0]\!].$$

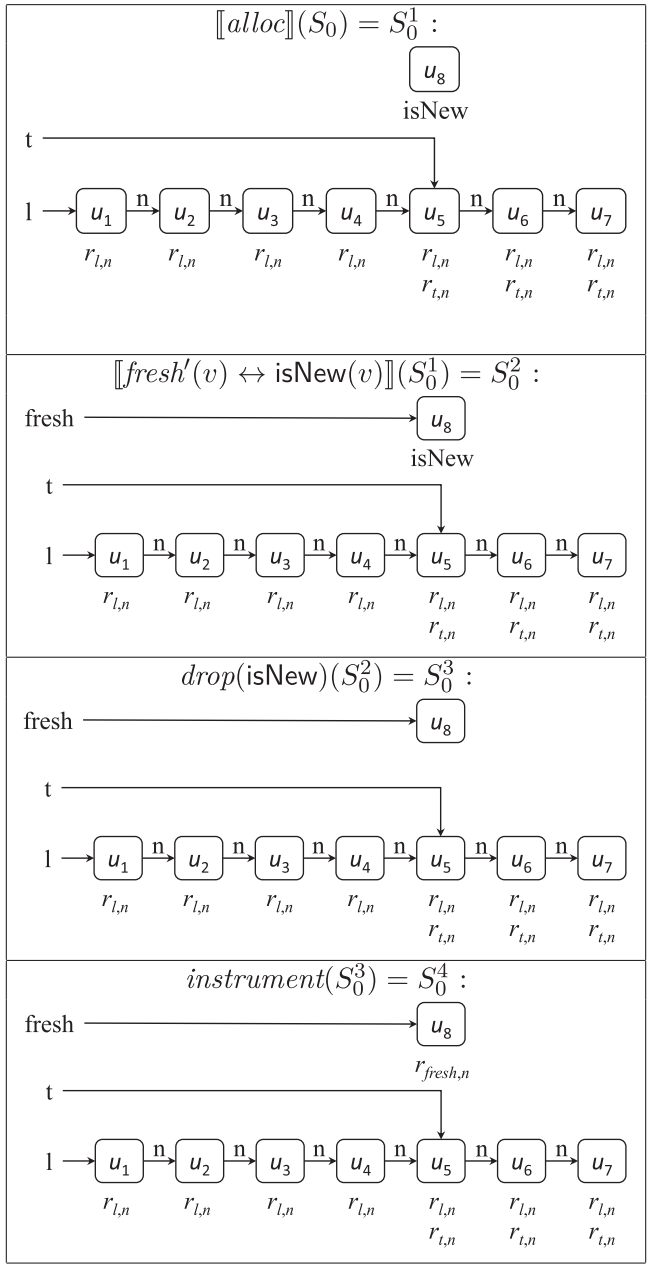


Figure 4.6: Analysis of memory allocation.

Modeling Field Loads: `x=y->f`

The semantics of field loading, which assumes that `y!=NULL` holds, updates the interpretation of `x` to hold for (the single) objects linked by `f` to the object referenced by `y`:

$$[\![\texttt{x=y->f}]\!](S) \equiv [\![x'(v) \leftrightarrow \exists u.\ y(u) \wedge f(u,v)]\!].$$

Modeling Field Stores: `x->f=y` and `x->f=NULL`

The semantics of field storing, which assumes that `x!=NULL` holds, updates the predicate f by first removing any f-links from `x` by conjoining $f(u,v)$ with $\neg x(u)$ and then adding a new link from the object referenced by `x` (given by $x(u)$) to the object referenced by `y` (given by $y(v)$):

$$[\![\texttt{x->f=y}]\!](S) \equiv [\![f'(u,v) \leftrightarrow (f(u,v) \wedge \neg x(u)) \vee (x(u) \wedge y(v))]\!]$$
$$[\![\texttt{x->f=NULL}]\!](S) \equiv [\![f'(u,v) \leftrightarrow (f(u,v) \wedge \neg x(u))]\!].$$

Example 4.9 (Continuing Example 4.8)**.** We demonstrate the execution of the statements `tn=t->n; fresh->n=tn; t->n=fresh` on S_0^4. For simplicity, we do not show the intermediate structures resulting from assertions (these are all successful in this example). We show the results in Figure 4.7.

4.3.5 Vanilla Instrumented Semantics

As shown earlier, instrumentation predicates are essential for improving the precision of the abstraction. The question is, how should the analysis maintain their interpretation across statements? More precisely, given a concrete semantics $[\![st]\!]$: *2-Struct*$[\mathcal{V}^{\text{core}}] \to$ *2-Struct*$[\mathcal{V}^{\text{core}}] \cup \{\bot, \top\}$, for a statement st, we are interested in an instrumented semantics $[\![st]\!]^\natural$: *2-Struct*$[\mathcal{V}^{\text{core}} \uplus \mathcal{V}^{\text{inst}}] \to$ *2-Struct*$[\mathcal{V}^{\text{core}} \uplus \mathcal{V}^{\text{inst}}] \cup \{\bot, \top\}$. The correctness requirement for such a semantics is the following:

$$\forall S \in \textit{2-Struct}[\mathcal{V}^{\text{core}}].\quad [\![st]\!](S) = S' \implies [\![st]\!]^\natural(\textit{instrument}(S)) = \textit{instrument}(S'). \tag{4.2}$$

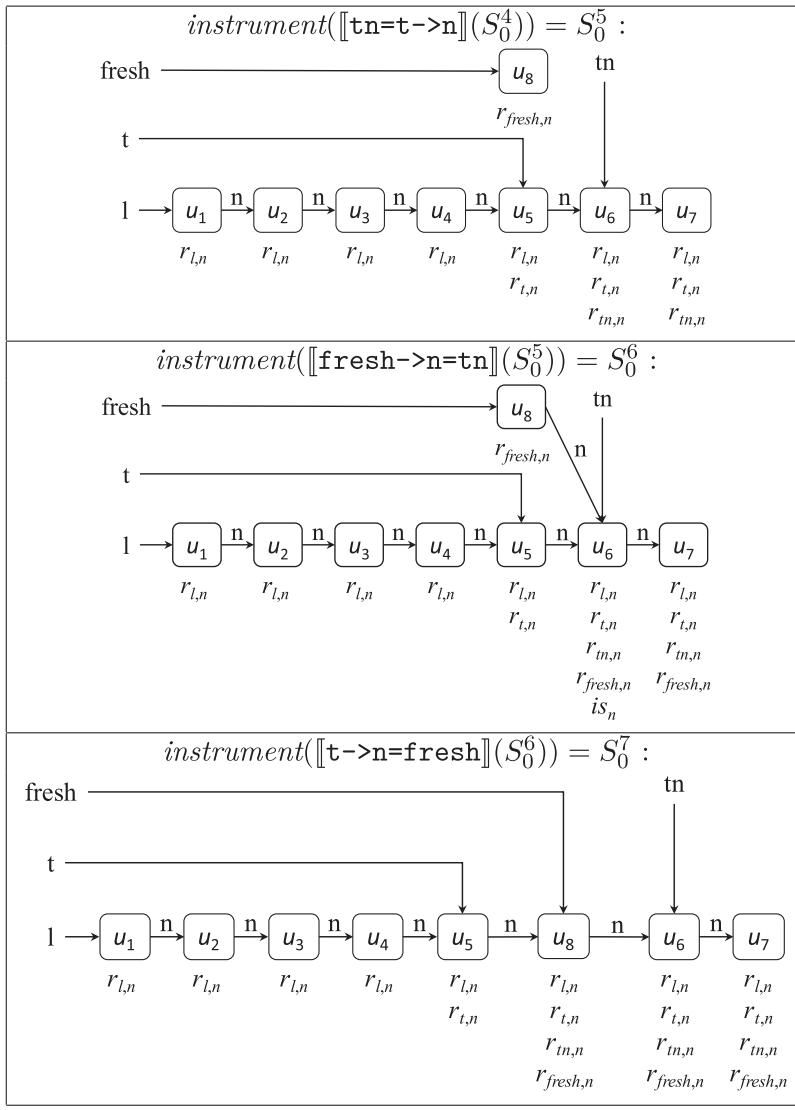


Figure 4.7: Execution of a series of assignment statements.

A straightforward approach is to define $[\![st]\!]^\natural = [\![st]\!] \circ \mathit{instrument}$. This is well defined, since $[\![st]\!](S)$ ignores the instrumentation predicates and *instrument* simply re-evaluates their interpretation from the interpretation of the core predicates resulting from the concrete semantics.

Examples 4.8 and 4.9 show the result of applying instrumentation to each structure.

Unfortunately, as we shall see in a later example, this approach leads to poor precision. The drastic loss of precision is due to the application of (an abstract version of) *instrument* to 3-valued structures.

4.3.6 Vanilla Abstract Transformers

We start by detailing a basic implementation of the abstract semantics of basic statements, which is sound yet often imprecise.

Interpreting FO^{TC} Formulas Over Abstract Structures

We define the meaning of an FO^{TC} formula φ over a 3-valued structure using the evaluation function $[\![\varphi]\!]^\sharp$: $\mathit{3\text{-}Struct}[\mathcal{V}] \to (FV(\varphi) \to \mathit{Node}) \to \{0, 1, \frac{1}{2}\}$, which is defined the same way as $[\![\cdot]\!]$, except that Boolean operations are interpreted over Kleene values as follows:

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

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

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

Theorem 4.2 (Embedding Theorem)**.** Let S and $S^\sharp$ be a 2-valued structure and a 3-valued structure, respectively, such that $S \sqsubseteq^\eta S^\sharp$. Then, for every formula φ where $FV(\varphi) = v_{1..k}$ and tuple $u_{1..k} \in U^k$ and matching tuple $\eta(u_1), \ldots, \eta(u_k) \in U^\sharp$, the following holds:

$$[\![\varphi]\!](S)(u_1, \ldots, u_k) \sqsubseteq [\![\varphi]\!]^\sharp(S^\sharp)(\eta(u_1), \ldots, \eta(u_k)).$$

The embedding theorem allows us to soundly evaluate assertions and conditionals, as we demonstrate in the next example.

Example 4.10. Evaluating the formulas in Example 4.4 on $S_0^\sharp$ yields the following results:

$$[\![\mathit{exists\_leak}]\!](S_0^\sharp) = \tfrac{1}{2}$$
$$[\![\mathit{exists\_cycle}]\!](S_0^\sharp) = \tfrac{1}{2}.$$

This is disappointing, since for every 2-valued structure $S \in \gamma_{precise}(S_0^\sharp)$, we have that

$$[\![exists\_leak]\!](S) = 0$$
$$[\![exists\_cycle]\!](S) = 0.$$

To fix this situation, we re-write the formulas using instrumentation predicates:

$$exists\_leak_{\text{inst}} \equiv \neg\left(\forall v. \bigvee_{z \in \{l,t\}} r_{z,n}(v)\right)$$
$$exists\_cycle_{\text{inst}} \equiv \exists v.\ c_n(v).$$

Evaluating the instrumentation-based formulas yields precise results:

$$[\![exists\_leak_{\text{inst}}]\!](S_0^\sharp) = 0$$
$$[\![exists\_cycle_{\text{inst}}]\!](S_0^\sharp) = 0$$

The embedding theorem allows us to lift predicate updates to 3-valued structures as follows (the extension to sets of updates is straightforward):

$$[\![p'(\overline{v}) \leftrightarrow \varphi(\overline{v})]\!]^\sharp(S^\sharp) \equiv \langle U, \iota[p \mapsto \lambda\overline{o}.\ [\![\varphi]\!]^\sharp(S^\sharp)\mu_{\overline{v},\overline{o}}]\rangle. \tag{4.3}$$

Theorem 4.3 (Soundness of Naive Updates)**.** Let S and $S^\sharp$ be a 2-valued structure and a 3-valued structure, respectively, such that $S \sqsubseteq S^\sharp$. Then, for every set of predicate updates $\{p_i'(\overline{v_i}) \leftrightarrow \varphi_i(\overline{v_i})\}_{i=1}^k$, the following holds:

$$[\![\{p_i'(\overline{v_i}) \leftrightarrow \varphi_i(\overline{v_i})\}_{i=1}^k]\!](S) \sqsubseteq [\![\{p_i'(\overline{v_i}) \leftrightarrow \varphi_i(\overline{v_i})\}_{i=1}^k]\!]^\sharp(S^\sharp).$$

We define the vanilla abstract semantics as follows:

$$[\![\texttt{assert}\ \varphi]\!]^\sharp(S) \equiv \begin{cases} S, & \text{if } [\![\hat{\varphi}]\!]^\sharp(S) = 1; \\ \top, & \text{otherwise.} \end{cases}$$

$$[\![\varphi]\!]^\sharp(S) \equiv \begin{cases} S, & \text{if } [\![\hat{\varphi}]\!]^\sharp \neq 0; \\ \bot, & \text{otherwise.} \end{cases}$$

$$[\![\texttt{x=malloc()}]\!]^\sharp \equiv \mathsf{blur} \circ \mathit{instrument} \circ \mathit{drop}(\mathsf{isNew}) \circ [\![\mathit{fresh}'(v) \leftrightarrow \mathsf{isNew}(v)]\!]^\sharp \circ \mathit{alloc}$$

$$\llbracket \texttt{x=NULL} \rrbracket^\sharp \equiv \mathsf{blur} \circ \mathit{instrument} \circ \llbracket x'(v) \leftrightarrow 0 \rrbracket^\sharp$$

$$\llbracket \texttt{x=y} \rrbracket^\sharp \equiv \mathsf{blur} \circ \mathit{instrument} \circ \llbracket x'(v) \leftrightarrow y(v) \rrbracket^\sharp$$

$$\llbracket \texttt{x=y->f} \rrbracket^\sharp \equiv \mathsf{blur} \circ \mathit{instrument} \circ \llbracket x'(v) \leftrightarrow \exists u.\ y(u) \wedge f(u,v) \rrbracket^\sharp$$

$$\llbracket \texttt{x->f=NULL} \rrbracket^\sharp \equiv \mathsf{blur} \circ \mathit{instrument} \circ \llbracket f'(u,v) \leftrightarrow (f(u,v) \wedge \neg x(u)) \rrbracket^\sharp$$

$$\llbracket \texttt{x->f=y} \rrbracket^\sharp \equiv \mathsf{blur} \circ \mathit{instrument} \circ \llbracket f'(u,v) \leftrightarrow (f(u,v) \wedge \neg x(u)) \vee (x(u) \wedge y(v)) \rrbracket^\sharp$$

The semantics is similar to the concrete semantics, except for the following differences: (i) the core predicate updates are carried out directly over 3-valued structures as explained above; (ii) instrumentation predicates are updated using *instrument*, which re-evaluates their interpretations once the values of the core predicates have been updated; (iii) blur ensures that the returned structures are bounded; and (iv) the transformers for conditionals consider the condition to hold if the formula evaluation returns a value different from 0, meaning either 1 or $\frac{1}{2}$.[10] Notice that the semantics of conditionals and assertions does not apply *instrument* nor blur. This is an optimization, as the input structure is assumed to be bounded and the statement does not modify the interpretations of any predicates.

The following example makes it clear why we refer to the semantics above as naive.

Example 4.11. Applying the abstract semantics of the statements `fresh=malloc(); tn=fresh->n; fresh->n=tn; t->n=fresh` to $S_0^\sharp$ yields the structures shown in Figure 4.8 (again, dropping the intermediate structures due to successful assertions).

The resulting structures are quite imprecise, as can be seen by the $\frac{1}{2}$ values assigned to the instrumentation predicates. Specifically, we cannot determine whether they represent heaps containing garbage cells, whether the list is cyclic, and whether there is sharing.

[10] A more precise semantics would abstractly conjoin the information from the formula expression with the input structure.

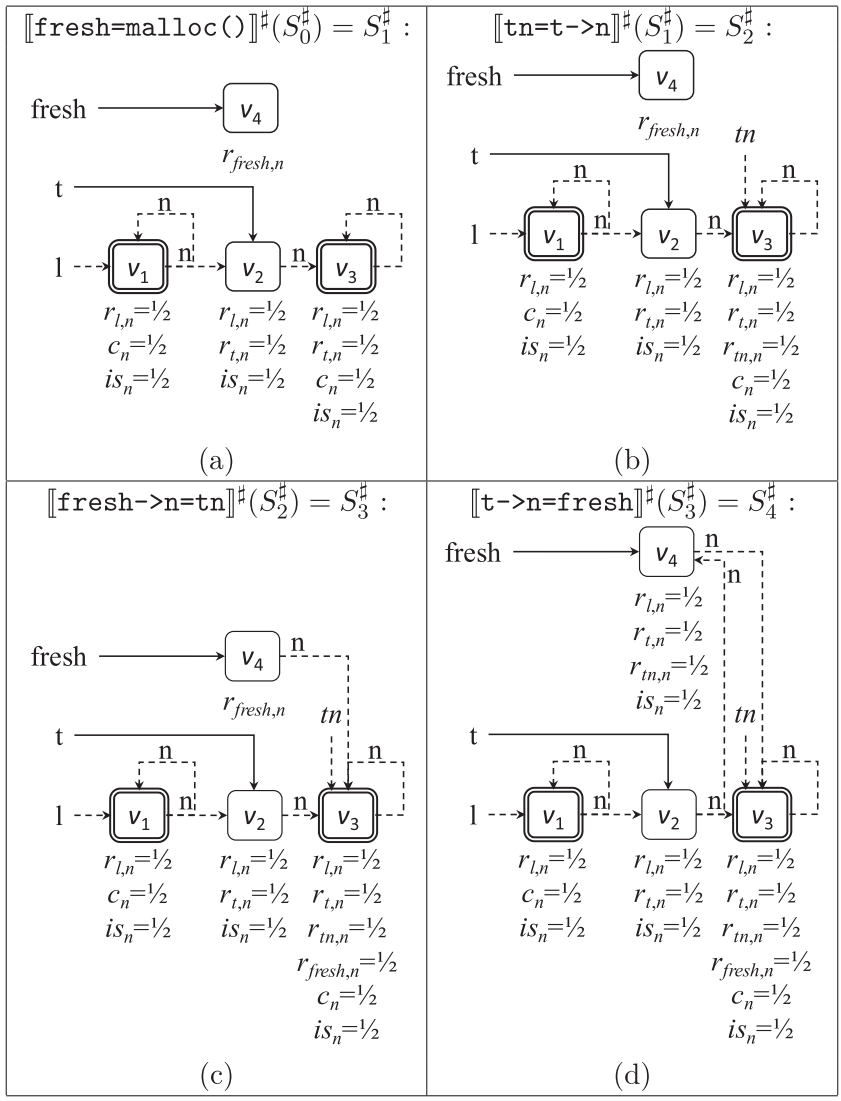


Figure 4.8: Structures arising from the application of naive abstract transformers.

4.3.7 Improving the Precision of Abstract Transformers

We now describe three techniques for increasing the precision of abstract transformers: (i) incrementally updating instrumentation predicates,

(ii) applying the *focus* partial concretization operation, and (iii) applying the *coerce* operation to sharpen structures.

Incremental Instrumented Semantics

Recall our previous comment about the vanilla instrumented semantics and consider $S_1^\sharp$. Notice that the concrete semantics of `fresh=malloc()` affects neither of the predicates n, l, t, on which the instrumentation predicates $is_n, r_{l,n}, r_{t,n}, c_n$ depend. Therefore, we would expect that their interpretation would remain the same as in $S_0^\sharp$. However, since we update the instrumentation predicates by re-evaluating them directly over (the core predicate interpretations of) the 3-valued structure $S_0^\sharp$, we obtain sound yet imprecise results.

The solution we take is to update the instrumentation predicates directly by supplying update formulas. Intuitively, if the update formulas re-use the values of the predicates in the input structure, avoiding quantification and transitive closure, their interpretation should be at least as precise. With the added predicate updates, we can use (4.3) to update all predicates and drop the use of *instrument*.

We now define incremental update formulas for instrumentation predicates, which in our example are $\{is_n, r_{l,n}, r_{t,n}, r_{fresh,n}, c_n\}$.

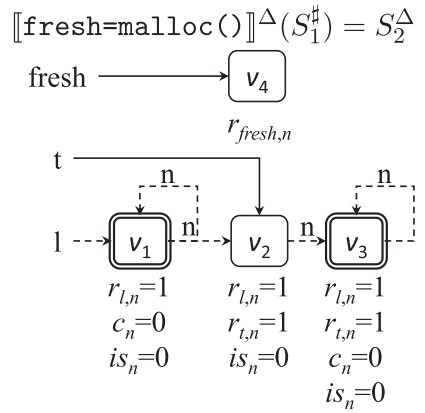
For memory allocation statement, we employ the following update formulas:

$$\begin{aligned}\texttt{x=malloc()}:&\{r'_{z,f}(v) \leftrightarrow r_{z,f}(v) \mid z \not\equiv x \in \mathbb{X}, f \in \mathbb{F}\} \cup \\ &\{r'_{x,f}(v) \leftrightarrow \mathsf{isNew}(v) \mid f \in \mathbb{F}\} \cup \\ &\{is'_f(v) \leftrightarrow is_f(v) \wedge \neg\mathsf{isNew}(v) \mid f \in \mathbb{F}\} \cup \\ &\{c'_f(v) \leftrightarrow c_f(v) \wedge \neg\mathsf{isNew}(v) \mid f \in \mathbb{F}\}.\end{aligned}$$

(The syntactic predicate $z \not\equiv x$ holds if `z` and `x` are different variables.)

With the updates above, applying abstract predicate updates to $S_0^\sharp$ yields the following structure (we explicitly depict some of the

0-valued interpretations, to contrast with $S_1^\sharp$ from the previous example):



While *instrument* is guaranteed to satisfy (4.2), the update formulas to instrumentation predicates may not. Reps *et al.* (2003) developed an algorithm based on finite differencing of logical formulas to automatically derive update formulas for instrumentation predicates. The updates are guaranteed to be precise under reasonable conditions (see the paper for details).

We continue by providing update formulas specific for singly-linked lists. That is, we assume the existence of a single field `n`.

`x=NULL:`	$\{r'_{z,n}(v) \leftrightarrow r_{z,n}(v) \mid z \not\equiv x \in \mathbb{X}\} \cup$ $\{r'_{x,n}(v) \leftrightarrow 0\} \cup$ $\{is'_n(v) \leftrightarrow is_n(v)\} \cup$ $\{c'_n(v) \leftrightarrow c_n(v)\}$
`x=y:`	$\{r'_{z,n}(v) \leftrightarrow r_{z,n}(v) \mid z \not\equiv x \in \mathbb{X}\} \cup$ $\{r'_{x,n}(v) \leftrightarrow r_{y,n}(v)\} \cup$ $\{is'_n(v) \leftrightarrow is_n(v)\} \cup$ $\{c'_n(v) \leftrightarrow c_n(v)\}$
`x=y->n:`	$\{r'_{z,n}(v) \leftrightarrow r_{z,n}(v) \mid z \not\equiv x \in \mathbb{X}\} \cup$ $\{r'_{x,n}(v) \leftrightarrow r_{y,n}(v) \wedge (c_n(v) \vee \neg y(v))\} \cup$ $\{is'_n(v) \leftrightarrow is_n(v)\} \cup$ $\{c'_n(v) \leftrightarrow c_n(v)\}$

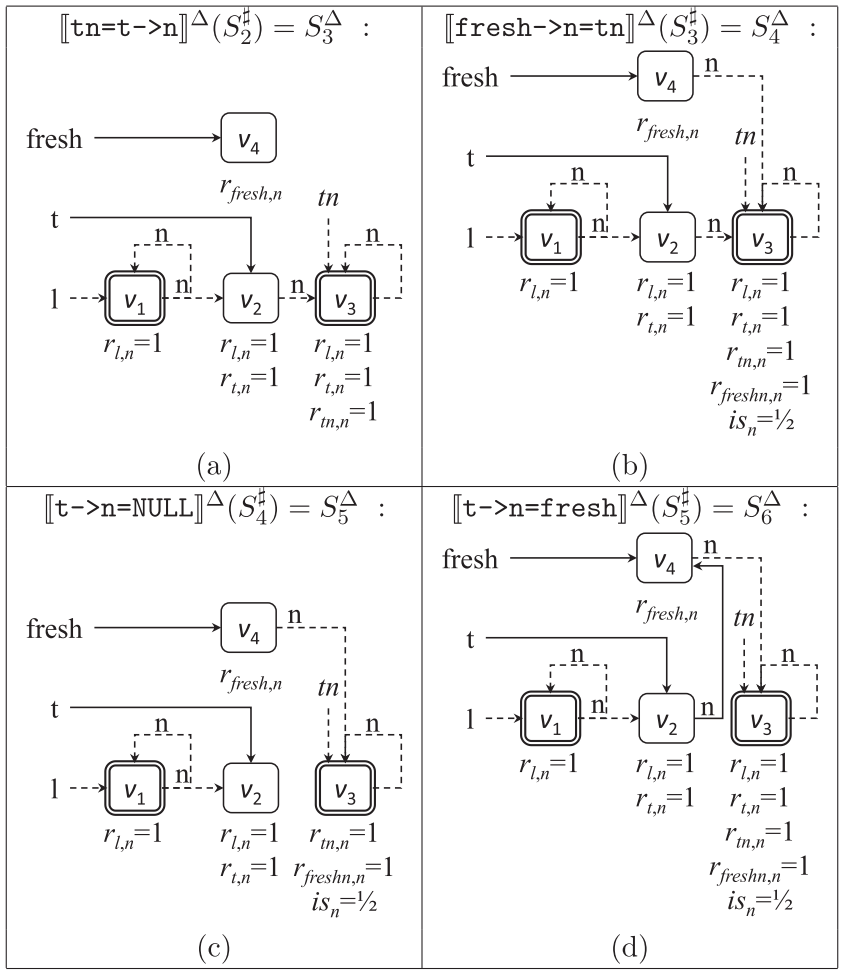


Figure 4.9: Structures arising from the application of abstract transformers employing incremental updates to instrumentation predicates.

With the update formulas above, we obtain the structure shown in Figure 4.9(b) for `tn=t->n`.

Updating reachability for field updates is more complicated. Intuitively, the "shape" of the heap is being mutated, which affects the set of n-paths in a non-trivial way, and in turn affects any predicate defined

via transitive closure (c_n and the $r_{z,n}$ family of predicates). We therefore use the simplifying transformation `x->n=y` $\rightsquigarrow$ `x->n=NULL; x->n=y`. This has the effect that the field updates either only remove a single n-link—via `x->n=NULL`—or add a single n-link—via `x->n=y` (since `x->n=NULL` was just executed and therefore `x->n==NULL` holds).

The updates formulas are then as follows:

`x->n=NULL`:

$$\{r'_{z,n}(v) \leftrightarrow c_n(v) \wedge r_{x,n}(v) \ ?\ \varphi_{r_{x,n}}(v) :$$
$$\quad r_{z,n}(v) \wedge \neg(\exists v'.\ r_{z,n}(v') \wedge x(v') \wedge r_{x,n}(v) \wedge \neg x(v)) \mid z \not\equiv x \in \mathbb{X}\} \cup$$
$$\{r'_{x,n}(v) \leftrightarrow x(v)\} \cup$$
$$\{is'_n(v) \leftrightarrow \exists v'.\ x(v') \wedge n(v', v)\ ?\ \varphi_{is_n}(v) : is_n(v)\} \cup$$
$$\{c'_n(v) \leftrightarrow c_n(v) \wedge \neg(\exists v'.\ x(v)' \wedge c_n(v') \wedge r_{x,n}(v))\}$$

`x->n=y`: (assuming `x==NULL` holds)

$$\{r'_{z,n}(v) \leftrightarrow r_{z,n}(v) \vee \exists v'.\ r_{z,n}(v') \wedge x(v') \wedge r_{y,n}(v) \mid z \in \mathbb{X}\} \cup$$
$$\{is'_n(v) \leftrightarrow \exists v'.\ y(v) \wedge n(v', v)\ ?\ \varphi_{is_n}(v) : is_n(v)\} \cup$$
$$\{c'_n(v) \leftrightarrow c_n(v) \vee \exists v'.\ x(v') \wedge r_{y,n}(v') \wedge r_{y,n}(v)\}$$

With the update formulas above, we obtain the structures shown in Figures 4.9(c) and (d), for `t->n=NULL` and `t->n=fresh`, respectively. Notice that these structures are more precise than the ones in Figure 4.8. Specifically, all reachability predicates are precise, which allows us to prove that no list cells have been lost. The same holds for the cyclicity predicate, which lets us prove that the resulting list is acyclic. However, the sharing predicate is imprecise for v_3 of S_6^Δ. This is because at S_4^Δ, the individual linked to v_4 is indeed shared, while the succeeding ones are not. Resolving this imprecision requires separating the summary individual in v_3 into two individuals—the "head" (the one linked to v_4) and the "tail" (the succeeding ones). We continue by defining the abstract operations to achieve this.

Bringing Structures into Focus

One way to improve the precision of an abstract transformer is by refining the input structure into a set of more precise structures. This

is done via the operation

$$\mathsf{focus}\colon (3\text{-}\mathit{Struct}[\mathcal{V}] \times \mathsf{FO}^{\mathsf{TC}}) \to \mathcal{P}(3\text{-}\mathit{Struct}[\mathcal{V}]) \cup \{\top\}.$$

focus uses the defining formula of an instrumentation predicate p, φ_p, to convert a 3-valued structure to a set of structures where the evaluation of φ_p is always definite (that is, for every assignment to its free variables):

$$\forall p^{(k)}.\ \langle U', \iota' \rangle \in \mathsf{focus}(\varphi_p)(S) \implies \forall \overline{u} \in U'^k.\ \iota'(\varphi_p)(\overline{u}) \in \{0, 1\}. \quad (4.4)$$

After the refinement, the (incremental) abstract transformer is applied to each of the resulting structures.

The focus operation either succeeds or returns $\top$ to indicate failure. If the operation succeeds, it preserves the meaning of the input structure, which is sometimes referred to as *semantic reduction*:

$$\gamma_{\mathit{precise}}(\mathsf{focus}(\varphi_p)(S)) = \gamma_{\mathit{precise}}(S). \quad (4.5)$$

A failure by focus indicates a set of structures satisfying both (4.4) and (4.5) may be infinite.

A detailed description of the focus operation appears in Sagiv *et al.* (2002) and a full algorithm appears in Lev-Ami and Sagiv (2000). The algorithm in Lev-Ami and Sagiv (2000) can statically detect a useful subset of the situations where focus may fail and detects the rest of the failing situations at runtime.

Coming up with a focus predicate is somewhat of an art, but usually they are derived from sub-formulas of the update formulas. Also, the focus predicate is usually chosen such that its free variables can be bound to at most a constant number of individuals in any structure.

Two useful focus predicates correspond to a variable dereference and a field dereference:

$$\begin{aligned} \mathit{VarDeref}_z(v) &\equiv z(w) \\ \mathit{FieldDeref}_{z,n}(v) &\equiv \exists w.\ z(w) \wedge n(w, v). \end{aligned}$$

A common practice, which we follow in our example, is to include the following focus predicates for the following types of statements:

Statement	Focus Predicates
`x=malloc()`	$\emptyset$
`x=NULL`	$\{VarDeref_x(v)\}$
`x=y`	$\{VarDeref_y(v)\}$
`x=y->f`	$\{FieldDeref_{y,f}(v)\}$
`x->f=NULL`	$\{VarDeref_x(f)\}$
`x->f=y`	$\emptyset$

In our example, the abstract transformer for `tn=t->n` resulted in `tn` pointing to a summary node. To separate the first individual represented by the tail of the list, we use the focus predicate $FieldDeref_{\mathtt{t},n}(v)$. Notice that this matches the right-hand side of the formula used to update `tn` and that, since $t(v)$ may hold for at most one individual so does $FieldDeref_{\mathtt{t},n}(v)$.

Example 4.12. The result of $\mathsf{focus}(FieldDeref_{\mathtt{t},n}(v))(S_2^\Delta)$ is shown in Figure 4.10.

Intuitively, the focus algorithm operates as follows: The formula $FieldDeref_{\mathtt{t},n}(v)$ is evaluated on every individual of S_2^Δ and it is determined that its interpretation is $\frac{1}{2}$ only for v_3. The algorithm then checks whether v_3 is a summary node, which happens to be the case in our example. Therefore, the algorithm creates three versions of S_2^Δ:

$S_{2.1}^\Delta$: A structure where $FieldDeref_{\mathtt{t},n}(v)$ must not hold for all concrete individuals represented by v_3. To achieve this, the algorithm checks the interpretation of the formula $\exists w.\ z(w) \wedge n(w,v)$ where v is bound to v_3. This is done by binding w to each individual, which yields $\frac{1}{2}$ only in the case where w is bound to v_2. Therefore, to falsify the formula, the algorithm sets $\iota^{S_{2.1}^\Delta}(n)(v_2,v_3)$ to 0.

$S_{2.2}^\Delta$: A structure where $FieldDeref_{\mathtt{t},n}(v)$ must hold for all concrete individuals represented by v_3. This is achieved in a similar way to the process for $S_{2.1}^\Delta$, except that $\iota^{S_{2.2}^\Delta}(n)(v_2,v_3)$ is set to 1.

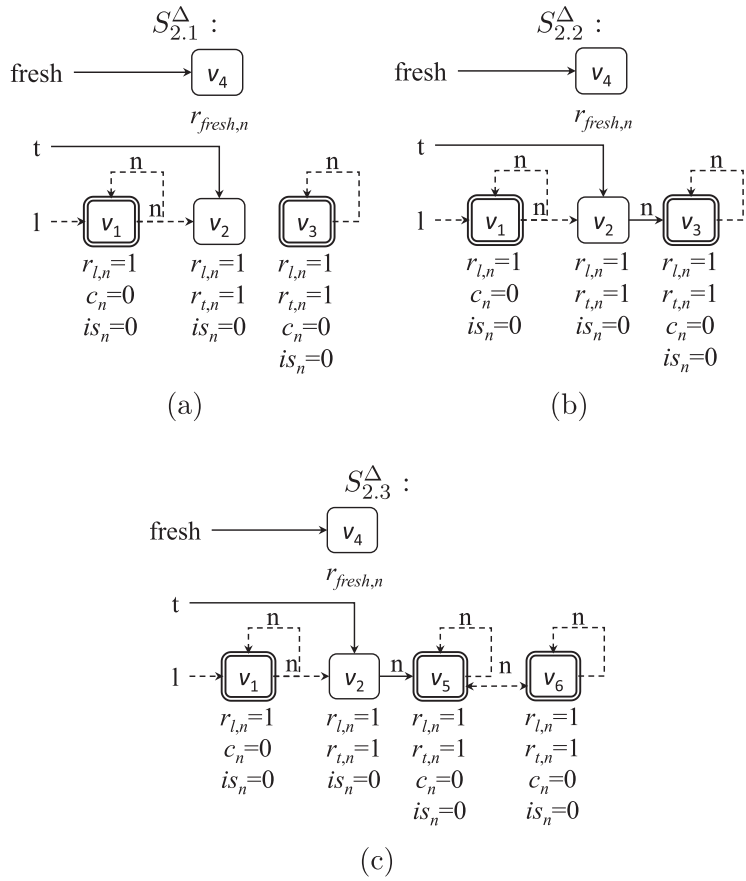


Figure 4.10: The structures resulting from $\mathsf{focus}(\mathit{FieldDeref}_{t,n}(v))(S_2^{\Delta})$.

$\boldsymbol{S^{\Delta}_{2.3}}$**:** A version where the predicate $\mathit{FieldDeref}_{\mathtt{t},n}(v)$ definitely holds for a subset of the concrete individuals represented by v_3 and definitely does not hold for another subset. To achieve the property stated above for $S^{\Delta}_{2.3}$, v_3 is bifurcated into v_5 (representing the subset of the individuals for which the predicate definitely holds) and v_6 (representing the subset of the individuals for which the predicate definitely does not hold). Finally, $\iota^{S^{\Delta}_{2.3}}(n)(v_2, v_5)$ is set to 1 and $\iota^{S^{\Delta}_{2.3}}(n)(v_2, v_6)$ is set to 0.

Here are some observations about these results and causes of imprecision: (1) $S^{\Delta}_{2.1}$ does not represent any concrete structure as v_3 is

unreachable from `l` and `t`, while $r_{n,l}$ $r_{n,t}$ indicate that it should be as both predicates evaluate to 1. (2) Intuitively, $S^{\Delta}_{2.2}$ should represent the case where the tail of the list, represented by v_3 contains exactly one element. This is reflected by the definite n-edge on (v_2, v_3), but not by the fact that v_3 is a summary individuals. (3) Intuitively, $S^{\Delta}_{2.3}$ should represent the case where the tail of the list contains one or more element where v_5 is its first element. However, v_5 is a summary individual and there is an indefinite n edge (v_6, v_5). These sources of imprecision can be corrected by *sharpening*, as we explain next.

Sharpening Structures via Coercing

To further improve precision, 3VSA employs the operation coerce_{cons}: *3-Struct*$[\mathcal{V}] \to$ *3-Struct*$[\mathcal{V}] \cup \{\bot\}$, which is parameterized by the set of integrity constraints *cons*.

The coerce operation accepts a 3-valued structure and returns either another, more precise, 3-valued structure or $\bot$, which indicates that the input structure is inconsistent with the integrity constraints. More specifically, coerce applies a forward inference algorithm in order to turn indefinite predicate values to definite predicate values. To achieve this, integrity constraints must be expressed in a specific form.

The Form of Integrity Constraints. To be useful in the context of a coerce operation, an integrity constraint must have the form $\varphi_b(\overline{v}) \Longrightarrow \varphi_h(\overline{v})$ where the following conditions hold: (i) $\varphi_b(\overline{v}), \varphi_h(\overline{v}) \in \mathsf{FO}^{\mathsf{TC}}$, (ii) $FV(\varphi_b(\overline{v})) = FV(\varphi_h(\overline{v})) = (\overline{v})$, and (iii) $\varphi_h(\overline{v})$ has the form $p(\overline{v})$ or $\neg p(\overline{v})$ (where $p \in \mathcal{V}$). We refer to the left-hand side of an integrity constraint as the *body* and to the right-hand side of the constraint as the *head*.

Similar to focus, coerce is also a *semantic reduction* and satisfies the following properties:

$$\mathsf{coerce}_{cons}(S) \sqsubseteq S. \tag{4.6}$$

$$\gamma_{precise}(\mathsf{coerce}_{cons}(S)) = \gamma_{precise}(S). \tag{4.7}$$

To understand how coerce achieves this, consider an integrity constraint $\varphi_b(\overline{v}) \Longrightarrow p(\overline{v})$ where $\overline{v} = k$ (the details are symmetric for $\neg p(\overline{v})$), a 3-valued structure $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$, and a node tuple $\overline{o} \in {U^\sharp}^k$.

Situation	**Action**
$[\![\varphi_b(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = 1$ and $[\![p(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = \frac{1}{2}$	$\iota^\sharp[p \mapsto \iota^\sharp(p)[\overline{o} \mapsto 1]]$
$[\![\varphi_b(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = 0$ and $[\![p(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = \frac{1}{2}$	$\iota^\sharp[p \mapsto \iota^\sharp(p)[\overline{o} \mapsto 0]]$
$[\![\varphi_b(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = 1$ and $[\![p(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = 0$	return $\bot$
$[\![\varphi_b(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = 0$ and $[\![p(\overline{v})]\!]^\sharp(S^\sharp)(\mu_{\overline{v},\overline{o}}) = 1$	return $\bot$

Figure 4.11 shows the effect of coerce in our example and its synergistic interaction with focus. Specifically, notice how $S^\Delta_{2.1}$ is filtered out, how the summary individual v_3 in $S^\Delta_{2.2}$ and v_5 in $S^\Delta_{2.3}$ are made into non-summary individuals, and how the indefinite edges around these individuals are removed.

4.3.8 Summary

In this subsection, we reviewed the key elements of the 3VSA approach for shape analysis, focusing on the way logical structures are used to represent and abstract concrete heaps and on the key reason for the precision of the analysis: explicitly recording key information using

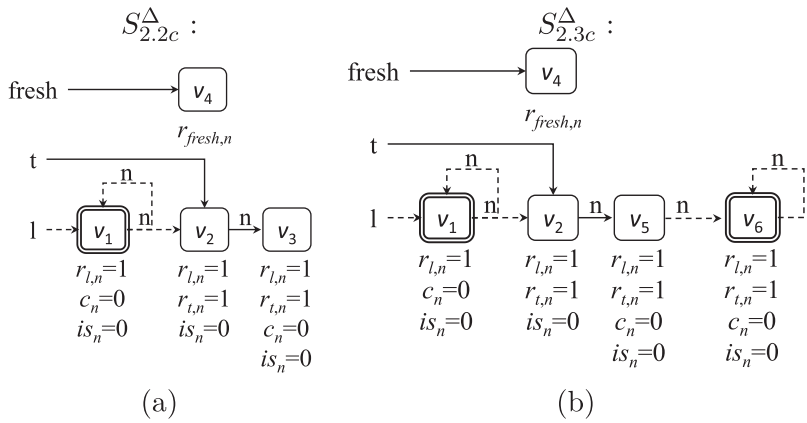
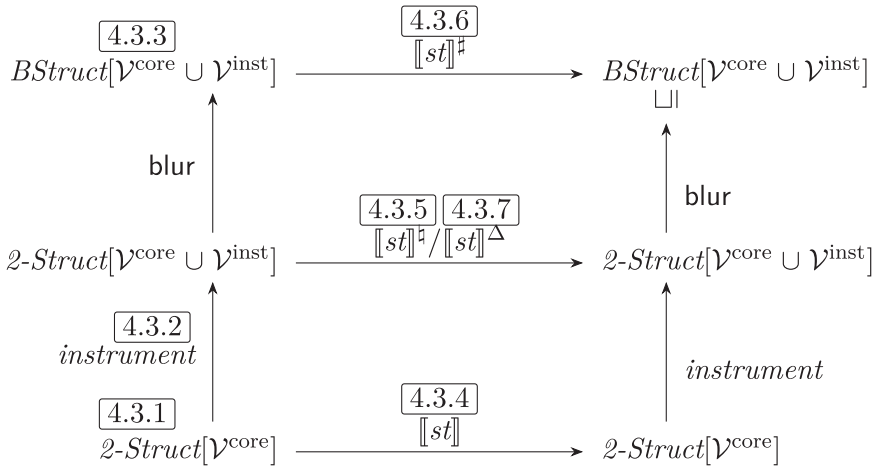


Figure 4.11: The structures resulting from coerce(focus($FieldDeref_{t,n}(v)$)(S^Δ_2)).

instrumentation predicates. The salient semantic elements and the relationships between them are illustrated in the following diagram:



TVLA. The TVLA tool (Lev-Ami and Sagiv, 2000) implements the 3VSA approach discussed here, along with other extensions. TVLA was used to generate different shape analysis and applied to solve different program analysis problems (Dor *et al.*, 2005). More recently, TVLA was extended with automatic termination reasoning (Manevich *et al.*, 2016) (automatic in the sense of not requiring any hints from the developers or users), allowing it to prove total correctness for a range of programs such as the running example used throughout this subsection. Moreover, Loginov *et al.* (2005) proposed a framework to compute the predicates to be used in order to describe structures using inductive learning.

Applications of Reachability Predicates. Reachability predicates have been used not only in shape analysis but also in other program verification contexts. Indeed, the notion of reachability is natural to capture information about the layout of linked structures of unbounded length. For instance, reachability has also been used in approaches based on predicate abstraction (Balaban *et al.*, 2007; Podelski and Wies, 2005). Such works typically lack the dynamic materialization used in shape analysis. Several works proposed solvers for logics that involve predicates similar to points-to predicates and reachability predicates (Itzhaky

et al., 2013; Nelson, 1983). Finally, Madhusudan *et al.* (2011) proposed a decidable logics to reason over families of inductive data-structures and to express constraints over data.

4.4 Separation Logic-Based Shape Abstraction

In this subsection, we focus on shape analyses that are based on separation logic and either manipulate logical formulas abstract syntax trees, or other data structures that actually encode some family of separation logic formulas. We first recall the main definitions of separation logic in Subsection 4.4.1, then we set up a shape abstraction based on in Subsection 4.4.2, and then we show the construction of a shape analysis based on these principles.

4.4.1 Separation Logic

In general, reasoning over programs that perform complex pointer operations over sophisticated data structures is very difficult, for many reasons. One of these reasons is that a pointer update deeply alters the shape of data structures. Furthermore, it is often hard to resolve precisely which concrete memory cell gets read or modified, especially when looking at abstract logical predicates describing complex memory states.

Let us consider the case of an update to a cell designated by a complex pointer expression. When the memory cell that is modified cannot be resolved precisely at all, a sound static analysis tool must assume that *any* cell could be modified, which amounts to losing all information, about all structures stored in memory. By contrast, when the modified cell can be localized, the analysis will easily conclude that many cells are not modified, so that the data structures that are stored in those cells are not impacted by the update.

Separation Logic: The Essence of Local Reasoning. This observation that reasoning over memory updates is easier when the modified cells can be determined precisely is at the foundation of the *logics of bunched implications* (Ishtiaq and O'Hearn, 2001; O'Hearn and Pym,

1999) and *separation logic* (Reynolds, 2002). A separation logic formula describes a set of memory states, using a set of basic logical predicates and connectors, some of which are specific to separation logic. The definition below presents a basic fragment of separation logic, that is adapted to present its main principles and characteristics.

Definition 4.7 (Separation Logic Formulas)**.** We let *address expressions* (or *heap location expressions*) be defined by:

$$\begin{array}{rcll} l & ::= & \mathtt{x} & \text{where } \mathtt{x} \in \mathbb{X} \text{ is a program variable} \\ & | & a \cdot \mathtt{f} & \text{where } a \text{ is an address and } \mathtt{f} \text{ a field} \end{array}$$

We let *separation logic formulas* (or *heap expressions*) be defined by:

$$\begin{array}{rcll} h & ::= & \mathbf{emp} & \text{(empty store)} \\ & | & l \mapsto a & \text{(atomic memory cell, where } a \text{ is an address)} \\ & | & h * h & \text{(separating conjunction)} \\ & | & h \wedge h & \text{(non separating, or classical conjunction)} \\ & | & \ldots & \text{(other constructions, to be defined)} \end{array}$$

The denotation of a separation logic formula is defined by a relation $\vdash$ over pairs made of a memory state and a formula. In the following, we write $[\![l_0 \mapsto a_0, \ldots, l_n \mapsto a_n]\!]$ for the partial function that maps location l_i into a_i (so that $[\![\,]\!]$ is the partial function with empty domain), and we also write $m_0 \uplus m_1$ for the partial function obtained by joining two partial functions m_0, m_1 with disjoint domains. Then, the definition of $\vdash$ proceeds as follows:

$$\frac{}{[\![\,]\!] \vdash \mathbf{emp}} \qquad \frac{}{[\![l \mapsto a]\!] \vdash l \mapsto a}$$

$$\frac{m_0 \vdash h_0 \qquad m_1 \vdash h_1}{m_0 \uplus m_1 \vdash h_0 * h_1} \qquad \frac{m \vdash h_0 \qquad m \vdash h_1}{m \vdash h_0 \wedge h_1}$$

As an example, the formula $\mathtt{x} \mapsto a * a \mapsto 7$ describes memory states where a single variable $\mathtt{x}$ is defined and stores a pointer to another cell that contains integer value 7. The formula $a \mapsto a * a \mapsto 3$ does not describe any memory state, as it is not possible to build two memory partial functions that are both defined over a and that have disjoint supports. The formula $a \mapsto b * b \mapsto a$ describes memory states that comprise exactly two cells, storing pointers to each other. On the other

hand, the formula $a \mapsto b \wedge b \mapsto a$ describes only memory states made of a single cell, that stores a pointer to itself: indeed, the classical, non separating conjunction operator asserts that two sub-formulas describe the same memory state, namely the same range of addresses, and corresponding values, so that, in this case, it implies that a and b are equal. This example illustrates the fundamental difference between $*$ and $\wedge$: the former divides memory into disjoint blocks whereas the second provides no separation.

We remark that separating conjunction is commutative and associative, in the sense that changing the order of terms does not modify the meaning of formulas. Therefore, we do not distinguish separation logic formulas that are derived from each other by rewriting based on commutativity and associativity of $*$. Moreover, for the sake of clarity, we abbreviate $\mathbb{c}_0 \cdot \underline{\emptyset} \mapsto \mathbb{c}_1$ into $\mathbb{c}_0 \mapsto \mathbb{c}_1$.

Local Reasoning. One great advantage of separation logic is that it enables local reasoning, which means that many program statements can be reasoned about while considering only a small part of the memory states, namely only the part that they read or update. More precisely, it supports the *Frame rule*, which expresses that an observation which can be made on an execution with a local memory context can still be made when adding a global memory context. Based on the semantics introduced in Subsection 3.1, it writes down as follows:

if $\mathtt{s}$ is such that
$\forall m_0, m_1,\ m_0 \vdash h_0 \wedge m_1 \in [\![\mathtt{s}]\!](m_0) \Longrightarrow m_1 \vdash h_1$
if $m \vdash h$,
and $\mathtt{s}$ does not write in any location that is a free variable in h
then, we also have
$\forall m_0, m_1,\ m_0 \vdash h_0 * h \wedge m_1 \in [\![\mathtt{s}]\!](m_0) \Longrightarrow m_1 \vdash h_1 * h.$

This property is equivalent to the more common presentation of the Frame rule, based on Hoare triples:

$$\frac{\{h_0\}\mathtt{s}\{h_1\} \qquad \mathbf{write}(\mathtt{s}) \cap \mathbf{freevar}(h) = \emptyset}{\{h_0 * h\}\mathtt{s}\{h_1 * h\}}$$

where $\{h_0\}\mathtt{s}\{h_1\}$ means that whenever $\mathtt{s}$ starts in a state that satisfies h_0 and terminates, it does so in a final state that satisfies h_1, $\mathbf{write}(\mathtt{s})$ denotes the set of memory locations that $\mathtt{s}$ may write to, and $\mathbf{freevar}(h)$ denotes the free variables in the formula h.

As an example, let us consider the formula $h = \mathtt{x} \mapsto a * \mathtt{y} \mapsto b * \mathtt{z} \mapsto c$ and the assignment $\mathtt{x} = \mathtt{y}$. Then, variable $\mathtt{z}$ plays no role at all. Considering only $\mathtt{x}$ and $\mathtt{y}$, we can observe the triple below is clearly satisfied:

$$\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b\}\mathtt{x} = \mathtt{y}\{\mathtt{x} \mapsto b * \mathtt{y} \mapsto b\}.$$

By the frame rule, we derive that:

$$\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b * \mathtt{z} \mapsto c\}\mathtt{x} = \mathtt{y}\{\mathtt{x} \mapsto b * \mathtt{y} \mapsto b * \mathtt{z} \mapsto c\}.$$

This example illustrates the principle of local reasoning and would generalize to the case where a part of the separation logic formula that is assumed as a pre-condition is a symbolic formula h':

$$\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b * h'\}\mathtt{x} = \mathtt{y}\{\mathtt{x} \mapsto b * \mathtt{y} \mapsto b * h'\}.$$

This form of local reasoning is one of the most important foundations of the family of shape analyses that we describe here, since the abstract operations (Subsection 4.4.3) extensively rely on it.

4.4.2 Abstractions Based on Separation Logic

We now construct a shape abstraction based on separation logic.

Abstract Memory States and Their Concretization. The logical fragment shown in Definition 4.7 does not feature any support for summarization, thus we now add summary predicates based on separation logic. The definition of these predicates follows the usual form of inductive predicates, except that they are based on separation logic connectors introduced in Definition 4.7, which means that they use separating conjunction, **emp** and points-to memory predicates. As summary predicates describe memory regions of variable size, therefore, we also need to account for a variable set of addresses, which is the reason why we need to introduce a notion of abstract address. These abstract address describe

the addresses used in address expressions (Definition 4.7). The refinement of summary predicates often results in case splits among abstract facts, thus we also include disjunctive abstract shapes. Moreover, the non-separating conjunction of heap expressions that was mentioned in Definition 4.7 is not supported by most shape analyses (or only in a very local and specialized way) so we omit it in the shape abstract domain definition. The following definition accounts for these changes:

Definition 4.8 (Abstract Shapes Based on Separation Logic)**.** We assume a set $\mathbb{S}$ of *symbolic addresses* (which are noted $\mathbb{a}, \mathbb{a}_0, \mathbb{a}_1, \ldots$). The $\mathbb{D}$ set of *abstract shapes* (noted $\mathbb{d}$), the set $\mathbb{H}$ of *abstract heaps* (noted $\mathbb{h}$), the *inductive separation logic predicates* (noted **ind**), and the set $\mathbb{P}$ of *pure predicates* (noted $\mathbb{p}$) are defined by the grammar below:

$$
\begin{array}{rcll}
\mathbb{d} & ::= & \mathbb{h} \wedge \mathbb{p} & \\
 & | & \bot & \text{unsatisfiable abstract shape} \\
 & | & \mathbb{d} \vee \mathbb{d} & \text{disjunctive abstract shape} \\
\mathbb{h} & ::= & \mathbf{emp} & \\
 & | & \mathbb{a}_0 \cdot \mathtt{f} \mapsto \mathbb{a}_1 & \\
 & | & \mathbb{h} * \mathbb{h} & \\
 & | & \mathbf{ind}(\mathbb{a}_0, \ldots, \mathbb{a}_n) & \text{inductive summary instance} \\
\mathbf{ind} & ::= & \exists \mathbb{a}'_0, \ldots, \mathbb{a}'_k \cdot \mathbb{d}_0 \vee \cdots \vee \mathbb{d}_n & \text{inductive summary definition} \\
\mathbb{p} & ::= & \mathbb{a} = \&\mathbf{x} \mid \mathbb{a} = 0 \mid \mathbb{a} \neq 0 \mid \ldots &
\end{array}
$$

The body of an inductive predicate uses a bunch of local variables, that are intuitively quantified existentially.

We now need to set up the concretization of these abstract shapes. Fortunately, the definition of the concretization mostly follows the $\vdash$ relation that was set up in Definition 4.7. There are only two differences: first, the symbolic addresses need to be turned into normal addresses as part of the concretization process; second, inductive summary predicates need to be eliminated, which is achieved by unfolding into non-summary predicates. Assuming some fixed set of inductive predicates, we define the meaning of abstract shapes as follows:

Definition 4.9 (Unfolding Relation, Valuation, Abstract Substitution and Concretization of Abstract Shapes)**.** The *unfolding relation* $\rightsquigarrow$ describes how an abstract heap can be rewritten into another one by replacing

one of the inductive predicates it contains with one of its inductive cases (note that the existentially quantified variables may need to be renamed by α-equivalence in the definition of inductive predicates in order to avoid collisions):

$$\frac{\begin{array}{c}\mathbf{ind}(\mathbb{a}_0,\ldots,\mathbb{a}_n) = \exists \mathbb{a}'_0,\ldots,\mathbb{a}'_k \cdot \mathbb{d}_0 \vee \cdots \vee \mathbb{d}_k \qquad 0 \leq i \leq k \\ \mathbb{a}'_0,\ldots,\mathbb{a}'_k \text{ are fresh in } \mathbb{d} \qquad \mathbb{d}_i = \mathbb{h}_i \wedge \mathbb{p}_i\end{array}}{(\mathbb{h} * \mathbf{ind}(\mathbb{a}_0,\ldots,\mathbb{a}_n)) \wedge \mathbb{p} \rightsquigarrow (\mathbb{h} * \mathbb{h}_i) \wedge (\mathbb{p} \wedge \mathbb{p}_i)} \quad .$$

We write $\rightsquigarrow^*$ for the iterated application of $\rightsquigarrow$, and $\mathbb{H}_\mathrm{f}$ for the set of *flat abstract heaps*, that is abstract heap that do not contain any inductive predicate.

A *valuation* is a function $\nu \colon \mathbb{S} \longrightarrow \mathbb{A}$ that maps symbolic addresses into addresses. The *address substitution* function Φ takes a valuation and an abstract shape with no inductive predicate (resp., a flat abstract heap or pure predicate) as arguments, and replaces symbolic addresses with addresses, based on the image of ν:

$$\begin{aligned}\Phi(\nu, \bot) &= \emptyset \\ \Phi(\nu, \mathbb{d}_0 \vee \mathbb{d}_1) &= \Phi(\nu, \mathbb{d}_0) \vee \Phi(\nu, \mathbb{d}_1) \\ \Phi(\nu, \mathbb{h} \wedge \mathbb{p}) &= \Phi(\nu, \mathbb{h}) \wedge \Phi(\nu, \mathbb{p}) \\ \Phi(\nu, \mathbb{a}_0 \cdot \mathtt{f} \mapsto \mathbb{a}_1) &= \nu(\mathbb{a}_0) \cdot \mathtt{f} \mapsto \nu(\mathbb{a}_1) \\ \Phi(\nu, \mathbf{emp}) &= \mathbf{emp} \\ \Phi(\nu, \mathbb{h}_0 * \mathbb{h}_1) &= \Phi(\nu, \mathbb{h}_0) * \Phi(\nu, \mathbb{h}_1)\end{aligned}$$

Then, we let the *concretization* of an abstract shape be defined by

$$\gamma(\mathbb{d}) = \{m \in \mathbb{M} \mid \exists (\mathbb{h} \wedge \mathbb{p}), \nu,\ \mathbb{d} \rightsquigarrow^* (\mathbb{h} \wedge \mathbb{p}) \wedge m \vdash \Phi(\nu, \mathbb{h} \wedge \mathbb{p}) \wedge \nu \vdash \mathbb{p}\}$$

where $\nu \vdash \mathbb{p}$ if and only the valuation ν satisfies the pure predicate $\mathbb{p}$.

The following paragraphs present a few common inductive predicates, and examples of such abstract shapes, and detail how value abstraction may be combined with separation logic based shape abstraction, and how the shape abstract states may be represented.

A Few Inductive Structures and Their Abstraction. Linked structures such as singly linked lists and binary trees can be described by simple inductive predicates in separation logic:

$$\mathbf{list}(\mathbb{o}) = \exists \mathbb{o}_0, \mathbb{o}_1 \cdot \begin{cases} & \mathbf{emp} \wedge \mathbb{o} = 0 \\ \vee & \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_0 * \mathbb{o} \cdot \mathtt{f} \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_0) \end{cases}$$

$$\mathbf{tree}(\mathbb{o}) = \exists \mathbb{o}_0, \mathbb{o}_1, \mathbb{o}_2 \cdot \begin{cases} & \mathbf{emp} \wedge \mathbb{o} = 0 \\ \vee & (\mathbb{o} \cdot \mathtt{l} \mapsto \mathbb{o}_0 * \mathbb{o} \cdot \mathtt{r} \mapsto \mathbb{o}_1 \\ & * \mathbb{o} \cdot \mathtt{f} \mapsto \mathbb{o}_2 * \mathbf{tree}(\mathbb{o}_0) * \mathbf{tree}(\mathbb{o}_1)) \end{cases}$$

The inductive predicate $\mathbf{list}(\mathbb{o})$, where the symbolic address $\mathbb{o}$ denotes the address of the head node distinguishes two cases: a singly linked list is either empty, or non empty; in the second case, it can be divided into a node with (at least two) fields, including a `n` field, and the list tail that is pointed to by the field `n` of the first node, and that can also be described by induction. The inductive predicate **tree** is very similar.

It is also often useful to describe an incomplete fragment of an inductive data structures. As an example, the singly linked list depicted in Figure 2.4 is naturally divided into two parts, which respectively consist of the list nodes between `l` and the cursor pointer `c`, and of the list nodes beyond the cursor pointer `c`. The above **list** inductive predicate can only describe a complete singly linked list. It is however also possible to describe such an incomplete "segment" of a structure using an inductive predicate, provided a second parameter symbolic address $\mathbb{o}'$ denotes the value of the `n` pointer of the last element:

$$\mathbf{lseg}(\mathbb{o}, \mathbb{o}') = \exists \mathbb{o}_0, \mathbb{o}_1 \cdot \begin{cases} & \mathbf{emp} \wedge \mathbb{o} = \mathbb{o}' \\ \vee & \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_0 * \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_1 * \mathbf{lseg}(\mathbb{o}_0, \mathbb{o}') \end{cases}$$

This inductive predicate is very similar to **list**. Actually, we remark, the only difference occurs in the base case, as an empty segment consists of an empty region, where both symbolic variables $\mathbb{o}, \mathbb{o}'$ describe the same address.

Using the above inductive predicates, we can now give a few examples of abstract shapes using our separation logic based abstraction. First, we consider the case of the memory states shown in Figure 2.3, and where the variable `l` stores the address of the first node of a singly

linked list. These memory states can all be described by the abstract shape below:

$$\begin{array}{ll} & \mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_1) \\ \wedge & \\ & \mathbb{o}_0 = \&\mathtt{l} \end{array}$$

In the above formula, the first like presents the abstract heap, and the second line the pure predicates. It uses two symbolic addresses $\mathbb{o}_0$ and $\mathbb{o}_1$ which respectively stand for the address and the content of the variable `l`.

In the case of the memory states shown in Figure 2.4, we need two summary predicates, where the first one represents the segment formed of the elements of the list that occur before the element pointed to by `c`, and the second one represents the tail of the list:

$$\begin{array}{ll} & \mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{o}_2 \mapsto \mathbb{o}_3 * \mathbf{lseg}(\mathbb{o}_1, \mathbb{o}_3) \\ & * \mathbb{o}_3 \cdot \mathtt{n} \mapsto \mathbb{o}_4 * \mathbb{o}_3 \cdot \mathtt{f} \mapsto \mathbb{o}_5 * \mathbf{list}(\mathbb{o}_4) \\ \wedge & \\ & \mathbb{o}_0 = \&\mathtt{l} \wedge \mathbb{o}_2 = \&\mathtt{c} \end{array}$$

The next two paragraphs briefly discuss the machine representation of abstract shapes in separation logic.

Representation of Abstract Heaps. We first discuss the representation of abstract heaps.

Based on Definition 4.8, the most straightforward representation relies on the abstract syntax tree of logical formulas. This representation is intuitive as it closely follows the definition. However, a drawback of this presentation is that formulas are equivalent modulo commutativity and associativity of $*$. Even representing an abstract heap as a list of terms does not fully solve this issue, since the order of terms does not change the meaning of the formula. This means that locating a term (e.g., to resolve the value of a given variable) is likely to be more costly than it should be.

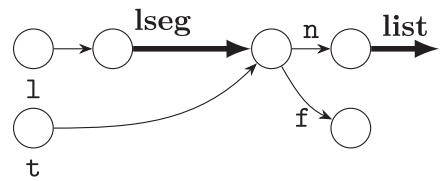
An alternate approach relies on a form of *shape graph*, where nodes stand for symbolic variables, and basic edges correspond to points-to predicates, or structure segments. Such a representation eases the search for the term that describes a variable or other location as it makes the

structure of abstract shapes even closer to the layout of concrete heaps. This structure comes at the cost of more complicated algorithms, but also allows efficiency gains. This approach has been implemented in Xisa (Chang *et al.*, 2007) and in Predator (Dudka *et al.*, 2011). To illustrate this approach, we show an abstract shape described by a separation logic formula in Figure 4.12(a) and its representation as a shape graph in Figure 4.12(b). In this picture, thin edges denote basic cells and thick edges stand for summarized regions. This representation shows the backbone of the inductive structure and the overall layout of the corresponding concrete states.

Representation of Pure Constraints and Combination with a Value Abstraction. We have observed that abstract shapes embed so called pure predicates, which may represent numerical constraints or other constraints on the values of the symbolic addresses. In the above examples, these pure predicates denote modest pointer properties (equality or disequality to null, equality to the address of a variable...), but we can imagine considering much more involved constraints (Chang and Rival, 2008). In this view, pure predicates are elements of a *value abstract domain*, which is typically a numerical domain (Cousot and Cousot, 1977, 1978). The dimensions of this domain range over symbolic variables in the abstract heap. The machine representation of the pure predicates boils down to the existing value abstract domain representation. From the static analysis point of view, this abstraction supports both the reduced product approach, where both shape and numerical predicates are computed in the same analysis (Chang and

$$\mathbb{0}_0 \mapsto \mathbb{0}_1 * \mathbb{0}_2 \mapsto \mathbb{0}_3$$
$$* \ \mathbb{0}_3 \cdot \mathtt{n} \mapsto \mathbb{0}_4 * \mathbb{0}_3 \cdot \mathtt{f} \mapsto \mathbb{0}_5$$
$$* \ \mathbf{lseg}(\mathbb{0}_1, \mathbb{0}_3) * \mathbf{list}(\mathbb{0}_4)$$
$$\wedge \ \mathbb{0}_0 = \&\mathtt{l} \wedge \mathbb{0}_2 = \&\mathtt{t}$$

(a) Separation logic formula



(b) Shape graph representation

Figure 4.12: Separation logic formula and shape graph representation.

Rival, 2008), and the design of separate shape and numerical analyses, where the shape analysis phase outputs not only shape invariants but also a purely numerical program, that can be analyzed in a second phase (Magill *et al.*, 2010). These combinations are discussed in detail in Subsection 5.1.

4.4.3 Computation of Post-Conditions

In the following paragraphs, we detail the definition of abstract operators to compute post-conditions for basic program statements such as assignments condition tests using separation logic based abstract shapes (Berdine *et al.*, 2005b). As several abstract operations need to *refine* abstract states so as to reason about memory regions described by summary predicates, we start with the presentation of an *unfolding* scheme for inductive predicates, which will be used in the following paragraphs.

Refinement Based on the Unfolding of Inductive Summaries. To illustrate the need for refinement, we consider a basic example. We let $\mathbb{d}$ be the abstract shape $\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_1) \wedge \mathbb{o}_0 = \&\mathtt{l}$, which was presented in the previous paragraph, and assume it as an abstract pre-condition. Intuitively, `l` points to a well-formed singly-linked list. Moreover, we consider the computation of a post-condition for a statement which reads or writes into either of the fields of the memory cell pointed to by `l`. Before the effect of this operation can be computed, the analysis needs to materialize the fields of the cell points to by `l`. This is not immediately doable here are this cell is part of the memory region that is summarized by $\mathbf{list}(\mathbb{o}_1)$. As we have defined **list** by induction, and as the disjunction of two cases, we can simply replace $\mathbf{list}(\mathbb{o}_1)$ with this disjunction (this is actually how we defined the concretization of inductive predicates in Definition 4.9), and the reorganize the terms, so as to produce a disjunctive abstract shape:

$$\begin{array}{ll} & \mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbf{emp} \wedge \mathbb{o}_0 = \&\mathtt{l} = 0 \\ \vee & \mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{o}_1 \cdot \mathtt{n} \mapsto \mathbb{o}_2 * \mathbb{o}_1 \cdot \mathtt{f} \mapsto \mathbb{o}_3 * \mathbf{list}(\mathbb{o}_2) \wedge \mathbb{o}_0 = \&\mathtt{l} \neq 0 \end{array}$$

In the first disjunct, `l` is actually a null pointer, so trying to access either of its fields would yield a runtime error or null pointer exception

depending on the language. In the second disjunct, $\mathtt{l}$ points to a block, the address of which is $\mathbb{a}_1$, and the two fields of this block are materialized explicitly. We also observe that pure predicates in both disjuncts take into account the pure predicates of the inductive definition **list**. In fact, pure predicates may even rule out some disjuncts, if the pure predicates are not compatible. This would happen in the above example if we started with a pre-condition that expresses $\mathtt{l} \neq 0$; in that case, only the disjunct that corresponds to a non-empty list is possible.

The purpose of the refinement operation is to make this unfolding step automatically. Given an abstract shape $\mathbb{h} \wedge \mathbb{p}$, and a symbolic address $\mathbb{a}$, the *unfolding operator* **unfold** should:

1. locate the term of $\mathbb{h}$ that describes the memory block pointed to by $\mathbb{a}$;

2. if this term is an inductive predicate, proceed to the unfolding of this predicate following the relation $\rightsquigarrow$, and produce a possibly disjunctive abstract shape.

The precise definition of **unfold**, and the way it achieves the localization of the term to unfold depend on the inductive predicates that are considered. When this term cannot be localized precisely, **unfold** may fail to refine the abstract shape it is applied to. The soundness of **unfold** boils down to:

$$\forall \mathbb{d}, \mathbb{a},\ \gamma(\mathbb{d}) \subseteq \gamma(\mathbf{unfold}(\mathbb{d}, \mathbb{a}))$$

Post-Condition for Memory Allocation. We consider the abstract operator $\mathbf{new}_{\mathtt{x}}$, for the analysis of memory allocation statement $\mathtt{x} = \mathbf{new}(\ )$ (Figure 3.3). As observed in Figure 3.2, this statement allocates a fresh memory block, and assigns its address to the variable $\mathtt{x}$. Obviously, it assumes that the variable $\mathtt{x}$ is already defined, thus the abstract pre-condition of $\mathbf{new}_{\mathtt{x}}$ should reflect that. Let us denote by $\mathbb{d}$ this abstract pre-condition, and first assume that it consists of a single term $\mathbb{h} \wedge \mathbb{p}$ (the case of empty or disjunctive abstract shapes are discussed afterwards). Since $\mathtt{x}$ is defined, $\mathbb{d}$ should be of the form:

$$(\mathbb{a}_0 \mapsto \mathbb{a}_1 * \mathbb{h}') \wedge (\mathbb{a}_0 = \&\mathbf{x} \wedge \mathbb{p}')$$

Given such a pre-condition, the analysis should first build up the abstraction of the new allocated block, and secondly update the value of `x` with the address of this new block. This amounts to producing the abstract shape below, where $\mathbb{o}'_0, \mathbb{o}'_1, \ldots$ are fresh symbolic variables, and where `f0`, `f1`, ... are the names of the fields of the newly allocated block:

$$\begin{aligned}&\mathbf{new}_{\mathtt{x}}((\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{h}') \wedge (\mathbb{o}_0 = \&\mathtt{x} \wedge \mathbb{p}'))\\ &= (\mathbb{o}_0 \mapsto \mathbb{o}_2 * \mathbb{o}_2 \cdot \mathtt{f0} \mapsto \mathbb{o}'_0 * \mathbb{o}_2 \cdot \mathtt{f0} \mapsto \mathbb{o}'_1 * \cdots * \mathbb{h}') \wedge (\mathbb{o}_0 = \&\mathtt{x} \wedge \mathbb{p}')\end{aligned}$$

We remark that symbolic variable $\mathbb{o}_1$ may be dropped if it is not referred to in $\mathbb{h}'$ or $\mathbb{p}'$.

The case of empty or disjunctive abstract shapes is straightforward:

$$\begin{aligned}\mathbf{new}_{\mathtt{x}}(\bot) &= \bot\\ \mathbf{new}_{\mathtt{x}}(\mathbb{d}_0 \vee \mathbb{d}_1) &= \mathbf{new}_{\mathtt{x}}(\mathbb{d}_0) \vee \mathbf{new}_{\mathtt{x}}(\mathbb{d}_1)\end{aligned}$$

Post-Condition for Assignments. We now consider the operator $\mathbf{assign}_{\mathtt{x.f} \leftarrow \mathtt{e}}$ for the computation of an abstract post-condition for an assignment statement `x.f = e`. As shown in Figure 3.2, the concrete semantics of such a statement carries out three successive steps: (1) it evaluates the l-value `x.f` into a memory cell, (2) it evaluates the r-value `e` into a value, and (3) it stores this value in the cell produced at step (1). The definition of the analysis follows this sequence of operations. The evaluation of the l-value should produce the abstraction of a cell, which is thus a points-to predicate $\mathbb{o}_0 \cdot \mathtt{f} \mapsto \mathbb{o}_1$. The evaluation of the r-value should produce the abstraction of a value, which is thus a symbolic variable $\mathbb{o}_2$. Last, the update step (3) replaces the points-to predicate $\mathbb{o}_0 \cdot \mathtt{f} \mapsto \mathbb{o}_1$ with the new points-to predicate $\mathbb{o}_0 \cdot \mathtt{f} \mapsto \mathbb{o}_2$. To sum up, if `e` boils down to just a variable `y`:

$$\begin{aligned}&\mathbf{assign}_{\mathtt{x.f} \leftarrow \mathtt{y}}((\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{o}_1 \cdot \mathtt{f} \mapsto \mathbb{o}_2 * \mathbb{o}_3 \mapsto \mathbb{o}_4 * \mathbb{h}')\\ &\qquad\qquad \wedge (\mathbb{o}_0 = \&\mathtt{x} \wedge \mathbb{o}_3 = \&\mathtt{y} \wedge \mathbb{p}'))\\ &= (\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{o}_1 \cdot \mathtt{f} \mapsto \mathbb{o}_4 * \mathbb{o}_3 \mapsto \mathbb{o}_4 * \mathbb{h}') \wedge (\mathbb{o}_0 = \&\mathtt{x} \wedge \mathbb{o}_3 = \&\mathtt{y} \wedge \mathbb{p}'))\end{aligned}$$

This definition follows the structure of separation logic formulas in a straightforward manner, and relies on the notion of local reasoning that was presented in Subsection 4.4.1 (Reynolds, 2002). The cases where

the r-value $\mathtt{e}$ is another, more complex pointer expression are similar, when all the fields that need to be read are exposed, as in the above example. This gives a straightforward definition of $\mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{e}}(\mathbb{d})$ for a wide family of expressions $\mathtt{e}$ and abstract shape $\mathbb{d}$.

However, this definition obviously does not work when either the l-value or the r-value is summarized as part of an inductive summary predicate. As we remarked in the beginning of the subsection, such cases require the unfolding of inductive summaries before applying the standard analysis algorithms. As an example:

$$\begin{aligned}&\mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{y}}((\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_1) * \mathbb{o}_3 \mapsto \mathbb{o}_4 * \mathbb{h}') \\ &\qquad\qquad \wedge (\mathbb{o}_0 = \&\mathtt{x} \wedge \mathbb{o}_3 = \&\mathtt{y} \wedge \mathbb{p}')) \\ &= \mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{y}}(\mathbf{unfold}((\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_1) * \mathbb{o}_3 \mapsto \mathbb{o}_4 * \mathbb{h}') \\ &\qquad\qquad \wedge (\mathbb{o}_0 = \&\mathtt{x} \wedge \mathbb{o}_3 = \&\mathtt{y} \wedge \mathbb{p}')), \mathbb{o}_1)\end{aligned}$$

This principle generalizes to the case where both the l-value and the r-value refer to memory cells that are part of memory regions which are summarized by inductive predicates. In this case, the computation of a post-condition for an assignment statement leads to several disjuncts.

Finally, the cases where $\mathbb{d}$ is either $\bot$ or a disjunctive abstract shape are handled as usual:

$$\begin{aligned}\mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{y}}(\bot) &= \bot \\ \mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{y}}(\mathbb{d}_0 \vee \mathbb{d}_1) &= \mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{y}}(\mathbb{d}_0) \vee \mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{y}}(\mathbb{d}_1)\end{aligned}$$

Post-Condition for Condition Tests. Given a condition $\mathtt{e}$, the abstract operator $\mathbf{test}_{\mathtt{e}}$ maps an abstract shape $\mathbb{d}$ to an over-approximation of the set of memory states in $\gamma(\mathbb{d})$ that satisfy condition $\mathtt{e}$. Intuitively, it should simply refine the pure constraints in $\mathbb{d}$, so as to express as precisely as possible the condition $\mathtt{e}$. As an example:

$$\begin{aligned}&\mathbf{test}_{\mathtt{l}=0}((\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{h}') \wedge (\mathbb{o}_0 = \&\mathtt{l} \wedge \mathbb{p}')) \\ &\quad = (\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{h}') \wedge (\mathbb{o}_0 = \&\mathtt{l} \wedge \mathbb{o}_1 = 0 \wedge \mathbb{p}')\end{aligned}$$

Just like $\mathbf{assign}_{\mathtt{x.f}\leftarrow\mathtt{e}}$, $\mathbf{test}_{\mathtt{e}}$ may need to unfold inductive summary predicates, when any of the locations that are read in $\mathtt{e}$ is part of such a summary. Moreover, when unsatisfiable constraints arise, abstract shapes can be reduced to $\bot$ as part of the computation of $\mathbf{test}_{\mathtt{e}}$. For instance,

in the case below, `l` is implicitly known to be a non null pointer because it points to the address of a regular memory cell as witnessed by the points-to predicate, therefore, the condition is unsatisfiable:

$$\mathbf{test}_{1=0}((\mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{o}_1 \mapsto \mathbb{o}_2) \wedge (\mathbb{o}_0 = \&\mathtt{l})) = \bot$$

The extension to the case of $\bot$ or disjunctive abstract shapes is similar as well.

Analysis of Straight Line Code. To conclude this subsection, we consider the analysis of a short program that consists only of conditions and assignments. We assume that the program below is ran from a state where `l` points to a well-formed singly linked list:

```
1  if( l != null ){
2      List c = new List( );
3      c.n = l.n;
4      l.n = c;
5  }
```

Essentially, this program allocates a new element and inserts it in the first position in the list pointed to by `l` when this list is not empty, and does nothing otherwise.

The analysis successively computes the abstract states shown in Figure 4.13. The abstract shapes computed for each program point are interleaved between the statements of the program. The initial abstract shape describes the list pointed to by `l` by an inductive summary predicate. At line 1, the condition operator adds predicates related to the nullness of `l`. At line 2, the memory allocation abstract operator synthesizes the representation of a new list element, and updates `c` with it. At line 3, the analysis of the assignment requires reading fields of the list pointed to by `l`, so the inductive summary needs to be unfolded; note that this unfolding produces a single disjunct as `l` is known to be non null at this point. At line 4, the analysis of the assignment is straightforward as all fields read or modified are already materialized.

4.4.4 Lattice Operations

The abstract operators shown in Subsection 4.4.3 allow to handle the analysis of straight line code, but cannot support loops. In the following,

$$(\mathbb{c}_0 \mapsto \mathbb{c}_1 * \mathbf{list}(\mathbb{c}_1)) \wedge \mathbb{c}_0 = \&\mathtt{l}$$

1 **if**(`l` != **null**){

$$(\mathbb{c}_0 \mapsto \mathbb{c}_1 * \mathbf{list}(\mathbb{c}_1)) \wedge (\mathbb{c}_0 = \&\mathtt{l} \wedge \mathbb{c}_1 \neq 0)$$

2 `List c = new List( );`

$$(\mathbb{c}_0 \mapsto \mathbb{c}_1 * \mathbf{list}(\mathbb{c}_1) * \mathbb{c}_2 \mapsto \mathbb{c}_3 * \mathbb{c}_3 \cdot \mathtt{n} \mapsto \mathbb{c}_4 * \mathbb{c}_3 \cdot \mathtt{f} \mapsto \mathbb{c}_5)$$
$$\wedge (\mathbb{c}_0 = \&\mathtt{l} \wedge \mathbb{c}_1 \neq 0 \wedge \mathbb{c}_2 = \&\mathtt{c})$$

3 `c.n = l.n;`

$$(\mathbb{c}_0 \mapsto \mathbb{c}_1 * \mathbb{c}_1 \cdot \mathtt{n} \mapsto \mathbb{c}_6 * \mathbb{c}_1 \cdot \mathtt{f} \mapsto \mathbb{c}_7 * \mathbf{list}(\mathbb{c}_6)) *$$
$$\mathbb{c}_2 \mapsto \mathbb{c}_3 * \mathbb{c}_3 \cdot \mathtt{n} \mapsto \mathbb{c}_6 * \mathbb{c}_3 \cdot \mathtt{f} \mapsto \mathbb{c}_5$$
$$\wedge (\mathbb{c}_0 = \&\mathtt{l} \wedge \mathbb{c}_1 \neq 0 \wedge \mathbb{c}_2 = \&\mathtt{c})$$

4 `l.n = c;`

$$(\mathbb{c}_0 \mapsto \mathbb{c}_1 * \mathbb{c}_1 \cdot \mathtt{n} \mapsto \mathbb{c}_3 * \mathbb{c}_1 \cdot \mathtt{f} \mapsto \mathbb{c}_7 * \mathbf{list}(\mathbb{c}_6) *$$
$$\mathbb{c}_2 \mapsto \mathbb{c}_3 * \mathbb{c}_3 \cdot \mathtt{n} \mapsto \mathbb{c}_6 * \mathbb{c}_3 \cdot \mathtt{f} \mapsto \mathbb{c}_7)$$
$$\wedge (\mathbb{c}_0 = \&\mathtt{l} \wedge \mathbb{c}_1 \neq 0 \wedge \mathbb{c}_2 = \&\mathtt{c})$$

4 }

$$(\mathbb{c}_0 \mapsto \mathbb{c}_1) \wedge (\mathbb{c}_0 = \&\mathtt{l} \wedge \mathbb{c}_1 = 0)$$
$$\vee (\mathbb{c}_0 \mapsto \mathbb{c}_1 * \mathbb{c}_1 \cdot \mathtt{n} \mapsto \mathbb{c}_3 * \mathbb{c}_1 \cdot \mathtt{f} \mapsto \mathbb{c}_7 *$$
$$\mathbb{c}_3 \cdot \mathtt{n} \mapsto \mathbb{c}_6 * \mathbb{c}_3 \cdot \mathtt{f} \mapsto \mathbb{c}_7 * \mathbf{list}(\mathbb{c}_6))$$
$$\wedge (\mathbb{c}_0 = \&\mathtt{l} \wedge \mathbb{c}_1 \neq 0)$$

Figure 4.13: Analysis of a code fragment.

we set up operations that approximate the classical lattice operations, such as inclusion check, join, and extrapolation of abstract iterates. From the shape analysis point of view, the operations defined in the following subsection accomplish a role dual to that of the refinement presented in Subsection 4.4.3: while refinement tends to unfold inductive summaries, these operations attempt to synthesize new summaries.

Abstract Inclusion Checking. We first consider *inclusion checking*. Given two abstract shapes $\mathbb{d}_0$ and $\mathbb{d}_1$, the inclusion checking algorithm attempts to decide whether $\gamma(\mathbb{d}_0) \subseteq \gamma(\mathbb{d}_1)$. It boils down to a conservative function **incl**: $\mathbb{D} \times \mathbb{D} \longrightarrow \mathbb{B}$, where $\mathbb{B}$ stands for the set of booleans $\{\mathbf{true}, \mathbf{false}\}$, and which is conservative in the sense that:

$$\mathbf{incl}(\mathbb{d}_0, \mathbb{d}_1) = \mathbf{true} \Longrightarrow \gamma(\mathbb{d}_0) \subseteq \gamma(\mathbb{d}_1)$$

The definition of **incl** depends on the nature of the inductive summary predicates that are used in the definition of $\mathbb{D}$. To make the presentation easier to follow, we provide the definition of a set of logical rules, which describe conditions under which the inclusion $\gamma(\mathbb{d}_0) \subseteq \gamma(\mathbb{d}_1)$ may be derived. Indeed, we can prove by induction on the derivation that, whenever the predicate $\vdash \mathbb{d}_0 \sqsubseteq \mathbb{d}_1$ is derived, the inclusion of concretization holds. The main logical rules are shown in Figure 4.14:

- the rules $(\sqsubseteq_\bot)$, $(\sqsubseteq_{\vee-l})$, $(\sqsubseteq_{\vee-r})$ and $(\sqsubseteq_{\vee-r'})$ show how to derive inclusion over disjunctive and possibly $\bot$ abstract shape, using classical disjunctive reasoning principles;
- the rule $(\sqsubseteq_{\text{pure}})$ separates inclusion checking over pure predicates (to be discharged using abstract operators from the numerical domain) and inclusion checking over abstract heaps;
- the rule $(\sqsubseteq_*)$ allows to derive inclusion by local checking, considering disjoint regions;
- the rule $(\sqsubseteq_{\text{Id}})$ expresses that inclusion is reflexive;
- last, the rule $(\sqsubseteq_{\rightsquigarrow})$ allows to compare inductive summary predicates with unfolded regions.

The precise definition of the inductive predicates may define additional rules to establish inclusion. As an example, we have defined in Subsection 4.4.2 the inductive predicates **list** and **lseg**, that respectively describe singly linked list and segments of singly linked lists. Using these predicates, the rule below is sound, and expresses that a list segment appended to a list forms another list:

$$\frac{}{\vDash \mathbf{lseg}(\mathbb{a}_0, \mathbb{a}_1) * \mathbf{list}(\mathbb{a}_1) \sqsubseteq \mathbf{list}(\mathbb{a}_0)}$$

While the rules of Figure 4.14 show how to derive inclusion, they do not fully specify how to implement an operator **incl** that takes two abstract shapes as arguments and attempts to determine whether inclusion holds. The full definition of **incl** requires to set up a proof search strategy using the rules presented in Figure 4.14, that attempts to apply the rules in a specific order. An ideal such order is difficult to

$$\frac{}{\vDash \bot \sqsubseteq \mathbb{d}_1}\ (\sqsubseteq_\bot) \qquad \frac{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}_1 \quad \vDash \mathbb{d}'_0 \sqsubseteq \mathbb{d}_1}{\vDash \mathbb{d}_0 \vee \mathbb{d}'_0 \sqsubseteq \mathbb{d}_1}\ (\sqsubseteq_{\vee - l})$$

$$\frac{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}_1}{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}_1 \vee \mathbb{d}'_1}\ (\sqsubseteq_{\vee - r}) \qquad \frac{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}'_1}{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}_1 \vee \mathbb{d}'_1}\ (\sqsubseteq_{\vee - r'})$$

$$\frac{\vDash (\mathbb{h}_0 \wedge \mathbb{p}_0) \sqsubseteq (\mathbb{h}_1 \wedge \mathbb{p}_0) \quad \mathbb{p}_0 \Longrightarrow \mathbb{p}_1}{\vDash (\mathbb{h}_0 \wedge \mathbb{p}_0) \sqsubseteq (\mathbb{h}_1 \wedge \mathbb{p}_1)}\ (\sqsubseteq_{\text{pure}})$$

$$\frac{\vDash (\mathbb{h}_0 \wedge \mathbb{p}_0) \sqsubseteq (\mathbb{h}_1 \wedge \mathbb{p}_1) \quad \vDash (\mathbb{h}'_0 \wedge \mathbb{p}_0) \sqsubseteq (\mathbb{h}'_1 \wedge \mathbb{p}_1)}{\vDash (\mathbb{h}_0 * \mathbb{h}'_0) \wedge \mathbb{p}_0 \sqsubseteq (\mathbb{h}_1 * \mathbb{h}'_1) \wedge \mathbb{p}_1}\ (\sqsubseteq *)$$

$$\frac{}{\vDash \mathbb{d} \sqsubseteq \mathbb{d}}\ (\sqsubseteq_{\text{Id}}) \qquad \frac{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}'_1 \quad \mathbb{d}_1 \rightsquigarrow \mathbb{d}'_1}{\vDash \mathbb{d}_0 \sqsubseteq \mathbb{d}_1}\ (\sqsubseteq_{\rightsquigarrow})$$

Figure 4.14: Logical rules for inclusion checking.

choose, and may depend on the specific inductive predicates that are used, so we do not fully make it explicit here.

To conclude this paragraph, we show an example inclusion derivation, where we assume, for the sake of clarity, that **list** defines singly linked lists where elements have a single field `n`, which points to the next element:

$$\frac{\dfrac{}{\vDash \mathbb{o}_0 \cdot \mathtt{n} \mapsto \mathbb{o}_1 \wedge \mathbb{o}_1 = 0 \sqsubseteq \mathbb{o}_0 \cdot \mathtt{n} \mapsto \mathbb{o}_1} \qquad \dfrac{\mathbf{list}(\mathbb{o}_1) \rightsquigarrow \mathbf{emp} \wedge \mathbb{o}_1 = 0}{\vDash \mathbf{emp} \wedge \mathbb{o}_1 = 0 \sqsubseteq \mathbf{list}(\mathbb{o}_1)}}{\dfrac{\vDash \mathbb{o}_0 \cdot \mathtt{n} \mapsto \mathbb{o}_1 \wedge \mathbb{o}_1 = 0 \sqsubseteq \mathbb{o}_0 \cdot \mathtt{n} \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_1)}{\vDash \mathbb{o}_0 \cdot \mathtt{n} \mapsto \mathbb{o}_1 \wedge \mathbb{o}_1 = 0 \sqsubseteq \mathbf{list}(\mathbb{o}_0)}}$$

Typical algorithms for **incl** carry out a proof search that is similar to that shown in the above proof tree.

The design of entailment checkers and theorem provers for separation logic has attracted a lot of interest beyond the shape analysis research field. In particular, Piskac *et al.* (2013) and Le *et al.* (2016) have proposed solvers modulo theory (SMT) for fragments of separation logic. Likewise, Qiu *et al.* (2013) and Pek *et al.* (2014) designed provers for fragments of separation logic. The inclusion checking described above may be viewed a specialized counterpart for such tools, that is aimed at performing well on entailment problems typically encountered in static analysis.

The next paragraphs put the inclusion rules of Figure 4.14 to work so as to define several forms of *weakening* algorithms.

Weakening of Abstract Shapes. First, we consider a *unary weakening operator* **weaken**: $\mathbb{D} \longrightarrow \mathbb{D}$, such that:

$$\forall \mathbb{d} \in \mathbb{D},\ \gamma(\mathbb{d}) \subseteq \gamma(\mathbf{weaken}(\mathbb{d}))$$

The purpose of such an operator is to carry out some generalization over a single input abstract shape. As opposed to **incl**, it does not start with two abstract shapes that it tries to compare. Instead, it takes just one abstract shape, and it should compute another one, that is weaker. Unary weakening operators have been employed in many shape analysis tools based on separation logic (Berdine *et al.*, 2005a, 2007).

The identity function satisfies the above soundness condition, yet it does not achieve any interesting weakening, so that it is not useful. Useful weakening operators should recognize memory regions that can be described in a more abstract manner, for instance by synthesizing new inductive summary predicates, and where doing so will not deteriorate too much the analysis precision.

The rules shown in Figure 4.14, that define $\sqsubseteq$ form a natural starting point to define an operator **weaken**. However, just like in the case of **incl**, the definition of **weaken** should rely on an adequate strategy to efficiently weaken abstract shapes, for instance by identifying portions of abstract heaps that could be weakened into an inductive summary. As an example, the following excerpt of an **weaken** operator performs such a folding:

$$\mathbf{weaken}(\mathbb{c}_0 \cdot \mathtt{n} \mapsto \mathbb{c}_1 \wedge \mathbb{c}_1 = 0) = \mathbf{list}(\mathbb{c}_0)$$

Abstract Union of Abstract Shapes. The abstract union operator **join** introduced in Subsection 3.3 also produces an over-approximation of its inputs, except that it takes two abstract shapes as arguments instead of one in the case of **weaken**. We remark that producing $\mathbb{d}_0 \vee \mathbb{d}_1$ as a result for $\mathbf{join}(\mathbb{d}_0, \mathbb{d}_1)$ is possible, though it does not accomplish any effective generalization. In practice, such a join operator may be used to analyze sequences of condition statements (as shown in Subsection 3.3),

yet it would not allow to cut down the size of symbolic disjunctions, so that it would let the logical formulas of the abstract shapes grow larger and larger. Shape analysis of non trivial programs require a smarter **join** operator to be used, at least for the analysis of loops.

The principle to design an **join** operator is similar to the construction of **weaken**. In particular, the most important step is to synthesize novel inductive summary predicates in order to over-approximate memory regions. However, contrarily to **weaken**, **join** may use two arguments to guide this summary synthesis process, which allows to define more powerful and simpler generalization schemes. For instance, we show a couple of commonly used join rules:

$$\frac{}{\mathbf{join}(\mathbb{d}, \mathbb{d}) = \mathbb{d}}$$

$$\frac{\mathbf{join}(\mathbb{d}_0, \mathbb{d}_1) = \mathbb{d} \qquad \mathbf{join}(\mathbb{d}'_0, \mathbb{d}'_1) = \mathbb{d}'}{\mathbf{join}(\mathbb{d}_0 * \mathbb{d}'_0, \mathbb{d}_1 * \mathbb{d}'_1) = \mathbb{d} * \mathbb{d}'}$$

$$\frac{\mathbf{incl}(\mathbb{d}_0, \mathbf{ind}(\mathbb{o})) = \mathbf{true}}{\mathbf{join}(\mathbb{d}_0, \mathbf{ind}(\mathbb{o})) = \mathbf{ind}(\mathbb{o})}$$

The first of these rule simply expresses that **join** does not need to perform any weakening when both of its arguments are already equal. The second of these two rules asserts that **join** can be computed locally (as all analysis operations that we have seen so far). The third one invites to weaken abstract shapes into inductive summaries, whenever either argument of **join** already contains a summary. We remark that this weakening rule relies on the inclusion checking algorithm to make sure that the rule applies.

As an example, we assume we consider the **tree** inductive predicate defined in Subsection 4.4.2, show only the fields for the pointers to left and right tree sub-trees, and consider the following two abstract shapes:

$$\begin{aligned}\mathbb{d}_0 = {} & \mathbb{o}_0 \cdot \mathtt{l} \mapsto \mathbb{o}_1 * \mathbb{o}_0 \cdot \mathtt{r} \mapsto \mathbb{o}_2 * \mathbb{o}_1 \cdot \mathtt{l} \mapsto \mathbb{o}_3 * \mathbb{o}_1 \cdot \mathtt{r} \mapsto \mathbb{o}_4 \\ & * \mathbf{tree}(\mathbb{o}_2) * \mathbf{tree}(\mathbb{o}_3) * \mathbf{tree}(\mathbb{o}_4)\end{aligned}$$

$$\begin{aligned}\mathbb{d}_1 = {} & \mathbb{o}_0 \cdot \mathtt{l} \mapsto \mathbb{o}_1 * \mathbb{o}_0 \cdot \mathtt{r} \mapsto \mathbb{o}_2 * \mathbb{o}_2 \cdot \mathtt{l} \mapsto \mathbb{o}_3 * \mathbb{o}_2 \cdot \mathtt{r} \mapsto \mathbb{o}_4 \\ & * \mathbf{tree}(\mathbb{o}_1) * \mathbf{tree}(\mathbb{o}_3) * \mathbf{tree}(\mathbb{o}_4)\end{aligned}$$

Intuitively, both $\mathbb{d}_0$ and $\mathbb{d}_1$ describe well-formed binary trees, but there is a slight difference between the structures that they describe: $\mathbb{d}_0$ describes

binary trees with at least two nodes, namely the root and a left child, whereas $\mathbb{d}_1$ describes binary trees with at least two nodes, namely the root and a right child. The three rules shown above are sufficient to compute the abstract shape below:

$$\mathbf{join}(\mathbb{d}_0, \mathbb{d}_1) = \mathbb{o}_0 \cdot \mathtt{l} \mapsto \mathbb{o}_1 * \mathbb{o}_0 \cdot \mathtt{r} \mapsto \mathbb{o}_2 * \mathbf{tree}(\mathbb{o}_1) * \mathbf{tree}(\mathbb{o}_2)$$

We note that this result is slightly less precise than $\mathbb{d}_0 \vee \mathbb{d}_1$, since it includes any binary tree with at least one node, including the case where both pointers to sub-trees are null and the tree has a single node.

Many shape analysis tools based on separation logic rely on the use of join operators (Chang *et al.*, 2007; Yang *et al.*, 2008).

Extrapolation Techniques. In the previous paragraphs, we have introduced several approaches to weaken abstract shapes. We now show how an operator **extrapol** can be defined from these, so as to compute in finite time sound loop invariants. We recall that **extrapol** (Subsection 3.3) applies to an abstract shape $\mathbb{d}$ that defines the pre-condition of a loop, and to a function $f\colon \mathbb{D} \to \mathbb{D}$ that describes the effect of one iteration of the body of the loop on abstract shapes. Remark that computing a series of abstract shapes of the form $\mathbb{d}_k = \mathbb{d} \vee f(\mathbb{d}) \vee f \circ f(\mathbb{d}) \vee \cdots \vee f^k(\mathbb{d})$ would not terminate, and would produce ever weaker (or more general) abstract shapes.

A first way to build an operator **extrapol** is to use a unary weakening operator **weaken**, with the additional constraint that the image of **weaken** should be a finite height sub-lattice of $\mathbb{D}$. Then, the sequence of abstract shapes defined below is increasing thus stationary, and it computes a sound loop invariant:

$$\mathbb{d}_0 = \mathbb{d}$$
$$\mathbb{d}_{k+1} = \mathbf{weaken}(\mathbb{d}_k \vee f(\mathbb{d}_k))$$

In practice, to ensure that the **weaken** operator returns in a finite height sub-lattice of $\mathbb{D}$, one simply needs to guarantee that it returns summary predicates often enough. Weakening rules that enforce this depend on the inductive predicates that are considered.

A second way to define **extrapol** is to require **join** to be a *widening operator* (Cousot and Cousot, 1977), which means that any sequence

computed by applying **join** as follows is ultimately stationary:

$$\mathbb{d}_0 = \mathbb{d}$$
$$\mathbb{d}_{k+1} = \mathbf{join}(\mathbb{d}_k, f(\mathbb{d}_k))$$

This sequence also over-approximates the iterations over the loop in the concrete semantics, hence produces a sound loop invariant, provided it converges. The definition of a **join** operator that has the widening property also requires to make sure that it synthesizes summary predicates often enough so as to avoid growing chains of precise shapes only based on points-to predicates.

Last, while this **extrapol** operator requires **join** to be a widening, it is also possible to let the analysis use two kinds of **join**, including one that is not necessarily a widening for the analysis of statements other than loops.

To conclude this paragraph, we show an example extrapolation. We consider a loop that constructs a singly linked list of arbitrary length, by initializing a null pointer and repeating a random number of times the addition of an element at the head of the structure. The abstract shapes synthesized over the first iterates are of the form below (all fields except pointers to next elements are omitted):

$$\mathbb{d}_0 = \mathbf{emp} \wedge \mathbb{o} = 0$$
$$\mathbb{d}_1 = \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_0 \wedge \mathbb{o}_0 = 0$$
$$\mathbb{d}_2 = \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_0 * \mathbb{o}_0 \cdot \mathtt{n} \mapsto \mathbb{o}_1 \wedge \mathbb{o}_1 = 0$$

Using either approaches to extrapolation mentioned above will produce the abstract shape below:

$$\mathbf{list}(\mathbb{o})$$

4.4.5 Example Analysis

To conclude this subsection, we show the results produced by a shape analysis based on separation logic. We consider the insertion of a node in a singly-linked list, at a random position as shown in Figure 4.15. We assume that:

- `l` initially points to a non-empty, well-formed singly-linked list;

- `t` points to a single list element (the `n` field is assumed to be uninitialized);
- `c` is uninitialized.

The intermediate abstract shapes that are computed during the analysis are inserted between the program statements. For the sake of clarity, we omit the fields other than `n` and we use the following shortcuts:

$$\mathbb{h} = \mathbb{o}_0 \mapsto \mathbb{o}_1 * \mathbb{o}_4 \mapsto \mathbb{o}_5$$

$$\mathbb{p} = \mathbb{o}_0 = \&\mathtt{l} \wedge \mathbb{o}_2 = \&\mathtt{c} \wedge \mathbb{o}_4 = \&\mathtt{t} \wedge \mathbb{o}_1 \neq 0$$

The abstract shape shown at the entry point simply formalizes the above assumptions. The abstract shapes attached to the other control states are computed by the algorithms described earlier in this subsection. In particular, the loop invariant comprises a list segment inductive predicates that is synthesized during the extrapolation process, and that was not initially present. It is a straightforward exercise to check that it is inductive. The other invariants are derived systematically from those at the previous point, using the transfer functions for assignments.

4.4.6 Applications and Extensions

The scope of separation logic based shape analyses is very wide, and many tools or analyses rely on this approach. Thus, to conclude this subsection, we mention a few shape analysis tools that rely on separation logic, and some analysis features that we could not discuss in detail.

A first application is the inference of the footprint of a piece of code, which is defined as the memory area that it may read or modify. The Smallfoot tool (Berdine *et al.*, 2005a; Calcagno *et al.*, 2007) relies on separation logic formulas to compute such information.

A second application is the verification of memory safety, including the absence of dereferences of null or invalid pointers. Many tools have been designed so as to infer a superset of the possible safety violations, including Slayer (Berdine *et al.*, 2011) or Facebook Infer (Calcagno and Distefano, 2011). Interesting results have been reported regarding to the analysis of pieces of system code and device drivers (Yang, 2007; Yang *et al.*, 2008). Another way to make separation logic analysis tools

$$(\mathbb{h} * \mathbf{list}(\mathbb{o}_1) * \mathbb{o}_2 \mapsto \mathbb{o}_3 * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_6) \wedge \mathbb{p}$$

`c = l;`

$$(\mathbb{h} * \mathbf{list}(\mathbb{o}_1) * \mathbb{o}_2 \mapsto \mathbb{o}_1 * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_6) \wedge \mathbb{p}$$

while(`c` != **null** && `c.next` != **null** && `random( )`){

$$(\mathbb{h} * \mathbf{lseg}(\mathbb{o}_1, \mathbb{o}_3) * \mathbb{o}_2 \mapsto \mathbb{o}_3 * \mathbb{o}_3 \cdot \mathtt{n} \mapsto \mathbb{o}_7 * \mathbf{list}(\mathbb{o}_7) * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_6) \wedge (\mathbb{p} \wedge \mathbb{o}_7 \neq 0)$$

`c = c.next;`

$$(\mathbb{h} * \mathbf{lseg}(\mathbb{o}_1, \mathbb{o}_3) * \mathbb{o}_2 \mapsto \mathbb{o}_7 * \mathbb{o}_3 \cdot \mathtt{n} \mapsto \mathbb{o}_7 * \mathbf{list}(\mathbb{o}_7) * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_6) \wedge (\mathbb{p} \wedge \mathbb{o}_7 \neq 0)$$

}

$$(\mathbb{h} * \mathbf{lseg}(\mathbb{o}_1, \mathbb{o}_3) * \mathbb{o}_2 \mapsto \mathbb{o}_3 * \mathbb{o}_3 \cdot \mathtt{n} \mapsto \mathbb{o}_7 * \mathbf{list}(\mathbb{o}_7) * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_6) \wedge \mathbb{p}$$

`t.next = c.next;`

$$(\mathbb{h} * \mathbf{lseg}(\mathbb{o}_1, \mathbb{o}_3) * \mathbb{o}_2 \mapsto \mathbb{o}_3 * \mathbb{o}_3 \cdot \mathtt{n} \mapsto \mathbb{o}_7 * \mathbf{list}(\mathbb{o}_7) * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_7) \wedge \mathbb{p}$$

`c.next = t;`

$$(\mathbb{h} * \mathbf{lseg}(\mathbb{o}_1, \mathbb{o}_3) * \mathbb{o}_2 \mapsto \mathbb{o}_3 * \mathbb{o}_3 \cdot \mathtt{n} \mapsto \mathbb{o}_5 * \mathbf{list}(\mathbb{o}_7) * \mathbb{o}_5 \cdot \mathtt{n} \mapsto \mathbb{o}_7) \wedge \mathbb{p}$$

Figure 4.15: Example shape analysis.

is to support a form of genericity with respect to structure contents so as to describe nested structures like lists of lists of lists (Berdine *et al.*, 2007). While some tools operate on whole programs, Facebook Infer performs modular analysis, and uses bi-abduction technique (Calcagno *et al.*, 2009) to infer procedure summaries.

Several tools have also incorporated support for numerical abstractions, which means that they can infer invariants about programs that manipulate both complex data structures and numerical properties. The Xisa approach (Chang and Rival, 2008) relies on a reduced product of

memory and value abstraction and allows them to exchange information during the analysis. The Thor approach (Magill *et al.*, 2010) composes two analyses that respectively focus on the shape part, and on the numerical part.

Separation logic based analyses have also served as a basis for shape abstraction composition operations. The motivation of this kind of techniques is the analysis of programs manipulating overlaid data structures (Lee *et al.*, 2011). This approach can be made more systematic thanks to general memory abstract domain composition operations such as reduced product (Toubhans *et al.*, 2013) and separating product (Toubhans *et al.*, 2014). The MemCAD tool (Li *et al.*, 2017) relies on such abstract domain construction to provide a wide set of logical predicates, without requiring an overly complex abstract domain.

While we consider the inductive predicates used for summaries a parameter of the separation logic abstraction, several works have proposed ways to compute them. In particular, Rival and Chang (2011) set up a widening operator which infers candidates of inductive predicates to be used for the summarization of the stack frame and the heap region that it points to; in this set up, the analysis assumes no inductive predicate and proceeds with summarization and predicate inference in the same time. More recently, Brockschmidt *et al.* (2017) rely on statistical machine learning to infer inductive predicates to be used for verification based on conventional techniques.

4.5 Automata-Based Shape Abstractions

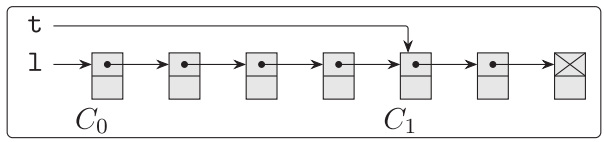
In this subsection, we give an overview of another family of shape analyses that view data-structures stored in the heap as collections of trees, and rely on automata to describe these trees. After we set up the main intuitions in Subsection 4.5.1, we present the abstraction in Subsection 4.5.2. Finally, Subsection 4.5.3 sketches the main analysis algorithms and Subsection 4.5.4 the

4.5.1 Heaps as Collections of Trees

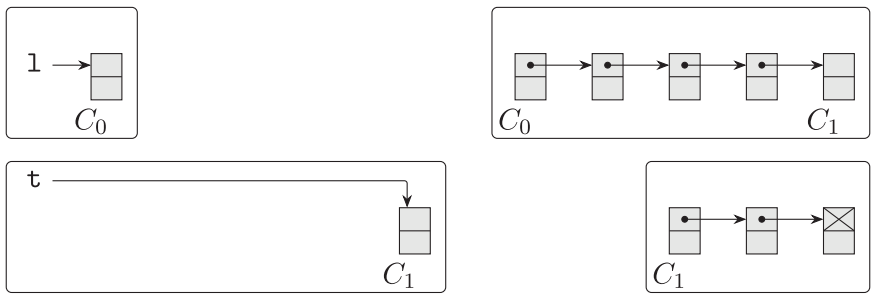
As we have observed previously, we can intuitively view memory states graphs of pointers. At first, let us assume that only non shared structures are used. Then, we observe that each program variable may point to an arborescent structure such as a tree or just a list. Since many relevant sets of trees can be described precisely using tree automata, this observation invites considering tree automata do abstract the heap regions pointed to by each variable. In the more involved case where structures involving sharing (such as doubly linked lists) are considered or structures are pointed to by several variables, this approach requires a bit more care in the sense that heaps first need to be split before tree automata can be applied to the abstraction of memory regions.

In fact, building upon such intuitions, automata-based approaches have been proposed early in a verification context (i.e., to verify user provided program invariants by Habermehl *et al.*, 2006). Other works have suggested using automata in order to describe the effect of programs manipulating heap data-structures and to verify them. For instance, Bouajjani *et al.* (2006) abstract programs that operate over singly linked lists using automata with counters and show that it is possible to use this abstraction as a step towards verification. Automata-based abstraction of program states have also been applied to the design of shape analyses that are able to infer precise invariants for programs manipulating complex data-structures such as lists and trees. In the rest of this subsection, we formalize an abstraction that is closed to that used in the Forester tool (Habermehl *et al.*, 2012) (though it is actually drastically simplified).

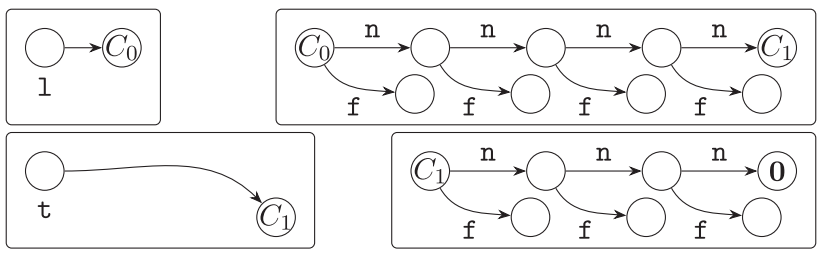
In the following, we consider the example shown in Figure 4.16(a) and give the intuition of its abstraction based on automata. This example was also considered in Figure 2.4, and served as a basis to illustrate the principles of other shape abstractions. It consists of two variables `l` and `t` that respectively point to the head of a complete singly linked list and to an element of that list. In the concrete heap shown in Figure 4.16(a), we mark two cells C_0 and C_1. These two cells are pointed directly by the variables `l` and `t` and the regions that spread in-between are pure singly linked list segments. Except for these, no cell is pointed to from



(a) Concrete heap



(b) Split heap



(c) Shape graph view of the split heap

Figure 4.16: Heap splitting into trees.

two different places or by a variable. These two cells can be used as a boundary to perform a splitting of the concrete heap into four regions, as shown in Figure 4.16(b):

- the two regions in the left respectively correspond to variables `l` and `t`;
- the region in the top right corner of Figure 4.16(b) is a list segment starting in C_0 and ending in C_1;

- the region in the bottom right corner of Figure 4.16(b) is a list tail starting in C_1.

Note that these four regions overlap only in C_0 and C_1. While they are not quite separated in the sense of separation logic (Reynolds, 2002), we can notice some similarity. Indeed, the only shared cells are known and explicitly marked.

We give a second representation of these four heap regions in Figure 4.16(c), using the same conventions as for the shape graphs introduced in Figure 4.12(b). In this representation, we do not duplicate the shared cells marked by C_0 and C_1 (only the nodes which stand for these addresses appear multiple times). This figure uses nodes to depict addresses and edges to denote memory cells.

As we can observe in this second representation each of these four regions is quite simple. The two regions in the left boil down to a single pointer. The two regions in the right can be viewed an iterated repetition of some specific pattern. The interesting point is that these patterns can be described using some techniques inspired by formal languages. Indeed, the top right region boils down to a number (that we may disregard, for the sake of the abstraction) of repetition of the pattern "a single list cell", that starts in C_0 and ends in C_1. The case of the bottom rig th region is similar. We remark that automata play here a role similar to that of inductive predicates shown in Subsection 4.4.

In the rest of this subsection, we formalize this abstraction in more detail.

4.5.2 Abstraction Based on Forest Automata

We first consider the abstraction of a single heap region, such as any of the four shown in Figure 4.16(c). As remarked above, such a pattern may be described using techniques generally used in order to study formal languages. Therefore, we first recall the definition of a finite, non-deterministic, bottom up tree automaton (or, for short, tree automaton—since we are not considering any other kind of automata).

Definition 4.10 (Trees and Tree Automata)**.** A *node constructor* N is defined by an arity n and sequence of field names $\mathtt{f}_1, \ldots, \mathtt{f}_n$. *Heap*

trees (or for short, when there is no ambiguity, trees) are constructed recursively by applying node constructors to sequences of trees such that the length of the sequence of arguments of each constructor matches its arity. A *tree automaton* $\mathcal{A}$ over an alphabet of node constructors Σ is a triple (Q, F, Δ) where Q is a finite set of states, $F \subseteq Q$ denotes the set of final states, and Δ is a finite set of transitions, such that each transition is a tuple $((q_1, \ldots, q_n), f, q)$, where $(q_1, \ldots, q_n)$ is a (possibly empty) sequence of states, f a node constructor of arity n, and q is the target state of the transition. We say that a heap tree t is recognized by state q of the tree automaton $\mathcal{A}$ (and write $\mathcal{A} \vdash t\colon q$) if and only if t is of the form $f(t_1, \ldots, t_n)$ and there exists a transition $((q_1, \ldots, q_n), f, q)$ such that for all $i \in \{1, \ldots, n\}$, heap tree t_i is recognized by state q_i in $\mathcal{A}$. Moreover, a heap tree t is recognized by the automaton $\mathcal{A}$ (which we write $\mathcal{A} \vdash t$) if and only if there exists a final state q of $\mathcal{A}$ such that $\mathcal{A} \vdash t : q$.

Note that the base case of the definition of the tree recognition property is the case of leaves, i.e., of node constructors with no argument (with arity $n = 0$).

As an example, we consider the heap region shown in the bottom right corner of Figure 4.16(c), which can be viewed as a tree. Indeed, a regular list element boils down to a constructor $L = (\mathtt{n}, \mathtt{f})$, and the null pointer at the end of the structure can be viewed a node constructor of arity 0. A heap region containing singly-linked list fragment such as the one shown in that figure can be recognized by a tree automaton with the following states:

- q_0, that describes a value stored in an element, with a single transition $((\,), \cdot, q_0)$;
- q_1 (final state), that describes a possibly empty singly linked list, with two transitions $((\,), \mathbf{0}, q_1)$ (empty list) and $((q_1, q_0), L, q_1)$ (non-empty list, constructed recursively).

This tree automaton is represented graphically in Figure 4.17. The heap region in the bottom right corner of Figure 4.16(c) boils down to a tree recognized by this tree automaton, and such that the root is C_1. The heap region in the top right corner of the figure is quite similar, with

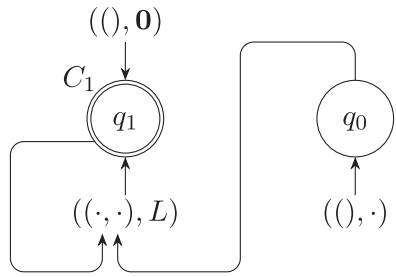


Figure 4.17: Tree automaton.

the only difference that the empty case for the second state corresponds to the node C_1 itself. Instead, we thus include a special state in the automaton to denote C_1 and unroll the list segment. We derive an automaton with three states:

- C_1, that describes only the tree consisting of a single node C_1, with a single transition describing this tree;
- q_0, that describes a value stored in an element, with a single transition $((\,), \cdot, q_0)$;
- q_1 (final state), that describes a possibly empty singly linked list, with two transitions $((C_1, q_0), L, q_1)$ (empty segment) and $((q_1, q_0), L, q_1)$ (non-empty segment).

The case of the two regions in the left is simpler as they involve no induction and can be depicted with trivial automata.

From these examples, we can derive a more general definition of an abstraction relation where abstract shapes boil down to collection of automata, with conditions on the roots. This definition is a simplified version of the one proposed by Habermehl *et al.* (2011).

Definition 4.11 (Abstraction of Heaps Based on Automata)**.** An abstract shape $\mathbb{a}$ is defined by a set of symbols S that is made of variable addresses (noted &x) and of cutpoint names (noted C_i), and a finite set $T = \{(s_0, \mathcal{A}_0), \ldots, (s_n, \mathcal{A}_n)\}$ of pairs $(s_i, \mathcal{A}_i)$ where $s_i \in S$ and $\mathcal{A}_i$ is a tree automaton.

Given an abstract shape $\mathbb{a} = (S, T)$ and $m \in \mathbb{M}$, $m \in \gamma(\mathbb{a})$ if and only if there exists $m_0, \ldots, m_n$ such that $m = m_0 \uplus \cdots \uplus m_n$ and for all $i \in \{0, \ldots, n\}$, m_i is isomorphic to a tree t_i such that $\mathcal{A}_i \vdash t_i$.

As we remarked above, regions storing inductive structures boil down to tree automata with cyclic transitions whereas basic edges boil down to tree automata that recognize a single tree. As an example, the combination of tree automata describing each of the four regions in Figure 4.16(b) provides an abstract shape that over-approximates concrete states such as the memory depicted in Figure 4.16(a).

Additionally, the abstraction can be extended with disjunctions of abstract shapes as in previous subsections.

4.5.3 Computation of Post-Conditions

Algorithms follow a structure similar to those shown in Subsection 4.4 therefore, we follow a less formal presentation here, and only sketch the main principles of the algorithms.

Unrolling of Tree Automata. First, we have observed in previous subsections that the computation of post-conditions often requires to materialize the memory cells that are read or written, when these are described by summary predicates. In particular, we have shown in Subsection 4.4.3 an algorithm to perform this operation on separation logic shape graphs. In the case of the shape abstraction introduced in Definition 4.11, summary predicates are described by tree automata, and it is also possible to refine tree automata. As an example, Figure 4.17 depicts an automaton that recognizes heap trees denoting possibly empty singly-linked lists. By duplicating state q_1 and the transition labeled by L to state q_1, it is possible to turn it into an automaton that recognizes non empty singly linked lists. The result of this transformation is shown in Figure 4.18. The materialization operation is more complex than a mere unrolling of a specific state/transition. Indeed, it also needs to perform disjunctive reasoning (to account for cases where the list is empty).

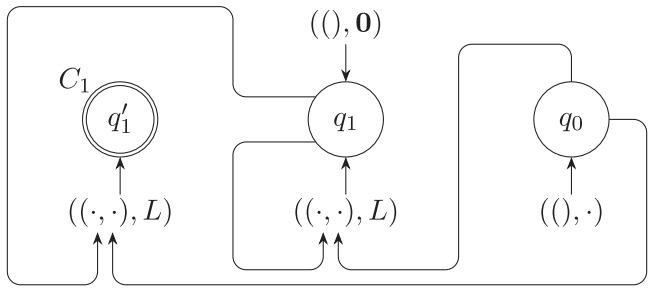


Figure 4.18: Unfolding a tree automaton.

Analysis of Updates. We consider the most general case of a destructive update of the form `x.f = y.g` and assume an abstract shape $\mathbb{a} = (S, T)$ as a pre-condition. Before a post-condition can be computed, the locations of `x.f` and `y.g` need to be determined among the automata in T. Second, materialization needs to be performed whenever either of the cells corresponding to `x.f` and `y.g` cannot be mapped into a unique transition, representing exactly that cell. Third, the affected transition needs to be updated. We remark that the heap partition S may also have to be modified when the assignment introduces sharing.

4.5.4 Lattice Operations and Analysis

To complete the design of an analysis based on the abstract shapes defined in Definition 4.11, we now need to provide algorithms for inclusion testing and union in the abstract level.

Inclusion Testing and Join. Tree automata support standard algorithms for operations such as inclusion testing and union, and these algorithms can be put to work almost directly for the computation of abstract operations. As an example, given two automata $\mathcal{A}_0, \mathcal{A}_1$, we can easily construct one that recognizes the union of the tree languages recognized by $\mathcal{A}_0$ and $\mathcal{A}_1$, by joining the sets of states, transitions and final states of these two automata. The application of this principle to abstract shapes also requires to combine only shapes with comparable heap partitions. Furthermore, Forester (Habermehl *et al.*, 2012) applies an abstraction to tree automata to enforce convergence.

Analysis and Extensions. Using the operations defined in Subsection 4.5.3 and in the previous paragraph, the full definition of a shape analysis follows the structure shown in Section 3. The Forester tool (Habermehl *et al.*, 2012) implements an analysis based on this approach using an abstraction that extends that shown in Definition 4.11.

In this subsection, we described only a simplified version of the analysis. In particular, the abstraction underlying Forester features hierarchical boxes (Habermehl *et al.*, 2011), which allow to express nested structures such as lists of lists. Furthermore, a scheme to infer boxes has been proposed so as to let the analysis require no user annotations in order to infer that a program fragment constructs, e.g., a binary tree. Last, an extension with data reasoning has been proposed by Abdulla *et al.* (2016).

In this subsection, we have alluded to a connection between the separation logic view and the representation based on automata. This connection has been further studied by Iosif *et al.* (2014).

5

Extension of Shape Abstractions

In Section 4, we have described several families of memory abstractions, and focused on the way they describe the layout of memory states. However, the correctness of programs typically relies on more than just the layout of the data structures that they manipulate. For instance, a function that operates on sorted singly-linked lists not only depends on the layout of these lists, but also on the fact that the values that they store satisfy some other constraints. It may need to deal with the low-level implementation of the high-level singly-linked list structure. It may also call other functions, to perform auxiliary tasks. Therefore, shape analyses also need to reason about other features of the semantics of programs than just the shape. Very often, one can derive accurate and useful abstractions for such program by combining or extending memory abstractions such as those presented in Section 4.

This section presents some of these extensions. Subsection 5.1 focuses on the extension of memory abstractions with value abstractions. Subsection 5.2 studies the abstraction of low-level memory operations. Subsection 5.3 lists the main approaches to interprocedural shape analysis.

5.1 Abstraction of Values Stored into Dynamic Structures

Programs rarely only manipulate pointer values, and their correctness generally not only depends on the memory layout but also on its contents, that is the base type values (integers, characters, floating point, or boolean values) that are stored in memory. Moreover, correctness may also rely on the preservation of some memory property that is parameterized by base type values, such as constraints over the length of a list, over the height of a tree, or like sortedness. Therefore, deploying shape analysis on real programs generally requires to describe not only the shape of data structures, but also their contents.

Difficulties Related to Shape Analysis in the Presence of Content Properties. Static analyses able to discover properties related to numeric or boolean values have been known long before shape analyses were proposed, and a very large number of value abstractions have been introduced to deal with all sorts of value analysis problems. However, the combination of shape and value analyses is challenging in terms of expressiveness. Indeed, a combined analysis should not only infer both shape and content properties, but also manage interactions between two very different sorts of abstract information, over the course of the invariant computation. Moreover, value analyses work in a rather different manner compared to shape analyses, and typically do not have a notion of refinement operation as shape analyses do, in order to unfold summaries.

To illustrate the intricacy of these issues, we briefly discuss a couple of examples:

- List length information. We consider the function below, that respectively compute the length of a list:

  ```
  int list_length(list l);
  ```

  When it is applied to a well-formed non-empty list, this function will always return a positive integer. Furthermore, it may be used to compute the length of several lists, and relations among these will induce certain numerical properties. For instance, if

`p` points to a successor of `l`, then `list_length(l)` will return a value that is greater than the result of `list_length(p)`. This example illustrates dependency of numerical information onto shape information.

- List initialization. We consider the functions below, that perform the initialization of all the numeric fields of a list to zero and test the elements of a list for equality to a given numerical value:

```
int list_init_zero(list l);
int list_check_equal(list l, int n);
```

These functions ensure or depend on properties of the contents of the data structures. In fact, even the control flow of their implementation depends both on the shape of the data structures and on their content. For instance, if a list is initialized to zero by a call to `list_init_zero`, the value of a subsequent call to `list_check_equal` can be precisely determined based on content information.

A first issue is to design an expressive combined abstraction and adequate analysis algorithms. This is challenging since shape analyses partition memory cells in a dynamic manner, thus, the domain value abstraction should apply to is necessarily dynamic. Therefore, the control of the dynamic partitioning of the memory, the operations over summaries and the information exchange between the shape and value components of the analysis all impact the expressiveness.

A second issue is the description of the content of summarized region, with accurate predicates, and the integration of such summary predicates into the analysis. The way these summary predicates are defined impacts not only the expressiveness of the analysis, but also the way shape and content information can be managed over the course of the abstract interpretation.

The following paragraph discuss the main approaches to combined shape and content analysis, and how they address these two main issues.

Structure of the Combined Abstraction and Analysis. As remarked above, a combined analysis should not only infer shape properties and

numeric properties, but it should also aggregate information of both sorts and manage the summarization of memory cells. We can cite two main techniques to achieve this.

The *pipeline approach* splits the combined analysis into two successive stages, one dealing with shape and the other with values. As an example, an instance of such an analysis could first perform a shape analysis as shown in Section 4, use the results of this analysis to transform the input program into a purely numerical program, and then carry out a value analysis in the last stage (Magill *et al.*, 2007, 2010). In this scheme the shape analysis phase should not only infer accurate partitions of memory cells and summaries for unbounded regions, but it should also reduce the inference of the value properties of interest to a pure value analysis. As an example the analysis proposed in the Thor tool (Magill *et al.*, 2007) considers list inductive predicates augmented with length information, and attempts to verify preservation and consistency of this length information. The first phase of this analysis infers shape properties and produces a purely numerical program, with assertions encoding the impact over list length of all operations in the initial program. To achieve that, it resolves all dynamic memory locations, and it also synthesizes additional numerical variables that stand for the length of each list. It would be possible to envision a similar process, where the value analysis occurs in the first stage, though we are not aware of any such analysis. We remark that, in both cases, the order in which the two static analyses are performed conditions the kind of dependencies between memory and value information that can be precisely reasoned about.

The *product approach* combines shape and value reasoning into a single analysis, and relies on the classical abstract interpretation product and reduced product operations (Cousot and Cousot, 1979). This means that the abstract interpretation relies on a single iterative abstract post-fixpoint computation, as part of which both shape predicates (including summaries) and value predicates are inferred. As an example, this approach has been implemented in the Xisa tool (Chang and Rival, 2008, 2013), which relies on user supplied inductive definitions that describe both shape invariants and content properties. It has also been implemented in the Celia tool (Bouajjani *et al.*, 2011, 2012). In the

product approach, information can be exchanged between the memory abstract domain and the value abstract domain at any time during the abstract interpretation, which means that in theory, such an analysis can cope with any kind of dependency between shape properties and value properties. This information exchange defines a specific case of *reduction* in abstract interpretation (Cousot and Cousot, 1979). As an example, reduction may put in evidence inconsistencies between shape and value predicates, and derive that certain abstract states are not reachable, as a consequence of the reduction. Moreover, as the product approach tightly ties the shape analysis and the value analysis, it also lets the former infer partitions of the set of memory cells that should be used by the latter. As an example, it is possible that different memory partitions and heap summaries get used at different stages of the analysis even in the same program location.

These two approaches are radically different.

The pipeline approach allows to reuse a value analysis with little modification, and even the shape analysis does not need deep changes as it does not need to explicitly deal with value invariants.

On the other hand, the product approach offers more expressiveness. Indeed, in the pipeline approach, one analysis precedes the other, so that the former cannot make use of semantic information computed by the latter. As an example, if the shape analysis is performed first, it cannot exploit value information, that is not available at this stage. For instance, this is problematic in the case of the list initialization example given above, since that example requires content information to infer properties on the control flow of `list_check_equal`. In the case of the product approach, this issue does not arise, since both shape and value analyses are performed in the same time, thus they can exchange information at any point. Due to this, the product approach is more expressive and powerful in general.

Furthermore, we also observed in the previous sections that a core feature of shape analysis is to manage the partitioning of memories in a dynamic manner, so that it can create or delete summary cells over the course of the analysis. This means that, in the product approach, the set of abstract memory cells handled in the value abstract domain is also dynamic, and benefits from all the information computed by the

shape analysis, and by the value analysis too since both progress in the same time. By contrast, in the pipeline approach, the abstract cells to be handled by the value analysis are fixed once and for all as soon as the shape analysis phase is over, thus the partitioning of memory cells and the definition of abstract summaries are performed without taking into account any value information. This shows another reason why the product approach is more expressive in general.

Abstraction of the Contents of Summarized Memory Regions. We now discuss considerations related to the way summary predicates are extended so as to describe value and content properties of summarized regions. We can cite two main techniques.

Encapsulated inductive predicates embed both the shape and the value properties into composite predicates. This means that a single inductive predicate summarizes both the structure of the data and information about their contents. As an example, the inductive predicate below describes lists data structures made of elements with a numeric data field, that lies in a range defined by a pair of parameters $\mathbb{o}_-, \mathbb{o}_+$:

$$\mathbf{list}(\mathbb{o}, \mathbb{o}_-, \mathbb{o}_+) = \begin{cases} \mathbf{emp} \wedge \mathbb{o} = 0 \\ \vee \quad \exists \mathbb{o}_0, \mathbb{o}_1 \cdot \begin{cases} \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_0 * \mathbb{o} \cdot \mathtt{f} \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_0) \\ \wedge \quad \mathbb{o} \neq 0 \wedge \mathbb{o}_- \leq \mathbb{o}_1 \leq \mathbb{o}_+ \end{cases} \end{cases}$$

Split inductive predicates separate the specification of the shape properties and of the contents properties. Most often, they take the form of shape predicates that can be parameterized with specific contents predicates. As an example, the **list** summary predicate describes well-formed singly linked lists. When it is used to describe lists of, e.g., scalars, it is natural to add additional annotations to convey the fact that the list is sorted or that its elements all lie in a given range.

Nature of Basic Values. In the previous paragraphs, we did not discuss in depth the type of the values we were considering, although we presented a series examples of examples based on integer values. The main reason why we did not start this subsection with a discussion of value types is that value types have no impact on the techniques and issues that were studied in the previous paragraphs (structure of the

combined analysis and structure of the summary predicates). Obviously, the cases of other kinds of scalar values (integers, characters, floating point, and boolean values) is similar. Content properties related to other kinds of base values such as strings do not bring radical differences.

On the other hand, it is possible to also treat as values elements that are not explicitly part of the concrete memory states. As an example, *set* properties are often most useful in order to verify properties of programs. For instance, to verify the preservation of the shape invariants and of the contents of a container data structure, one needs to check that the set of elements is also preserved. To achieve this, the semantics of the program should be augmented with some auxiliary variables of type set, and the tracking of information related to the sets boils down to a basic value static analysis. This approach has been used both in shape analysis tools like Celia (Bouajjani *et al.*, 2011, 2012) and in verification tools that use similar sets of abstract predicates (Chin *et al.*, 2007). In these tools, the handling of sets is similar to that of scalar values in the previous paragraphs. Moreover, set abstraction may be applied to pointer values as well, so as to capture sharing properties in data structures described in separation logic (Li *et al.*, 2015): in that view, the structure of graphs in adjacency list representation can be captured with inductive definitions that fix the backbone of the structure, and auxiliary set predicates that express the additional shared pointers.

5.2 Abstraction of Low-Level Memory Models

The previous sections have presented the foundations of the main families of shape analyses, using as a starting point a fairly simple memory model. Compared to the memory model of many real programming languages, this language model restricts the way memory accesses are carried out. In particular, structure fields were simply assumed to be defined by names of type string. By contrast, C or C++ are very commonly used, and feature a much more complex semantics regarding to memory access and management. In this subsection, we study how shape analyses discussed in Section 4 can be extended so as to cope with such semantics.

Full Featured Memory Access and Management Concrete Semantics. Before we discuss the extension of shape abstractions, we list the main issues related to full featured memory access and management, and how they impact reasoning over programs.

- **Existence of pointers to object fields.** In the previous sections, we assumed the value of a pointer may only be the address of an object. However, some languages feature nested objects, and allow a pointer to store *the address of a field of an object*, which complicates the operations on pointers. Indeed, to reason over such programs, one needs to keep track of the internal structure of objects.

- **Type of pointer values.** While Java pointers are purely symbolic, C and C++ support *numerical pointers* that can be computed directly, or converted into or from integers. In that case, a pointer should be seen as a pair (b, o) formed by the base address b of a memory block (typically, the address of a statically or dynamically allocated object), and of an offset o inside that block, which means that numerical properties may be involved in the computation of indexes. Furthermore, when the physical representation of aggregate data-types (structure and union types) is fully exposed, memory cell sizes and padding bits also need to be considered. In such models, *pointer arithmetics operations* are also allowed. For instance, the sum of a pointer value (b, o) and of an integer n is the pointer value $(b, o + n)$ defined by the same base address and the sum of the offsets. Similarly, the difference of two pointer values (b, o_0) and (b, o_1) that lie in the same block can be defined as the difference $o_0 - o_1$ of the offsets.

- **Management of allocated blocks.** When memory management is fully explicit, and when pointers may store numerical values or the address of the field of an object, several complications arise. In particular, a pointer in a deallocated block may become invalid if this block is freed. Moreover, only pointers to the base address of an object may be safely deallocated.

- **Nesting and overlaying of structures.** General purpose programming languages offer a rich type system that allows to define nested structures. As an example, an element of a singly linked list may in turn store pointers to other dynamic data structures besides the next element in the list. These may include other lists or trees. For instance a "forest" data structure is defined as a linked list each element of which refers to a tree data structure, and where trees corresponding to distinct elements are disjoint.

- **Arrays and string buffers.** So far, we studied only recursive structures such as lists and trees. By contrast, arrays and buffers have no recursive structures. They may be statically or dynamically allocated, and their size may not be known before run-time. Furthermore, they may also be combined with recursive structures either by nesting or by overlaying.

- **Procedures and stack frame.** Procedures bring many important issues. Nested procedure calls may maintain pointers to different nodes of a same structure, which makes shapes very complicated. They also imply that an update to a structure from a callee may completely modify the structure the callers operate on. Last, programming language with higher-order procedures and closures rely on a much more involved notion of environment as well.

The issues related to procedures and stack frame are discussed in the subsection on interprocedural shape analysis (Subsection 5.3) whereas the other aforementioned points are considered further in this subsection.

Abstraction of Complex Memory Access Features, and Applications to Shape Analysis. We first consider the static analysis of programming languages that feature a low-level semantics for pointer and datatypes, like C or C++ do. The semantics of memory accesses (read or write) is fundamentally more complex than in the basic model that we have used in Section 3 and used as a basis in Section 4. Indeed, this semantics also takes into account the numerical offset and size of memory cells and it allows numerical arithmetic operations on pointers.

However, it does not change the high-level shape of pointer structures such as lists, trees, or graphs. As an example, a list data structure is made of objects with two fields that respectively store a pointer to the next element, and some data. Using an inductive definition in separation logic, and assuming that fields are symbolic names, we used the following inductive summary predicate in Section 4:

$$\mathbf{list}(\mathbb{o}) = \exists \mathbb{o}_0, \mathbb{o}_1 \cdot \begin{cases} & \mathbf{emp} \wedge \mathbb{o} = 0 \\ \vee & \mathbb{o} \cdot \mathtt{n} \mapsto \mathbb{o}_0 * \mathbb{o} \cdot \mathtt{f} \mapsto \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_0) \end{cases}$$

If we take numerical offsets into account, assume a 64-bits architecture (a pointer or a base value is 8 bytes long), and assume that the compiler allocates the `n` field first, and the `f` second, we obtain the following inductive predicate:

$$\mathbf{list}(\mathbb{o}) = \exists \mathbb{o}_0, \mathbb{o}_1 \cdot \begin{cases} & \mathbf{emp} \wedge \mathbb{o} = 0 \\ \vee & \mathbb{o} \cdot 0 \mapsto_8 \mathbb{o}_0 * \mathbb{o} \cdot 8 \mapsto_8 \mathbb{o}_1 * \mathbf{list}(\mathbb{o}_0) \end{cases}$$

The only differences are that field names are replaced with field offsets (0 for `n` and 8 for `f`), and the fact that points-to predicates are annotated with memory cell sizes (as arrow subscripts).

This observation implies that the shape abstractions and analysis algorithms presented in Section 4 remain relevant, but need to be extended so as to deal with addresses that are numerical values, and not mere symbolic values. This extension is fairly independent from the core principles of shape analysis, and it relies on techniques known outside of shape analysis. In particular, Miné (2006) extends a static analysis for C programs, so as to handle pointer arithmetics, pointer casts, and union data-types. More recently, Sotin *et al.* (2010) classifies the classes of concrete semantics of pointers so as to make the extension required for static analysis more salient.

The shape analyses of Section 4 can be extended using similar approaches, by simply letting the abstract elements and analysis algorithms reflect for the manipulation of numerical addresses. For instance, Kreiker *et al.* (2010) proposed a version of three-valued logic based shape analysis (Subsection 4.3) that supports numerical pointers and accesses to memory cells of various sizes. It encodes the structure of memory blocks with numeric offsets, nested structures and pointers to fields

into the general three-valued logic framework. Moreover, Calcagno *et al.* (2006) set up a shape analysis based on separation logic (Subsection 4.4) that is able to reason about memory blocks of variable size (as in lists of structures where the size of each element is specified by one of its fields), and to reason on pointer accesses within such a block. Also, Laviron *et al.* (2010) extended a separation logic-based shape analysis so as to deal with pointers to fields, pointer casts, pointer arithmetic, and union types. It relies on an encoding of the separation logic formulas based on shape graphs that contains full numerical pointer information including offsets, memory cell sizes, and padding bits. These shape analyses all utilize the same fundamental abstractions and algorithms as the analyses presented in Section 4: they also rely on summaries (that also account for pointer properties), and on refinement/generalization operations.

Abstraction of Nested and Overlaid Dynamic Data Structures, and of Combinations of Data Structures. Shape abstractions presented in Section 4 focus on rather standard data structures, yet programmers often construct their own advanced structures, by nesting or overlaying basic structures.

Nested structures are collections of structures of different types (lists, trees . . .), and such that one of these types contains fields storing pointers to elements of another type. As an example, a forest is a list of trees. The abstraction of such structures relies on several distinct kinds of summary predicates that account for each of the kinds of elements of the overlaid structure. Such predicates are defined in the same way as in Section 4 although they may require several mutually inductive predicates. When several structures share the same pattern, it is also possible to define more general parametric predicates (Berdine *et al.*, 2007), that can be used in order to summarize lists, lists of lists, etc. Moreover, sharing relations found in structures such as graphs in adjacency list representation can be captured using constraints over pointers, such as set constraints (Li *et al.*, 2015).

Overlaid structures are collections made of a single kind of objects, that are elements of several distinct structures (such as a list and a tree) with the same footprint. Such structures require significantly

more sophisticated summary predicates. For instance, Lee *et al.* (2011) provide an abstraction for a tree such that nodes are also chained (using a different field) so as to form a list. In some cases, it is possible to implement a product of shape domains, where the product implements the non-separating conjunction of separation logic. This approach has been implemented by Toubhans *et al.* (2013).

The *combination of arrays and recursive dynamic structures* is also relevant. Array and string buffer data structures also require abstractions that are able to describe memory regions of unbounded size, and operations over numerical indexes. Therefore, many techniques inherited from shape analyses may be employed to build array abstractions. We discuss array specific abstractions in detail in Subsection 6.1 and focus here on combination of arrays and recursive structures. Calcagno *et al.* (2006) consider arrays nested into dynamic structures, and propose a shape analysis based on separation logic, that is able to reason over list elements of varying size. Memory allocators and process tables often resort to dynamic structures stored in static arrays. To abstract such structures, Liu *et al.* (2018) proposes a coalescing abstraction that ties shape abstract predicates to array regions.

5.3 Interprocedural Shape Analysis

We consider in this subsection analysis techniques for programs with procedure calls. The language in Figure 3.1 allows only one procedure call, namely `new()`, which implements an object creation. In the following, we consider an extended language, that allows any user defined procedure to be called.

Naive Approach. The analysis techniques discussed so far are applicable to any program with procedure calls, as long as these procedures are not recursive. A naive way to reuse them consists in inlining the code of the called procedures. This method reduces the analysis of an inter-procedural program to the analysis of an intra-procedural program, that is a program without procedure calls. However, this naive technique is not applicable to programs with recursive procedures, because the number of recursive calls is potentially unbounded. Even in the absences

of recursive procedures, inlining might cause scalability problems as it blows up the program.

There are two main approaches to reason about inter-procedural programs introduced by Sharir and Pnueli (1981): the call stack approach and the relational approach.

Call Stack. The call stack approach helps overcoming the precision loss due to invalid paths. The context of the call is recorded, by keeping a string that simulates the call stack, and so the analysis distinguishes between different function calls. However, call strings have bounded depth and recursion creates an unbounded number of calling contexts. Therefore the call string approach in the presence of recursive functions loses accuracy. Several approaches were developed to improve the precision of the call string approach, developing abstractions of the call stack (Jeannet *et al.*, 2004). As we discuss a bit further in this survey the call stack itself is amenable to shape abstractions techniques.

Relational Procedure Summaries. The relational approach computes procedure summaries, that is a relation between the procedure's input and output parameters. These relations can be computed bottom-up, by propagating the summaries of individual statements, or top-down, using tabulation to associate input entries with the corresponding output values. Either way the summary is computed in isolation, taking into account only the values of the variables local to the procedure (w.l.o.g. we assume that global variables are input and output parameters of any procedure). The challenge of this approach consists in finding abstractions that are precise enough to re-establish eventual connections with the context of the call (i.e., with variables recorded in the call stack when returning from a call).

The two approaches presented so far are general methods to reason about programs with procedures. They were rather successfully applied on numeric programs however applying them on programs with pointers and dynamically allocated data structures poses new challenges to inter-procedural analysis.

Difficulty. Achieving a precise inter-procedural analysis requires understanding the relation between the parameters of a call and the call stack.

In numeric programs, procedures have no side effects except those captured by the output parameters of the call. Therefore, precise analyses try to capture the relation between the parameters of the analyzed procedure and the top most entry record of the call stack.

In the presence of pointers side effects are much harder to localize. The actual parameters of a call might be related (by the points-to or the alias relation) to any variables in the call stack, not just the top most entry. In the presence of recursive functions the number of program variables grows unboundedly, each entry of the call stack storing the state of one recursive call. Therefore, in the worst case, a precise analysis needs to maintain the relation between the procedure's local variables and unboundedly many variables recorded in the call stack.

For example, free-ing an allocated object in a procedure, would invalidate all variables in the call stack pointing to the memory region of the object that is being freed. In the case of pointer analysis, the properties that an analysis tracks are typically equality predicates between the procedure's local variables and call stack entries. In case of shape analysis, the problem is more complicated, the analysis needs to track predicates that describe data structures, i.e., summary predicates.

Consider the example in Figure 5.1. Procedure `main` allocates a list of size one, pointed to by `l` and calls `insert` on this list. Function `insert` does more than "adding" a new element pointed to by `x` to the beginning of list `l`; it also calls `foo` which modifies the second element in the new list pointed to by `x` (the only element of the original list). This modification could be a free instruction, in which case this information needs to be propagated all the way to the main function, to detect double free which might be possible if `ll` is freed. Another more difficult situation is when the modification creates a cyclic list, by executing `x->next->next=x` for example. In this case the entire summary predicate stating the circularity of the list pointed to by `ll` needs to be propagated to the `main` function.

```
1  void foo(list *x){
2    if(x != NULL){
3      if (x→next != NULL)  modify(temp);
4    }
5  }
6  list* insert(list* y){
7    list *x = (list *) malloc(sizeof(list));
8    x→next = y;
9    foo(x);
10   return x;
11 }
12 int main(){
13   list* l =(list *) malloc(sizeof(list));
14   l->next = NULL;
15   list *ll = l;
16   l=insert(l);
17   return 1;
18 }
```

Figure 5.1: Example interprocedural program.

Call String Approaches. Call-String based approaches explicitly maintain the relation with between the procedure's parameters and the call stack up to a certain precision. Such techniques are fairly orthogonal the abstraction of program memories, and can thus be readily applied to any of the shape analysis techniques discussed in Section 4. Moreover, classical techniques to tune the level of sensitivity apply straightforwardly (Shivers, 1991).

Relational Approaches. Designing a relational shape analysis interprocedural approach is possible if one can ensure that all side effects are captured by the output parameters of the call. The definition of a relational analysis relies on the design of an abstraction for the effect of a function (Reps *et al.*, 1995). While a simple way is to use table of input/output states abstractions, more specific forms of abstract predicates can often be defined. For instance, Jeannet *et al.* (2004) and Jeannet *et al.* (2010) define procedure summaries using a variant of the three-valued logic shape abstraction described in Subsection 4.3, by letting distinct individuals describe entities *before* and *after* a function

call. Also, Illous *et al.* (2017) define a set of novel logical connectors that are inspired by separation logic and allow to express what a function has modified and how it has modified it, and build an abstract domain to describe relations using these connectors. Rinetzky *et al.* (2005a,b) introduce the notion of *cutpoints*, where the absence of cutpoints guarantees that all side effects are captured by the output parameters. Gotsman *et al.* (2006) study an interprocedural shape analysis that characterizes regions of the heap that have been preserved, so as to deal with only finitely many cutpoints.

The computation of function summaries can be tacked either in top-down or bottom-up manners. The bi-abduction approached defined by Calcagno *et al.* (2009) allows to analyze procedures out of context, and build tables of input/output abstraction pairs using a powerful combination of forward and backward reasoning.

Shape Abstraction of the Call Stack Frame. When recursive procedures are applied to an inductive structure such as a list or a tree, the combination of the stack frame, and of the heap form a very complex structure. As an example, at any point of the execution of a recursive infix binary tree traversal procedure, the stack contains a series of pointers into the tree, that follow a (possibly incomplete) branch in that tree. Such combined structures can also be viewed more complex recursive data structures, which means that the stack frame itself is amenable to shape analysis techniques, either using a Three-Valued Logic-based abstraction (Rinetzky and Sagiv, 2001) or a separation logic-based abstraction (Rival and Chang, 2011): in both of these analyses, not only the inductive summarization of the shape abstraction follows the structure of the call stack, but also the characteristic operations of shape analysis also support the analysis of function calls and returns. In fact, the summarization of call stacks observed by repeated recursive calls relies on the generalization mechanism, whereas the retrieval of the caller stack region during the return from the callee relies on the local refinement operation.

We can also remark that less expressive pointer analyses have also been applied to the abstraction of the call stack (as for instance by Sotin and Jeannet, 2011).

Higher-order procedures (procedures that take other procedures as arguments) and closures (partial application of a procedure with several arguments) also result in intricate call states, where the environment layout can be viewed an inductively defined structure. Therefore, shape analysis techniques also apply in such cases, as remarked in Might (2010).

6

Abstractions Exploiting Shape Analysis Principles

In the previous sections, we have shown shape analysis aims at describing memory regions of unknown and unbounded size, that require the computation of summaries during the analysis, as no purely syntactic criteria will provide a reasonable partitioning of the memory into summaries. The fundamental techniques to achieve this comprise the definition of expressive inductive summaries and of operations to decompose and recompose summaries during the analysis. These fundamental techniques have been used in other forms of static analysis, that aim at computing invariants for programs that manipulate very different kinds of structures than the basic, inductively defined structures seen so far, like singly-linked lists or binary trees. This section presents at a very high-level two examples of such analyses: Subsection 6.1 focuses on the abstraction of array structures and Subsection 6.2 studies the abstraction of dictionary structures.

6.1 Abstraction of Arrays

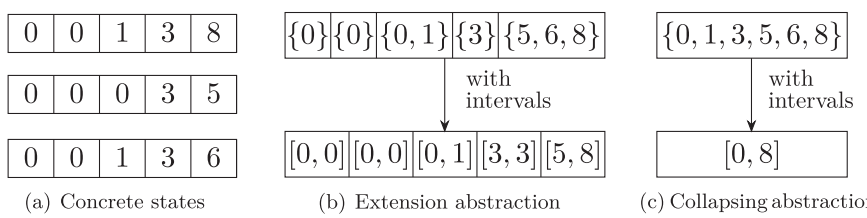
Although array structures may seem simpler to deal with than the pointer structures that were considered in the previous section, they allow to express complex algorithms, that rely on sophisticated index

relations. Moreover, programs may manipulate arrays the size of which is either very large, or even unbounded. This observation entails that the abstraction of array states is also a very difficult problem in general. In the following of this subsection, we discuss a range of array abstractions, including some that share many of the principles of the shape analyses exposed in Section 4.

Basic Array Abstractions. In this paragraph, we detail a few basic array abstractions that do not require any technique from shape analysis. We assume a program that manipulates an array `a` of fixed, and known length. Figure 6.1(a) depicts a few concrete states, under the assumption that `a` is of length 5 and contains numerical values.

Since the array has a known number of cells, it is possible to represent its structure precisely, by considering each cell a separate variable. This *extension abstraction* collects the values that can be read in each cell, as shown in the top of Figure 6.1(b). From that point, classical numerical abstractions may easily be applied to each component, as shown in the bottom of the figure. In general, this approach incurs a resource usage (both in terms of analysis time and memory consumption) when arrays are large. Furthermore, it does not work when the size of arrays cannot be determined statically. On the other hand, it is quite precise. As an example, it can accurately characterize the fact that the first two cells of the array store value 0.

The *collapsing abstraction* goes the opposite route, and merges all the values that can be observed in all the cells of the array into a single set, before applying a value abstraction. This abstraction is depicted in Figure 6.1(c). We remark that this abstraction is much less precise than



(a) Concrete states (b) Extension abstraction (c) Collapsing abstraction

Figure 6.1: Basic array abstractions.

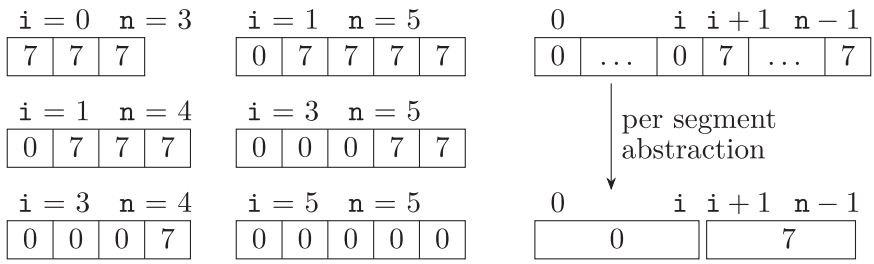
the previous one, as it does not allow to say that the cells of indexes 0 and 1 store value 0, and that the cell of index 3 stores value 3. On the other hand, it results in more scalable analyses and still applies even when the array size cannot be determined statically.

Array Abstractions Based on Dynamic Summaries. Neither of the two abstractions described in the previous paragraph can abstract precisely arrays of unbounded size. As an example, we consider the Java function shown in Figure 6.2 that performs the creation and initialization of an array of integers. It takes two arguments `n` and `c` that respectively fix the length of the array and the initial value to be stored in each cell. The Java semantics states that the array allocation performed at line 2 produces an array all the cells of which are initialized to zero. The initialization of each cell to `c` takes place in the loop.

To verify the correctness of this program, we need to infer a non trivial loop invariant. Indeed, at the beginning of the i-th iteration, the first i cells of the array store value `c`, whereas the cells beyond that position still store zeroes. A few such states are shown in Figure 6.3(a), under the assumption that `c` $= 7$, and for various values of `n`. When the execution exits the loop, `i` is equal to the length of the array, which implies that all cells contain value `c`. The extension abstraction of Figure 6.1(b) does not apply here, since the length of the array is unknown statically, and could be any positive integer (note that even enriching it with disjunctions would not work, as we would still need infinitely many case splits). The collapsing abstraction of Figure 6.1(c) also fails, since it abstracts together the two groups of cells mentioned above, and can thus only express that any cell in the array stores either

```
1 int [ ] create( int n, int c ){
2     int [ ] t = new int[n];
3     for( int i = 0; i < n; i ++ ){
4         t[i] = c;
5     }
6     return t;
7 }
```

Figure 6.2: An array creation and initialization program.



(a) Concrete states (b) Loop invariant

Figure 6.3: Array abstraction with dynamic summaries.

0 or c, which is not sufficient to express the loop invariant required to verify the correctness of the program.

To achieve this verification task, we need to a use an abstraction that is both expressive enough to deal with arrays of unknown length, and able to *segment* the array into two parts the size of which depends on the executions. This is very similar to what shape abstractions achieve. The principle of this per-segment abstraction is depicted in Figure 6.3(b), and relies on the partitioning of the array into two zones, that respectively span over the already updated cells, and the cells left to update. The size of each zone varies depending on the concrete state, but the general shape invariant holds true for all stores observed at the loop head point. Unlike the shape abstractions of Section 4, all the cells of the array occupy a contiguous memory region, that can be accessed using numerical indexes. However, the abstraction also utilizes *summary array predicates* to describe array zones of unknown and unbounded size. Furthermore, it also needs to partition an unbounded size memory area (the array) in a way that depends on the memory states that are considered, and on the executions.

There exist several instances of array abstractions based on this principle. As an example, Halbwachs and Péron (2008) relies on a semantic pre-analysis to choose array segments, and then utilizes the abstraction shown in Figure 6.3(b) so as to prove properties such as sorting. Moreover, Cousot *et al.* (2011) uses a similar array abstraction, but infers both array segments as part of the analysis that utilizes them, which makes this approach more precise than the previous one.

The difference between these two analyses is rather similar to the dividing line found in Subsection 4.1, between static analyses with fixed summaries and shape analyses that infer summaries. The analysis proposed in Liu and Rival (2015) and Liu *et al.* (2018) uses a variant of the abstraction of Cousot *et al.* (2011), where the partition of the array consists of groups of cells that are not necessarily contiguous. It also infers the array partition as part of the array analysis.

Example Array Analysis Using Dynamic Partitions. We now discuss the main static analysis algorithms to support an abstraction of arrays with summary predicates. In fact, these are very close the shape analysis algorithms that were presented in Section 2.

First, post-condition for operations such as the reading of the update of an array cell are easy to compute when that cell is fully exposed (i.e., not accounted for by a summary predicate). This means, that the computation of a post-condition for the statement $\mathtt{t}[\mathtt{i}] = \mathtt{c}$ boils down to the update of the value stored in $\mathtt{t}[\mathtt{i}]$ provided this cell is described precisely in the abstract memory.

Second, let us consider the computation of a post-condition for such a statement, in the case where the cell that is read or updated is described by a summary predicate. For instance, this situation arises at line 4, in the program shown in Figure 6.2: the statement $\mathtt{t}[\mathtt{i}] = \mathtt{c}$ operates on a cell that is described by a segment of unknown length, depicted in Figure 6.3(b). Attempting to analyze it directly would result in a week-update, and the loss of any precise information over the whole segment $[\mathtt{i}, \mathtt{n} - 1]$. The solution is similar to that used in shape analysis: the summary predicate should first be *refined* before a strong update can be achieved on a single cell. Indeed, we can turn the summary predicate over $[\mathtt{i}, \mathtt{n} - 1]$ into the conjunction of two separate predicates, over the cell of index $\mathtt{i}$, and over the segment $[\mathtt{i} + 1, \mathtt{n} - 1]$. Then, the assignment operates over a precisely known, single cell array region.

Last, the inference of summary predicates proceeds by the *generalization* over abstract states. Let us intuitively show how this process works, on the example program of Figure 6.3(b). Before the loop is executed, the whole array is initialized to zero and $\mathtt{i}$ is equal to 0. After the first iteration, the first cell stores value $\mathtt{c}$, whereas the rest is still

equal to zero, and `i` is equal to 1. At this stage, the abstract state of Figure 6.3(b) can be produced by generalization, as it says all cells of index ranging from 0 to `i`-1 are set to `c`, whereas the other cells are still set to 0.

6.2 Abstraction of Dictionary Structures

In this subsection, we consider *dictionary* data structures, that map unsorted keys into values. Like arrays, they consist of a set of fields that can be accessed from a same base address. However, they present several important differences: first, the set of indexes is not ordered; second, this set is most often not static, so that it is possible to add or remove keys at any time of the execution of programs. On one hand, the impossibility to do arithmetic operations over indexes somewhat simplifies their semantics. On the other hand, it also means that the notions of segments presented in Subsection 6.1 does not apply here. Besides, the extensibility property of these structures significantly complicates reasoning. Moreover, keys are often designated by strings, that can be computed dynamically, which is in fact even harder to deal with than numerical indexes.

In practice, dictionary data structures are either native features of the programming languages, as is the case of *open objects* in JavaScript, or implemented in standard libraries. In the latter case, they are typically implemented as standard containers such as balanced trees or hash-tables. Practical uses for dictionary data structures range from the storage of collections of homogeneous data such as data-bases (then the elements usually all have the same type) to the description of higher-level programming languages constructs such as objects (in the sense of object oriented programming). This entails that they analysis of programs using such structures cannot rely on a single solution, and that a wide range of abstractions may be used depending on the applications.

Basic Abstractions for Dictionaries and Open Objects. Basic abstractions similar to those previously applied to arrays may be used to abstract memory states containing dictionary structures as well.

In particular, the *extension abstraction*, where all the elements that are present are enumerated explicitly applies in a straightforward manner, when the set of keys is finite.

The case of the *collapsing abstraction*, where a single piece of abstract information is used to represent all the elements in the structure, is more subtle: indeed, unlike arrays, dictionary structures have a dynamic set of keys, and abstract states need to characterize this set precisely as well. The most simple way to achieve that is to describe a dictionary with two elements:

1. an abstraction of the set of keys in the dictionary;
2. an abstraction of the set of values associated to these keys.

Dictionary Abstractions Based on Summaries. The limitations of these basic abstractions are the same as in the case of arrays. The extension abstraction fails whenever the set of keys that may appear in dictionaries is not bounded within a finite set. The collapsing abstraction makes it hard to precisely describe the updates to individual fields.

To overcome these issues, a similar form of summarization is required as in the case of array abstractions. More precisely, a precise analysis of programs that manipulate dictionary data structures should be able to express constraints over unbounded sets of keys/contained elements, and to single out an entry being read or written. As we have shown in Section 4, this is exactly what all shape analyses aim at, thus dictionary data structures make similar kinds of abstractions necessary. In the following paragraphs, we comment on two such abstractions.

The abstraction proposed by Dillig *et al.* (2011) expresses constraints over the set of keys and over the mapping from keys to contained values, so as to abstract dictionary data structures. This abstraction is agnostic to the representation of the dictionary data structures that are considered. It also relies on a graph-based representation.

The abstraction proposed by Cox *et al.* (2014) partitions JavaScript open objects in disjoint regions, and maintains constraints over the sets of keys corresponding to each region. It is thus very similar to shape analyses based on separation logic (Reynolds, 2002): the analysis of

updates requires the refinement of abstract states to single out fields, and the analysis of loops is based on a widening, the purpose of which is to generalize partitions and set constraints. Furthermore, this analysis also supports attribute trackers (Cox *et al.*, 2015) in order to identify properties that were copied from one object to another and reinforce the initial abstraction with relations across distinct objects.

7

Conclusion

The main contribution of this survey is to identify the common characterizing features of a wide range of shape analyses and static analyses inspired by them.

Characterization. Among other static analyses that compute semantic properties of pointer manipulating programs, shape analyses can be characterized by their ability to *partition* heaps containing data structures of unbounded size based on *semantic criteria* and to *summarize* them.

Summarization is essential in order to describe precisely structures that can be arbitrarily large, such as variants of lists, trees, and graphs. It deeply impacts the abstraction underlying shape analyses, as it requires specific and expressive summary predicates. Summary predicates may describe structural properties either as a whole (e.g., "this structure is a list"), or using a collection of local predicates (e.g., "this structure is connected, acyclic, and made of a elements reachable from a common root").

The inference of summary predicates is very challenging for several reasons. First, it requires to identify the memory footprint of each summary predicate (i.e., which relation can be characterized by it). Second, it requires to select the appropriate predicate. A fundamental characteristic of shape analyses is to achieve this based on information available in the abstract states themselves. Intuitively, the decision to introduce a summary predicate that abstracts a well-formed singly-linked list should emerge from the other abstract information. Therefore, shape analyses should be more robust than syntactic analyses, or semantic analyses that exploit mainly syntactic information such as allocation sites.

On the other hand, this increased expressiveness comes at the cost of sophisticated analysis algorithms that are able to perform static analysis on complex abstract states, containing summary predicates. Intuitively, the two main analysis operations respectively *refine summary predicates into more precise descriptions* and *synthesize summary predicates by generalization*. The refinement of summary predicates should take place whenever analyzing some operations (such as reads and writes) that affect a memory region described by a summary predicate, and that cannot be reasoned over in a precise manner at the summary level. On the other hand, the synthesis of summary predicates aims at inferring higher level abstract information, at keeping abstract states finite (even though they describe unbounded regions), and enforcing termination of the analysis. While shape analyses typically proceed like fairly standard abstract interpretation based analyses, their design and implementation depends heavily on the way these two fundamental operations are carried out. Refinement typically occurs in operators for the computation of abstract post-conditions, whereas generalization may be performed either step-by-step, or as a widening operation, or both.

We observe that these fundamental features characterize not only shape analyses based on radically different abstractions, but also static analyses aimed at non recursive data structures. The main shape analyses aimed at recursive data structures utilize reachability predicates in three valued logic or some fragment of separation logic (possibly with a different representation of abstract predicates, based on graphs or automata). Similar ideas (including summarizing abstractions and

analysis algorithms based on refinement and generalization of summary predicates) allow to analyze precisely programs manipulating non-recursive structures such as arrays or dictionaries (like, for instance, open objects).

Perspectives. Even though shape analysis has received a lot of attention from the research community and a large number of results already achieved, many outstanding and important research problems remain unsatisfactorily resolved as of today. In the following, we list a few salient such issues, even though one could identify many more open questions.

First, shape analyses are heavily based on inductive summary predicates, and perform a large amount of reasoning over inductive properties. This incurs a number of difficulties that are not fully resolved as of now. In particular, static analyses may need to compare inductive predicates of different forms, which is still hard in general (even outside the specific context of a shape analysis issue). As an example, there exist many ways to define the property that a list is well-formed and sorted for some order over its elements. The comparison of such definitions is undecidable in general.

A second issue is the very wide spectrum of combined data structures. While existing shape analyses deal reasonably well with basic structures such as lists or trees, there is still a gap to close in order to handle the full complexity of memory states constructed by complex programs such as operating systems. When basic structures are mixed together with several levels of nesting and overlaying, or are combined with non recursive structures such as arrays, memory states are well-beyond what existing shape analyses can compute reasonably well.

Programming languages that lack a static typing discipline such as Python or JavaScript raise a third range of problems. This complicates the inference of shapes, since it makes it harder to even identify what data structures programmers actually intend to build. On the other hand, shape analysis is adapted to such issues, since it naturally tries to use the semantic information that it computes to select summaries in order to describe memory regions.

More generally, as static analysis techniques are increasingly popular, and are applied to a wide range of programming languages of various paradigms (object oriented, functional, ...), we believe that shape abstractions are very relevant and that one can expect novel shape analysis research problems to emerge.

References

Abdulla, P. A., L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar (2016). "Verification of heap manipulating programs with ordered data by extended forest automata". *Acta Informatica*. 53(4): 357–385.

Andersen, L. O. (1994). "Program analysis and specialization for the C programming language". *PhD thesis*. DIKU, University of Copenhagen.

Balaban, I., A. Pnueli, and L. D. Zuck (2007). "Shape analysis of single-parent heaps". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Cook and A. Podelski. Vol. 4349. *Lecture Notes in Computer Science*. Springer. 91–105.

Berdine, J., C. Calcagno, and P. W. O'Hearn (2005a). "Smallfoot: Modular automatic assertion checking with separation logic". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1–4, Revised Lectures*. 115–137.

Berdine, J., C. Calcagno, and P. W. O'Hearn (2005b). "Symbolic execution with separation logic". In: *APLAS*. Springer. 52–68.

Berdine, J., C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang (2007). "Shape analysis for composite data structures". In: *International Conference on Computer Aided Verification (CAV)*. Springer. 178–192.

Berdine, J., T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv (2008). "Thread quantification for concurrent shape analysis". In: *International Conference on Computer Aided Verification (CAV)*. Ed. by A. Gupta and S. Malik. Vol. 5123. *Lecture Notes in Computer Science*. Springer. 399–413.

Berdine, J., B. Cook, and S. Ishtiaq (2011). "Slayer: Memory safety for systems-level code". In: *Computer Aided Verification – 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14–20, Proceedings*. 178–183.

Bouajjani, A., M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar (2006). "Programs with lists are counter automata". In: *International Conference on Computer Aided Verification (CAV)*. 517–531.

Bouajjani, A., C. Drăgoi, C. Enea, and M. Sighireanu (2011). "On inter-procedural analysis of programs with lists and data". In: *Conference on Programming Languages Design and Implementation (PLDI)*. 578–589.

Bouajjani, A., C. Drăgoi, C. Enea, and M. Sighireanu (2012). "Abstract domains for automated reasoning about list-manipulating programs with infinite data". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 1–22.

Brockschmidt, M., Y. Chen, P. Kohli, S. Krishna, and D. Tarlow (2017). "Learning shape analysis". In: *Static Analysis Symposium (SAS)*. Ed. by F. Ranzato. Vol. 10422. *Lecture Notes in Computer Science*. Springer. 66–87.

Calcagno, C. and D. Distefano (2011). "Infer: An automatic program verifier for memory safety of C programs". In: *NASA Formal Methods – Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, Proceedings*. 459–465.

Calcagno, C., D. Distefano, P. W. O'Hearn, and H. Yang (2006). "Beyond reachability: Shape abstraction in the presence of pointer arithmetic". In: *Static Analysis Symposium (SAS)*. Springer. 182–203.

Calcagno, C., D. Distefano, P. W. O'Hearn, and H. Yang (2007). "Footprint analysis: A shape analysis that discovers preconditions". In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22–24, Proceedings*. 402–418.

Calcagno, C., D. Distefano, P. W. O'Hearn, and H. Yang (2009). "Compositional shape analysis by means of bi-abduction". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Savannah, GA, USA, January 21–23. 289–300.

Chang, B.-Y. E. and X. Rival (2008). "Relational inductive shape analysis". In: *Symposium on Principles of Programming Languages (POPL)*. ACM. 247–260.

Chang, B.-Y. E. and X. Rival (2013). "Modular construction of shape-numeric analyzers". In: *Semantics, Abstract Interpretation, and Reasoning About Programs: Essays Dedicated to David A. Schmidt on the Occasion of His Sixtieth Birthday, Manhattan, KS, USA, September 19–20*. 161–185.

Chang, B.-Y. E., X. Rival, and G. Necula (2007). "Shape analysis with structural invariant checkers". In: *Static Analysis Symposium (SAS)*. Springer. 384–401.

Chase, D. R., M. Wegman, and F. K. Zadeck (1990). "Analysis of pointers and structures". In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. PLDI '90*. New York, USA: ACM. 296–310.

Chin, W.-N., C. David, H. H. Nguyen, and S. Qin (2007). "Automated verification of shape, size and bag properties". In: *12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*. 307–320.

Cooper, K. D. and K. Kennedy (1989). "Fast interprocedural alias analysis". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM. 49–59.

Cousot, P. and R. Cousot (1977). "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Symposium on Principles of Programming Languages (POPL)*. ACM. 238–252.

Cousot, P. and R. Cousot (1979). "Systematic design of program analysis frameworks". In: *Symposium on Principles of Programming Languages (POPL)*. 269–282.

Cousot, P. and N. Halbwachs (1978). "Automatic discovery of linear restraints among variables of a program". In: *Symposium on Principles of Programming Languages (POPL)*. Tucson, AZ: ACM. 84–97.

Cousot, P., R. Cousot, and F. Logozzo (2011). "A parametric segmentation functor for fully automatic and scalable array content analysis". In: *POPL*. Austin, TX, USA: ACM. 105–118.

Cox, A., B.-Y. E. Chang, and X. Rival (2014). "Automatic analysis of open objects in dynamic language programs". In: *Static Analysis Symposium (SAS)*. Springer. 134–150.

Cox, A., B.-Y. E. Chang, and X. Rival (2015). "Desynchronized multi-state abstractions for open programs in dynamic languages". In: *European Symposium on Programming (ESOP)*. Springer. 483–509.

Deutsch, A. (1994). "Interprocedural may-alias analysis for pointers: Beyond k-limiting". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM. 230–241.

Dillig, I., T. Dillig, and A. Aiken (2011). "Precise reasoning for programs using containers". In: *Symposium on Principles of Programming Languages (POPL)*. ACM. 187–200.

Distefano, D., P. O'Hearn, and H. Yang (2006). "A local shape analysis based on separation logic". In: *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 287–302.

Dor, N., J. Field, D. Gopan, T. Lev-Ami, A. Loginov, R. Manevich, G. Ramalingam, T. W. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, E. Yahav, and G. Yorsh (2005). "Automatic verification of strongly dynamic software systems". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10–13, Revised Selected Papers and Discussions*. 82–92.

Dudka, K., P. Peringer, and T. Vojnar (2011). "Predator: A practical tool for checking manipulation of dynamic data structures using separation logic". In: *International Conference on Computer Aided Verification (CAV)*. Springer. 372–378.

Ghiya, R. and L. J. Hendren (1996). "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C". In: *Symposium on Principles of Programming Languages (POPL)*. 1–15.

Gotsman, A., J. Berdine, and B. Cook (2006). "Interprocedural shape analysis with separated heap abstractions". In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29–31, Proceedings*. 240–260.

Graf, S. and H. Saïdi (1997). "Construction of abstract state graphs with PVS". In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22–25, Proceedings*. 72–83.

Habermehl, P., R. Iosif, and T. Vojnar (2006). "Automata-based verification of programs with tree updates". In: *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 350–364.

Habermehl, P., L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar (2011). "Forest automata for verification of heap manipulation". In: *International Conference on Computer Aided Verification (CAV)*. Springer. 424–440.

Habermehl, P., L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar (2012). "Forest automata for verification of heap manipulation". *Formal Methods in System Design (FMSD)*. 41(1): 83–106.

Halbwachs, N. and M. Péron (2008). "Discovering properties about arrays in simple programs". In: *PLDI*. Tucson, AZ, USA: ACM. 339–348.

Holík, L., O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar (2013). "Fully automated shape analysis based on forest automata". In: *International Conference on Computer Aided Verification (CAV)*. Springer. 740–755.

Illous, H., M. Lemerre, and X. Rival (2017). "A relational shape abstract domain". In: *NASA Formal Methods – 9th International Symposium, NFM 2017*. Vol. 10227. *Lecture Notes in Computer Science*. Springer. 212–229.

Iosif, R., A. Rogalewicz, and T. Vojnar (2014). "Deciding entailments in inductive separation logic with tree automata". In: *Automated Technology for Verification and Analysis (ATVA)*. Springer. 201–218.

Ishtiaq, S. S. and P. W. O'Hearn (2001). "BI as an assertion language for mutable data structures". In: *Symposium on Principles of Programming Languages (POPL)*. ACM. 14–26.

Itzhaky, S., A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv (2013). "Effectively-propositional reasoning about reachability in linked data structures". In: *International Conference on Computer Aided Verification (CAV)*. Ed. by N. Sharygina and H. Veith. Vol. 8044. *Lecture Notes in Computer Science*. Springer. 756–772.

Jeannet, B., A. Loginov, T. W. Reps, and S. Sagiv (2004). "A relational approach to interprocedural shape analysis". In: *Static Analysis Symposium (SAS)*. 246–264.

Jeannet, B., A. Loginov, T. W. Reps, and M. Sagiv (2010). "A relational approach to interprocedural shape analysis". *ACM Trans. Program. Lang. Syst.* 32(2): 5:1–5:52.

Jones, N. D. and S. S. Muchnick (1979). "Flow analysis and optimization of LISP-like structures". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA: ACM. 244–256. URL: http://doi.acm.org/10.1145/567752.567776. Reprinted in *Program Flow Analysis: Theory and Application*, Muchnick, Steven S. and Jones, Neil D., 1981, published by Prentice Hall Professional Technical Reference.

Jones, N. D. and S. S. Muchnick (1982). "A flexible approach to interprocedural data flow analysis and programs with recursive data structures". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA: ACM. 66–74.

Jonkers, H. B. and H. B. M. Jonkers (1981). *Abstract Storage Structures*. Afdeling Informatica: IW. Afdeling Informatica, Mathematisch Centrum.

Kleene, S. C. (1952). *Introduction to Metamathematics*. Vol. 483. New York, NY: D. Van Nostrand Co., Inc.

Kreiker, J., H. Seidl, and V. Vojdani (2010). "Shape analysis of low-level C with overlapping structures". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer. 214–230.

Larus, J. R. and P. N. Hilfinger (1988). "Detecting conflicts between structure accesses". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI '88*. New York, NY, USA: ACM. 24–31.

Laviron, V., B.-Y. E. Chang, and X. Rival (2010). "Separating shape graphs". In: *European Symposium on Programming (ESOP)*. Springer. 387–406.

Le, Q. L., J. Sun, and W.-N. Chin (2016). "Satisfiability modulo heap-based programs". In: *International Conference on Computer Aided Verification (CAV)*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9779. *Lecture Notes in Computer Science*. Springer. 382–404.

Lee, O., H. Yang, and R. Petersen (2011). "Program analysis for overlaid data structures". In: *International Conference on Computer Aided Verification (CAV)*. 592–608.

Lev-Ami, T. and S. Sagiv (2000). "TVLA: A system for implementing static analyses". In: *Static Analysis Symposium (SAS)*. Springer. 280–301.

Li, H., X. Rival, and B.-Y. E. Chang (2015). "Shape analysis for unstructured sharing". In: *Static Analysis Symposium (SAS)*. 90–108.

Li, H., F. Berenger, B.-Y. E. Chang, and X. Rival (2017). "Semantic-directed clumping of disjunctive abstract states". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Paris, France, January 18–20. 32–45.

Liu, J. and X. Rival (2015). "Abstraction of arrays based on non contiguous partitions". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Mumbai, India: Springer. 282–299.

Liu, J., L. Chen, and X. Rival (2018). "Automatic verification of embedded system code manipulating dynamic structures stored in contiguous regions". In: *International Conference on Embedded Software (EMSOFT)*.

Loginov, A., T. W. Reps, and S. Sagiv (2005). "Abstraction refinement via inductive learning". In: *International Conference on Computer Aided Verification (CAV)*. Ed. by K. Etessami and S. K. Rajamani. Vol. 3576. *Lecture Notes in Computer Science*. Springer. 519–533.

Madhusudan, P., G. Parlato, and X. Qiu (2011). "Decidable logics combining heap structures and data". In: *Symposium on Principles of Programming Languages (POPL)*. Ed. by T. Ball and M. Sagiv. ACM. 611–622.

Magill, S., J. Berdine, E. M. Clarke, and B. Cook (2007). "Arithmetic strengthening for shape analysis". In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22–24, Proceedings*. 419–436.

Magill, S., M.-H. Tsai, P. Lee, and Y.-K. Tsay (2010). "Automatic numeric abstractions for heap-manipulating programs". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, January 17–23. 211–222.

Manevich, R., T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine (2008). "Heap decomposition for concurrent shape analysis". In: *Static Analysis Symposium (SAS)*. Ed. by M. Alpuente and G. Vidal. Vol. 5079. *Lecture Notes in Computer Science*. Springer. 363–377.

Manevich, R., B. Dogadov, and N. Rinetzky (2016). "From shape analysis to termination analysis in linear time". In: *Computer Aided Verification – 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, Proceedings, Part I*. 426–446.

Might, M. (2010). "Shape analysis in the absence of pointers and structure". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer. 263–278.

Miné, A. (2006). "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics". In: *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM. 54–63.

Nelson, G. (1983). "Verifying reachability invariants of linked structures". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. New York, NY, USA: ACM. 38–47.

O'Hearn, P. W. and D. J. Pym (1999). "The logic of bunched implications". *Bulletin of Symbolic Logic*. 5(2): 215–244.

O'Hearn, P. W., J. C. Reynolds, and H. Yang (2001). "Local reasoning about programs that alter data structures". In: *International Conference on Computer Science Logics (CSL)*. 1–19.

Pek, E., X. Qiu, and P. Madhusudan (2014). "Natural proofs for data structure manipulation in C using separation logic". In: *Conference on Programming Languages Design and Implementation (PLDI)*. Ed. by M. F. P. O'Boyle and K. Pingali. ACM. 440–451.

Piskac, R., T. Wies, and D. Zufferey (2013). "Automating separation logic using SMT". In: *International Conference on Computer Aided Verification (CAV)*. Ed. by N. Sharygina and H. Veith. Vol. 8044. *Lecture Notes in Computer Science*. Springer. 773–789.

Podelski, A. and T. Wies (2005). "Boolean heaps". In: *Static Analysis Symposium (SAS)*. Ed. by C. Hankin and I. Siveroni. Vol. 3672. *Lecture Notes in Computer Science*. Springer. 268–283.

Qiu, X., P. Garg, A. Stefanescu, and P. Madhusudan (2013). "Natural proofs for structure, data, and separation". In: *Conference on Programming Languages Design and Implementation (PLDI)*. Ed. by H.-J. Boehm and C. Flanagan. ACM. 231–242.

Reps, T. W., S. Horwitz, and M. Sagiv (1995). "Precise interprocedural dataflow analysis via graph reachability". In: *Symposium on Principles of Programming Languages (POPL)*. ACM. 49–61.

Reps, T., M. Sagiv, and A. Loginov (2003). "Finite differencing of logical formulas for static analysis". In: *European Symposium on Programming (ESOP)*. 380–398.

Reynolds, J. C. (1968). "Automatic computation of data set definitions". In: *Information Processing 68: Proceedings of the IFIP Congress*. North-Holland. 296–310.

Reynolds, J. C. (2002). "Separation logic: A logic for shared mutable data structures". In: *Symposium on Logics in Computer Science (LICS)*. IEEE. 55–74.

Rinetzky, N. and S. Sagiv (2001). "Interprocedural shape analysis for recursive programs". In: *International Conference on Compiler Construction (CC)*. Springer. 133–149.

Rinetzky, N., J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm (2005a). "A semantics for procedure local heaps and its abstractions". In: *Symposium on Principles of Programming Languages (POPL)*. 296–309.

Rinetzky, N., M. Sagiv, and E. Yahav (2005b). "Interprocedural shape analysis for cutpoint-free programs". In: *Static Analysis Symposium (SAS)*. 284–302.

Rival, X. and B.-Y. E. Chang (2011). "Calling context abstraction with shapes". In: *Symposium on Principles of Programming Languages (POPL)*. ACM. 173–186.

Sagiv, M., T. Reps, and R. Wilhelm (1998). "Solving shape-analysis problems in languages with destructive updating". *ACM Trans. Program. Lang. Syst.* 20(1): 1–50.

Sagiv, S., T. Reps, and R. Wilhelm (1999). "Parametric shape analysis via 3-valued logic". In: *Symposium on Principles of Programming Languages (POPL)*. 105–118.

Sagiv, M., T. Reps, and R. Wilhelm (2002). "Parametric shape analysis via 3-valued logic". *Transactions on Programming Languages and Systems (TOPLAS)*. 24(3): 217–298.

Sharir, M. and A. Pnueli (1981). "Two approaches to interprocedural data flow analysis". In: *Program Flow Analysis: Theory and Applications*. Englewood Cliffs, NJ: Prentice-Hall, Inc. Chap. 7.

Shivers, O. (1991). "Control-flow analysis of higher-order languages". *PhD thesis*. Carnegie Mellon University.

Smaragdakis, Y. and G. Balatsouras (2015). "Pointer analysis". *Found. Trends Program. Lang.* 2(1): 1–69.

Sotin, P. and B. Jeannet (2011). "Precise interprocedural analysis in the presence of pointers to the stack". In: *European Symposium on Programming (ESOP)*. 459–479.

Sotin, P., B. Jeannet, and X. Rival (2010). "Concrete memory models for shape analysis". *Electronic Notes in Theoretical Computer Science*. 267(1): 139–150.

Srivastava, S., N. Immerman, and S. Zilberstein (2011). "A new representation and associated algorithms for generalized planning". *Artificial Intelligence*. 175(2): 615–647.

Steensgaard, B. (1996). "Points-to analysis in almost linear time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM. 32–41.

Stransky, J. (1992). "A lattice for abstract interpretation of dynamic (LISP-like) structures". *Inf. Comput.* 101(1): 70–102.

Toubhans, A., B.-Y. E. Chang, and X. Rival (2013). "Reduced product combination of abstract domains for shapes". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 375–395.

Toubhans, A., B.-Y. E. Chang, and X. Rival (2014). "An abstract domain combinator for separately conjoining memory abstractions". In: *Static Analysis Symposium (SAS)*. 285–301.

Vafeiadis, V. (2010). "Automatically proving linearizability". In: *International Conference on Computer Aided Verification (CAV)*. Ed. by T. Touili, B. Cook, and P. B. Jackson. Vol. 6174. *Lecture Notes in Computer Science*. Springer. 450–464.

Yang, H. (2007). "Towards shape analysis for device drivers". In: *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14–16, Proceedings*. 267.

Yang, H., O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn (2008). "Scalable shape analysis for systems code". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7–14, Proceedings*. 385–398.