TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

Raymond and Beverly Sackler Faculty of Exact Sciences
The Blavatnik School of Computer Science

PARTIALLY DISJUNCTIVE SHAPE ANALYSIS

Thesis submitted for the degree of Doctor of Philosophy

by Roman Manevich

This work was carried out under the supervision of

Prof. Mooly Sagiv

and the consultation of

Dr. Ganesan Ramalingam

Submitted to the Senate of Tel-Aviv University

February 2009

Abstract

Partially Disjunctive Shape Analysis

Roman Manevich
Doctor of Philosophy
The Blavatnik School of Computer Science
Tel-Aviv University

Modern programs rely significantly on the use of dynamically-allocated linked data structures. Shape analysis algorithms statically analyze a program to determine information about these data structures, e.g, “does variable x point to an acyclic list?” and “is it possible to reach an object via pointer traversals from two different variables?” These algorithms are conservative (sound), that is, the discovered information is true for every program input, and thus can be applied for various uses, such as program verification, optimization, parallelization, etc.

Disjunctive shape analyses operate by abstracting the concrete memory stores into (bounded) *shape graphs*. At control flow join points of the program, the shape graphs are merged by using disjunction (set union), which often leads to an exponential explosion in the number of shape graphs. In concurrent programs this problem is even more acute, due to the interleaving of different threads.

We present new “partially disjunctive” shape analyses aimed at taming the size of the state space by abstracting disjunctions, as well as soundly approximating program statements. We implemented and applied these analyses to prove properties of sequential programs and fine-grained concurrent programs. We were able to prove a variety of challenging properties, including cleanness properties, shape invariants, and linearizability of concurrent data structure implementations. The new shape analyses scale better than the disjunctive shape analyses, usually running faster by orders of magnitude, and still able to prove the desired properties.

Acknowledgements

First and foremost, I would like to express my deep gratitude and appreciation to my advisor, Prof. Mooly sagiv. I thank him for having the patience required to educate me and the instincts to provide me with the right kind of education at the right moments, during my notable period of research with him. His energy and constant belief in me helped me to keep going whereas otherwise I would have given up. He also taught me that a good researcher should always keep looking and understanding problems in new and better ways.

I have been fortunate to have Dr. Ganesan Ramalingam as my second, unofficial, advisor. Ramalingam was always there when I needed someone to listen to new ideas, provide an objective opinion, and show me how to improve them and present them in clearer ways.

I thank my collaborators on papers on which this thesis is based: Mooly Sagiv, Ganesan Ramalingam, John Field, Eran Yahav, Josh Berdine, Byron Cook, and Tal Lev-Ami [MSRF04, MYRS05, MBC+07, MLAS+08].

I enjoyed two fruitful and educational internships at MSR Redmond. For the first one, during the summer of 2003, I thank Manuvir Das, Stephen Adams, Zhe Yang, and Manu Sridharan. For the second one, during the summer of 2005, I thank Tom Ball, Shuvendu Lahiri, and Shaz Qadeer.

During the summer of 2006 I had an amazing internship at MSR India. I thank Ramalingam and Sriram Rajamani's group there for an amazing internship during the summer of 2006. I had such a great environment to work at and discuss my ideas with members of the lab, and experience India.

I thank Tom Henzinger for interesting discussions and joint work during my visit to EPFL.

I thank Thomas Reps, Reinhard Wilhelm, and their students for many discussions, including my visit to the University of Wisconsin and meetings in Dagstuhl seminars.

I thank Mooly's programming languages group at Tel Aviv University for being such a wonderful combination of research colleagues and friends: Ran Shaham, Noam Rinetzky, Greta Yorsh, Gilad Arnold, Nurit Dor, Daphna Amit, Ohad Shacham, and Michal Segalov. I thank Nir Andelman for many interesting discussions and moral support.

I also thank Anat Lotan, Gilit Zohar-Oren, and the anonymous referees who commented on earlier drafts of this thesis as well as on papers upon which this thesis is based.

Finally, I thank my beloved family for their love and support.

The research in this thesis was support in part by generous financial support of the Israeli Clore Programme and the Israeli Academy of Sciences.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we are interested in statically inferring properties of programs manipulating linked data structures, which is referred to as *shape analysis* and is a special case of *abstract interpretation* [CC77]. The main application of shape analysis in this thesis is verification of safety properties, including absence of null dereferences, absence of memory leaks, data structure invariants, absence of concurrent modification exceptions in Java, and checking *linearizability* of concurrent data structures.

Requisites and Theoretical Backgrounds. A requisite for understanding the material in this thesis is familiarity with static program analysis via abstract interpretation (for an introduction to abstract interpretation see Nielson et al. [NNH99]), including understanding of the following concepts: *abstraction*, *concretization*, *abstract domains*, *abstract transformers*, and inference of program invariants by *chaotic iteration* (fixed-point computation). We next include a brief reminder to these concepts. We also include the necessary parts of the theory of parametric shape analysis via 3-valued logic [SRW02] in Chapter 2. In this framework, concrete states and abstract states are represented by logical structures, which can be thought of as directed graphs with multiple types of edges and Boolean properties associated with the vertices.

Abstract Interpretation Essentials. In this thesis, we will assume that an abstract domain A is given by a complete lattice $D_A = \langle \sqsubseteq_A, \sqcup_A, \sqcap_A, \bot_A, \top_A \rangle$ where A is the set of elements; $\sqsubseteq_A$ is a partial ordering on the elements; $\sqcup_A$ is the least upper bound, or *join*, operator; $\sqcap_A$ is the greatest lower bound, or *meet*, operator; $\bot_A$ is the least element of the lattice; and $\top_A$ is the greatest element of the lattice. We say that an element c_1 is more *precise* than an element c_2 when $c_1 \sqsubseteq c_2$.

In abstract interpretation [CC77], an abstraction function $\alpha^{C,A} : C \rightarrow A$ maps an element of the concrete domain C to the most precise element that represents it in the *abstract domain* A. The meaning of an abstract element $a \in A$ is given by a *concretization* function $\gamma^{A,C} : A \rightarrow C$. That is, we say that $a \in A$ represents any element $c \in C$ such that $c\gamma^{A,C}(a)$. Moreover, the pair $(\gamma^{A,C}, \alpha^{C,A})$ forms a *Galois Connection*.

In the sequel, we will drop the subscripts and superscripts denoting the semantic domains when no confusion is likely.

A semantic function $F^\sharp : A \to A$ is a *sound* over-approximation of a semantic function $F : C \to C$ if the following holds:

$$F(\gamma(a)) \sqsubseteq \gamma(F^\sharp(a)) \ .$$

We call the function F the concrete transformer and the function $F^\sharp$ the abstract transformer. In this thesis, we will often be interested in over-approximating the meaning of a program statement $[\![st]\!]$ over a finite abstract domain A.

The semantics of a program is given in terms of a least fixed point $\mathit{lfp}(F)$ and its abstract semantics is given by $\mathit{lfp}(F^\sharp)$[1]. Properties of a program can be conservatively inferred by starting from an initial element a_0 and then applying the function $F^\sharp$ over and over until reaching the fixed point. This process is guaranteed to end when the height of the lattice A is finite.

Finally, we note that two abstract domains A_1 and A_2 may be equivalent, i.e., isomorphic, offering different encodings of the same information. That is, for every concrete element $c \in C$, we have $\gamma^{A_1,C}(\alpha^{C,A_1}(c)) = \gamma^{A_2,C}(\alpha^{C,A_2}(c))$.

Disjunctive Abstractions vs. Partially Disjunctive Abstractions. An abstract domain A (and the corresponding abstraction) is said to be *disjunctive* when the following holds for every two abstract elements $a_1, a_2 \in A$:

$$\gamma(a_1) \sqcup_C \gamma(a_2) = \gamma(a_1 \sqcup_A a_2) \ .$$

Otherwise (when $\gamma(a_1) \sqcup_C \gamma(a_2) \sqsubset \gamma(a_1 \sqcup_A a_2)$ is possible), the domain is said to be *partially disjunctive*.

We now compare the two forms of abstraction:

- Disjunctive abstractions are popular in the model checking community, where an abstraction is defined by finitely partitioning the set of concrete states. Partially disjunctive abstractions are more general, since they allow defining an abstraction in terms of overlapping sets of states (closed under join and meet).

- Disjunctive abstract domains are powersets of the set of equivalence classes of the concrete domain, with respect to A. Thus, they can be quite expensive for static analyses. For instance, when a static analyzer interprets control flow join points, the size of the joined element can be double the size of each of the elements in the branches. In contrast, a partially disjunctive domain can over-approximate the elements from two branches in a such a way that the size remains feasible (especially in terms of the computer representation) and the fixed point computation terminates faster. Of course, the abstract element computed in this way might be less precise and thus less useful for, e.g., program verification. Therefore, partially disjunctive abstractions have to be chosen carefully to support two opposite needs — taming the cost of the analysis and providing information that is precise enough for the goals of the analysis.

[1] In actuality, the fixed-point is usually taken for the sequence of iterates defined by $X_0 = a_0$ and $X_{n+1} = X_n \sqcup F^\#(X_n)$

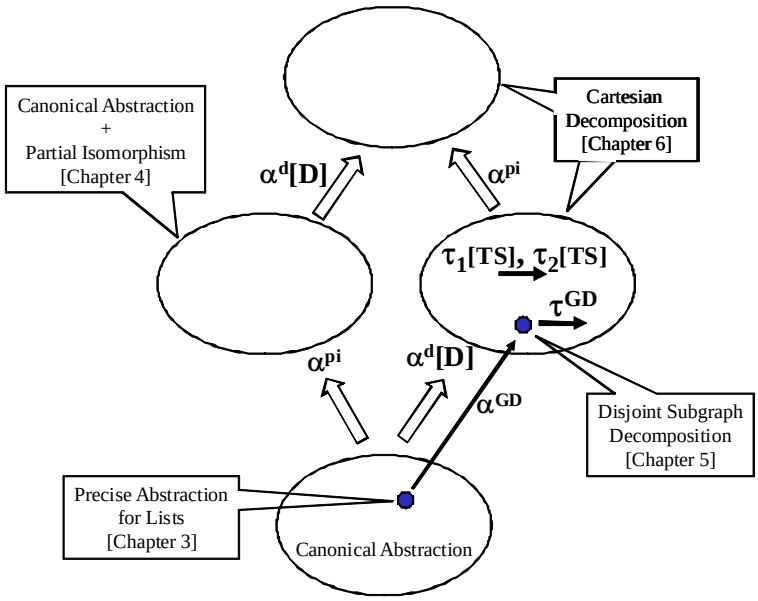


Figure 1.1: Abstractions and transformers developed in thesis

Goals. Our goal in this thesis is to: (i) find precise new abstractions for linked data structures; and (ii) find new partially disjunctive abstractions and efficient transformers that can be used to analyze heap-manipulating programs with similar precision as analysis using disjunctive abstractions, but with better performance. In particular, we seek partially disjunctive abstractions that enable the analysis designer to intentionally abstract away information that he considers to be irrelevant for proving a certain property (e.g., abstracting away the correlation between the properties of disjoint lists to prove absence of null dereferences).

1.1 Overview

The material in this thesis is based on four conference papers [MSRF04, MYRS05, MBC+07, MLAS+08], each one with its corresponding technical chapter. This chapter contains an informal overview of the thesis, describing the contributions of each of the papers, and the connections between them.

Figure 1.1 illustrates the various abstractions and transformer algorithms developed in this thesis. The concrete domain, which is not shown in the figure, is either a powerset of 2-valued structures (in chapters 3,4, and 6) or a powerset of shape graphs (in Chapter 5).

1.1.1 A Precise Abstraction for Singly-Linked Lists

In chapter 3, we introduce a rather precise disjunctive abstraction for programs containing a finite number of (possibly cyclic) lists (of unbounded length). These ideas have been adapted and extended in subsequent works of other researchers [APV06, APV08, LAIS06].

Abstract Domain Encoding. We show how to encode the abstract elements in Predicate Abstraction [GS97] and in Canonical Abstraction [SRW02] and prove these encodings are equiv-

alent in the sense defined above. (In Chapter 5, we use a more direct, specialized, encoding of the abstract elements by shape graphs.)

Application. We use this abstraction to prove basic safety properties and data structure invariants in standard list-manipulating procedures.

Main Contributions: (i) We define an instance of Canonical Abstraction [SRW02] that abstracts cyclic lists more precisely than existing instances based on Canonical Abstraction; (ii) we compare Predicate Abstraction and Canonical Abstraction in terms of the number of predicates needed to encode the same abstraction and show the equivalence of the two encodings for the list abstraction; and (iii) we report on an empirical evaluation of an analysis based on the new abstraction on a suite of benchmarks.

1.1.2 Partial Isomorphism Abstraction

In Chapter 4, we introduce a partially disjunctive abstraction on top of Canonical Abstraction. The idea is to use an equivalence relation on structures, using *partial isomorphism*, to define a notion of similarity between them. We merge similar structures into a single structure, thus reducing the number of structures after a join. We do not merge structures that are not similar, since we consider them to be distinguished by properties that may be important for the analysis.

Abstract Domain Encoding. We build on top of the 3-valued shape analysis theory and encode the abstract elements by 3-valued structures. This enables us to define a very general abstraction and reuse the abstract transformers available in that framework.

Application. We applied the analysis to a wide variety of benchmarks, which were defined over the years by different users of the TVLA system [LAS00], including sequential and concurrent benchmarks, checking list- and tree-manipulating procedures, and checking concurrent modification exception in Java.

Main Contributions: (i) A generic type of abstraction (variations of this idea were adapted by other researchers [YLB+08]); (ii) a robust implementation in TVLA; and (iii) empirical evaluation on a wide variety of benchmarks showing dramatic speed-ups to the performance of TVLA.

1.1.3 Disjoint Subgraph Decomposition

In Chapter 5, we define a partially disjunctive abstraction for exploiting loose coupling between different, disjoint, data structures. This abstraction can reduce exponential factors in analysis of programs manipulating multiple disjoint data structures.

Abstract Domain Encoding. We formulate the concrete states and the abstract states using specialized shape graphs. This is done for the simplicity of the presentation. However, recasting the results in terms of logical structures is straightforward.

Application. We have implemented and applied the analysis to programs manipulating multiple cyclic singly-linked lists, including programs modeled after windows device drivers.

Main Contributions: (i) We introduce a new type of shape abstraction that exploits disjointness of data structures to reduce the height of the abstract domain; (ii) we study the complexity of the abstract transformers and show that the most precise transformers are NP-complete; (iii) we propose polynomial, efficient, transformers (τ^{GD}) that are less precise than the most precise transformer but are usually good in practice, that is, the resulting analysis is as precise as the one based on disjunctive abstraction for most benchmarks; and (iv) we have implemented and showed significant speed-ups of the analysis on a set of benchmarks manipulating lists.

1.1.4 Cartesian Subheap Decomposition

In Chapter 6, we introduce a framework for constructing partially disjunctive abstractions, based on the idea of decomposing logical structures into sub-structures and using Cartesian abstraction. A user of the framework can specify different kinds of decompositions and different kinds of transformers, ensured to result in a sound analysis.

Abstract Domain Encoding. We present the ideas using concrete stores. The algorithms are incorporated into TVLA and use logical structures. This enables a very generic system.

Application. We applied the ideas to analyze concurrent fine-grained programs manipulating list data structures to prove basic safety properties (e.g., absence of null dereferences) and linearizability [HW90], by building on top of the analysis of Amit et al. [ARR+].

Main Contributions: (i) We define the concept of heap decomposition and show how it can be utilized with Cartesian abstraction; (ii) we develop techniques for sound and efficient transformers ($\tau_1[TS], \tau_2[TS]$) that can be parameterized by the analysis designer; (iii) the algorithms are incorporated into TVLA where an analysis designer can specify a decomposition parameter D and a transformer parameter *TS* and automatically obtain a sound analysis to experiment with; and (iv) we show the usefulness of the framework for analyzing concurrent fine-grained programs and checking linearizability. We have shown that using these techniques is prudent for efficiently checking linearizability for program with an unbounded number of threads [BLAM+08].

Cartesian decomposition abstraction can capture disjoint subgraph decomposition and the efficient transformers developed in Chapter 5.

The partial isomorphism abstraction, α^{pi}, and the Cartesian decomposition abstractions, $\alpha^{d}[D]$, can be easily combined. We use both of these abstractions to obtain the analyses and the results reported in Chapter 6 and in a subsequent work [BLAM+08].

1.2 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 contains background materials for Canonical Abstraction [SRW02];
- Chapter 3 presents a finitary abstraction for stores containing a bounded number of singly-linked lists (of unbounded length) and describes how to encode the abstraction using different formalisms;
- Chapter 4 presents a partially disjunctive abstraction on top of Canonical Abstraction based on merging similar abstract states and its application to a wide variety of analyses;
- Chapter 5 presents a partially disjunctive abstraction based on decomposing (abstract) heaps into their sets of disjoint subheaps and its application to analyzing sequential programs;
- Chapter 6 presents a parametric framework for partially disjunctive abstractions based on decomposing (abstract) heaps into sets of — not-necessarily disjoint — subheaps, and its application to analyzing sequential and concurrent programs;
- Chapter 7 concludes the thesis and discusses possible future research directions.

Chapter 2

3-valued Shape Analysis Background

In this section, we provide a brief introduction into the theory of parametric shape analysis via 3-valued logic [SRW02] and, in particular, define Canonical Abstraction.

2.1 Concrete Program States

We represent the state of a program using a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects.

Definition 2.1.1 *A 2-valued logical structure over a vocabulary (set of predicates) $\mathcal{P}$ is a pair $S = \langle U^S, \iota^S \rangle$ where U^S is the universe of the 2-valued structure, and ι^S is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) : U^{S^k} \to \{0, 1\}$.*

In the following, we use $p^S(v)$ as alternative notation for $\iota^S(p)(v)$; we also omit the superscript S, when no confusion is likely.

In the context of shape analysis, a logical structure is used as a shape descriptor, with each individual corresponding to a heap-allocated object and predicates of the structure corresponding to properties of heap-allocated objects.

We denote the set of all 2-valued logical structures over a set of predicates $\mathcal{P}$ by 2-STRUCT$_{\mathcal{P}}$. In the sequel, we assume that the vocabulary $\mathcal{P}$ is fixed, and abbreviate 2-STRUCT$_{\mathcal{P}}$ to 2-STRUCT.

Table 4.1 shows the predicates we typically use to record properties of individuals. A unary predicate $x(v)$ holds when the object v is pointed-to by the reference variable x. We assume

Table 2.1: Typical predicates used for representing concrete program states

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	v_1 is equal to v_2
$\{\, x(v) : x \in PVar \,\}$	reference variable x points to the object v
$n(v_1, v_2)$	next field of the object v_1 points to the object v_2

Table 2.2: Predicate-update formulae that define the semantics of heap-manipulating statements

Statement	Update formulae
`x = null`	$x'(v) = 0$
`x = t`	$x'(v) = t(v)$
`x = t.n`	$x'(v) = \exists v_1 : t(v_1) \wedge n(v_1, v)$
`x.n = null`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1)$
`x.n = t` (assuming x.n == null)	$n'(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge t(v_2))$

that the set of predicates includes a unary predicate for every reference variable in a program. We use *PVar* to denote the set of all reference variables in a program. A binary predicate $n(v_1, v_2)$ records the value of the reference field `n`.

2.1.1 Concrete Semantics

Program statements are modelled by *actions* that specify how statements transform an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the predicates in the outgoing structure using formulae of first-order logic with transitive closure over the incoming structure [SRW02]. The update formulae for heap-manipulating statements are shown in Table 3.2. For brevity, we omit the treatment of the allocation statement `new T()`, the interested reader may find the details in [SRW02].

To simplify update formulae, we assume that every assignment to the `n` field of an object is preceded by first assigning null to it.

2.2 Canonical Abstraction

The goal of an abstraction is to create a finite representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [SRW02], which extends boolean logic by introducing a third value $1/2$ denoting values that may be 0 or 1.

We represent an abstract state of a program using a 3-valued first-order structure.

Definition 2.2.1 *A 3-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = \langle U, \iota \rangle$ where U is the universe of the 3-valued structure (an individual in U may represent multiple heap-allocated objects), and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota(p) : U^k \rightarrow \{0, 1, 1/2\}$.*

An abstract state may include summary nodes, *i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. A summary node u has $eq(u, u) = 1/2$, indicating that it may represent more than a single individual. In the rest of the thesis, we assume that the set of predicates P includes a distinguished unary predicate* sm *to indicate if an individual is a summary individual.*

Table 2.3: Predicates used for the Canonical Abstraction of singly-linked lists in [SRW02], and their meaning

Predicates	**Intended Meaning**	**Defining formulae**
$\{ x(v) : x \in PVar \}$	reference variable x points to v	
$n(u, v)$	next field of u points to v	
$\{ r_x(v) : x \in PVar \}$	v is reachable from x by dereferencing n fields	$\exists v_x . x(v_x) \wedge n^*(v_x, v)$
$c_n(v)$	v resides on a cycle of n fields	$n^+(v, v)$
$is(v)$	v is heap-shared	$\exists v_1, v_2 . n(v_1, v) \wedge n(v_2, v) \wedge (v_1 \neq v_2)$

2.2.1 Embedding

We now formally define how states are represented using abstract states. The idea is that each individual from the (concrete) state is mapped into an individual in the abstract state. More generally, it is possible to map individuals from an abstract state into an individual in another, less precise, abstract state.

Formally, let $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$ be abstract states. A function $f : U \to U'$ such that f is surjective is said to *embed* S *into* S' if for each predicate p of arity k, and for each $u_1, \ldots, u_k \in U$, one of the following holds:

$$\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \quad \text{or} \quad \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2 \quad .$$

We say that S' *represents* S when there exists such an embedding f.

One way of creating an embedding function f is by using *Canonical Abstraction*. Canonical Abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual.

Table 3.3 presents the set of predicates used in [SRW02] to abstract singly-linked lists. The predicates $r_x(v)$, $c_n(v)$, and $is(v)$, referred to in [SRW02] as *instrumentation predicates*, record derived information and are used to refine the abstraction.

Chapter 3

A Precise Abstraction for Singly-Linked Lists

Predicate abstraction and canonical abstraction are two finitary abstractions used to prove properties of programs. We study the relationship between these two abstractions by considering a very limited case: abstraction of (potentially cyclic) singly-linked lists.

We provide a new and rather precise family of abstractions for potentially cyclic singly-linked lists. The main observation behind this family of abstractions is that the number of shared nodes in linked lists can be statically bounded. Therefore, the number of possible "heap shapes" is also bounded. We present the new abstraction in both predicate abstraction form as well as in canonical abstraction form.

As we illustrate in the chapter, given any canonical abstraction, it is possible to define a predicate abstraction that is equivalent to the canonical abstraction. However, with this straightforward simulation, the number of predicates used for the predicate abstraction is exponential in the number of predicates used by the canonical abstraction.

An important feature of the family of abstractions we present in this chapter is that the predicate abstraction representation we define is far more practical as it uses a number of predicates that is quadratic in the number of predicates used by the corresponding canonical abstraction representation. In particular, for the most abstract abstraction in this family, the number of predicates used by the canonical abstraction is linear in the number of program variables, while the number of predicates used by the predicate abstraction is quadratic in the number of program variables.

We have encoded this particular predicate abstraction and corresponding transformers in TVLA, and used this implementation to successfully verify safety properties of several list manipulating programs, including programs that were not previously verified using predicate abstraction or canonical abstraction.

3.1 Introduction

Abstraction and abstract interpretation [CC79] are essential techniques for automatically proving properties of programs. The main challenge in abstract interpretation is to develop abstractions that are precise enough to prove the required property and efficient enough to be applicable to realistic applications.

Predicate abstraction [GS97] abstracts the program into a Boolean program which conservatively simulates all potential executions. Every safety property which holds for the Boolean program is guaranteed to hold for the original program. Furthermore, abstraction refinement [CGJ+00, BR02] can be used to refine the abstraction when the analysis produces a "false alarm". When the process terminates, it yields a concrete error trace in which the property is violated, or successfully verifies the property. In principle, the whole process can be fully mechanized given a sufficiently powerful theorem prover. This process was successfully used in SLAM [Mic01] and BLAST [HJMS02] to prove safety properties of device drivers.

Canonical abstraction [SRW02] is a finitary abstraction that was specially developed to model properties of unbounded memory locations (inspired by [JM81b]). This abstraction has been implemented in TVLA [LAS00], and successfully used to prove various properties of heap-manipulating programs (e.g., [RWF+02, YR04, SYKS03]).

3.1.1 Main Results

In this chapter, we study the utility of predicate abstraction to prove properties of programs operating on singly-linked lists. We also compare the expressive power of predicate abstraction and canonical abstraction.

The results in this chapter can be summarized as follows:

- We show that current state-of-the-art iterative refinement techniques fail to prove interesting properties of singly-linked lists such as pointer equalities and absence of null dereferences in a fully automatic manner. This means that on many simple programs the process of refinement will diverge when the program is correct. This result is inline with the experience of Blanchet et al. [BCC+03].

- We show that predicate abstraction can simulate arbitrary finitary abstractions and, in particular, canonical abstraction. This trivial result is not immediately useful because of the number of predicates used. The number of predicates required to simulate canonical abstraction is, in the worst case, exponential in the number of predicates used by the canonical abstraction (usually, this means exponential in the number of program variables).

- We develop a new family of abstractions for heaps containing (potentially cyclic) singly-linked lists. The main idea is to summarize list elements on unshared list segments not pointed-to by local variables. For programs manipulating singly-linked lists, this abstraction is finitary since the number of shared list elements reachable from program variables is bounded. Abstractions in this family vary in their level of precision, which is controlled by the level of sharing-relationships recorded.

- We show that the abstraction recording only one-level sharing relationships (i.e., the least precise member of the family that records sharing) is sufficient for successfully verifying all our example programs, including programs that were not verified earlier using predicate abstraction or canonical abstraction.

- We show how to code the one-level-sharing abstraction using both canonical abstraction (with a linear number of unary predicates) and predicate abstraction (with a quadratic number of nullary predicates).

```
      //head points to the first element of an acyclic list
      //tail points to the last element of the same list
1     curr = head;
2     while (curr != tail) {
3        assert (curr != null);
4        curr = curr.n;
5     }
```

Figure 3.1: A simple program on which counterexample-guided refinement diverges

3.1.2 Motivating Examples

Figure 3.1 shows a program that traverses a singly-linked list with a head-pointer `head` and a tail-pointer `tail`. This is a trivial program since it only uses an acyclic linked list, and does not contain destructive pointer updates. When counterexample-guided iterative refinement is applied to this program to assure that the assertion at line 3 is never violated, it will diverge. At the i-th iteration it will generate an assertion of the form $\texttt{curr}(\texttt{.n})^i! = \texttt{null}$. However, no finite value of i will suffice. Indeed, the problem of proving the absence of null-dereferences is undecidable even in programs manipulating singly-linked lists and even under the (non-realistic) assumption that all control flow paths are executable [Cha03].

In contrast, the TVLA abstract interpreter [LAS00] proves the absence of null dereferences in this program in 2 seconds, consuming 0.6MB of memory. TVLA uses canonical abstraction which generalizes predicate abstraction by allowing first-order predicates (relation symbols) that can have arguments. Thus, nullary (0-arity) predicates correspond to predicates in the program and in predicate abstractions. Unary predicates (1-arity) are used to denote sets of unbounded locations and binary (2-arity) predicates are used to denote relationships between unbounded locations.

A curious reader may ask herself: *Are there program properties that can be verified with canonical abstractions but not with predicate abstractions?*

It is not hard to see that the answer is negative, since any finitary abstraction can be simulated by a suitable predicate abstraction. For example, consider an abstraction mapping $\alpha : C \rightarrow A$, from a concrete domain C to a finite abstract domain of indexed elements $A = \{1, \ldots, n\}$. Define the predicate $\text{BIT}[j]$ to hold for the set of concrete states $\{c \mid \text{the } j\text{th bit of } \alpha(c), \text{ in its binary representation, is } 1\}$. Now, the set of predicates $\{\text{BIT}[j]\}_{j=1}^{\lceil \log n \rceil}$ yields a predicate abstraction that simulates A. This simulation is usually not realistic, since it contains too many predicates. The number of predicates required by predicate abstraction to simulate canonical abstraction can be exponential in the number of predicates used by the canonical abstraction.

Fortunately, the only nullary predicate crucial to prove the absence of null dereferences in this program is the fact that `tail` is reachable from `curr` by a path of `n` selectors (of some length). Similar observations were suggested independently in [JJNS97, BRS99, IO01]. In this chapter, we define a quadratic set of nullary predicates that captures the invariants in many programs manipulating (potentially cyclic) singly-linked lists.

Figure 3.2 shows a simple program removing a contiguous segment from a cyclic singly-linked list pointed-to by `x`. For this example program, we would like to verify that the resulting

```
      // x points to a cyclic singly-linked list
      // low and high are two integer values, low < high
1     t = null;
2     y = x;
3     while (t != x && y.data < low) {
4        t = y.n; y = t;
5     }
6     z = y;
7     while (z != x && z.data < high) {
8        t = z.n; z = t;
9     }
10    t = null;
11    if (y != z) {
12       y.n = null;
13       y.n = z;
14    }
```

Figure 3.2: A simple program that removes the segment between low and high from a linked list

structure pointed-to by x remains a cyclic singly-linked list. Unfortunately, using TVLA's canonical abstraction with the standard set of predicates turns out to be insufficient. The problem stems from the fact that canonical abstraction with the standard set of predicates loses the ordering between the 3 reference variables that point to that cyclic singly-linked list (this is further explained in the next section).

In this chapter, we provide two abstractions — a predicate abstraction, and a canonical abstraction — that are able to correctly determine that the result of this program is indeed a cyclic singly-linked list.

3.1.3 Outline

The rest of this chapter is organized as follows. Section 3.2 provides background on the basic concrete semantics we are using, Canonical Abstraction, and Predicate Abstraction. Section 3.3 presents an instrumented concrete semantics that records list interruptions. Section 3.4 shows a quite precise predicate abstraction for singly-linked lists. Section 3.5 shows a quite precise canonical abstraction of singly-linked lists. In Section 3.6, we show that the predicate abstraction of Section 3.4 and the canonical abstraction of Section 3.5 are equivalent. Section 3.7 describes our experimental results.

3.2 Background

In this section, we provide basic definitions that we will use throughout the chapter. In particular, we provide a reminder of Canonical Abstraction with some examples, and we define Predicate Abstraction.

Table 3.1: Predicates used for representing concrete program states

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	v_1 is equal to v_2
$\{\, x(v) : x \in \mathit{PVar} \,\}$	reference variable `x` points to the object v
$n(v_1, v_2)$	next field of the object v_1 points to the object v_2

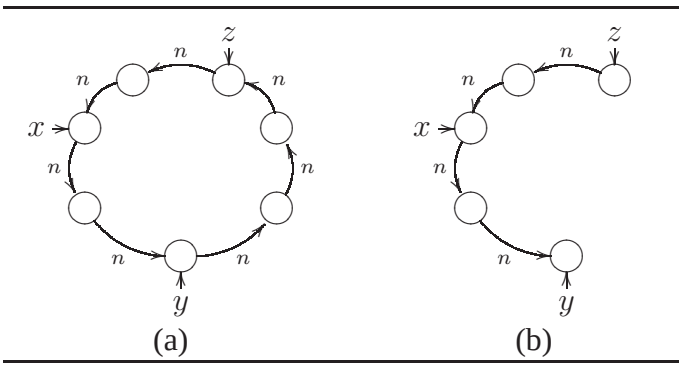


Figure 3.3: The effect of the statement `y.n=null` in the concrete semantics. (a) a possible state of the program of Figure 3.2 at line 12; (b) the result of applying `y.n=null` to (a)

3.2.1 Concrete Program States

In our setting, we represent the state of a program using a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects.

Table 4.1 shows the predicates we use to record properties of individuals. A unary predicate $x(v)$ holds when the object v is pointed-to by the reference variable `x`. We assume that the set of predicates includes a unary predicate for every reference variable in a program. We use *PVar* to denote the set of all reference variables in a program. A binary predicate $n(v_1, v_2)$ records the value of the reference field `n`.

Concrete Semantics

Recall that the semantics of program statements over 2-valued structures is modelled by *actions* that specify how statements transform an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the predicates in the outgoing structure using formulae of first-order logic with transitive closure over the incoming structure [SRW02]. The update formulae for heap-manipulating statements are shown in Table 3.2. For brevity, we omit the treatment of the allocation statement `new T()`, the interested reader may find the details in [SRW02].

To simplify update formulae, we assume that every assignment to the `n` field of an object is preceded by first assigning null to it. Therefore, the statement at line 12 of the example program of Figure 3.2 assigns null to `y.n` before the next statement assigns it the new value `z`.

Table 3.2: Predicate-update formulae that define the semantics of heap-manipulating statements

Statement	**Update formulae**
`x = null`	$x'(v) = 0$
`x = t`	$x'(v) = t(v)$
`x = t.n`	$x'(v) = \exists v_1 : t(v_1) \wedge n(v_1, v)$
`x.n = null`	$n'(v_1, v_2) = n(v_1, v_2) \wedge \neg x(v_1)$
`x.n = t` (assuming x.n == null)	$n'(v_1, v_2) = n(v_1, v_2) \vee (x(v_1) \wedge t(v_2))$

Table 3.3: Predicates used for the Canonical Abstraction in Figure 3.4, and their meaning

Predicates	**Intended Meaning**	**Defining formulae**
$\{ x(v) : x \in PVar \}$	reference variable x points to v	
$n(u, v)$	next field of u points to v	
$\{ r_x(v) : x \in PVar \}$	v is reachable from x by dereferencing n fields	$\exists v_x . x(v_x) \wedge n^*(v_x, v)$
$c_n(v)$	v resides on a cycle of n fields	$n^+(v, v)$
$is(v)$	v is heap-shared	$\exists v_1, v_2 . n(v_1, v) \wedge n(v_2, v) \wedge (v_1 \neq v_2)$

Example 3.2.1 *Applying the action* `y.n = null` *to the concrete structure of Figure 3.3(a), results with the concrete structure of Figure 3.3(b). Throughout this chapter we assume that all heaps are garbage-free, i.e., every element is reachable from some program variable, and that the concrete program semantics reclaims garbage elements immediately after executing program statements. Thus, the two objects between* `y` *and* `z` *are collected when* `y.n` *is set to null, as they become unreachable.*

3.2.2 Canonical Abstraction

The goal of an abstraction is to create a finite representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size. The abstractions we use are based on 3-valued logic [SRW02], which extends Boolean logic by introducing a third value $1/2$ denoting values that may be 0 or 1.

We represent abstract states of a program using a 3-valued first-order structures.

Table 3.3 presents the set of predicates used in [SRW02] to abstract singly-linked lists. The predicates $r_x(v)$, $c_n(v)$, and $is(v)$, referred to in [SRW02] as *instrumentation predicates*, record derived information and are used to refine the abstraction.

This set of predicates has been used for successfully verifying many programs manipulating singly-linked lists, but is insufficient for verifying that the output of the example program of Figure 3.2 is a cyclic singly-linked list pointed-to by x.

Example 3.2.2 *Figure 3.4(b) shows the Canonical Abstraction of the concrete state of Figure 3.4(a), using the predicates of Table 3.3. The node with double-line boundaries is a* summary

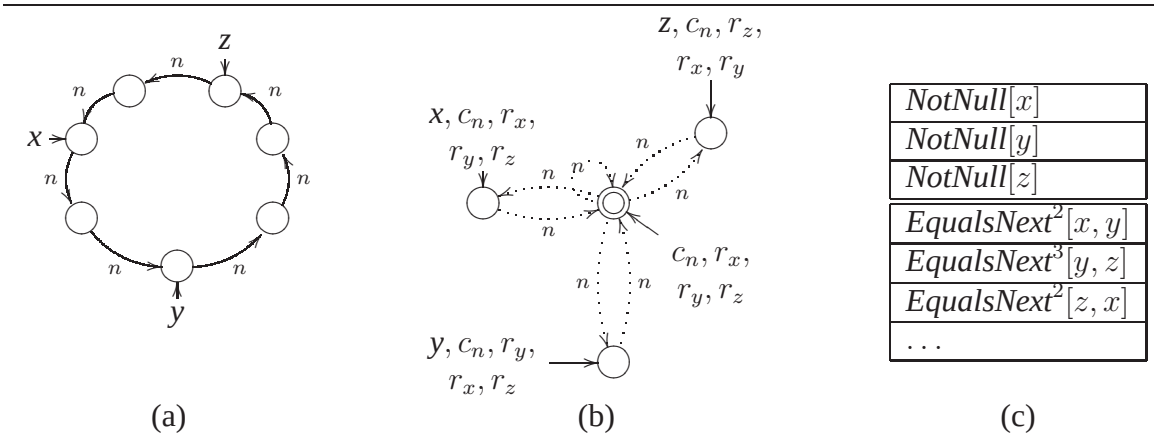


Figure 3.4: (a) a concrete possible state of the program of Figure 3.2 at line 12, (b) its canonical abstraction in TVLA, (c) its predicate abstraction with the set of predicates in Table 3.4

node, *possibly representing more than a single concrete node. The dashed edges are* $1/2$ *edges, a dashed edge exists between* v_1 *and* v_2 *when* $n(v_1, v_2) = 1/2$*. The abstract state of Figure 3.4(b) records the fact that* x,y*, and* z *point to a cyclic list (using the* $c_n(v)$ *predicate), and that all list elements are reachable from all* 3 *reference variables (using the* $r_x(v)$,$r_y(v)$*, and* $r_z(v)$ *predicates). This abstract state, however, does not record the order between the reference variables. In particular, it does not record that* x *does not reside between* y *and* z *(the segment that is about to be removed by the program statement at line* 12*). As a result, applying the abstract effect of* y.n=z *to this abstract state results with a possible abstract state in which the cyclic list is broken.*

3.2.3 Predicate Abstraction

Predicate Abstraction abstracts a concrete state into a truth-assignment for a finite set of propositional (nullary) predicates.

A Predicate Abstraction is defined by a vocabulary $P^A = \{P_1, \ldots, P_m\}$, where each P_i is associated with a defining formula φ_i that can be evaluated over concrete states. An abstract state is a truth assignment to the predicates in P^A. Given an abstract state A, we denote the value of P_i in A by A_i.

Let 2-STRUCT$[\mathcal{P}]$ denote the set of all 2-valued logical structures over the set of predicates $\mathcal{P}$. A concrete state S over a vocabulary P^C, is mapped to an abstract state A by an abstraction mapping β: 2-STRUCT$[\mathcal{P}^C] \rightarrow$ 2-STRUCT$[\mathcal{P}^A]$. The abstraction mapping evaluates the defining formulae of the predicates in $\mathcal{P}^A$ over S and sets the appropriate values to the respective predicates in A. Formally, for every $1 \leq i \leq m$, $A_i = [\![\varphi_i]\!]_2^S$.

Table 3.4 shows an example set of predicates similar to the ones used in [BMMR01,DN03].

Table 3.4: Predicates used for the Predicate Abstraction in Figure 3.4, and their meaning. Note that the maximal tracked length K is fixed a priori

Predicates	**Intended meaning**	**Defining formulae**
$\{$ *NotNull*$[x]$: $x \in$ *PVar* $\}$	x is not null	$\exists v_x.x(v_x)$
$\{$ *EqualsNext*$^k[x,y]$: $x, y \in$ *PVar*, $0 \leq k \leq K$ $\}$	the node pointed-to by y is reachable by k n fields from the node pointed-to by x	$\exists v_0, \ldots, v_k.x(v_0) \wedge y(v_k) \wedge \bigwedge_{0 \leq i < k} n(v_i, v_{i+1})$

Example 3.2.3 *Figure 3.4(c) shows the Predicate Abstraction of the concrete state shown in Figure 3.4(a) using the predicates of Table 3.4. A predicate of the form NotNull$[x]$ records the fact that* x *is not null. In Figure 3.4(c), all three variables* x*,*y*,and* z *are not null. A predicate of the form EqualsNext$^k[x,y]$ records that the node pointed-to by* y *is reachable by k steps over the* n *fields from the node pointed-to by* x *(Note that K, the maximal tracked length, is fixed a priori). For example, in Figure 3.4(c), the list element pointed-to by* y *is reachable from the list element pointed-to by* x *in 2 steps over the* n *field, and therefore EqualsNext$^2[x,y]$ holds.*

3.3 Recording List Interruptions

In this section, we instrument the concrete semantics to record a designated set of nodes, called *interruptions*, in singly-linked lists. The instrumented concrete semantics presented in this section serves as the basis for the predicate abstraction and the canonical abstraction presented in the following sections.

3.3.1 The Intuition

The intuition behind our instrumented concrete is that a garbage-free heap, containing only singly-linked lists, is characterized by two factors: (i) the "shape" of the heap, i.e., the connectivity relations between a set of designated nodes (interruptions); and (ii) the length of "simple" list segments connecting interruptions, but not containing interruptions themselves. This intuition is similar to proofs of small model properties (e.g., [RSY04]).

Considering this characterization, we observe that the number of shapes that are equivalent, up to lengths of simple list segments, is bounded. We therefore instrument our concrete semantics to record interruptions, which are an essential ingredient of the sharing patterns.

The abstractions presented in the next sections, abstract the lengths of simple list segments into a fixed set of abstract lengths (thereby obtaining a finite representation). These abstractions retain the general shape of the heap but lose any correlations between the actual lengths of different simple list segments. Our experience indicates that the correctness of program properties usually depends on the shape of heap, rather than on the lengths of simple list segments.

In the rest of this section, we formally define the notions of interruptions and simple list segments, and formally define the information recorded by our instrumented concrete semantics.

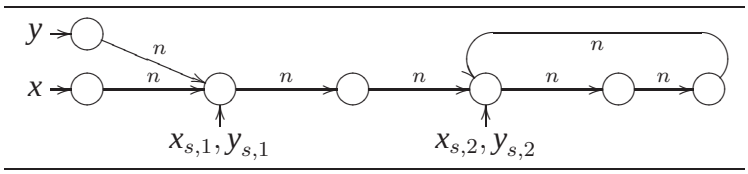


Figure 3.5: Two lists sharing the same tail, and their representation in the instrumented concrete semantics

3.3.2 Basic Definitions

We say that a list node v is an *interrupting node*, or simply an *interruption*, if it is pointed-to by a program variable or it is heap-shared. Figure 3.5 shows a heap with 4 interruptions: (i) the node pointed-to by x, (ii) the node pointed-to by y, (iii) the node pointed-to by $x_{s,1}$ and $y_{s,1}$, and (iv) the node pointed-to by $x_{s,2}$ and $y_{s,2}$.

Definition 3.3.1 (Uninterrupted Lists) *We say that there is an* uninterrupted list *between list node u and list node v, denoted by UList(u, v), when there is a non-empty path between them, such that, every node on the path between them (i.e., not including u and v) is non-interrupting.*

We also say that there is an uninterrupted list between list node v and null, denoted by UListNULL(v), when there is a non-empty path from v to null, such that, every node on the path, except possibly v, is non-interrupting.

Table 3.5 formulates UList(u, v) and UListNULL(v) as formulae in FO^{TC}.

Given a heap, we are actually interested in a subset of its uninterrupted lists. We say that an uninterrupted list is *maximal* when it is not contained in a longer uninterrupted list.

The heap in Figure 3.5 contains 4 maximal uninterrupted lists: (i) from the node pointed-to by x and the node pointed-to by $x_{s,1}$ and $y_{s,1}$, (ii) from the node pointed-to by y and the node pointed-to by $x_{s,1}$ and $y_{s,1}$, (iii) from the node pointed-to by $x_{s,1}$ and $y_{s,1}$ to the node pointed-to by $x_{s,2}$ and $y_{s,2}$, and (iv) from the node pointed-to by $x_{s,2}$ and $y_{s,2}$ to itself.

3.3.3 Statically Naming Heap-Shared Nodes

We now explain how to use a quadratic number of auxiliary variables to statically name all heap-shared nodes. This will allow us to name all maximal uninterrupted lists using nullary predicates for the predicate abstraction, and using unary predicates for the canonical abstraction.

Proposition 3.3.2 *A garbage-free heap, consisting of only singly-linked lists with n program variables, contains at most n heap-shared nodes and at most $2n$ interruptions.*

Corollary 3.3.3 *In a garbage-free heap, consisting of only singly-linked lists with n program variables, list node v is reachable from list node u if and only if it is reachable by a sequence of $k < n$ uninterrupted lists. Similarly, there is a path from node v to null if and only if there is a path from v to null by a sequence of $k < n$ uninterrupted lists.*

Table 3.5: Shorthand notations used throughout this chapter

Shorthand	**Meaning**	**Formula**
HeapShared(v)	v is heap-shared	$\exists a, b.n(a, v) \wedge n(b, v) \wedge (a \neq b)$
PtByVar(v)	v is pointed-to by some variable	$\bigvee_{var \in PVar} var(v)$
Interruption(v)	v is an interrupting list node	*HeapShared*$(v) \vee$ *PtByVar*(v)
UList$_1(u, v)$	there is an uninterrupted list of length 1 from u to v	$n(u, v)$
UList$_2(u, v)$	there is an uninterrupted list of length 2 from u to v	$\exists m.\neg$*Interruption*$(m) \wedge$ $n(u, m) \wedge n(m, v)$
UList$_{>2}(u, v)$	there is an uninterrupted list of length > 2 from u to v	$\exists m_1, m_2 : n(u, m_1) \wedge n(m_2, v) \wedge$ (TC $a, b : n(a, b) \wedge \neg$*Interruption*$(a) \wedge$ $\neg$*Interruption*$(b))(m_1, m_2)$
UList(u, v)	there is an uninterrupted list of some length from u to v	*UList*$_1(u, v) \vee$ *UList*$_2(u, v) \vee$ *UList*$_{>2}(u, v)$
UListNULL$_1(v)$	there is an uninterrupted list of length 1 from v to null	$\forall w.\neg n(v, w)$
UListNULL$_2(v)$	there is an uninterrupted list of length 2 from v to null	$\exists m.n(v, m) \wedge \neg$*Interruption*$(m) \wedge$ *UListNULL*$_1(m)$
UListNULL$_{>2}(v)$	there is an uninterrupted list of length > 2 from v to null	$\exists m_1, m_2 : n(v, m_1) \wedge$ *UListNULL*$_1(m_2)$ (TC $a, b : n(a, b) \wedge \neg$*Interruption*$(a) \wedge$ $\neg$*Interruption*$(b))(m_1, m_2)$
UListNULL(v)	there is a list of some length from v to null	*UListNULL*$_1(v) \vee$ *UListNULL*$_2(v) \vee$ *UListNULL*$_{>2}(v)$

Proof: By Proposition 3.3.2, every simple path (from u to v or from v to null) contains at most n interruptions, and, therefore, at most n maximal uninterrupted lists.

For every program variable *x*, we define a set of auxiliary variables $\{x_{s,k} | k = 1 \ldots n - 1\}$. Auxiliary variable $x_{s,k}$ points to a heap-shared node u when there exists a simple path consisting of k maximal uninterrupted lists from the node pointed by x-to to u, such that all of the interrupting nodes on the path are not pointed-to by program variables (i.e., they are heap-shared). Formally, we define the set of auxiliary variables derived for program variable *x* by using the following set of formulae in FO^{TC}.

$$\begin{aligned}
x_{s,1}(v) \equiv \quad & \exists v_x . x(v_x) \wedge \mathit{UList}(v_x, v) \wedge \mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v), \\
\ldots \\
x_{s,k+1}(v) \equiv \quad & \exists v_k . x_{s,k}(v_k) \wedge \mathit{UList}(v_k, v) \wedge \mathit{HeapShared}(v) \wedge \\
& \neg \mathit{PtByVar}(v) \wedge \neg (\bigvee_{m=1\ldots k} x_{s,m}(v)) \ .
\end{aligned}$$

We denote the set of auxiliary variables by *AuxVar* and the set of all (program and auxiliary) variables by $\mathit{Var} = \mathit{PVar} \cup \mathit{AuxVar}$.

Proposition 3.3.4 *Every heap-shared node is pointed-to by a variable in Var. Also,* $x_{s,k}(v)$ *holds for at most one node, for every reference variable* x *and* k*.*

3.3.4 Parameterizing the Concrete Semantics

Let n denote the number of (regular) program variables. Notice that $|AuxVar| = O(n^2)$. In the following sections, we will see that using the full set of auxiliary variables yields a canonical abstraction with a quadratic ($O(n^2)$) number of unary predicates, and a predicate abstraction with a bi-quadratic ($O(n^4)$) number of predicates.

We use a parameter k to define different subsets of *Var* as follows: $\mathit{Var}_k = \mathit{PVar} \cup \{x_{s,i}(v) | x \in \mathit{PVar}, i \leq k\}$. By varying the "heap-shared depth" parameter k, we are able to distinguish between different sets of heap-shared nodes. We discovered that, in practice, heap-shared nodes with depth > 1 rarely exist (they never appear in our examples), and, therefore, restricting k to 1 is usually enough to capture all maximal uninterrupted lists. Using Var_1 as the set of variables to record, we obtain a canonical abstraction with a linear number of unary predicates $(O(n))$ and a predicate abstraction with a quadratic $(O(n^2))$ number of variables.

Figure 3.5 shows a heap containing a heap-shared node of depth 2 (pointed by $x_{s,2}$ and $y_{s,2}$). By setting the heap-shared depth parameter k to 1, we are able to record the following facts about this heap: (i) there is a list of length 1 from the node pointed-to by *x* to a heap-shared node, (ii) there is a list of length 1 from the node pointed-to by *y* to a heap-shared node, (iii) the heap-shared node mentioned in (i) and (ii) is the same (we record aliasing between variables), and (iv) there is a partially cyclic list (i.e., a non-cyclic list connected to a cyclic list) from the heap-shared node mentioned in (iii). We know that the list from the first heap-shared node does not reach null (since we record lists from interruptions to null) and it is not a cycle from the first-heap shared node to itself (otherwise there would be no second heap-shared node and the cycle would be recorded). The information lost, due to the fact that $x_{s,2}$ and $y_{s,2}$ are not recorded, is that the list from the first heap-shared node to second has length 2 and the cycle from the second heap-shard node to itself is also of length 2.

Table 3.6: Predicates used for the Predicate Abstraction and their meaning

Predicates	**Defining formulae and intended meaning**
$\{\, \mathit{Aliased}[x, y] : x, y \in \mathit{Var} \,\}$	$\exists v : x(v) \wedge y(v)$ variables x and y point to the same object
$\{\, \mathit{UList}_1[x, y] : x, y \in \mathit{Var} \,\}$	$\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)$ the n field of the object pointed-to by x and the variable y point to the same object
$\{\, \mathit{UList}_2[x, y] : x, y \in \mathit{Var} \,\}$	$\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge \mathit{UList}_2(v_x, v_y)$ there is an uninterrupted list of length 2 from the object pointed-to by x to the object pointed-to by y
$\{\, \mathit{UList}[x, y] : x, y \in \mathit{Var} \,\}$	$\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge \mathit{UList}(v_x, v_y)$ there is an uninterrupted list of length 1 or more from the object pointed-to by x to the object pointed-to by y
$\{\, \mathit{UList}_1[x, \mathrm{null}] : x \in \mathit{Var} \,\}$	$\exists v_x : x(v_x) \wedge \mathit{UListNULL}_1(v_x)$ there n field of the object pointed-to by x points to null
$\{\, \mathit{UList}_2[x, \mathrm{null}] : x \in \mathit{Var} \,\}$	$\exists v_x : x(v_x) \wedge \mathit{UListNULL}_2(v_x)$ there is an uninterrupted list of length 2 from the object pointed-to by x to null
$\{\, \mathit{UList}[x, \mathrm{null}] : x \in \mathit{Var} \,\}$	$\exists v_x : x(v_x) \wedge \mathit{UListNULL}(v_x)$ there is an uninterrupted list of length 1 or more from the object pointed-to by x to null

The Instrumented Concrete Semantics

The instrumented concrete semantics operates by using the update formulae presented in Table 3.2 and then using the defining formulae of the auxiliary variables to update their values.

3.4 A Predicate Abstraction for Singly-Linked Lists

We now describe the abstraction used to create a finite (bounded) representation of a potentially unbounded set of 2-valued structures (representing heaps) of potentially unbounded size.

3.4.1 The Abstraction

We start by defining a vocabulary P^A of nullary predicates, which we use in our abstraction. The predicates are shown in Table 3.6.

Intuitively, the heap is partitioned into a linear number of uninterrupted list segments and each list segment is delimited by some variables. The predicates in Table 3.6 abstract the path length of list segments into one of the following abstract lengths: 0 (via the $\mathit{Aliased}[x, y]$ predicates), 1 (via the $\mathit{UList}_1[x, y]$ predicates), 2 (via the $\mathit{UList}_2[x, y]$ predicates), or any length ≥ 1 (via the $\mathit{UList}[x, y]$ predicates), and infinity (i.e., there is no uninterrupted path and thus all of the previously mentioned predicates are 0).

$Aliased[x,x], Aliased[y,y], Aliased[z,z]$
$UList_2[x,y], UList_2[z,x]$
$UList[x,y], UList[y,z], UList[z,x]$

(a)

$Aliased[x,x], Aliased[y,y], Aliased[z,z]$
$UList_1[y, \text{null}]$
$UList_2[x,y], UList_2[z,x]$
$UList[x,y], UList[z,x], UList[y, \text{null}]$

(b)

Figure 3.6: The abstract effect of `y.n=null` under Predicate Abstraction. (a) Predicate Abstraction of the state of Figure 3.3(a); (b) result of applying the abstract transformer of `y.n=null` to (a)

The abstraction function $\beta_{PredAbs}$: 2-STRUCT$[P^C] \rightarrow$ 2-STRUCT$[P^A]$ operates as described Section 3.2.3 where P^A is the set of predicates in Table 3.6.

Example 3.4.1 *Figure 3.6(a) shows an abstract state abstracting the concrete state of Figure 3.3(a). The predicates Aliased$[x,x]$,Aliased$[y,y]$, Aliased$[z,z]$ represent the fact that the reference variables* `x`*,* `y`*, and* `z` *are not null. The predicate UList$_2[x,y]$ represents the fact that there is an uninterrupted list of length exactly 2 from the object pointed-to by* `x` *to the object pointed-to by* `y`*. This adds on the information recorded by the predicate UList$[x,y]$, which represents the existence of a list of length 1 or more. Similarly, the predicate UList$_2[z,x]$ records the fact that a list of exactly length 2 exists from* `z` *to* `x`*. Note that the uninterrupted list between* `y` *and* `z` *is of length 3, a length that is abstracted away and recorded as a uninterrupted list of an arbitrary length by UList$[y,z]$.*

3.4.2 Abstract Semantics

Rabin [Rab69] showed that monadic second-order logic of theories with one function symbol is decidable. This immediately implies that first-order logic with transitive closure of singly-linked lists is decidable, and thus the best transformer can be computed as suggested in [RSY04]. Moreover, Rabin also proved that every satisfiable formula has a small model of limited size, which can be employed by the abstraction. For simplicity and efficiency, we directly define the abstractions and the abstract transformer. The reader is referred to [IRR+04] which shows that reasonable extensions of this logic become undecidable. We believe that our techniques can be employed even for undecidable logics but the precision may vary. In particular, the transformer we provide here is the *best transformer* and operates in polynomial time.

Example 3.4.2 *In order to simplify the definition of the transformer for* `y.n = null`*, we split it to 5 different cases (shown in Appendix A.1) based on classification of the next list interruption. The abstract state of Figure 3.6(a) falls into the case in which the next list interruption is a node pointed-to by some regular variable (*`z` *in this case) and not heap-shared (case 3).*

Table 3.7: Predicates used for the Canonical Abstraction and their meaning. We use the shorthand $\mathit{UList}(u, v)$ as defined in Definition 3.3.1

Predicates	**Intended Meaning**	**Defining Formulae**
$\{\, x(v) : x \in \mathit{Var} \,\}$	object v is pointed-to by x	
$\{\, \mathit{cul}[x](v) : x \in \mathit{Var} \,\}$	there exists an uninterrupted list to v, starting from the node pointed-to by x	$\exists v_x : x(v_x) \wedge \mathit{UList}(v_x, v)$

The update formulae for this case are the following:

$$
\begin{array}{lcl}
\mathit{UList}_1[z_1, z_2]' & = & \mathit{UList}_1[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y] \\
\mathit{UList}_1[z_1, \mathit{null}]' & = & \mathit{UList}_1[z_1, \mathit{null}] \vee \mathit{Aliased}[z_1, y] \\
\mathit{UList}_2[z_1, z_2]' & = & \mathit{UList}_2[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y] \\
\mathit{UList}[z_1, z_2]' & = & \mathit{UList}[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y] \\
\mathit{UList}[z_1, \mathit{null}]' & = & \mathit{UList}[z_1, \mathit{null}] \vee \mathit{Aliased}[z_1, y]
\end{array}
$$

Applying this update to the abstract state of Figure 3.6(a) yields the abstract state of Figure 3.6(b).

In Appendix A.1, we show that these formulae are produced by manual construction of the best transformer.

3.5 Canonical Abstraction for Singly-Linked Lists

In this section, we show how canonical abstraction, with an appropriate set of predicates, provides a rather precise abstraction for (potentially cyclic) singly-linked lists.

3.5.1 The Abstraction

As in Section 3.4, the idea is to partition the heap into a linear number of uninterrupted list segments, where each segment is delimited by a pair of variables (possibly including auxiliary variables). The predicates we use for canonical abstraction are shown in Table 3.7. The predicates of the form $\mathit{cul}[x](v)$, for $x \in \mathit{Var}$, record uninterrupted lists starting from the node pointed-to by x.

Example 3.5.1 *Figure 3.7(a) shows an abstract state abstracting the concrete state of Figure 3.3(a). The predicates* $\mathit{cul}[x](v)$,$\mathit{cul}[y](v)$, *and* $\mathit{cul}[z](v)$ *record uninterrupted list segments. Note that, in contrast to the abstract state of Figure 3.4(b) (which uses the standard TVLA predicates), the abstract configuration of Figure 3.7(a) records the order between the reference variables, and is therefore able to observe that* x *is not pointing to an object on the list from* y *to* z.

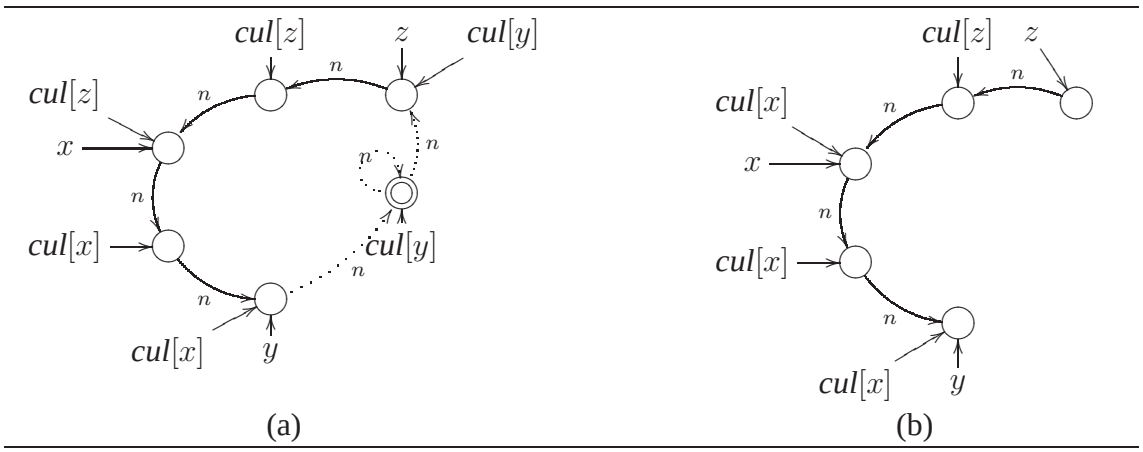


Figure 3.7: The abstract effect of `y.n=null` under Canonical Abstraction. (a) Canonical Abstraction of the state of Figure 3.3(a); (b) result of applying the abstract transformer of `y.n=null` to (a)

3.6 Discussion

Equivalence of the Canonical Abstraction and the Predicate Abstraction

We first show that the two abstractions — the Predicate Abstraction of Section 3.4, and the Canonical Abstraction of Section 3.5 — are equivalent. That is, both observe the same set of distinctions between concrete heaps.

Theorem 3.6.1 *The abstractions presented in Section 3.4 and in Section 3.5 are equivalent.*

*Proof:*See Appendix A.2.

The Number of Predicates Used by the Abstractions.

The next proposition shows that in fact only a logarithmic number of auxiliary variables is required for every regular program variable, in order to name all heap-shared nodes.

Proposition 3.6.2 *The* heap-sharing depth *in any heap is bounded from above by* $m = \lfloor \log n \rfloor + 1$. *In other words, auxiliary variables* $x_{s,k}$ *where* $k > m$ *never point to nodes.*

Proof: See Appendix A.3.□

Using Proposition 3.6.2, we can reduce the number of unary predicates needed for the Canonical Abstraction to $O(n \log n)$, and the number of predicates needed for the Predicate Abstraction to $O((n \log n)^2)$, without affecting precision.

In general, the number of predicates needed by a Predicate Abstraction to simulate a given Canonical Abstraction is exponential in the number of unary predicates used by the Canonical Abstraction. It is interesting to note that, in this case, we were able to simulate the Canonical Abstraction using a sub-exponential number of nullary predicates.

Recording Numerical Relationships

We believe that our abstractions can be generalized along the lines suggested by Deutsch in [Deu94], by capturing numerical relationships between list lengths. This will allow us to prove properties of programs which traverse correlated linked lists, while maintaining the ability to conduct strong updates, which could not be handled by Deutsch. Indeed, in [GDN+04] numerical and Canonical Abstractions were combined in order to handle such programs.

3.7 Experimental Results

We implemented in TVLA the analysis based on the predicates and abstract transformers described in Section 3.2.3. We applied it to verify various specifications of programs operating on lists, described in Table 5.3. For all examples, we checked the absence of null dereferences and memory leaks. For the running example and reverse_cyclic we also verified that the output list is cyclic and partially cyclic, respectively.

The experiments were conducted using TVLA version 2, running with SUN's JRE 1.4, on a laptop computer with a 796 MHZ Intel Pentium Processor with 256 MB RAM.

The results of the analysis are shown in Table 5.3. In all of the examples, the analysis produced no false alarms. In contrast, TVLA, with the abstraction predicates in Table 4.1, is unable to prove that the output of reverse_cyclic is a partially cyclic list and that the output of removeSegment is a cyclic list.

The dominating factor in the running times and memory consumption is the loading phase, in which the predicates and update formulae are created (and explicitly represented). For example, the time and space consumed during the chaotic iteration of the `merge` example is 8 seconds and 7.4 MB, respectively.

Table 3.8: Time, space and number of errors measurements. Rep. Err. is the number of errors reported by the analysis, and Act. Err. is the number of real errors

Benchmark	**Description**	**Time (sec)**	**Space (MB)**	**Rep. Err./ Act. Err.**
create	Dynamically allocates a new linked list	3	1.8	0/0
delete	Removes an element from a list	7	9.1	0/0
deleteAll	Deallocates a list	3	2.7	0/0
getLast	Retrieves the last element in a list	4	4	0/0
insert	Inserts an element into a sorted list	9	13.5	0/0
merge	Merges two sorted lists into a single list	15	29.6	0/0
removeSegment	The running example	7	8.4	0/0
reverse	Reverses an acyclic list in-place	5	6	0/0
reverse_cyclic	reverse, applied to a partially cyclic list	2	7.1	0/0
rotate	Moves the first element after the last element	6	7.9	0/0
search	Searches for an element with a specified value	3	2.1	0/0
search_nullderef	Erroneous implementation of search that dereferences a null pointer	3	2.4	1/1
swap	Swaps the first two elements in a list	6	8.8	0/0

Chapter 4

Partially Disjunctive Heap Abstraction

One of the continuing challenges in abstract interpretation is the creation of abstractions that yield analyses that are both *tractable* and *precise enough* to prove interesting properties about real-world programs. One source of difficulty is the need to handle programs with different behaviors along different execution paths. Disjunctive (powerset) abstractions capture such distinctions in a natural way. However, in general, powerset abstractions increase space and time costs by an exponential factor. Thus, powerset abstractions are generally perceived as very costly.

In this chapter, we partially address this challenge by presenting and empirically evaluating a new heap abstraction. The new heap abstraction works by merging shape descriptors according to a partial isomorphism similarity criteria, resulting in a partially disjunctive abstraction.

We implemented this abstraction in TVLA—a generic system for implementing program analyses.We conducted an empirical evaluation of the new abstraction and compared it with the powerset heap abstraction. The experiments show that analyses based on the partially disjunctive heap abstraction are as precise as the ones based on the powerset heap abstraction. In terms of performance, analyses based on the partially disjunctive heap abstraction are often superior to analyses based on the powerset heap abstraction. The empirical results show considerable speedups, up to 2 orders of magnitude, enabling previously non-terminating analyses, such as verification of the Deutsch-Schorr-Waite scanning algorithm, to terminate with no negative effect on the overall precision. Indeed, experience indicates that the partially disjunctive shape abstraction improves performance across all TVLA analyses uniformly, and in many cases is essential for making precise shape analysis feasible.

4.1 Introduction

One of the continuing challenges in abstract interpretation [CC77] is the creation of abstractions that yield analyses that are both *tractable* and *precise enough* to prove interesting properties about real-world programs. In this chapter we partially address this challenge by presenting and empirically evaluating a new heap abstraction, i.e., an abstraction for the (potentially unbounded) dynamically allocated storage manipulated by programs (e.g., see [JM81b, LH88, CWZ90, Lar89, Str92, SRW98, SRW02]). Heap abstractions are of fundamental importance to static analysis and verification of programs written in modern languages. Heap abstractions have been used, for instance, in the context of shape analysis (e.g., for prov-

ing that a program fragment preserves certain tree structure invariants), as well as in verifying that a client program satisfies certain conformance constraints for the correct usage of a library.

We present our abstraction in the context of the parametric abstract interpretation framework of [SRW02], which is based on the idea of representing program states using 3-valued logical structures. While it is very natural to view the abstraction we present as a heap abstraction, it can be used for abstracting other domains as well.

The TVLA framework presented in [SRW02] uses a disjunctive (powerset) heap abstraction: the abstract value at every program point is a *set* of shape descriptors (of bounded size) and set union is used as the join operation. In particular, this abstraction does not attempt to combine (or merge) different shape descriptors into one and relies on the fact that there are only finitely many shape descriptors (as they are of bounded size). This leads to powerful and sophisticated analyses for proving interesting program properties but is usually too expensive to be applied to real-world programs. (The number of distinct shape descriptors is doubly exponential in the size of the program in the worst case.)

The heap abstractions most commonly used in practice, especially when scalability is important, tend to be *single-shape* heap abstractions, which use a single shape descriptor to describe all possible program states at a program point [LH88, CWZ90, SRW98]. The current TVLA implementation provides options to utilize such single-shape heap abstractions. However, our experience has been that for the kind of applications that we have used TVLA for (mostly verification problems), the single-shape abstraction tends to be imprecise and causes a number of "false alarms" (i.e., verification fails for correct programs). Hence, this abstraction is not widely used by TVLA users. (A detailed discussion of the single-shape abstractions is beyond the scope of this thesis, because of the complexity of formalizing the single-shape abstractions within the framework of 3-valued-logic.)

This chapter presents a *partially disjunctive* heap abstraction which, in our experience, is significantly more efficient than the powerset heap abstraction, but has turned out to be precise enough for all the applications we have experimented with. Indeed, this abstraction has turned out to be the abstraction of choice for all TVLA users. The main idea behind this abstraction is to reduce the set of shape descriptors arising at a program point by merging "similar" shape descriptors but keeping "dissimilar" shape descriptors apart.

4.1.1 Running Example

Figure 4.1 shows a method implementing the mark phase of a mark-and-sweep garbage collector. The challenge here is to show that this procedure is partially correct, i.e., to establish that "upon termination, an element is marked if and only if it is reachable from the root." This simple program serves as a running example in this chapter.

The partial correctness of this program was established using abstract interpretation in [RSW01]. This abstract interpretation was created using TVLA—a generic system for implementing program analyses [LAS00]. The default implementation of TVLA uses the powerset heap abstraction. Verification of the above property using the powerset heap abstraction took 584 cpu seconds and generated $189,772$ different shape descriptors—definitely too many for such a simple program and simple property. The situation is worse for verifying a similar property for an implementation of the Deutsch-Schorr-Waite scanning procedure [Lin73]. This verification took 4 hours when the powerset heap abstraction was used.

Powerset heap abstractions are costly since they may distinguish between too many shape

```
// @Ensures marked == REACH(root)
void mark(Node root, NodeSet marked) {
   Node x;
   if (root != null) {
      NodeSet pending = new NodeSet();
      pending.add(root);
      marked.clear();
      while (!pending.isEmpty()) {
         x = pending.selectAndRemove();
         marked.add(x);
         if (x.left != null)
            if (!marked.contains(x.left))
               pending.add(x.left);
         if (x.right != null)
            if (!marked.contains(x.right)
               pending.add(x.right);
      }
   }
}
```

Figure 4.1: A simple Java-like implementation of the mark phase of a mark-and-sweep garbage collector

descriptors, which may not be necessary in order to verify program properties. In this chapter, we define a partially disjunctive heap abstraction, which is coarser than the powerset heap abstraction. The main idea is to reduce the set of shape descriptors arising at a program point by merging "similar" shape descriptors. In the mark example, verification using the partially disjunctive heap abstraction took 3 cpu seconds and generated $1,133$ shape descriptors—a two orders of magnitude improvement over verification using the powerset heap abstraction—with the same precision. Similarly, the verification of an implementation of the Deutsch-Schorr-Waite scanning procedure terminated successfully in 158 cpu seconds using the partially disjunctive heap abstraction.

4.1.2 Main Results

A New Abstraction.

We define a new heap abstraction, which we refer to as the *partial-isomorphism* heap abstraction. The new abstraction is coarser than the powerset heap abstraction and yet keeps certain shape descriptors apart. Our abstraction is parametric. It allows the user to specify which heap properties are of importance for a given analysis, and this guides the abstraction in determining which shape descriptors are merged together.

Table 4.1: Predicates used to verify the running example

Predicates	**Intended Meaning**
$x(v)$	Does reference variable x point to object v?
$root(v)$	Does reference variable $root$ point to object v?
`left`(v_1, v_2)	Does field `left` of object v_1 point to object v_2?
`right`(v_1, v_2)	Does field `right` of object v_1 point to object v_2?
$r[root](v)$	Is object v heap-reachable from reference variable $root$?
$set[marked](v)$	Is object v a member of the $marked$ set?
$set[pending](v)$	Is object v a member of the $pending$ set?

Robust Implementation.

We implemented our abstraction in TVLA. This abstraction has turned out to be the abstraction of choice for all TVLA users (e.g., see [YR04]). We believe that it is simple enough to be implemented in other systems besides TVLA (e.g., [TKB02]).

Empirical Evaluation.

We empirically evaluated our abstraction by comparing it with the powerset heap abstraction. In the largest benchmark, `SQLExecuter`, powerset heap abstraction did not terminate within $20,000$ cpu seconds. In contrast, the new abstraction took $9,673$ cpu seconds and proved correct usage of JDBC objects and absence of null-dereferences.

4.1.3 Outline

The rest of this chapter is organized as follows. In Section 4.2, we give a reminder of 3-valued-logic based program analysis with some examples. In Section 4.3, we describe the partial-isomorphism heap abstraction. In Section 4.4, we provide an empirical evaluation of the partial-isomorphism heap abstraction and powerset heap abstraction. In Section 4.5, we outline several other heap abstractions that we are investigating as ongoing work. In Section 6.6, we discuss related work.

4.2 3-valued Shape Analysis Primer

We now present a short reminder of 3-valued based shape analysis, providing examples and additional details relevant for this chapter.

Concrete Program Configurations

Recall that in our setting, concrete program states are represented using 2-valued logical structures over a fixed vocabulary of predicate symbols.

Table 4.1 shows the predicates used to record properties of individuals for the analysis of our running example. A unary predicate $ref(v)$ holds when the reference (or pointer) variable

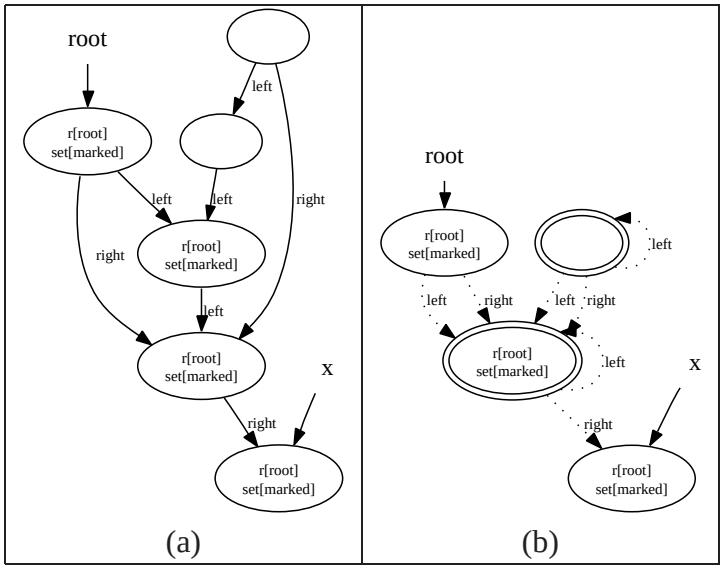


Figure 4.2: (a) A concrete program configuration arising at the exit label of the mark procedure, where all non-garbage nodes have been marked; (b) An abstract program configuration that approximates the concrete configuration in (a)

`ref` points to the object v; in our example `ref` $\in \{x, root\}$. Similarly, a binary predicate $fld(v_1, v_2)$ records the value of a reference (or pointer-valued) field `fld`; in our example `fld` $\in \{$`left`, `right`$\}$. A unary predicate $set[s](v)$ holds when the object v belongs to the set s; in our example $s \in \{marked, pending\}$.

In this thesis, 2-valued logical structures are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate $p(u)$, which holds for a node u, is drawn inside the node u. If a unary predicate represents a reference variables it is shown by having an arrow drawn from its name to the node pointed by the variable. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with p.

Figure 4.2(a) shows a concrete configuration arising at the exit label of the mark procedure, where all the individuals that are reachable from $root$ are marked, as indicated by the value of the $set[marked]$ predicate. The individuals represented by the empty nodes correspond to garbage objects.

Abstract Program Configurations

Recall that a 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure. This is achieved by letting an abstract configuration (i.e., a 3-valued logical structure) include *summary individuals*, i.e., an individual which corresponds to one or more individuals in a concrete configuration represented by that abstract configuration.

In this thesis, 3-valued logical structures are also depicted as directed graphs, where binary predicates with $1/2$ values are shown as dotted edges and summary individuals are shown as double-circled nodes.

We denote the set of all 3-valued logical structures over a set of predicates P by 3-STRUCT$_P$, usually abbreviating it to 3-STRUCT.

Bounded Program Configurations

Note that the size of a 3-valued structure is potentially unbounded and that 3-STRUCT is infinite. The abstractions studied in this thesis rely on a fundamental abstraction function for converting a potentially unbounded structure (either 2-valued or 3-valued) into a bounded 3-valued structure, which we define now. This abstraction function $\beta_{blur[A]}$ is parameterized by a special set of unary predicates A referred to as the *abstraction* predicates.

Let A be a set of unary predicates. An individual u_1 in a structure S_1 is said to be *A*-compatible to an individual u_2 in a structure S_2 iff for every predicate $p \in A$, $p^{S_1}(u_1) \sqsubseteq p^{S_2}(u_2)$ or $p^{S_2}(u_2) \sqsubseteq p^{S_1}(u_1)$. (Recall that the partial order $\sqsubseteq$ on $\{0, 1, 1/2\}$ is defined by $x \sqsubseteq y$ iff $x = y$ or $y = 1/2$.)

A 3-valued structure is said to be *A*-bounded if no two different individuals in its universe are *A*-compatible. A structure that is *A*-bounded can have at most $2^{|A|}$ individuals. We denote the set of all 3-valued *A*-bounded structures over a set of predicates by B-STRUCT$_{P,A}$, and, as usual, omit the subscripts when no confusion is likely.

The abstraction function β_{blur} : 3-STRUCT $\rightarrow$ B-STRUCT, which converts a (potentially unbounded) 3-valued structure into a bounded 3-valued structure, is defined as follows: we obtain an *A*-bounded structure from a given structure by merging all pairs of *A*-compatible individuals. $\beta_{blur}(\langle U_1, I\rangle) = \langle U_2, J\rangle$, where U_2 is the set of *A*-compatible equivalence classes of U_1, and the interpretation J is defined by:

$$p^J(c_1, \ldots, c_k) = \bigsqcup_{u_1 \in c_1, \ldots, u_k \in c_k} p^I(u_1, \ldots, u_k) \ .$$

Figure 4.2(b) shows an *A*-bounded structure obtained from the structure in Figure 4.2(a) with $A = \{x, root, r[root], set[marked], set[pending]\}$.

The abstraction function β_{blur} serves as the basis for abstract interpretation in TVLA. In particular, it serves as the basis for defining various different abstractions for the (potentially unbounded) *set of* 2*-valued logical structures* that arise at a program point.

4.2.1 Powerset Heap Abstraction

This abstraction is based on the fact that there can only be a finite number of bounded structures that are not *isomorphic* to one another. (Two structures are isomorphic when there is a bijection between their universes that preserves all predicate values.) The powerset abstraction function operates by bounding 2-valued structures with respect to a subset of the unary predicates, and removing duplicates (isomorphic structures).

For the sake of simplicity we will work with *canonic* bounded structures. Note that the individuals of an *A*-bounded structure are uniquely identified by the set of values of the predicates in A; we refer to such a set of predicate values as the individual's *canonical name*. For example, the individual pointed by $root$ in Figure 4.2(b) has the canonical name $u_{\{x=0, root=1, r[root]=1, set[marked]=1, set[pending]=0\}}$. A canonic bounded structure is a bounded structure in which the individuals are identified by their canonical names. We refer to the set of all canonic bounded structures by CB-STRUCT$_{P,A}$. Note that for a given P and A, CB-STRUCT$_{P,A}$ is finite. The *canonic* abstraction function $\beta_{canonic}$: 2-STRUCT $\rightarrow$ CB-STRUCT is defined as follows: $\beta_{canonic}(S)$ is obtained by renaming the individuals of $\beta_{blur}(S)$, giving them canonic names.

The powerset heap abstraction function $\alpha_{pow} : 2^{\text{2-STRUCT}} \rightarrow 2^{\text{CB-STRUCT}}$ is defined by

$$\alpha_{pow}(\mathit{XS}) = \{\beta_{canonic}(S) \mid S \in \mathit{XS}\} \ .$$

4.3 The Partial-Isomorphism Heap Abstraction

The idea behind partial-isomorphism heap abstraction is fairly simple. The powerset heap abstraction keeps all the canonic bounded structures that arise at a program point separate. Single-shape heap abstraction merges all canonic bounded structures arising at a program point into one structure. The partial-isomorphism heap abstraction, in contrast, merges canonic bounded structures into one structure only when they have the same universe.

We say that a pair of canonic bounded structures are *universe congruent* iff the two structures have the same universe. Universe congruence induces an equivalence relation over sets of canonic bounded structures. This equivalence relation lets us define an abstraction function $\alpha_{pi} : 2^{\text{2-STRUCT}} \rightarrow 2^{\text{CB-STRUCT}}$ that merges all universe congruent structures. Given a set of canonic bounded structures *XS* with the same universe U, we define the merged structure $\bigsqcup \mathit{XS} = \langle U, I \rangle$ that has the same universe as all structures in *XS* and the following interpretation of predicates. For every predicate p of arity k and tuple of individuals $\langle u_1, \ldots, u_k \rangle \in U^k$:

$$p^{\bigsqcup \mathit{XS}}(u_1, \ldots, u_k) = \bigsqcup_{S \in \mathit{XS}} p^S(u_1, \ldots, u_k) \ .$$

We are now ready to define the partial-isomorphism heap abstraction function α_{pi}:

$$\alpha_{pi}(\mathit{XS}) = \left\{ \bigsqcup C \mid C \subseteq \alpha_{pow}(\mathit{XS}) \text{ is a universe congruence equivalence class} \right\} \ .$$

Thus, partial-isomorphism heap abstraction is less precise than the powerset heap abstraction[1]. As the empirical results presented later show, the partial-isomorphism heap abstraction seems to work as well as (i.e., is as precise as) the powerset heap abstraction, *in practice*. The following propositions may help explain why.

Proposition 4.3.1 *If a pair of bounded structures S_1 and S_2 are universe congruent, then the merged structure $S_1 \bigsqcup S_2$ is the least bounded structure that approximates (embeds) both S_1 and S_2.*

When partial-isomorphism abstraction is applied to a pair of structures S_1 and S_2, there are two possibilities:

- Structures S_1 and S_2 are not universe congruent. In this case, the result of the abstraction is $\alpha_{pi}(\{S_1, S_2\}) = \{S_1, S_2\}$, which is the least upper-bound of the powerset abstraction—the most precise approximation of both structures.

- Structures S_1 and S_2 are universe congruent. In this case, the result of the abstraction is $\alpha_{pi}(\{S_1, S_2\}) = S_1 \bigsqcup S_2$, which is the most precise upper bound among all (singleton sets of) bounded structures.

[1] Here, precision is used in the sense of a Galois Connection between a pair of abstract domains.

Proposition 4.3.2 *Partial-isomorphism heap abstraction preserves the values of abstraction predicates.*

In other words, partial-isomorphism heap abstraction only loses the same kind of distinctions that can also be lost by β_{blur}—values of non-abstraction predicates.

In terms of worst-case complexity, partial-isomorphism heap abstraction has the same complexity as powerset heap abstraction—doubly-exponential in the number of abstraction predicates. This is due to the number of sets of canonical names, which is the dominant factor in the worst-case complexity. However, partial-isomorphism heap abstraction can save an exponential factor due to binary predicates, which is the dominant factor in many cases, in practice.

4.3.1 Illustrating Example

To illustrate the operation of partial-isomorphism heap abstraction, consider the abstract program configuration shown in Figure 4.2(b) and the abstract program configuration shown in Figure 4.3(a). Both configurations represent cases where all of the non-garbage nodes have been marked and non-garbage nodes have not been marked, i.e., the program property we want to verify holds for those configurations. The difference between the configurations is in the position of the node pointed by `x` in the part of the heap that has been marked. In this case, the partial-isomorphism heap abstraction results in the structure shown in Figure 4.3(b), which ignores the precise position of the node pointed by `x` inside the part of the heap that was marked.

The mark program non-deterministically selects an object and removes it from the pending set. This non-determinism allows many different ways of traversing the set of objects reachable from `root`, which results in many different abstract program configurations that sustain the program property we want to verify and only differ by values of binary predicates. Partial-isomorphism heap abstraction ignores the values of the binary predicates, but keeps precise the overall property for an abstract configuration of having sets of nodes with the same garbage/non-garbage and mark/unmarked properties. This allows the analysis to merge many similar structures without losing the information needed to prove the partial correctness of the mark program.

4.4 Implementation and Empirical Evaluation

We implemented the partial-isomorphism abstraction described in the previous section in TVLA, and the implementation is publicly available [LAS00]. We applied it to verify various specifications for the Java programs described in Table 4.2. To translate Java programs and their specifications to TVP (TVLA's input language), we used a front-end for Java, which is based on the Soot framework [VRHS+99]. For all benchmarks, we checked the absence of null dereferences in addition to the properties described in Table 4.2. Our specifications include correct usage of JDBC objects, correct usage of Java I/O streams, correct usage of Java collections and iterators, and additional small but interesting specifications.

The experiments were conducted using TVLA version 2, running with SUN's JRE 1.4, on a 1 GHZ Intel Pentium Processor machine with 1.5 GB RAM. We optimized for precision and simplicity by using TVLA's Focus and Coerce operations in all benchmarks. We compared partial isomorphism to the full powerset abstraction in terms of time and space performance and precision.

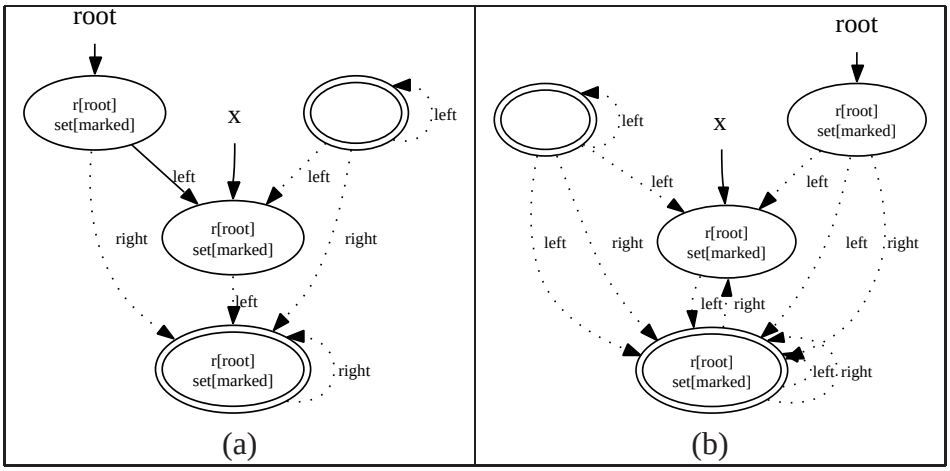


Figure 4.3: (a) An abstract program configuration arising at the exit label of the mark procedure, where all non-garbage nodes have been marked and `x` points to a node adjacent to `root`; (b) The result of merging the structure in (a) and the structure in Figure 4.2(b)

Table 4.2: Benchmarks and properties used for comparing the analysis based on powerset heap abstraction with the analysis based on partial-isomorphism heap abstraction. Treeness means preservation of tree structure invariants

Benchmark	**Description**	**Property**
GC.mark	Figure 4.1	Partial correctness
DSW	Deutsch-Schorr-Waite	Partial correctness of tree scanning + Treeness
ISPath	Input streams	Correct usage of Java IOStreams
InputStream5	Input stream holders	Correct usage of Java IOStreams
InputStream5b	Input stream holders with error	Correct usage of Java IOStreams
InputStream6	Input stream holders	Correct usage of Java IOStreams
SQLExecutor	A JDBC framework	Correct usage of JDBC objects
KernelBench.1	CMP benchmark [RWF+02]	Absence of concurrent modification exceptions
InsertSorted	Insertion into sorted trees	Tree sortedness + Treeness
DeleteSorted	Deletion from sorted trees	Tree sortedness

Table 4.3: Time, space and number of errors measurements. Rep. Err. is the number of errors reported by the analysis, and Act. Err. is the number of errors that indicate real problems. Time and space measurements for non-terminating benchmarks are prefixed with > to indicate the measurements taken when the analysis timed out. The number of reported errors is the same for both the analysis based on the powerset heap abstraction and the analysis based on partial-isomorphism heap abstraction on all (terminating) benchmarks. For benchmarks that did not terminate with the powerset heap abstraction, the numbers are taken from the analysis based on partial-isomorphism heap abstraction

Benchmark	**Time in seconds**		**Space in Mb.**		**Rep. Err. / Act. Err.**
	Powerset	**Partial iso.**	**Powerset**	**Partial iso.**	
GC.mark	584	3	56	1.4	0/0
DSW	14,364	157	116.3	5.6	0/0
ISPath	79	79	2.8	2.9	0/0
InputStream5	4,530	1,706	14.0	11.9	1/0
InputStream5b	3,492	1,394	9.8	9.1	1/0
InputStream6	15,558	3,929	23.6	15.9	1/0
SQLExecutor	>20,000	9,673	>109.3	104.8	0/0
KernelBench.1	7,393	5,355	13.3	10.8	1/1
InsertSorted	264	37	4.5	2.4	0/0
DeleteSorted	>20,000	3,271	>62.6	21.8	0/0

The results of the analyses are shown in Table 5.3. In all the benchmarks the analysis based on the partial-isomorphism heap abstraction achieved the same precision as the analysis based on the powerset heap abstraction, and other TVLA users reported the same phenomena. In all but one example, the analysis based on partial-isomorphism heap abstraction achieved significant performance improvements.

4.4.1 Implementation Independent Results

Although the results shown in Table 5.3 measure the time and space consumption of analyses using different abstractions, they are also influenced by the various implementation details of the abstractions.

In Table 4.4, we supply implementation independent measurements. We measured the total number of abstract configurations generated by the analysis and the maximal number of abstract configurations that exist in the transition system at any given time during the analysis. The total number of abstract configurations and the maximal number of abstract configurations are always the same with the powerset heap abstraction, since structures are only accumulated in the transition system. For the partial-isomorphism heap abstraction, the maximal number of abstract configurations is often lower than the total number of abstract configurations, indicating that structures discovered in different iterations were merged together.

The results show a consistency between the improvements in time and space performance of the partial-isomorphism heap abstraction, relative to the powerset heap abstraction, and the reduced number of abstract configurations.

Table 4.4: Implementation-independent measurements. Total #structs is the total number of abstract configurations that arose during the analysis, and Max #structs is the maximal number of abstract configurations that existed in the transition system at any time during the analysis. The results of non-terminating benchmarks are prefixed with $>$ to indicate the measurements taken when the analysis timed out

Benchmark	**Total #structs**		**Max #structs**	
	Powerset	**Partial iso.**	**Powerset**	**Partial iso.**
GC.mark	189,772	1,133	189,772	748
DSW	320,387	6,480	320,387	2,986
ISPath	2,168	2,168	2,168	2,168
InputStream5	8,164	3,366	8,164	2,204
InputStream5b	5,973	2,598	5,973	1,729
InputStream6	24,461	6,678	24,461	4,411
SQLExecutor	>8,824	4,107	>8,824	2,164
KernelBench.1	12,594	9,296	12,594	5,748
InsertSorted	7,487	1,318	7,487	905
DeleteSorted	>158,780	30,386	>158,780	25,673

4.5 Extensions and Future Work

The partial-isomorphism heap abstraction has so far performed quite satisfactorily in our experience with TVLA. However, we cannot assume that this will always be adequate. Analysis and verification of larger programs may require more aggressive abstractions, while in some cases we may require more precise abstractions. In this section we describe various other abstractions that may be of value. We are currently in the process of evaluating the effectiveness of some of the abstractions described below.

Parametric Partial Isomorphism

We now present a parametric abstraction that includes both the powerset heap abstraction and the partial-isomorphism heap abstraction as special cases.

Definition 4.5.1 *We say that a pair of bounded structures $S_1 = \langle U_1, I_1 \rangle$ and $S_2 = \langle U_2, I_2 \rangle$ are* ***partially isomorphic*** *with respect to a set of predicates R, denoted by $S_1 \equiv_R S_2$, iff there exists a bijection $f^{pi} : U_1 \rightarrow U_2$, such that, for every predicate $p \in R$ of arity k and tuple of nodes $\langle u_1, \ldots, u_k \rangle \in U_1^k$, the following holds:*

$$p^{S_1}(u_1, \ldots, u_k) = p^{S_2}(f^{pi}(u_1), \ldots, f^{pi}(u_k)) \ .$$

Note that $\equiv_R$ is an equivalence relation among 3-valued structures. Given any set of predicates R that includes the set of all abstraction predicates A, we define an abstraction function

$\alpha_{pi[R]} : 2^{\text{2-STRUCT}} \rightarrow 2^{\text{CB-STRUCT}}$ as follows:

$$\alpha_{pi[R]}(XS) = \left\{ \bigsqcup C \mid C \subseteq \alpha_{pow}(XS) \text{ is a } \equiv_R \text{ equivalence class} \right\} \quad .$$

This function defines a whole family of abstractions. Further, $\alpha_{pow} = \alpha_{pi[P]}$ (where P is the set of all predicates) is the most precise among this family of abstractions, and $\alpha_{pi} = \alpha_{pi[A]}$ is the least precise among this family of abstractions.

The reason we restrict ourselves to sets R that contain the set of all abstraction predicates A is the following. If R includes A, then for any two $\equiv_R$-equivalent bounded structures, the bijection between the universes of the two structures that preserves the values of predicates in R is uniquely determined, and this bijection is used to determine which individuals should be "merged" together.

This parametric definition allows users to choose abstractions in a more fine-grained fashion, by specifying the set of predicates R. The parametric abstraction could also be used by an appropriate iterative refinement technique, which starts with $R = A$ and iteratively adds predicates to R, until a sufficiently precise abstraction is obtained or $R = P$.

Deflating Reductions

Deflating reductions can potentially yield performance improvements without a loss of precision. A very simple deflating reduction is the following: consider a set of 3-valued structures X containing structures S_1 and S_2, such that $S_1 \sqsubseteq S_2$. Clearly, the set $X' = X - \{S_1\}$ is semantically equivalent to X, and removing S_1 involves no loss of precision (even when the abstract transformer that is used is not the best). This reduction is referred to as "non-redundancy" in [BHZ03]. Making this reduction feasible requires testing for the partial order relation over 3-valued structures, which can be done in polynomial time for bounded 3-valued structures. The key question with this reduction is whether the subsequent (performance) benefits of doing the reduction outweigh extra cost of performing the reduction. Our initial experience shows that this reduction is worth using. This reduction transforms TVLA's preorder over sets of 3-valued structures into a proper (Hoare powerdomain) partial ordering.

4.6 Related Work

A substantial body of literature exists on abstractions for various different domains and for creating new abstractions from existing abstractions. The distinguishing aspect of our work is its focus on heap abstractions and its focus on an empirical evaluation of the effectiveness of the proposed heap abstraction.

Function Space Domain Construction.

Function space domain construction is one way of creating abstractions that are "partly disjunctive". Examples of previous work using such a domain construction include [Deu94], where the abstraction is composed of two components—a lattice of symbolic access paths and a parametric numerical lattice. In this abstraction, abstract elements with the same symbolic access path component are merged by joining the numerical lattice component. The ESP system [DLS02] also utilizes a similar function space domain construction, but not for heap abstractions.

Least Disjunctive Basis.

In [GR98], a technique is defined for obtaining the "least disjunctive basis", which is the most abstract domain inducing the same disjunctive completion as another domain. Unfortunately, this may result in larger sets of abstract elements, as abstract elements are substituted by sets of other abstract elements, causing inflation.

Deflating Operators and Widening Operators.

In [BHZ03], different widening operators and congruence relations are considered for the powerset polyhedra domain, and in more general settings.

Chapter 5

Disjoint Subgraph Decomposition

Programs commonly maintain multiple linked data structures. Correlations between multiple data structures may often be *non-existent or irrelevant to verifying that the program satisfies certain safety properties or invariants*. In this chapter, we show how this *independence* between different (singly-linked) data structures can be utilized to perform shape analysis and verification more efficiently. We present a new abstraction based on decomposing graphs into sets of subgraphs, and show that, in practice, this new abstraction leads to very little loss of precision, while yielding substantial improvements to efficiency.

5.1 Introduction

We are interested in verifying that programs satisfy various safety properties (such as the absence of null dereferences, memory leaks, dangling pointer dereferences, etc.) and that they preserve various data structure invariants.

Many programs, such as web-servers, operating systems, network routers, etc., commonly maintain multiple linked data-structures in which data is added and removed throughout the program's execution. The Windows IEEE 1394 (firewire) device driver, for example, maintains separate cyclic linked lists that respectively store bus-reset request packets, data regarding CROM calls, data regarding addresses, and data regarding ISOCH transfers. These lists are updated throughout the driver's execution based on events that occur in the machine. Correlations between multiple data-structures in a program, such as those illustrated above, may often be *non-existent or irrelevant to the verification task of interest*. In this chapter, we show how this *independence* between different data-structures can be utilized to perform verification more efficiently.

Many scalable heap abstractions typically maintain no correlation between different *points-to* facts (and can be loosely described as *independent attribute* abstractions in the sense of [JM81a]). Such abstractions are, however, not precise enough to prove that programs preserve data structure invariants. More precise abstractions for the heap that use shape graphs to represent *complete* heaps [SRW02], however, lead to exponential blowups in the state space.

In this chapter, we focus on (possibly cyclic) singly-linked lists and introduce an approximation of the *full heap abstraction* presented in Chapter 3. The new *graph decomposition abstraction* is based on a decomposition of (shape) graphs into sets of (shape) subgraphs (without maintaining correlations between different shape subgraphs). In our initial empirical evalua-

tion, this abstraction produced results almost as precise as the full heap abstraction (producing just one false positive), while reducing the state space significantly, sometimes by exponential factors, leading to dramatic improvements to the performance of the analysis. We also hope that this abstraction will be amenable to abstraction refinement techniques (to handle the cases where correlations between subgraphs are necessary for verification), though that topic is beyond the scope of this thesis.

One of the challenges in using a subgraph abstraction is the design of safe and precise transformers for statements. We show in this chapter that the computation of the most precise transformer for the graph decomposition abstraction is FNP-complete.

We derive efficient, polynomial-time, transformers for our abstraction in several steps. We first use an observation by Distefano et al. [DOY06] and show how the most precise transformer can be computed more efficiently (than the naive approach) by: (a) identifying *feasible combinations of subgraphs referred to by a statement*, (b) composing only them, (c) transforming the composed subgraphs, and (d) decomposing the resulting subgraphs. Next, we show that the transformers can be computed in polynomial time by omitting the feasibility check (which entails a possible loss in precision). Finally, we show that the resulting transformer can be implemented in an *incremental* fashion (i.e., in every iteration of the fixed point computation, the transformer reuses the results of the previous iteration).

We have developed a prototype implementation of the algorithm and compared the precision and efficiency (in terms of both time and space) of our new abstraction with that of the full heap abstraction over a standard suite of shape analysis benchmarks as well as on models of a couple of Windows device drivers. Our results show that the new analysis produces results as precise as the full heap-based analysis in almost all cases, but much more efficiently.

5.1.1 Outline

The rest of the chapter is organized as follows. Section 5.2 gives a motivation for our analysis. Section 5.3 describes a concrete semantics for programs with linked lists and a full heap abstraction. Section 5.4 describes the graph decomposition abstraction. In Section 5.5 we develop efficient transformers for the graph decomposition abstraction. Section 5.6 presents experimental results and compares the full heap abstraction with the graph decomposition abstraction. Section 5.5 discusses related work.

5.2 Overview

In this section, we provide an informal overview of our approach. Later sections provide the formal details.

Figure 5.1 shows a simple program that adds elements into independent lists: a list with a head object referenced by a variable `h1` and a tail object referenced by a variable `t1`, and a list with a head object referenced by a variable `h2` and a tail object referenced by a variable `t2`. This example is used as the running example throughout the chapter. The goal of the analysis is to prove that the data structure invariants are preserved in every iteration, i.e., at label `L1` variables `h1` and `t1` and variables `h2` and `t2` point to disjoint acyclic lists, and that the head and tail pointers point to the first and last objects in every list, respectively.

```
//@assume h1!=null && h1==t1 && h1.n==null &&
//        h2!=null && h2==t2 && h2.n==null
//@invariant Reach(h1,t1) && Reach(h2,t2) &&
//           DisjointLists(h1,h2)
EnqueueEvents() {
L1:  while (...) {
       List temp = new List(getEvent());
       if (nondet()) {
L2:      t1.n = temp;
L3:      t1 = temp;
       } else {
         t2.n = temp;
         t2 = temp;
}   }  }
```

Figure 5.1: A program that enqueues events into one of two lists. `nondet()` returns either `true` or `false` non-deterministically

The shape analysis presented in Chapter 3 is able to verify the invariants by generating, at program label `L1`, the 9 abstract states shown in Figure 5.2. These states represent the 3 possible states that each list can have: a) a list with one element, b) a list with two elements; and c) a list with more than two elements. This analysis uses a *full heap abstraction*: it does not take advantage of the fact that there is no interaction between the lists, and explores a state-space that contains all 9 possible combinations of cases $\{a, b, c\}$ for the two lists.

The shape analysis using a *graph decomposition abstraction* presented in this chapter, represents the properties of each list separately and generates, at program label `L1`, the 6 abstract states shown in Figure 5.3. For a generalization of this program to k lists, the number of states generated at label `L1` by using a graph decomposition abstraction is $3 \times k$, compared to 3^k for an analysis using a full heap abstraction, which tracks correlations between properties of all k lists. In many programs, this exponential factor can be significant. Note that in cases where there is no *correlation* between the different lists, the new abstraction of the set of states is as precise as the full heap abstraction: e.g., Figure 5.3 and Figure 5.2 represent the same set of concrete states.

We note that in the presence of pointers, it is not easy to decompose the verification problem into a set of sub-problems to achieve similar benefits. For example, current (flow-insensitive) alias analyses would not be able to identify that the two lists are disjoint.

5.3 A Full Heap Abstraction for Lists

In this section, we describe the concrete semantics of programs manipulating singly-linked lists and a full heap abstraction for singly-linked lists.

A Simple Programming Language for Singly-Linked Lists.

We now define a simple language and its concrete semantics. Our language has a single data type $List$ (representing a singly-linked list) with a single reference field `n` and a data field,

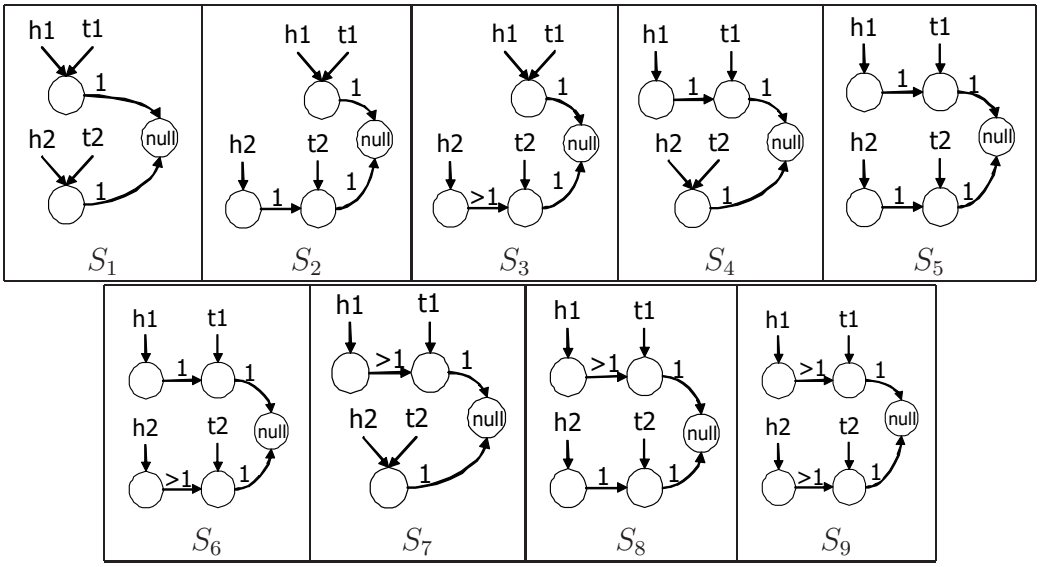


Figure 5.2: Abstract states at program label `L1`, generated by an analysis of the program in Figure 5.1 using a powerset abstraction. Edges labeled by 1 indicate list segments of length 1, whereas edges labeled by >1 indicate list segments of lengths greater than 1

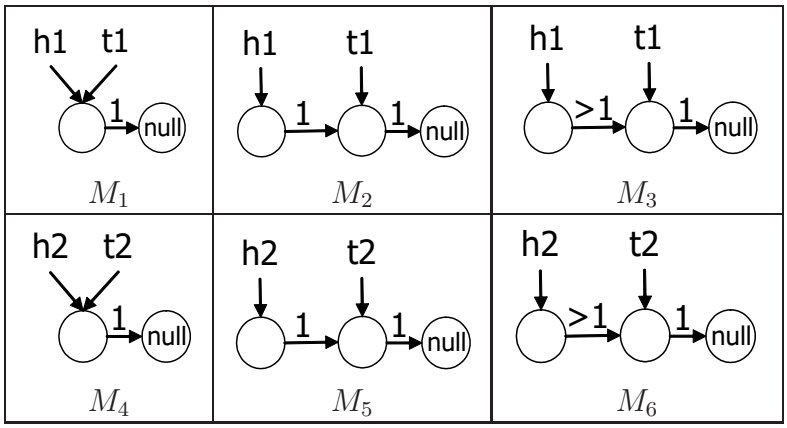


Figure 5.3: Abstract states at program label `L1`, generated by an analysis of the program in Figure 5.1, using the graph decomposition abstraction

Table 5.1: Concrete semantics of program statements. Primed symbols denote post-execution values. We write x,y, and x' to mean *env*(x), *env*(y), and *env*$'(x)$, respectively

Statement	**Condition**	**Update**
`x=new List()`		$x' = v_{new}$, where v_{new} is a fresh List object $n' = \lambda\, v\, .\, (v = v_{new}\ ?\ \mathit{null}\ :\ n(v))$
`x=null`		$x' = \mathit{null}$
`x=y`		$x' = y$
`x=y.n`	$y \neq \mathit{null}$	$x' = n(y)$
`x.n=y`	$x \neq \mathit{null}$	$n' = \lambda\, v\, .\, (v = x\ ?\ y\ :\ n(v))$
`assume(x!=y)`	$x \neq y$	
`assume(x==y)`	$x = y$	

which we conservatively ignore.

There are five types of heap-manipulating statements: (1) `x=new List()`, (2) `x=null`, (3) `x=y`, (4) `x=y.n`, and (5) `x.n=y`. Control flow is achieved by using `goto` statements and `assume` statements of the form `assume(x==y)` and `assume(x!=y)`. For simplicity, we do not present a deallocation, `free(x)`, statement and use garbage collection instead. Our implementation supports memory deallocation, assertions, and detects (mis)use of dangling pointers.

Concrete States. Let *PVar* be a set of variables of type $List$. A concrete program state is a triple $C \stackrel{\text{def}}{=} (U^C, \mathit{env}^C, n^C)$ where U^C is the set of heap objects, an environment $\mathit{env}^C : \mathit{PVar} \cup \{\mathit{null}\} \rightarrow U^C$ maps program variables (and *null*) to heap objects, and $n^C : U^C \rightarrow U^C$, which represents the `n` field, maps heap objects to heap objects. Every concrete state includes a special object v_{null} such that *env*$(\mathit{null}) = v_{null}$.We denote the set of all concrete states by $States$.

Concrete Semantics. We associate a transition function $[\![st]\!]$ with every statement *st* in the program. Each statement *st* takes a concrete state C, and transforms it to a state $C' = [\![st]\!](C)$. The semantics of a statement is given by a pair $(condition, update)$ such that when the condition specified by $condition$ holds the state is updated according to the assignments specified by $update$. The concrete semantics of program statements is shown in Table 5.1.

5.3.1 Abstracting List Segments

The abstraction is based on the one presented in Chapter 3; we now briefly repeat the essential details.

The core concepts of the abstraction are *interruptions* and *uninterrupted list*. An object is an *interruption* if it is referenced by a variable (or *null*) or shared (i.e., has two or more predecessors). An uninterrupted list is a path delimited by two interruptions that does not contain interruptions other than the delimiters.

Definition 5.3.1 (Shape Graphs) *A shape graph* $G \stackrel{\text{def}}{=} (V^G, E^G, \mathit{env}^G, \mathit{len}^G)$ *is a quadruple where* V^G *is a set of nodes,* E^G *is a set of edges,* $\mathit{env}^G : \mathit{PVar} \cup \{\mathit{null}\} \rightarrow V^G$ *maps variables*

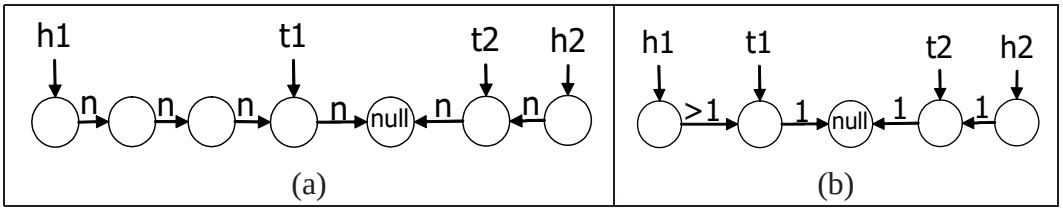


Figure 5.4: (a) A concrete state, and (b) The abstraction of the state in (a)

(and null) to nodes, and $\mathit{len}^G : E^G \to \mathit{pathlen}$ *assigns labels to edges. In this chapter, we use* $\mathit{pathlen} \stackrel{\text{def}}{=} \{1, >1\}$.[1]

We denote the set of shape graphs by SG_{PVar}, omitting the subscript if no confusion is likely, and define equality between shape graphs by isomorphism. We say that a variable x points to a node $v \in V^G$ if $\mathit{env}^G(\mathsf{x}) = v$.

We now describe how a concrete state $C \stackrel{\text{def}}{=} (U^C, \mathit{env}^C, n^C)$ is abstracted into a shape graph $G \stackrel{\text{def}}{=} (V^G, E^G, \mathit{env}^G, \mathit{len}^G)$ by the function $\beta^{FH} : States \to SG$. First, we remove any node in U^C that is not reachable from a (node pointed-to by a) program variable. Let $\mathit{PtVar}(C)$ be the set of objects pointed-to by some variable, and let $\mathit{Shared}(C)$ the set of heap-shared objects. We create a shape graph $\beta^{FH}(C) \stackrel{\text{def}}{=} (V^G, E^G, \mathit{env}^G, \mathit{len}^G)$ where $V^G \stackrel{\text{def}}{=} \mathit{PtVar}(C) \cup \mathit{Shared}(C)$, $E^G \stackrel{\text{def}}{=} \{(u, v) \mid (u, \ldots, v) \text{ is an uninterrupted list}\}$, env^G restricts env^C to V^G, and $\mathit{len}^G(u, v)$ is 1 if the uninterrupted list from u to v has one edge and >1 otherwise. The abstraction function α^{FH} is the point-wise extension of β^{FH} to sets of concrete states[2]. We say that a shape graph is *admissible* if it is in the image of β^{FH}.

Proposition 5.3.2 *A shape graph is admissible iff the following properties hold: (i) Every node has a single successor; (ii) Every node is pointed-to by a variable (or null) or is a shared node, and (iii) Every node is reachable from (a node pointed-to by) a variable.*

We use Proposition 5.3.2 to determine if a given graph is admissible in linear time and to conduct an efficient isomorphism test for two shape graphs in the image of the abstraction. It also provides a bound on the number of admissible shape graphs: $2^{5n^2+10n+8}$, where $n \stackrel{\text{def}}{=} |\mathit{PVar}|$.

Example 5.3.3 *Figure 5.4(a) shows a concrete state that arises at program label* L1 *and Figure 5.4(b) shows the shape graph that represents it.*□

Concretization.

The function $\gamma^{FH} : SG \to 2^{States}$ returns the set of concrete states that a shape graph represents: $\gamma^{FH}(G) \stackrel{\text{def}}{=} \{C \mid \beta^{FH}(C) = G\}$. We define the concretization of sets of shape graphs by using its point-wise extension. We now have the Galois Connection $\langle 2^{States}, \alpha^{FH}, \gamma^{FH}, 2^{SG} \rangle$.

[1]The abstraction in Chapter 3 is more precise, since it uses the abstract lengths $\{1, 2, > 2\}$. We use the lengths $\{1, > 1\}$, which we found to be sufficiently precise, in practice.

[2]In general, the point-wise extension of a function $f : D \to D$ is a function $f : 2^D \to 2^D$, defined by $f(S) \stackrel{\text{def}}{=} \{f(s) \mid s \in S\}$. Similarly, the extension of a function $f : D \to 2^D$ is a function $f : 2^D \to 2^D$, defined by $f(S) \stackrel{\text{def}}{=} \bigcup_{s \in S} f(s)$.

Abstract Semantics.

The most precise, a.k.a *best*, abstract transformer [CC77] of a statement is given by $[\![st]\!]^{\#} \stackrel{\text{def}}{=} \alpha^{FH} \circ [\![st]\!] \circ \gamma^{FH}$. An efficient implementation of the most precise abstract transformer is shown in the full version [MBC+].

5.4 A Graph Decomposition Abstraction for Lists

In this section, we introduce the abstraction that is the basis of our approach as an approximation of the abstraction shown in the previous section. We define the domain we use—2^{ASSG}, the powerset of atomic shape subgraphs—as well as the abstraction and concretization functions between 2^{SG} and 2^{ASSG}.

5.4.1 The Abstract Domain of Shape Subgraphs

Intuitively, the graph decomposition abstraction works by decomposing a shape graph into a set of *shape subgraphs*. In principle, different graph decomposition strategies can be used to get different abstractions. However, in this chapter, we focus on decomposing a shape graph into a set of subgraphs induced by its *(weakly-)connected components*. The motivation is that different weakly connected components mostly represent different "logical" lists (though a single list may occasionally be broken into multiple weakly connected components during a sequence of pointer manipulations) and we would like to use an abstraction that decouples the different logical lists. We will refer to an element of SG_{PVar} as a shape graph, and an element of SG_{Vars} for any $Vars \subseteq PVar$ as a shape subgraph. We denote the set of shape subgraphs by SSG and define $Vars(G)$ to be the set of variables that appear in G, i.e., mapped by env^G to some node.

5.4.2 Abstraction by Graph Decomposition

We now define the decomposition operation. Since our definition of shape graphs represents *null* using a special node, we identify connected components *after excluding the null node*. (Otherwise, all *null*-terminated lists, i.e. all acyclic lists, will end up in the same connected component.)

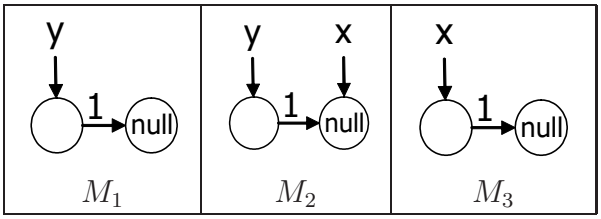
Definition 5.4.1 (Projection) *Given a shape subgraph $G \stackrel{\text{def}}{=} (V, E, env, len)$ and a set of nodes $W \subseteq V$, the subgraph of G induced by W, denoted by $G|_W$, is the shape subgraph (W, E', env', len'), where $E' \stackrel{\text{def}}{=} E \cap (W \times W)$, $env' \stackrel{\text{def}}{=} env \cap (Vars(G) \times W)$, and $len' \stackrel{\text{def}}{=} len \cap (E' \times pathlen)$.*

Definition 5.4.2 (Connected Component Decomposition) *For a shape subgraph $G \stackrel{\text{def}}{=} (V, E, env, len)$, let $R \stackrel{\text{def}}{=} E'^*$ be the reflexive, symmetric, transitive closure of the relation $E' \stackrel{\text{def}}{=} E \setminus \{(v_{null}, v), (v, v_{null}) \mid v \in V\}$. That is, R does not represent paths going through null. Let $[R]$ be the set of equivalence classes of R. The connected component decomposition of G is given by*

$$Components(G) \stackrel{\text{def}}{=} \{G|_{C'} \mid C' = C \cup \{v_{null}\}, C \in [R]\} \ .$$

```
if (?)
    x = new List();
else
    x = null;
y = new List();
```

(a)



(b)

Figure 5.5: (a) A code fragment; and (b) Shape subgraphs arising after executing `y=new List()`. M_1: `y` points to a list and `x` is not null, M_2: `y` points to a list and `x` is null; and M_3: `x` points to a list and `y` is not null

Example 5.4.3 *Referring to Figure 5.2 and Figure 5.3, we have Components*$(S_2) = \{M_1, M_5\}$.

Abstracting Away Null-value Correlations.

The decomposition *Components* manages to decouple distinct lists in a shape graph. However, it fails to decouple lists from null-valued variables.

Example 5.4.4 *Consider the code fragment shown in Figure 5.5(a) and the shape subgraphs arising after* `y=new List()`*.* `y` *points to a list (with one cell), while* `x` *is null or points to another list (with one cell). Unfortunately, the* `y` *list will be represented by two shape subgraphs in the abstraction, one corresponding to the case that* `x` *is null (*M_2*) and one corresponding to the case that* `x` *is not null (*M_1*). If a number of variables can be optionally null, this can lead to an exponential blowup in the representation of other lists! Our preliminary investigations show that this kind of exponential blow-up can happen in practice.*□

The problem is the occurrence of shape subgraphs that are isomorphic except for the *null* variables. We therefore define a coarser abstraction by decomposing the set of variables that point to the *null* node. To perform this further decomposition, we define the following operations:

- *nullvars* : *SSG* $\rightarrow 2^{PVar}$ returns the set of variables that point to *null* in a shape subgraph.
- *unmap* : *SSG* $\times\ 2^{PVar} \rightarrow$ *SSG* removes the mapping of the specified variables from the environment of a shape subgraph.
- *DecomposeNullVars* : *SSG* $\rightarrow 2^{SSG}$ takes a shape subgraph and returns: (a) the given subgraph without the null variables, and (b) one shape subgraph for every null variable, which contains just the null node and the variable:

$$\begin{aligned} \mathit{DecomposeNullVars}(G) \stackrel{\text{def}}{=} & \{unmap(G, \mathit{nullvars}(G))\} \cup \\ & \{unmap(G|_{v_{null}}, \mathit{Vars}(G) \setminus \{var\} \mid var \in \mathit{nullvars}(G)\} \ . \end{aligned}$$

In the sequel, we use the point-wise extension of *DecomposeNullVars*.

We define the set *ASSG* of *atomic* shape subgraphs to be the set of subgraphs that consist of either a single connected component or a single *null*-variable fact (i.e., a single variable pointing to the *null* node). Non-atomic shape subgraphs correspond to conjunctions of atomic shape subgraphs and are useful intermediaries during concretization and while computing transformers.

The abstraction function $\beta^{GD} : SG \rightarrow 2^{ASSG}$ is given by

$$\beta^{GD}(G) \stackrel{\text{def}}{=} DecomposeNullVars(Components(G)) \ .$$

The function $\alpha^{GD} : 2^{SG} \rightarrow 2^{ASSG}$ is the point-wise extension of β^{GD}. Thus, $ASSG = \alpha^{GD}(SG)$ is the set of shape subgraphs in the image of the abstraction.

Note: We can extend the decomposition to avoid exponential blowups created by different sets of variables pointing to the same (non-*null*) node. However, we believe that such correlations are significant for shape analysis (as they capture different states of a single list) and abstracting them away can lead to a significant loss of precision. Hence, we do not explore this possibility in this chapter.

5.4.3 Concretization by Composition of Shape Subgraphs

Intuitively, a shape subgraph represents the set of its super shape graphs. Concretization consists of connecting shape subgraphs such that the intersection of the sets of shape graphs that they represent is non-empty. To formalize this, we define the following binary relation on shape subgraphs.

Definition 5.4.5 (Subgraph Embedding) *We say that a shape subgraph* $G' \stackrel{\text{def}}{=} (V', E', env', len')$ *is* embedded *in a shape subgraph* $G \stackrel{\text{def}}{=} (V, E, env, len)$*, denoted* $G' \sqsubseteq G$*, if there exists a function* $f : V \rightarrow V'$ *such that: (i)* $(u, v) \in E$ *iff* $(f(u), f(v)) \in E'$*; (ii)* $f(env(x)) = env'(x)$ *for every* $x \in Vars(G)$*; and (iii) for every* $x \in Vars(G') \setminus Vars(G)$*,* $f^{-1}(env'(x)) \cap V = \emptyset$ *or* $env'(x) = env'(null)$*.*[3]

Thus, for any two atomic shape subgraphs G and G', $G' \sqsubseteq G$ iff $G = G'$.

We make $\langle SSG, \sqsubseteq \rangle$ a complete partial order by adding a special element $\perp$ to represent infeasible shape subgraphs, and define $\perp \sqsubseteq G$ for every shape subgraph G. We define the operation *compose* : $SSG \times SSG \rightarrow SSG$ that accepts two shape subgraphs and returns their greatest lower bound (w.r.t. to the $\sqsubseteq$ ordering). The operation naturally extends to sets of shape subgraphs.

Example 5.4.6 *Referring to Figure 5.2 and Figure 5.3, we have* $S_1 \sqsubseteq M_1$ *and* $S_1 \sqsubseteq M_4$*, and* $compose(M_1, M_4) = S_1$*.*$\square$

The concretization function $\gamma^{GD} : 2^{ASSG} \rightarrow 2^{SG}$ is defined by

$$\gamma^{GD}(XG) \stackrel{\text{def}}{=} \{G \mid G = compose(Y), Y \subseteq XG, \text{ G is admissible}\} \ .$$

This gives us the Galois Connection $\langle 2^{SG}, \alpha^{GD}, \gamma^{GD}, 2^{ASSG} \rangle$.

[3] We define $f^{-1}(x) \stackrel{\text{def}}{=} \{y \in V \ . \ f(y) = x\}$.

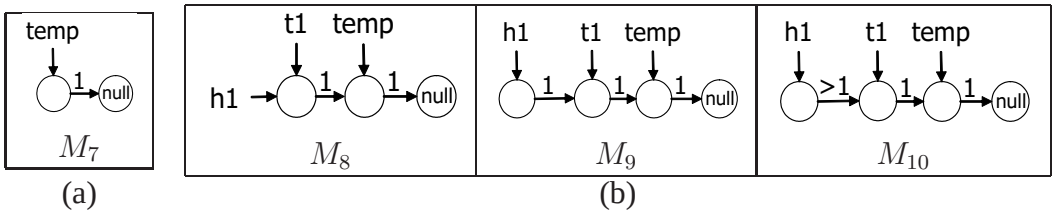


Figure 5.6: (a) A subgraph at label `L2` in Figure 5.1, and (b) Subgraphs at `L3` in Figure 5.1

Properties of the Abstraction.

Note that there is neither a loss of precision nor a gain in efficiency (e.g., such as a reduction in the size of the representation) when we decompose a single shape graph, i.e., $\gamma^{GD}(\beta^{GD}(G)) = \{G\}$. Both potentially appear when we abstract a *set of shape graphs* by decomposing each graph in a set. However, when there is no logical correlation between the different subgraphs (in the graph decomposition), we will gain efficiency without compromising precision.

Example 5.4.7 *Consider the graphs in Figure 5.2 and Figure 5.3. Abstracting* S_1 *gives* $\beta^{GD}(S_1) = \{M_1, M_4\}$. *Concretizing back, gives* $\gamma^{GD}(\{M_1, M_4\}) = \{S_1\}$. *Abstracting* S_5 *yields* $\beta^{GD}(S_5) = \{M_2, M_5\}$. *Concretizing* $\{M_1, M_2, M_4, M_5\}$ *results in* $\{S_1, S_2, S_4, S_5\}$, *which overapproximates* $\{S_1, S_5\}$. □

5.5 Efficient Abstract Transformers for the Graph Decomposition Abstraction

In this section, we show that it is hard to compute the most precise transformer for the graph decomposition abstraction in polynomial time and develop sound and efficient transformers. We demonstrate our ideas using the statement `t1.n=temp` in the running example and the subgraphs in Figure 5.6 and Figure 5.3.

An abstract transformer $T_{st} : 2^{ASSG} \rightarrow 2^{ASSG}$ is *sound* for a statement *st* if for every set of shape subgraphs *XG* the following holds:

$$(\alpha^{GD} \circ [\![st]\!]^{\#} \circ \gamma^{GD})(XG) \subseteq T_{st}(XG) \quad . \tag{5.1}$$

5.5.1 The Most Precise Abstract Transformer

We first show how the *most precise transformer* $[\![st]\!]^{GD} \stackrel{\text{def}}{=} \alpha^{GD} \circ [\![st]\!]^{\#} \circ \gamma^{GD}$ can be computed *locally*, without concretizing complete shape graphs. As observed by Distefano et al. [DOY06], the full heap abstraction transformer $[\![st]\!]^{\#}$ can be computed by considering only the *relevant* part of an abstract heap. We use this observation to create a local transformer for our graph decomposition abstraction.

The first step is to identify the subgraphs "referred" to by the statement *st*. Let $Vars(st)$ denote the variables that occur in statement *st*. We define:

- The function $modcomps_{st} : 2^{SSG} \rightarrow 2^{SSG}$ returns the shape subgraphs that have a variable in $Vars(st)$: $modcomps_{st}(XG) \stackrel{\text{def}}{=} \{G \in XG \mid Vars(G) \cap Vars(st) \neq \emptyset\}$.

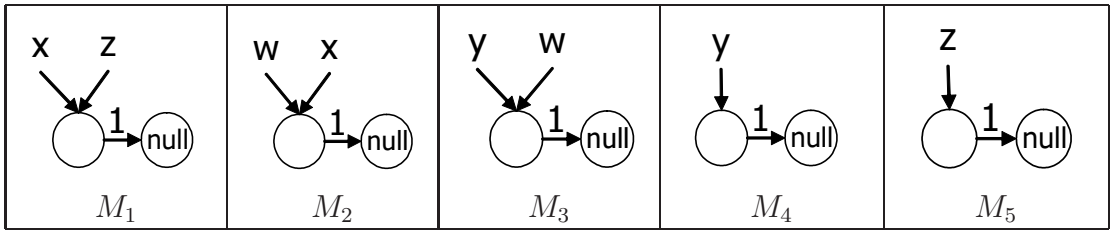


Figure 5.7: A set of shape subgraphs over the set of program variables {x,y,z,w}

- The function $\mathit{samecomps}_{st} : 2^{SSG} \rightarrow 2^{SSG}$ returns the complementary subset: $\mathit{samecomps}_{st}(XG) \stackrel{\text{def}}{=} XG \setminus \mathit{modcomps}_{st}(XG)$.

Example 5.5.1 $\mathit{modcomps}_{\texttt{t1.n=temp}}(\{M_1, \ldots, M_7\}) = \{M_1, M_2, M_3, M_7\}$ *and* $\mathit{samecomps}_{\texttt{t1.n=temp}}(\{M_1, \ldots, M_7\}) = \{M_4, M_5, M_6\}$. □

Note that the transformer $[\![st]\!]^{\#}$ operates on *complete* shape graphs. However, the transformer can be applied, in a straightforward fashion, to any shape *subgraph* G as long as G contains all variables mentioned in *st* (i.e., $\mathit{Vars}(G) \supseteq \mathit{Vars}(st)$). Thus, our next step is to compose subgraphs in $\mathit{modcomps}_{st}(XG)$ to generate subgraphs that contain all variables of *st*. However, not every set of subgraphs in $\mathit{modcomps}_{st}(XG)$ is a candidate for this composition step.

Given a set of subgraphs *XG*, a set $XG' \subseteq XG$, is defined to be *weakly feasible* in *XG* if $\mathit{compose}(XG') \neq \perp$. Further, we say that XG' is *feasible* in *XG* if there exists a subset $XR \subseteq XG$ such that $\mathit{compose}(XG' \cup XR)$ is an admissible shape graph (i.e., $\exists G \in SG : XG' \subseteq \alpha^{GD}(G) \subseteq XG$).

Example 5.5.2 *The subgraphs* M_1 *and* M_7 *are feasible in* $\{M_1, \ldots, M_7\}$*, since they can be composed with* M_4 *to yield an admissible shape graph. However,* M_1 *and* M_2 *contain common variables and thus* $\{M_1, M_2\}$ *is not (even weakly) feasible in* $\{M_1, \ldots, M_7\}$*. In Figure 5.7, the shape subgraphs* M_1 *and* M_4 *are weakly-feasible but not feasible in* $\{M_1, \ldots, M_5\}$ *(there is no way to compose subgraphs to include* w*, since* M_1 *and* M_2 *and* M_3 *and* M_4 *are not weakly-feasible.).*□

Let *st* be a statement with $k \stackrel{\text{def}}{=} |\mathit{Vars}(st)|$ variables ($k \leq 2$ in our language). Let $M^{(\leq k)}$ denote all subsets of size k or less of a set M. We define the transformer for a heap-mutating statement *st* by:

$$
\begin{aligned}
T^{GD}_{st}(XG) \stackrel{\text{def}}{=} \ & \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid M = \mathit{modcomps}_{st}(XG), R \in M^{(\leq k)}, \\
& \qquad G = \mathit{compose}(R), \mathit{Vars}(st) \subseteq \mathit{Vars}(G), \\
& \qquad R \text{ is feasible in } XG\} \\
& \textbf{in } \mathit{samecomps}_{st}(XG) \cup \alpha^{GD}(Y) \ .
\end{aligned}
$$

The transformer for an assume statement *st* is slightly different. An assume statement does not modify incoming subgraphs, but filters out some subgraphs that are not consistent with the condition specified in the assume statement. Note that it is possible for even subgraphs

in $\mathit{samecomps}_{st}(XG)$ to be filtered out by the assume statement, as shown by the following definition of the transformer:

$$\begin{aligned} T^{GD}_{st}(XG) \stackrel{\text{def}}{=} \ & \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid R \in XG^{(\leq k+1)}, \\ & \qquad G = \mathit{compose}(R), \mathit{Vars}(st) \subseteq \mathit{Vars}(G), \\ & \qquad R \text{ is feasible in } XG\} \\ & \textbf{in } \alpha^{GD}(Y) \ . \end{aligned}$$

Example 5.5.3 *The transformer $T^{GD}_{\texttt{t1.n=temp}}$: (a) composes subgraphs: $\mathit{compose}(M_1, M_7)$, $\mathit{compose}(M_2, M_7)$, and $\mathit{compose}(M_3, M_7)$; (b) finds that the three pairs of subgraphs are feasible in $\{M_1, \ldots, M_7\}$; (c) applies the local full heap abstraction transformer $[\![\texttt{t1.n=temp}]\!]^{\#}$, producing M_8, M_9, and M_{10}, respectively; and (d) returns the final result: $T^{GD}_{\texttt{t1.n=temp}}(\{M_1, \ldots, M_7\}) = \{M_4, M_5, M_6\} \cup \{M_8, M_9, M_{10}\}$.* □

Theorem 5.5.4 *The transformer T^{GD}_{st} is the most precise abstract transformer.*

Although T^{GD}_{st} applies $[\![st]\!]^{\#}$ to a polynomial number of shape subgraphs and $[\![st]\!]^{\#}$ itself can be computed in polynomial time, the above transformer is still exponential in the worst-case, because of the difficulty of checking the feasibility of R in XG. In fact, as we now show, it is impossible to compute the most precise transformer in *polynomial time*, unless P=NP.

Definition 5.5.5 (Most Precise Transformer Decision Problem) *The decision version of the most precise transformer problem is as follows: for a set of atomic shape subgraphs XG, a statement st, and an atomic shape subgraph G, does G belong to $[\![st]\!]^{GD}(XG)$?*

Theorem 5.5.6 *The most precise transformer decision problem, for the graph decomposition abstraction presented above, is NP-complete (even when the input set of subgraphs is restricted to be in the image of α^{GD}). Similarly, checking if XG' is feasible in XG is NP-complete.*

Proof:[sketch] By reduction from the EXACT COVER problem: given a universe $U = \{u_1, \ldots, u_n\}$ of elements and a collection of subsets $A \subseteq 2^U$, decide whether there exists a subset $B \subseteq A$ such that every element $u \in U$ is contained in exactly one set in B. EXACT COVER is known to be NP-complete [GJ79]. □

5.5.2 Sound and Efficient Transformers

We safely replace the check for whether R is feasible in XG by a check for whether R is weakly-feasible (i.e., whether $\mathit{compose}(R) \neq \bot$) and obtain the following transformer. (Note that a set of subgraphs is weakly-feasible iff no two of the subgraphs have a common variable; hence, the check for weak feasibility is easy.) For a heap-manipulating statement st, we define the transformer by:

$$\begin{aligned} \widehat{T^{GD}_{st}}(XG) \stackrel{\text{def}}{=} \ & \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid M = \mathit{modcomps}_{st}(XG), R \in M^{(\leq k)}, \\ & \qquad G = \mathit{compose}(R) \neq \bot, \mathit{Vars}(st) \subseteq \mathit{Vars}(G)\} \\ & \textbf{in } \mathit{samecomps}_{st}(XG) \cup \alpha^{GD}(Y) \ . \end{aligned}$$

For an assume statement *st*, we define the transformer by:

$$\begin{aligned}\widehat{T^{GD}_{st}}(XG) \stackrel{\text{def}}{=} \;& \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid R \in XG^{(\leq k+1)}, \\ & \qquad G = \mathit{compose}(R) \neq \bot, \mathit{Vars}(st) \subseteq \mathit{Vars}(G)\} \\ & \textbf{in } \alpha^{GD}(Y) \;\; .\end{aligned}$$

By definition, (5.1) holds for $\widehat{T^{GD}_{st}}$. Thus, $\widehat{T^{GD}_{st}}$ is a sound transformer.

We apply several engineering optimizations to make the transformer $\widehat{T^{GD}_{st}}$ efficient in practice: (i) by preceding statements of the form `x=y` and `x=y.n` with an assignment `x=null`, we specialize the transformer to achieve linear time complexity; (ii) we avoid unnecessary compositions of shape subgraphs for statements of the form `x.n=y` and `assume(x==y)`, when a shape subgraph contains both `x` and `y`; and (iii) assume statements do not change subgraphs, therefore we avoid performing explicit compositions and propagate atomic subgraphs.

5.5.3 An Incremental Transformer

The goal of an *incremental* transformer is to compute $\widehat{T^{GD}_{st}}(XG \cup \{D\})$ by reusing $\widehat{T^{GD}_{st}}(XG)$. We define the transformer for a heap-manipulating statement *st* by:

$$\begin{aligned}\widehat{T^{GD}_{st}}(XG \cup \{D\}) \stackrel{\text{def}}{=} \;& \textbf{if } D \in \mathit{modcomps}_{st}(\{D\}) \\ & \quad \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid M = \mathit{modcomps}_{st}(XG \cup \{D\}), \\ & \qquad R \in M^{(\leq k)}, D \in R, \\ & \qquad G = \mathit{compose}(R) \neq \bot, \mathit{Vars}(st) \subseteq \mathit{Vars}(G)\} \\ & \quad \textbf{in } \widehat{T^{GD}_{st}}(XG) \cup \alpha^{GD}(Y) \\ & \textbf{else} \\ & \quad \widehat{T^{GD}_{st}}(XG) \cup \{D\} \;\; .\end{aligned}$$

Here, if the new subgraph D is not affected by the statement, we simply add it to the result. Otherwise, we apply the local full heap abstraction transformer only to subgraphs composed from the new subgraph (for sets of subgraphs not containing D, the result has been computed in the previous iteration).

For an assume statement *st*, we define the transformer by:

$$\begin{aligned}\widehat{T^{GD}_{st}}(XG \cup \{D\}) \stackrel{\text{def}}{=} \;& \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid R \in (XG \cup \{D\})^{(\leq k+1)}, \\ & \qquad D \in R, G = \mathit{compose}(R) \neq \bot, \mathit{Vars}(st) \subseteq \mathit{Vars}(G)\} \\ & \textbf{in } \widehat{T^{GD}_{st}}(XG) \cup \alpha^{GD}(Y) \;\; .\end{aligned}$$

Again, we apply the transformer only to (composed) subgraphs containing D.

5.6 Prototype Implementation and Empirical Results

Implementation. We implemented the analyses based on the full heap abstraction and the graph decomposition abstraction described in previous sections in a system that supports memory deallocation and assertions of the form `assertAcyclicList(x)`, `assertCyclicList(x)`, `assertDisjointLists(x,y)`, and `assertReach(x,y)`. The analysis checks absence of null dereferences, absence of memory leakage, misuse of dangling pointers, and (manually added) shape assertions. The system supports non-recursive procedure calls via call strings and unmaps variables as they become dead.

Example Programs. We use a set of examples, described in Table 5.2, to compare the full heap abstraction-based analysis with the graph decomposition-based analysis. The first set of examples consists of standard list manipulating algorithms operating on a single list (except for `merge`). The second set of examples consists of programs manipulating multiple lists. We created the serial port driver example incrementally, first modeling 4 of the lists used by the device and then 5. The `queue_2_stacks` program was constructed to show a case where the graph decomposition-based analysis loses precision—determining that a queue is empty requires maintaining a correlation between the two (empty) lists. The code appears in Appendix B.1.

Precision. The results of running the analyses appear in Table 5.3. The graph decomposition-based analysis failed to prove that the pointer returned by `getLast` is non-null[4], and that a dequeue operation is not applied to an empty queue in `queue_2_stacks`. On all other examples, the graph decomposition-based analysis has the same precision as the analysis based on the full heap abstraction.

Performance. The graph decomposition-based analysis is slightly less efficient than the analysis based on the full heap abstraction on the standard list examples. For the examples manipulating multiple lists, the graph decomposition-based analysis is faster by up to a factor of 212 (in the `serial_5_lists` example) and consumes considerably less space. These results are also consistent with the number of states generated by the two analyses.

5.7 Related Work

Single-graph Abstractions. Some early shape analyses used a single shape graph to represent the set of concrete states [JM81b, CWZ90, SRW98]. As noted earlier, it is possible to generalize our approach and consider different strategies for decomposing shape graphs. Interestingly, the single shape graph abstractions can be seen as one extreme point of such a generalized approach, which relies on a decomposition of a graph into its set of edges. The decomposition strategy we presented in this chapter leads to a more precise analysis.

[4] A simple feasibility check while applying the transformer of the assertion would have eliminated the subgraph containing the null pointer.

Table 5.2: Benchmarks used to compare the full heap analysis with the graph decomposition analysis

Benchmark	**Description**
`create`	Creates new elements and adds them to an acyclic list
`delete`	Deletes a cell chosen non-deterministically in an acyclic list
`deleteAll`	Deletes all elements of an acyclic list
`getLast`	Returns a pointer to the last element of an acyclic list
`getLast_cyclic`	Returns a pointer to the last element of a cyclic list
`insert`	Inserts an element to an acyclic list in a position chosen non-deterministically
`merge`	Merges two acyclic lists (simulates merging ordered lists)
`removeSeg`	Removes a sublist from a cyclic list
`reverse`	Reverses an acyclic list
`reverse_cyclic`	Applying reversal to a cyclic list
`reverse_pan`	Applying reversal to a panhandle list
`rotate`	Moves the first element of an acyclic list to the tail
`search_nullderef`	A buggy implementation of a list search
`swap`	Swaps the first two elements of an acyclic list
Benchmarks with Multiple Lists	
`enqueueEvents`	The running example
`queue_2_stacks`	Test for an implementation of a queue using two lists
`join_5`	A program joining 5 acyclic lists
`split_5`	A program that splits a list into 5 lists
`1394diag`	Modeling aspects of the diagnostics program for the 1394 firewire device driver
`serial_4_lists`	Modeling aspects of 4 lists in the serial port device driver
`serial_5_lists`	Modeling aspects of 5 lists in the serial port device driver

Table 5.3: Time, space, number of states (shape graphs for the analysis based on full heap abstraction and subgraphs for the graph decomposition-based analysis), and number of errors reported. Rep. Err. and Act. Err. are the number of errors reported, and the number of errors that indicate real problems, respectively. #Loc indicates the number of CFG locations. F.H. and G.D. stand for full heap and graph decomposition, respectively

Benchmark		**Time** (sec.)		**Space** (Mb.)		**#States**		**R. Err./A. Err.**	
(**#Loc**)		**F.H.**	**G.D.**	**F.H.**	**G.D.**	**F.H.**	**G.D.**	**F.H.**	**G.D.**
`create`	(11)	0.03	0.19	0.3	0.3	27	36	0/0	0/0
`delete`	(25)	0.17	0.27	0.8	0.9	202	260	0/0	0/0
`deleteAll`	(12)	0.05	0.09	0.32	0.36	35	64	0/0	0/0
`getLast`	(13)	0.06	0.13	0.42	0.47	67	99	0/0	1/0
`getLast_cyclic`	(13)	0.08	0.09	0.39	0.41	53	59	0/0	0/0
`insert`	(23)	0.14	0.28	0.75	0.82	167	222	0/0	0/0
`merge`	(37)	0.34	0.58	2.2	1.7	517	542	0/0	0/0
`removeSeg`	(23)	0.19	0.33	0.96	1.0	253	283	0/0	0/0
`reverse`	(13)	0.09	0.12	0.47	0.46	82	117	0/0	0/0
`reverse_cyclic`	(14)	0.14	0.36	0.6	1.4	129	392	0/0	0/0
`reverse_pan`	(12)	0.2	0.6	0.9	2.2	198	561	0/0	0/0
`rotate`	(17)	0.05	0.08	0.3	0.4	33	50	0/0	0/0
`search_nulldref`	(7)	0.06	0.1	0.4	0.4	48	62	1/1	1/1
`swap`	(13)	0.05	0.09	0.3	0.4	35	62	0/0	0/0
`enqueueEvents`	(49)	0.2	0.2	1.2	0.7	248	178	0/0	0/0
`queue_2_stacks`	(61)	0.1	0.2	0.6	0.7	110	216	0/0	1/0
`join_5`	(68)	12.5	0.5	67.0	2.4	14,704	1,227	0/0	0/0
`split_5`	(47)	28.5	0.3	126.2	1.7	27,701	827	0/0	0/0
`1394diag`	(180)	26.2	1.8	64.7	8.5	10,737	4,493	0/0	0/0
`serial_4_lists`	(248)	36.9	1.7	230.1	11.7	27,851	6,020	0/0	0/0
`serial_5_lists`	(278)	552.6	2.6	849.2	16.4	89,430	7,733	0/0	0/0

Partially Disjunctive Heap Abstraction. In Chapter 4, we described a heap abstraction based on merging sets of graphs with the same set of nodes into one (approximate) graph. The abstraction in the current chapter is based on decomposing a graph into a set of subgraphs. The abstraction in Chapter 4 suffers from the same exponential blow-ups as the full heap abstraction for our running example and examples containing multiple independent data structures.

Heap Analysis by Separation. Yahav and Ramalingam [YR04] and Hackett and Rugina [HR05] decompose heap abstractions to separately analyze different parts of the heap (e.g., to establish the invariants of different objects). A central aspect of the separation-based approach is that the analysis/verification problem is itself decomposed into a set of problem instances, and the heap abstraction is specialized for each problem instance and consists of one sub-heap consisting of the part of the heap relevant to the problem instance, and a coarser abstraction of the remaining part of the heap ([HR05] uses a points-to graph). In contrast, we simultaneously maintain abstractions of different parts of the heap and also consider the interaction between these parts. (E.g., it is possible for our decomposition to dynamically change as components get connected and disconnected.)

Application to Other Shape Abstractions. Lev-Ami et al. [LAIS06] present an abstraction that could be seen as an extension of the full heap abstraction in this chapter to more complex data structures, e.g., doubly-linked lists and trees. We believe that applying the techniques in this chapter to their analysis is quite natural and can yield a more scalable analysis for more complex data structures. Distefano et al. [DOY06] present a full heap abstraction based on separation logic, which is similar to the full heap abstraction presented in this chapter. We therefore believe that it is possible to apply the techniques in this chapter to their analysis as well. TVLA [LAS00] is a generic shape analysis system that uses canonical abstraction. We believe it is possible to decompose logical structures in a similar way to decomposing shape subgraphs and extend the ideas in this chapter to TVLA.

Decomposing Heap Abstractions for Interprocedural Analysis. Gotsman et al. [GBC06] and Rinetzky et al. [RBR+05, RSY05] decompose heap abstractions to create procedure summaries for full heap+ abstractions. This kind of decomposition, which does not lead to loss of precision (except when cutpoints are abstracted), is orthogonal to our decomposition of heaps, which is used to reduce the number of abstract states generated by the analysis. We believe it is possible to combine the two techniques to achieve a more efficient interprocedural shape analysis.

Chapter 6

Cartesian Subheap Decomposition

We demonstrate shape analyses that can achieve a state space reduction exponential in the number of threads compared to the state-of-the-art analyses, while retaining sufficient precision to verify sophisticated properties such as *linearizability*. The key idea is to abstract the global heap by decomposing it into (not necessarily disjoint) subheaps, abstracting away some correlations between them. These new shape analyses are instances of an analysis framework based on heap decomposition. This framework allows rapid prototyping of complex static analyses by providing efficient abstract transformers given user-specified decomposition schemes. Initial experiments confirm the value of heap decomposition in scaling concurrent shape analyses.

6.1 Introduction

The problem of verifying concurrent programs that manipulate heap-allocated data structures is challenging: it requires considering arbitrarily interleaved threads manipulating unbounded data structures. Both heap-allocated data structures and concurrency can introduce state explosion. Their combination only makes matters worse. This chapter develops new static analysis algorithms that address the state space explosion problem in a systematic and generic way. The result of these analyses can be used to automatically establish interesting properties of concurrent heap-manipulating programs such as the absence of null dereferences, the absence of memory leaks, the preservation of data structure invariants, and *linearizability* [HW90].

The Intuition.

Typical programs manipulate a large number of (instances of) data structures (possibly nested within other data structures). Each individual data structure can usually be in one of several different states (even in an abstract representation). This can lead to a combinatorial explosion in the number of distinct abstract states that can arise during abstract interpretation.

The essential idea we pursue is that of *decomposing* the heap into multiple subheaps and abstracting away some correlations between the subheaps. Decomposition allows reusing subheaps that were decomposed from different heaps, thus representing a set of heaps more compactly (and more abstractly). For example, consider a program maintaining k disjoint lists. A powerset-based shape analysis such as the one in [SRW02] uses a lattice whose height is exponential in k. An abstraction that ignores the correlations between the k lists reduces the

lattice height to be linear in k, leading to exponentially faster analysis. (The savings come from not maintaining the correlations between different states of the different lists, which we observe are often irrelevant for a specific property of interest.) Similar situations arise in the kind of multithreaded programs discussed earlier, where the size of the state space is a function of the number of threads rather than the number of data structures. In this chapter, we allow decomposing the heap into non-disjoint (i.e., overlapping) subheaps, which is important for handling programs with fine-grained concurrency (where different threads can simultaneously access the same objects) in a thread-modular way.

Fine-Grained Concurrency.

Fine-grained concurrent heap-manipulating programs allow multiple threads to use the same data structure *simultaneously*. They trade the simplicity of the single-thread-owning-a-data-structure model, which is at the heart of the coarse-grained concurrency approach, to achieve a higher degree of concurrency. However, the additional performance comes with a price: these programs are notoriously hard to develop and prove correct, even when the manipulated data structures are singly-linked lists (see, e.g., [DDG+04]).

It is hard to employ thread-modular approaches that exploit locking [GBCS07] to analyze fine-grained concurrent programs because they have *intentional* (benign) data-races. Thus, state-of-the-art shape analyses capable of verifying intricate properties of fine-grained concurrent heap-manipulating programs, e.g., linearizability (explained in Section 6.3), track all correlations between the states of all the threads [ARR+]. This makes these analyses hard to scale. For example, the shape analysis in [ARR+] handles at most 3 threads.

It is interesting to observe, however, that it is often the case that although proving properties of these programs requires tracking sophisticated correlations between every thread and the part of the heap that it manipulates, the correlations between the states of different threads is often irrelevant. Intuitively, this is because fine-grained concurrent programs are often written in a way which *attempts* to ensure the correct operation of every thread *regardless* of the actions taken by other threads. This programming paradigm makes these programs an ideal match with our approach explained below.

The Conceptual Framework.

To permit the use of heap decomposition in several settings, we first present it as a parametric abstraction that can be tuned by the analysis designer in three ways:

Decomposition: Specify along what lines a concrete heap should be decomposed into (possibly overlapping) subheaps. One of the strengths of the specification mechanism is that the decomposition of a heap depends on its properties. This allows us, for example, to decompose the state of a concurrent program based on the association between threads and data-structures in that state, which is usually not known a priori.

Subheap abstraction: Create a bounded abstract heap representation from concrete subheaps (which are unbounded). Subheap abstractions can be obtained from existing whole-heap abstractions that satisfy certain properties.

Combiner Sets: The framework is parametric with respect to transformers. Computing sound and precise transformers for statements is quite challenging with a heap decomposition. Transforming each subheap independently can end up being very imprecise (or potentially

incorrect, if not done carefully), especially when subheaps overlap. At the other extreme, combining subheaps together into a full heap prior to transforming it can be very inefficient and defeats the purpose of using heap decomposition. Achieving the desired precision and efficiency, without compromising soundness, can be tricky. Our framework allows the analysis designer to specify only which subheaps should be combined together for a given transformer, called combiner sets. The framework automatically generates a corresponding sound transformer, letting the analysis designer easily explore alternatives without worrying about soundness.

HeDec.

We implemented our conceptual framework for the family of canonical abstractions [SRW02] in a system called HeDec (for **He**ap **Dec**omposition), which is publicly available. This implementation retains the parametricity of the conceptual framework, which allows analysis designers to rapidly prototype different shape analysis algorithms by defining heap decomposition schemes.

Instances of the Framework.

We have used our framework to develop several shape analyses, including the following, and have implemented these analyses in HeDec.

(a) A shape analysis for sequential programs manipulating singly-linked lists that abstracts away the correlations between disjoint lists . The resultant shape analysis algorithm emulates the algorithm of Chapter 5, with some interpretative overhead. Unlike the tedious proof of soundness of Chapter 5, the soundness of this instance immediately follows from the soundness of the underlying subheap abstraction.

(b) A new shape analysis for sequential programs manipulating singly-linked lists and trees by abstracting away the correlations between segments which do not contain an element pointed-to by a variable. We confirmed that it is precise enough to prove memory safety and preservation of data-structure invariants. This is encouraging for scaling shape analysis for programs with densely connected heaps.

(c) A shape analysis for fine-grained concurrent programs with a bounded number of threads which is precise enough to prove memory safety and preservation of data-structure invariants. Here, we obtain exponential speed-up in terms of time and space, in comparison to similar whole-heap analysis without decomposition. Our algorithm goes beyond [GBCS07] by supporting fine-grained concurrency and handling programs with intentional data races.

(d) A shape analysis algorithm for concurrent programs with a bounded number of threads that manipulate singly-linked lists, which proves linearizability. The resultant algorithm is exponentially faster than the one in [ARR$^+$], being polynomial in the number of threads. Our initial empirical results confirm that our algorithm is able to prove linearizability with 20 threads, ten times more than in [ARR$^+$].

Main Results.

The contributions of this chapter can be summarized as follows:

1. We present a generic analysis framework (in an abstract interpretation setting) for exploiting state decomposition effectively. The main technical contributions are in introducing a family of sound abstract transformers that admit flexibly exploring the efficiency/precision spectrum.
2. We propose scalable analyses for several interesting problems involving coarse-grained as well as fine-grained concurrency, including proving linearizability. These algorithms scale much better (e.g., polynomially) over the number of threads than the previous algorithms for these problems.
3. The implementation of the framework for canonical abstraction is publicly available, together with the above mentioned analyses, as well as other benchmarks, which show the benefit of the approach.

6.1.1 Outline

The rest of this chapter is organized as follows. In Section 6.2, we demonstrate heap decomposition for fine-grained concurrent programs. In Section 6.3, we describe an analysis based on heap decomposition for proving linearizability of non-blocking data structures. In Section 6.4 we present the technical details of our abstract domain and its transformers. In Section 6.5 we report on our experiments with HeDec. In Section 6.6, we discuss related work, and in Section 6.7, we conclude the chapter. Appendix C.1 contains formal proofs for Section 6.4. Appendix C.2 describes optimizations implemented in HeDec. Appendix C.3 contains a case-study of heap decomposition for the two-lock queue algorithm.

6.2 Heap Decomposition for Fine-Grained Concurrency

In this section, we develop decomposition schemes for performing shape analysis of fine-grained concurrent programs and show that HeDec can be used to automatically obtain shape analysis implementations that are precise enough to prove the desired properties of programs (the absence of null pointer dereferences, absence of memory leaks, and data structure invariants) while scaling up to a large number of threads. The material in this section is presented informally, deferring formal definitions and technical details to Section 6.4.

6.2.1 Decomposing Non-blocking Implementations

A Running Example. Figure 6.1 shows a simple running example of a non-blocking stack implementation from [Tre86]. Producers push elements onto the stack by allocating an element, copying the current global pointer to the top of the stack, connecting the new element to that copied top, and then using CAS (**C**ompare **A**nd **S**wap) to atomically check that the top of the stack has not changed and replace it with the new element. Consumers pop elements from the stack by copying the current global pointer to top and recording its next element and then using CAS to atomically check that the top of the stack has not changed and replace it with the new top, i.e., the recorded next element. In both cases, a failed CAS results in a restart.

```
#define EMPTY -1
typedef int data_type;
typedef struct node t {
      data_type d;
      struct node t *n
} Node;

typedef struct stack t {
      struct node t *Top;
} Stack;

[1]   void push(Stack *S, data_type v){
[2]     Node *x = alloc(sizeof(Node));
[3]     x->d = v;
[4]     do {
[5]       Node *t = S->Top;
[6]       x->n = t;
[7]     } while (!CAS(&S->Top,t,x));
[8]   }

[9]   data_type pop(Stack *S){
[10]    do {
[11]      Node *t = S->Top;
[12]      if (t == NULL)
[13]        return EMPTY;
[14]      Node *s = t->n;
[15]      data_type r = t->d;
[16]    } while (!CAS(&S->Top,t,s));
[17]    return r;
[18]  }
```

Figure 6.1: A non-blocking stack implementation

The goal here is to prove the absence of null pointer dereferences, absence of memory leaks, and the preservation of data structure invariants, i.e., that `stack` points to an acyclic list.

Concrete Execution. Figure 6.2(a) shows an example of two states occurring in the non-blocking implementation shown in Figure 6.1; for now ignore the $corr$ annotations (which is used by the linearizability analysis in the next section). The figure shows two consumer threads and two producer threads. Both **cons1** and **prod1** can succeed with the CAS if they are the next threads to be scheduled. Concrete states are depicted by graphs. To avoid clutter the `data` field is not shown. Hexagonal nodes denote thread objects and square nodes denote list elements. The program label of every thread is written inside the hexagon. Edges from text labels to nodes correspond to global pointers (`Top`). Labeled edges from thread nodes to list nodes denote thread-local pointer variables (`t` and `x`). Edges between list nodes, labeled by `n` correspond to the `next` field of the list.

Exponential State Space. There are several sources of exponential explosion in the state space exploration of the stack algorithm. The first one is the correlation between the program locations of the different threads. The second source is the next pointers of the just allocated elements. The stack can grow after the next pointer has already been set, but before the CAS, thus the next pointers of the different producers can point to all possible stack elements and have all possible aliasing between each other. The third source of state-space explosion is the recorded next pointer of the consumer threads. Note that the state space explosion occurs even if the list has a bounded number of elements. This is a general problem when maintaining correlations between the properties of different threads. Exponential blow-ups also occur in sequential programs because of aliasing. However, for the purpose of our analysis, these correlations are unimportant and tracking them is pointless and only reduces the efficiency of the analysis.

Heap Decomposition Abstraction. We reduce the size of the state space by decomposing the heap into a set (or tuple) of subheaps and abstractly interpreting the program over the subheaps.

For each subheap to be used in the decomposition, a user of HeDec specifies the part of the heap it should include. This is done by defining a *location selection predicate*, which specifies the subset of the nodes in the state for which abstract properties (such as aliasing, heap-reachability, etc.) are maintained. For each location selection predicate, the program state is projected onto the nodes satisfying that predicate, thus obtaining a *substate* of the original state. We refer to the domain of substates pertaining to a location selection predicate pt as the *subdomain* of pt.

The Decomposition Scheme. For the purpose of our analysis, we define for each thread t the location selection predicate $pt[t]$ that holds for: (a) the thread object of t, (b) the objects pointed-to by its local variables (`t` and `x`), and (c) the objects pointed-to by the global variables (`Top`). In addition, we define the location selection predicate *Globals*, which holds for the objects reachable from global variables.

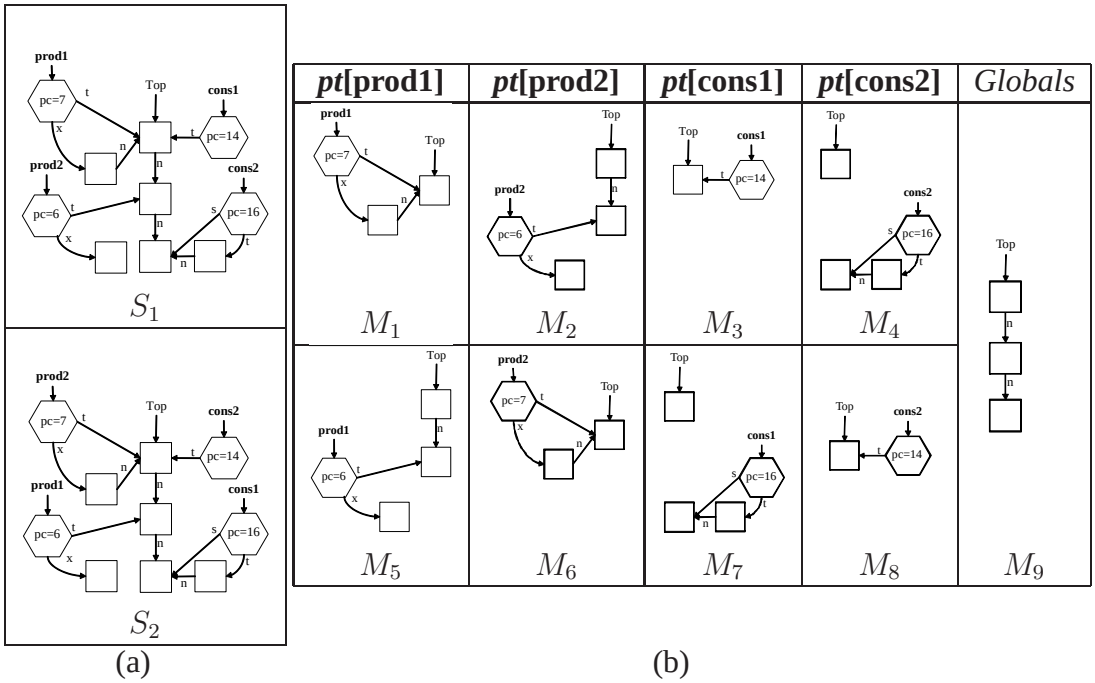


Figure 6.2: (a) Two concrete states in the non-blocking stack implementation shown in Figure 6.1; and (b) The decomposed states abstracting the full states in (a). The names of the subdomains appear above the substates

Figure 6.2(b) shows the result of applying the decomposition scheme explained above to the states in Figure 6.2(a). Notice that different location selection predicates may occasionally overlap. For example, in the decomposition explained above, the objects reachable from the global variables appear in each subheap.

Intuitively, the meaning of a substate M, decomposed by a location selection predicate $p(v)$, is the set of all full states that contain M and any disjoint substate M', such that the objects in M satisfy $p(v)$ and the objects in M' do not satisfy $p(v)$. A sequence of sets of substates $\{M_1, M_5\} \times \{M_2, M_6\} \times \{M_3, M_7\} \times \{M_4, M_8\} \times \{M_9\}$ represents the set of full states obtained by choosing one structure from each subdomain and intersecting their meanings. For example, composing the substates $\{M_1, M_2, M_3, M_4, M_9\}$ together yields S_1 and composing the substates $\{M_5, M_6, M_7, M_8, M_9\}$ together yields S_2. The loss of precision by the abstraction can be observed by the fact that other compositions, such as $\{M_1, M_6, M_7, M_8, M_9\}$ yield full states other than S_1 and S_2.

State Space Savings. In general, for n threads, if the set of objects reachable from a thread is bounded, then the number of substates resulting from the reachability-based decomposition is linear in n (even though the number of full states generated by the program is exponential in n). Although we do not show the state space reduction in the figures, one can imagine how running the program with n threads generates states similar to the ones in Figure 6.2(a). By permuting the thread ids between producers threads and between consumer threads, we obtain an exponential number of full states that are all reachable by the program execution. Decomposing these states results in a number of substates that is linear in n.

Transformers. HeDec is guaranteed to be sound, in the sense that when the analysis terminates all reachable concrete states are represented by some abstract state.

While the abstraction ignores correlations between substates, transforming substates in isolation using an "independent-attribute" style of analysis [NNH99] leads to debilitating loss of precision. For example, the analysis executes the statement `6: x->n=t` where thread **prod1** is scheduled. Substate M_3 does not contain information about the local variables of thread **prod1**. Therefore, M_3 also represents a state S_{bad} in which the local variables `t` and `x` of thread **prod1** point to the first cell and to the last cell of the list, respectively. Thus, a conservative transformer of `6: x->n=t` must emit a warning about a possible creation of a cyclic list.

To avoid this kind of loss of precision, a user of HeDec can specify which substates, obtained from different location selection predicates, should be (temporarily) composed by the transformer. This is done in terms of *combiner sets*, which are subsets of node selection predicates. In this example, for the transformer of `6: x->n=t`, we can specify the combiner sets $\{pt[\textbf{prod1}], pt[\textbf{prod2}]\}$, $\{pt[\textbf{prod1}], pt[\textbf{cons1}]\}$, $\{pt[\textbf{prod1}], pt[\textbf{cons2}]\}$, and $\{pt[\textbf{prod1}], pt[Globals]\}$. Then, the generated transformer composes, separately, the substates $\{M_1, M_5\}$ with each of the sets of substates $\{M_2, M_6\}$, $\{M_3, M_7\}$, $\{M_4, M_8\}$, and $\{M_9\}$. For the substates composed with M_5 (which is the only substate in the **prod1**-subdomain that can execute `6: x->n=t`) the transformer updates the `n` field appropriately, avoiding the false alarm. Finally, the transformer decomposes the substates again into each one of the subdomains. The resulting abstract substates are the same as in Figure 6.2, except that M_5 has an `n`-link between the object pointed-to by `t` and the object pointed-to by `x` and its program counter is 7.

This example shows how, by combining a small number (linear in the number of location selection predicates, in this case) of substates decomposed by different predicates, the transformer is able to increase precision without incurring an unreasonable time/space blow-up.

A Methodology for Combiner Sets.

We now briefly discuss the issue of choosing combiner sets for a transformer (which is done by the analysis designer in our framework). Every transformer can be thought of as having a *frame* as well as a *footprint*. The frame identifies the part of a program state that is completely irrelevant to the transformer. Thus, it contains no information that is either used or modified by the transformer. The footprint is the complement and contains adequate information to perform the transformer as precisely as possible.

A straightforward approach for computing the footprint of an operation affecting several subdomains is combining all the affected subdomains. Unfortunately, this approach might be too expensive. We apply a more efficient approach, which according to our experience is precise enough. Specifically, for each operation we choose a set of *core subdomains* which contain the heap objects and variables that participate in the operation. We compute the *core footprint* by combining the core subdomains (in practice, there are usually no more than two). We then independently combine the core footprint with the other affected subdomains. For example, the core subdomains for a statement of the form "`x->f = g`", where `x` of thread t is a local variable and `g` is a global variable, are the subdomains containing thread t and the subdomain of the global variable `g`. The affected subdomains are any subdomains which may alias these variables.

Conditional branches pose an interesting puzzle. Note that because the condition essentially filters states it can affect *all* subdomains. Thus, for a conditional "`if (x == g)`", we identify the core subdomains to be the ones containing (the nodes pointed-to by) `x` and `g`. However, we will independently combine them with all other subdomains.

6.3 Using Decomposition to Prove Linearizability

Linearizability [HW90] is one of the main correctness criteria for implementations of concurrent data structures. Informally, a concurrent data structure is said to be linearizable if the concurrent execution of a set of operations on it is equivalent to some sequential execution of the same operations, in which the global order between non-overlapping operations is preserved. The equivalence is based on comparing the arguments and results of operations (responses). The permitted behavior of the concurrent object is defined in terms of a specification of the desired behavior of the object in a sequential setting. Linearizability is a widely-used concept, and there are numerous non-automatic proofs of linearizability for concurrent objects.

Verifying linearizability is challenging because it requires correlating any concurrent execution with a corresponding permitted sequential execution. Verifying linearizability for concurrent dynamically allocated linked data structures is particularly challenging, because it requires correlating executions that may manipulate memory states of unbounded size. Interestingly, proving linearizability does not require directly proving safety properties such as preservation of data structure invariants. Instead, one can first prove that the sequential implementation satisfies the required safety properties and then prove that the concurrent implementation is linearizable, thereby, satisfies the safety property. Finally, linearizability of complex systems can be shown by separately proving the linearizability of each of the individual data structure implementations.

Intuitively, we verify linearizability by representing, in the concrete state, both the state of the concurrent program and the state of the reference sequential program. Each element entered into the data structure is correlated at linearization points with the matching object from the sequential execution. This works well under abstraction when the differences between the heaps of the sequential and concurrent implementations are bounded. The details are described in [ARR+].

In order to guarantee that the shape analysis scales-up in the number of threads, in HeDec we have defined a decomposition scheme that abstracts away the correlations between the threads (as in Section 6.2). Also, there is no need to track reachability from program variables. Instead, the subheap abstraction tracks elements whose values in the sequential and the concurrent implementations are correlated.

6.3.1 A Decomposition Scheme for Linearizability Analysis

In HeDec, we have defined such a decomposition scheme by decomposing the heap into $n + 1$ components where n is the number of threads: (i) For each thread the objects pointed-to by local variables of the thread and objects pointed-to by global variables. This captures the relationships between local pointer variables and global pointer variables. Each subheap abstracts away the values of the local variables of the other threads. (ii) A separate subheap with the objects pointed-to by global variables and the part of the heap already correlated with the se-

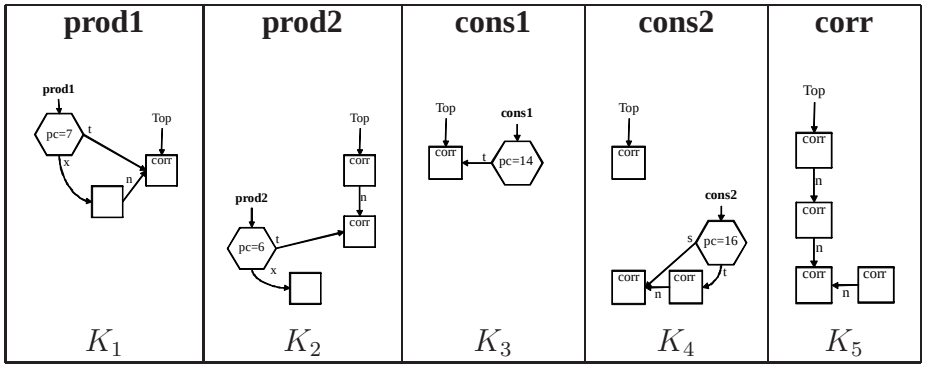


Figure 6.3: The decomposed states abstracting the full state S_1 in Figure 6.2(a). The names of the sub-domains appear above each substate

quential execution. Here, the values of the local variables of all the threads are abstracted away. We call this the *corr* subdomain as it represents the correlated elements. Figure 6.3 shows the effect of applying this decomposition to the full state S_1 in Figure 6.2(a).

Intuitively, this decomposition is appropriate for verifying linearizability for the program in Figure 6.1 because of the following. The list consisting of correlated objects changes locally when a thread executes a successful `CAS` operation. In fact, successful `CAS` operations are the linearization points for this program. Precisely interpreting these operations (`CAS(&S->Top,t,x)` and `CAS(&S->Top,t,s)`) in the analysis requires tracking correlations between local and global variables, which we do in the subheap we decompose for each thread.

The subheap captured by the *corr* subdomain is important only during successful `CAS` operations, which is when a (non-correlated) node allocated by a thread is passed into the list. Maintaining the subheap of the *corr* subdomain for each thread is wasteful, and thus we separate these correlations into different subdomains.

The important thing to notice is that all the exponential explosion in the state space that is due to the number of threads in the full heap is eliminated by this decomposition. The number of possible subheaps of each thread becomes independent of the number of threads in the system (for more than two threads).

Transformers. The combiner sets used in the transformers of the analysis are the application of the methodology described in Section 6.2.1 to this decomposition scheme. For example, copying a global variable into a local variable does not require decomposition as the executing thread has all the needed information. Copying a local variable into a global variable combines the subdomain of the executing thread with each of the other subdomains. Other operations that change the global state such as changes to pointer fields and performing CAS operations behave the same. Dereferencing a pointer requires composing the subdomain for the current thread and the *corr* subdomain as the information on the next element of the stack is not available in the thread's subdomain.

6.4 The Heap Decomposition Abstraction

In this section, we formally define our new parametric heap abstraction and a family of sound abstract transformers.

6.4.1 Heap Decomposition as a Cartesian Product of Subheaps

We first define the (parameterized) abstract domain of decomposed heaps.

Let $(\Sigma, \preceq, \otimes)$ be a semilattice, where elements of Σ represent (total and partial) states, $\preceq$ is a partial ordering on Σ capturing the "is a substate of" relation, and $\otimes$ is the join operation with respect to $\preceq$ (which composes substates together). We extend $\otimes$ to sets of states as follows. Let $X_1 \subseteq \Sigma$ and $X_2 \subseteq \Sigma$. We define $X_1 \otimes X_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in X_1, \sigma_2 \in X_2\}$. For purposes of abstraction, we shall also make use of the information ordering defined by $\sigma \sqsubseteq \sigma'$ iff $\sigma' \preceq \sigma$.

Let $(\mathcal{P}(\Sigma), \sqsubseteq)$ denote the powerset domain of Σ with the Hoare ordering: i.e., for every $X, Y \subseteq \Sigma$, we write $X \sqsubseteq Y$ iff $\forall x \in X : \exists y \in Y : x \sqsubseteq y$.

A *substate extraction* function is a function $\eta : \Sigma \to \Sigma$ that satisfies $\eta(\sigma) \preceq \sigma$. Assume we have a sequence of k substate extraction functions η_1 to η_k. We use the k-fold product $\mathcal{P}(\Sigma)^k$ = $\mathcal{P}(\Sigma) \times \cdots \times \mathcal{P}(\Sigma)$ as our domain of abstract states. The abstraction function $\alpha : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)^k$ is defined by:

$$\alpha(S) = (\hat{\eta}_1(S), \ldots, \hat{\eta}_k(S)) \tag{6.1}$$

where $\hat{\eta}_i$ is the pointwise extension of η_i defined by:

$$\hat{\eta}_i(S) = \{\eta_i(\sigma) \mid \sigma \in S\} \tag{6.2}$$

We define the meaning, or *concretization*, of a tuple $I_1, \ldots, I_k \in \mathcal{P}(\Sigma)^k$ by

$$\gamma(I_1, \ldots, I_k) = I_1 \otimes \cdots \otimes I_k. \tag{6.3}$$

Example 6.4.1 *Let S denote the set of states $\{S_1, S_2\}$ shown in Figure 6.2(a). For any thread t, we define the predicate pt$[t]$ to be true for: (a) the thread object of t, (b) the objects pointed-to by its local variables (`t` and `x`), and (c) the objects pointed-to by the global variables (`Top`). In addition, we define the location selection predicate Globals, which holds for the objects reachable from global variables. Given any predicate p, the substate extraction function δ_p maps a state σ to the substate consisting only of the locations satisfying p. We define η_1 to be $\delta_{pt[\textbf{prod1}]}$, η_2 to be $\delta_{pt[\textbf{prod2}]}$, η_3 to be $\delta_{pt[\textbf{cons1}]}$, η_4 to be $\delta_{pt[\textbf{cons2}]}$, and η_5 to be $\delta_{Globals}$. Now, $\eta_1(S_1) = M_1$, $\eta_2(S_1) = M_2$, $\eta_3(S_1) = M_3$, $\eta_4(S_1) = M_4$, and $\eta_5(S_1) = M_9$.*

6.4.2 Abstract Transformers

We now turn our attention to the more challenging aspect of decomposition: computing sound abstract transformers.

The semantics of a program statement is given by a function $\tau : \Sigma \to \mathcal{P}(\Sigma)$. We make the standard assumption that the transformer is monotonic in the information order, i.e., if $\sigma_1 \sqsubseteq \sigma_2$ then $\tau(\sigma_1) \sqsubseteq \tau(\sigma_2)$. We extend this function pointwise to $\tau : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, by defining $\tau(S) = \bigcup\{\tau(\sigma) \mid \sigma \in S\}$. (Note that the extended transformer is monotone in the information order as well.) For purposes of abstract interpretation, we need to define a corresponding sound

abstract transformer on $\mathcal{P}(\Sigma)^k$. Given an input value $I = (I_1, \ldots, I_k)$, the abstract transformer needs to compute the output value $O = (O_1, \ldots, O_k)$.

A straightforward sound transformer is the pointwise transformer τ^{pw} defined as follows:

$$\tau^{pw}(I_1, \ldots, I_k) = (\hat{\eta}_1(\tau(I_1)), \ldots, \hat{\eta}_k(\tau(I_k))). \tag{6.4}$$

Proposition 6.4.2 *The pointwise transformer τ^{pw} is sound. That is, for every input value $I = (I_1, \ldots, I_k)$ where $I \in \mathcal{P}(\Sigma)^k$, the following holds:*

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau^{pw}(I)) \ . \tag{6.5}$$

Proof: Note that since $\preceq$ and $\sqsubseteq$ are reversed, $\otimes$ is a meet (i.e., greatest lower bound) operator on $\mathcal{P}(\Sigma)$. Let j be any index in $\{1, \ldots, k\}$.

$$I_1 \otimes \ldots \otimes I_k \sqsubseteq I_j \tag{6.6}$$

$\triangleright$ $\otimes$ is a meet operator

$$\gamma(I) \sqsubseteq I_j \tag{6.7}$$

$\triangleright$ by (6.6) and (6.3)

$$\tau(\gamma(I)) \sqsubseteq \tau(I_j) \tag{6.8}$$

$\triangleright$ τ is monotone

$$\tau(\gamma(I)) \sqsubseteq \hat{\eta}_j(\tau(I_j)) \tag{6.9}$$

$\triangleright$ by (6.8) and since $\hat{\eta}_j$ is extensive

$$\tau(\gamma(I)) \sqsubseteq \hat{\eta}_1(\tau(I_1)) \otimes \ldots \otimes \hat{\eta}_k(\tau(I_k)) \tag{6.10}$$

$\triangleright$ $\otimes$ is a meet operator

$$\tau(\gamma(I)) \sqsubseteq \gamma(\hat{\eta}_1(\tau(I_1)), \ldots, \hat{\eta}_1(\tau(I_k)) \tag{6.11}$$

$\triangleright$ by (6.3) and (6.10)

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau^{pw}(I)) \tag{6.12}$$

$\triangleright$ by (6.4) and (6.11)

□

Example 6.4.3 *While the pointwise transformer is simple and efficient, it can lead to imprecise results when the transformer has to update a substate that does not have all the relevant information. Recall the example from Section 6.2, and consider the substate M_3. Substate M_3 does not contain information about the local variables of other threads. Therefore, M_3 also represents a state S_{bad} in which the local variables* `t` *and* `x` *of thread* ***prod1*** *point to the first cell and to the last cell of the list, respectively. Thus, a conservative transformer of* `6: x->n=t`*, when* ***prod1*** *serves as the scheduled thread, must emit a warning about a possible creation of a cyclic list. As explained in Section 6.2, we can avoid this imprecision by composing substate M_3 with other substates (M_1) to produce a more precise substate that can be transformed without making such worst-case assumptions. This motivates the following definitions.* □

A *combiner set* is a set $R \subseteq \{1, \ldots, k\}$ identifying a set of subheap domains. We define the *partial concretization function* γ_R, which combines the information from the specified set of subdomains $R = \{j_1, \ldots, j_m\}$, as follows:

$$\gamma_R(I_1, \ldots, I_k) = \bigotimes_{r \in R} I_r = I_{j_1} \otimes I_{j_2} \cdots \otimes I_{j_m} \ . \tag{6.13}$$

One-Level Composition.

We define the *partial transformer* $\tau_1[R, i]$, which computes the substate corresponding to the i-th subdomain using the subdomains identified by R, by

$$\tau_1[R, i](I) = \hat{\eta}_i(\tau(\gamma_R(I))). \tag{6.14}$$

We use the term *one-level* transformer to indicate that combining (or composing) information from a set of subdomains (identified by R above) occurs in one step.

We define a *one-level transformer specification* TS to be a tuple $(\mathit{TS}_1, \ldots, \mathit{TS}_k)$ where each $\mathit{TS}_i \subseteq \{1, \ldots, k\}$. We define the transformer $\tau_1[\mathit{TS}]$ by

$$\tau_1[\mathit{TS}](I) = (\tau_1[\mathit{TS}_1, 1](I), \ldots, \tau_1[\mathit{TS}_k, k](I)). \tag{6.15}$$

Theorem 6.4.4 *For any one-level transformer specification* TS*, the transformer* $\tau_1[\mathit{TS}]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*:* $\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\mathit{TS}](I))$.

Theorem 6.4.5 *Let* $\mathit{TS} = (\mathit{TS}_1, \ldots, \mathit{TS}_k)$ *where each* $\mathit{TS}_i \subseteq \{1, \ldots, k\}$ *be a one-level transformer specification. Then, the one-level transformer* $\tau_1[\mathit{TS}]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*, the following holds:*

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\mathit{TS}](I)) \ . \tag{6.16}$$

Two-Level Composition.

We now present a generalization of the above definition. As motivation for this generalization, consider a situation where we want to compute an output value O_j by combining the input values from a set of subdomains R_1 or by combining the input values from a set of subdomains R_2 (but we are unable to say which of these combinations to use statically). We could, of course, combine the input values from the set of subdomains $R_1 \cup R_2$, but this could be expensive. Instead, we can utilize the two combinations *independently* of each other by using

$$(\hat{\eta}_j(\tau(\gamma_{R_1}(I)))) \sqcap (\hat{\eta}_j(\tau(\gamma_{R_2}(I))))$$

as the desired output value. We call transformers derived in this fashion two-level transformers, as the use of the meet operation $\sqcap$ constitutes a second stage of combining (composing) information.

Let Y be a set of combiner sets. We define the *partial transformer* $\tau_2[Y, i]$, which computes the substate corresponding to the i-th subdomain using the combiner sets in Y independently, as follows:

$$\tau_2[Y, i](I) = \prod_{R \in Y} \tau_1[R, i](I) \tag{6.17}$$

We define a *two-level transformer specification* TS to be a tuple $(\mathit{TS}_1, \ldots, \mathit{TS}_k)$ where each $\mathit{TS}_i \subseteq \mathcal{P}(\{1, \ldots, k\})$. We define the transformer $\tau_2[\mathit{TS}]$ by

$$\tau_2[\mathit{TS}](I) = (\tau_2[\mathit{TS}_1, 1](I), \ldots, \tau_2[\mathit{TS}_k, k](I)). \tag{6.18}$$

(Note that the computation of the above transformer involves a partial concretization for every R in every TS_i. In practice, different TS_i and TS_j may have common elements, and it is sufficient for the transformer implementation to do the corresponding partial concretization just once.)

Theorem 6.4.6 *For any two-level transformer specification* τs*, the transformer* $\tau_2[\tau s]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*:* $\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\tau s](I))$*.*

Theorem 6.4.7 *Let* $\tau s = (\tau s_1, \ldots, \tau s_k)$ *where each* $\tau s_i \subseteq 2^{\{1,\ldots,k\}}$ *be a two-level transformer specification. Then, the two-level transformer* $\tau_2[\tau s]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*, the following holds:*

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\tau s](I)) \quad . \tag{6.19}$$

6.5 Empirical Results

We implemented the HeDec system in Java on top of the TVLA system [LAS00]. HeDec allows analysis designers to rapidly prototype different shape analysis algorithms by defining heap decomposition schemes. HeDec, however, is not a panacea — the designer needs to carefully select suitable heap decompositions. Nevertheless, HeDec relieves the designer from the task of developing and implementing the static analysis algorithms, including the transformers.

Table 6.2 and Table 6.1 compare the results of our decomposition-based analysis with a full heap analysis.[1]

Concurrent Benchmarks.

We use the analysis of [1] as the underlying shape analysis.

Both analyses successfully prove linearizability and absence of null dereferences for the three concurrent programs. For a given number of threads, t, the table shows the time and the number of states resulting in the analysis of t threads invoking an arbitrary sequence of operations on a single instance of the analyzed concurrent data structure. Stack is the non-blocking stack example of Section 6.2.1. TLQ is the two-lock queue implementation described in [MS96]. NBQ is a non-blocking queue implementation from [DGLM04]. [2]

Note that while [ARR+] can analyze at most 3 threads, our approach, on the other hand,runs for 15 threads or more. Furthermore, [ARR+] runs out of memory when analyzing 3 threads manipulating a non-blocking-queue.

Sequential Benchmarks.

Both analyses successfully prove absence of null dereferences, absence of memory leaks, and data structure invariants for the following sequential benchmarks: 6`-list-prepend` adds elements, non-deterministically, into one of 6 lists; 6`-list-join` joins 6 lists into one list; and 4`-tree-insert` inserts nodes, non-deterministically, into one of 4 binary search trees.

[1] All benchmarks except NBQ were run on a 2.4 GHz E6600 Core 2 Duo processor with 2 GB of memory running Linux.

[2] This benchmark was run on a 2.66 GHz Quad Xeon with 16 GB of memory running Windows XP 64 bit.

Table 6.1: Empirical results for concurrent benchmarks

		Full Heap		Decomposition	
Example	# of threads	# of states	secs.	# of substates	secs.
Stack	2	3,424	3	1,608	7
	3	10,6296	71	4,103	13
	4	MemOut	-	7,728	22
	20	-	-	212,048	3,421
TLQ	3	8,783	12	8,911	30
	5	44,285	35	23,585	90
	8	MemOut	-	58,796	307
	15	-	-	202,555	2,122
NBQ	2	39,583	69	20,646	263
	3	MemOut	-	57,065	694
	15	-	-	2,017,280	1 day

Table 6.2: Empirical results for sequential benchmarks

	Full Heap		Decomposition	
Example	# of states	secs.	# of substates	secs.
6-list-prepend	17,496	16	557	5
6-list-join	37,689	40	1,282	6
4-tree-insert	43,031	44	5,316	29

6.6 Related Work

The framework of Cartesian abstraction via state decomposition we have presented is relevant to a number of previous lines of work.

Heterogeneous Abstractions. Yahav and Ramalingam [LAIS06] defined a notion of heterogeneous abstractions. There, Cartesian abstractions are used as a way to achieve decomposition (or separation, in the terminology of that paper). One contribution of this chapter is to show that that previous analysis is based on a (simple form of) Cartesian abstraction. On the other hand, in that work, heterogeneity was used only within a single structure (to abstract the substructure of interest differently from its context), where our framework supports different abstractions for different factors of the product, yielding heterogeneity across different structures. Furthermore, while Yahav and Ramalingam [LAIS06] rely on the point-wise transformer, we introduce a generalized family of transformers that allow (de)composition when transformers are applied. This generalization allows specifying more precise transformers, and gives us dynamic separation/decomposition.

Region-based Heap Analyses. Like [LAIS06], [HR05] also decomposes heap abstractions to independently analyze different parts of the heap. There the analysis/verification problem is itself decomposed into a set of problem instances, and the heap abstraction is specialized for each instance and consists of one subheap for the part of the heap relevant to the instance, and a coarser abstraction of the remaining part of the heap, e.g. a points-to graph. In contrast, we simultaneously maintain abstractions of different parts of the heap and also consider the interaction between these parts. (E.g., our decomposition dynamically changes as components get connected and disconnected.)

Local Transformers. The importance of modularity for the ability to compute transformers is well known. For example, the first proof rule for procedure calls, the *rule of adaptation*, was given in [Hoa71]. It allows reusing a proof of a procedure body in different invocations of the procedure.

Local reasoning [ORY01, Rey02] enables reasoning about programs that alter heap-allocated data by combining claims about disjoints parts of the heap. The use of decomposition here is intuitively similar to that of separation in [ORY01]. The chief difference is that here a decomposition may be used that is finer than the transformers in the underlying domain are precise for, which we react to by performing composition in the transformers. The transformers used in analyses based on separation logic [BCO05], on the other hand, when applied to substates either produce exactly as precise information as on full states, or produce top. Our treatment of decomposition as an abstraction allows more flexibility in this regard. This flexibility is central to the concurrency analysis we presented: By not basing decomposition on disjointness, the analysis does not necessarily need to be thread-modular. In particular, we have the option of introducing predicates which track important correlations between different threads' local states. Approaches based on disjointness such as [GBCS07] have trouble with such situations unless auxiliary state is added to the invariants, which is beyond the ability of the existing automatic analyses.

Partially Disjunctive Heap Abstraction. In Chapter 4 we describe a heap abstraction based on merging sets of graphs with the same set of nodes into one (approximate) graph. The abstraction in this chapter is based on decomposing a graph into a set of subgraphs. The abstraction in Chapter 4 is orthogonal to the one in this chapter.

Handling Concurrency for an Unbounded Number of Threads. In [2], we use thread quantification to analyze programs with an unbounded number of threads. Thread quantification can be thought of as an unbounded variant of a particular decomposition strategy, which we use to abstract away correlations between local variables of different threads. In the thread quantification analysis, we report that using an additional heap decomposition abstraction in order to abstract away correlations between values of some local variables and global variables effects drastic state-space savings. This made the analysis feasible in the example of proving linearizability of a non-blocking queue implementation.

Proving Linearizability of Data Structures. Shape analysis of concurrent programs with unbounded dynamic allocation have been investigated. The analysis in [Yah01] addresses an unbounded number of threads by losing distinctions that cannot be made based on thread-independent information. This analysis has been extended to verify linearization [ARR$^+$] of programs with a bounded number of threads. Here we use the decomposition abstraction to define an analysis that can be exponentially faster than that in [ARR$^+$].

Manual linearizability proofs using rely-guarantee have been given in [VHHS06], and using a manual translation to automata followed by an interactive proof in PVS in [CDG05]. Recently, [Vaf09] automatically verifies linearizability from manual specifications in a combination of rely-guarantee and separation logic, using the proof technique of [ARR$^+$].

6.7 Conclusions

We present systematic and generic techniques for scaling up shape analyses using heap decomposition, implemented in the HeDec system. A user of HeDec can quickly prototype a shape analysis by: (a) defining any heap decomposition she believes is appropriate for the class of programs and properties of interest, and (b) supplying for every type of program statement any (possibly empty) combiner set she believes supplies the right balance between efficiency and precision. HeDec then automatically generates a sound analysis.

Chapter 7

Conclusions

In this thesis, we have shown that partially disjunctive abstractions can be used to greatly improve the performance of precise shape analyses. An analysis designer may start by designing a disjunctive abstraction and use it over a set of simple benchmarks to obtain confidence that the abstraction is precise enough for proving the desired properties. Then, the designer would observe the abstract states arising from such an analysis to find useless information that is captured by the abstraction. For example, a predicate may provide local distinctions inside an abstract state (3-valued logical structure) that are not needed in order to distinguish between different abstract states. This can suggest merging abstract states that are distinguished by that predicate by using partial isomorphism join (see Chapter 3). In considering programs with multiple data structures or concurrency, the analysis designer can define appropriate subheap decomposition techniques to cope with the exponential factors, e.g., due to thread interleaving (see Chapter 6).

Indeed, in the beginning of this thesis (Chapter 3), we define a rather precise abstraction for programs containing a finite number of singly-linked lists. The precision of the analysis is experimentally verified on a number of benchmarks manipulating one or two lists. Later (see Chapter 5), we consider programs such as device drivers that simultaneously manipulate multiple instances of cyclic linked lists. Using a disjunctive abstraction proves to be infeasible, since it incurs exponential state space blow-ups. We create a further abstraction by abstracting away the correlations between disjoint subheaps, which usually contain different lists, thus reducing the exponential factors. These correlations are usually not important for the safety properties we wish to verify and thus we are able to improve the performance of the analysis without a significant increase to the number of false alarms.

We note that the abstractions based on decomposition are incomparable, in terms of precision, with the one based on partial isomorphism, in Chapter 4. For example, applying the partial isomorphism abstraction on top of the list abstraction of Chapter 3, results in the same abstraction, since no two shape graphs in the image of the list abstraction are partially isomorphic. On the other hand, partial isomorphism abstraction is able to merge multiple similar structures containing a single connected component, whereas graph decomposition would not result in an identity abstraction (since the structures cannot be decomposed). This also means that analyses based on these abstraction are incomparable in terms of performance: Disjoint subgraph decomposition can help reduce exponential factors where partial isomorphism abstraction cannot and vice versa.

It is fortunate, that the two kinds of partially disjunctive abstraction — the one based on

partial isomorphism and the ones based on decomposition — can be combined to yield very useful abstractions, which can many times complement each other in terms of the kinds of exponential factors that they are able to reduce.

7.1 Suggestions for Further Work

Automatically Refining Partially Disjunctive Abstractions. The theory of automatically refining abstractions has been heavily studied by the model checking community using techniques such as counterexample-guided abstraction refinement for *Predicate Abstraction* [CGJ+00, BMMR01, HJMM04]. The theory of automatic abstraction refinement for partially disjunctive abstractions has been studied [GQ01, CGR07], but not as extensively. Intuitively, the problem of automatically refining abstractions is harder for partially disjunctive abstractions, since the analyzer has to learn new abstractions from multiple control flow paths simultaneously. An interesting direction of research is modifying a system based on predicate abstraction to use "Cartesian Predicate Abstraction" and automatically finding the relevant predicates by considering multiple counterexample paths.

Localized Heap Abstractions The main property that was useful in developing precise and efficient transformers for the analysis in Chapter 5 and the analyses in Chapter 6 is that the granularity of the abstraction of a concrete state matched the level of granularity of the concrete transformer. This property is also utilized in separation based shape abstractions [DOY06]. We are interested in precisely characterizing abstractions that have this property and in the possibility of automatically generating these abstractions based on a given set of transformers.

Bibliography

[AMSS06] G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *Proc. Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 33–48, 2006.

[APV06] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proc. Intl. SPIN Workshop on Model Checking of Software*, 2006.

[APV08] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstraction. *Proc. Intl. Journal on Software Tools for Technology Transfer*, 2008. To appear.

[ARR+] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Proc. Intl. Conf. on Computer Aided Verification*, pages 477–490.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, M. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In Jr. James B. Fenwick and Cindy Norris, editors, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 196–207, 2003.

[BCO05] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proc. Asian Symp. on Programming Languages and Systems*, pages 52–68, 2005.

[BHZ03] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Proc. Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 135–148, 2003.

[BLAM+08] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *Proc. Intl. Conf. on Computer Aided Verification*, pages 399–413, 2008.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 203–213, June 2001.

[BR02] T. Ball and S.K. Rajamani. Generating abstract explanations of spurious counterexamples in c programs. Report MSR-TR-2002-09, Microsoft Research, Microsoft Redmond, January 2002. http://research.microsoft.com/slam/.

[BRS99] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proc. European Symp. on Programming*, pages 2–19, 1999.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[CDG05] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *Electr. Notes Theor. Comput. Sci.*, 137(2):93–110, 2005.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. Intl. Conf. on Computer Aided Verification*, pages 154–169, 2000.

[CGR07] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *Proc. Static Analysis Symp.*, page 333, 2007.

[Cha03] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 115–125. ACM Press, 2003.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 296–310. ACM Press, 1990.

[DDG+04] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and Jr. G. L. Steele. Dcas is not a silver bullet for non-blocking algorithm design. In *Proc. Symp. on Parallelism in Algorithms and Architectures*, page 216, 2004.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 230–241. ACM Press, 1994.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *Proc. Intl. Conf. on Formal Techniques for Networked and Distributed Systems*, page 97, 2004.

[DLS02] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, January 2002.

[DN03] D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proc. Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 310–324, 2003.

[DOY06] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302, 2006.

[EY04] H. Eo and K. Yi. A differential fixpoint iteration method for static analysis specifications. Technical Memorandum ROPAS-2004-21, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University, February 2004.

[GBC06] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Proc. Static Analysis Symp.*, pages 240–260, 2006.

[GBCS07] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 266–277. ACM, 2007.

[GDN+04] D. Gopan, F. DiMaio, N.Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529, 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[GQ01] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *Proc. Static Analysis Symp.*, pages 356–373, 2001.

[GR98] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming*, 32(1-3):177–210, 1998.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. Intl. Conf. on Computer Aided Verification*, pages 72–83, 1997.

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 232–244. ACM Press, 2004.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. *Lecture Notes in Mathematics*, 188:102–116, 1971.

[HR05] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 310–323, 2005.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Progrmming Languages and Systems*, 12(3):463–492, 1990.

[IO01] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 14–26. ACM Press, 2001.

[IRR+04] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. *Proc. Computer Science Logic*, pages 160–174, 2004.

[JJNS97] J.L. Jensen, M.E. Joergensen, N.Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 226–236, 1997.

[JM81a] N. D. Jones and S. S. Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to dijkstra. In *Program Flow Analysis: Theory and Applications*, chapter 12. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[JM81b] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Proc. Intl. Conf. on Computer Aided Verification*, pages 547–561, 2006.

[Lar89] J.R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, Univ. of Calif., Berkeley, CA, May 1989.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In *Proc. Static Analysis Symp.*, pages 280–301, 2000.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 21–34, 1988.

[Lin73] G. Lindstrom. Scanning list structures without stacks or tag bits. *Information Processing Letters*, 2(2):47–51, June 1973.

[MBC+] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. Technical Report TR-2007-11-85453, Tel-Aviv University, Tel-Aviv, Israel.

[MBC+07] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–18, 2007.

[Mic01] Microsoft Research. The SLAM project. http://research.microsoft.com/slam/, 2001.

[MLAS+08] R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. In *Proc. Static Analysis Symp.*, pages 363–377, 2008.

[MS96] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[MSRF04] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Proc. Static Analysis Symp.*, pages 265–279, 2004.

[MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proc. Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 181–198, 2005.

[NNH99] F. Nielson, H. R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, 2142, 2001.

[Rab69] M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc*, 141(1):1–35, 1969.

[RBR+05] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 296–309, 2005.

[Rey02] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. Symp. on Logic in Computer Science*, pages 55–74, 2002.

[RSW01] T. Reps, M. Sagiv, and R. Wilhelm. Automatic verification of a simple mark and sweep garbabge collector. Presented in the 2001 University of Washington and Microsoft Research Summer Institute, Specifying and Checking Properties of Software, http://research.microsoft.com/specncheck/, 2001.

[RSY04] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Proc. Verification, Model Checking, and Abstract Interpretation*, pages 252–266. Springer-Verlag, 2004.

[RSY05] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proc. Static Analysis Symp.*, pages 284–302, 2005.

[RWF+02] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 83–94, 2002.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Progrmming Languages and Systems*, 20(1):1–50, January 1998.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Progrmming Languages and Systems*, 24(3):217–298, 2002.

[Str92] J. Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Information and Computation*, 101(1):70–102, November 1992.

[SYKS03] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. Static Analysis Symp.*, 2003.

[TKB02] T.Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Static Analysis Symposium*, pages 69–84, 2002.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[Vaf09] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proc. Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 335–348, 2009.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proc. Symp. on Principles and Practice of Parallel Programming*, pages 129–136, 2006.

[VRHS+99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proc. Conf. of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, 1999.

[Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3), 2001.

[YLB+08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *Proc. Static Analysis Symp.*, 2008.

[YR04] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstraction. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 25–34, 2004.

Appendix A

Proofs and Additional Details for Chapter 3

A.1 Deriving the Abstract transformer for `y.n=null`

In order to simplify the definition of the transformer for `y.n = null`, we split it to five different cases, shown in Table A.1, based on classification of the next list interruption. The table uses the following shorthand notations:

$$
\begin{array}{lcl}
\mathit{ListToInDegree2}[y] & = & \bigvee_{z_1 \in \mathit{Var}} \mathit{UList}[z_1, y_s] \wedge \neg \mathit{Aliased}[y, z_1] \wedge \\
 & & \quad \bigwedge_{z_2 \in \mathit{Var}} \mathit{UList}[z_2, y_s] \rightarrow (\mathit{Aliased}[z_2, y] \vee \mathit{Aliased}[z_2, z_1]) \\
\mathit{ListRegularVar}[y] & = & \bigvee_{w \in \mathit{PVar}} \mathit{UList}[y, w] \\
\mathit{ListToHeapShared}[y] & = & \bigvee_{w \in \mathit{PVar}} \mathit{UList}[y, w_s]
\end{array}
$$

We show that manual construction of the best transformer results with the same formulae provided in Section 3.4. The derivation is shown Table A.2. For each predicate, we first show its defining formula after applying the concrete effect of the statement `y.n=null`. We then rewrite this formula to an equivalent formula that is folded into the nullary predicates of our predicate-abstraction vocabulary (of Table 3.6). In the process of rewriting, we use transformations of FO^{TC} under the assumption that formulae describe heap configurations satisfying the integrity rules of the following definition:

Definition A.1.1 (Integrity Rules) *We require that every heap configuration satisfies the following integrity rules:*

1. *for every unary predicate $x(v)$ representing a reference variable,*
   $\forall v_1, v_2 . x(v_1) \wedge x(v_2) \rightarrow v_1 = v_2$
2. *for the predicate $n(v_1, v_2)$ representing the `n` field,*
   $\forall v, v_1, v_2 . n(v, v_1) \wedge n(v, v_2) \rightarrow v_1 = v_2$

In the process of rewriting, we also use the rewrite rules of the following lemma. When a rule from the lemma is used in the rewriting, we note its number in brackets. We use $[*]$ to denote a rewrite using FO^{TC} transformations (assuming formulae describe heap configurations that satisfy the above consistency rules).

Lemma A.1.2 *The following always hold:*

Table A.1: The different cases considered when defining the abstract transformer for the statement `y.n = null`

Case	Next List Interruption	Precondition
1	is a heap-shared node not pointed by any regular variable, with in-degree $= 2$	$\neg(\mathit{UList}[y, \text{null}] \vee \mathit{ListRegularVar}[y]) \wedge \mathit{ListToInDegree2}[y]$
2	is null	$\mathit{UList}[y, \text{null}]$
3	is a node pointed by some regular variable and not heap shared	$\mathit{ListRegularVar}[y] \wedge \neg\mathit{ListToHeapShared}[y]$
4	is a heap-shared node with in-degree > 2	$\mathit{ListToHeapShared}[y] \wedge \neg\mathit{ListToInDegree2}[y]$
5	is a node pointed by a regular variable and heap shared, with in-degree $= 2$	$\mathit{ListRegularVar}[y] \wedge \mathit{ListToHeapShared}[y] \wedge \mathit{ListToInDegree2}[y]$

(I) $\neg PtByVar(u) \Rightarrow \neg y(u)$

(II) $\neg Interruption(u) \Rightarrow \neg y(u)$

The following hold under the precondition of case 3*:*

(III) $\mathit{Interruption}'(u) = \mathit{Interruption}(u)$

(IV) $\mathit{UList}'(v_1, v_2) = \mathit{UList}(v_1, v_2) \wedge \neg y(v_1)$

(V) $\mathit{UListNULL}'(v_1, v_2) = \mathit{UListNULL}(v_1, v_2) \vee y(v_1)$

where the primed values of shorthands denote their value after a after applying the effect of the statement `y.n = null`*.*

Proof: The first claims in the Lemma are mostly immediate from the definitions of the shorthand notations.

(I)

$$\begin{aligned} \neg\mathit{PtByVar}(u) &= \neg \bigvee_{\mathit{var} \in \mathit{PVar}} \mathit{var}(u) \\ &\Rightarrow \neg y(u) \end{aligned}$$

(II)

$$\begin{aligned} \neg Interruption(u) &= \neg\mathit{HeapShared}(u) \wedge \neg\mathit{PtByVar}(u) \\ &\Rightarrow \neg y(u) \end{aligned}$$

(III) we begin by showing that $\mathit{HeapShared}'(u) = \mathit{HeapShared}(u)$

$$\begin{aligned} \mathit{HeapShared}'(u) &= \exists a, b.n(a, u) \wedge \neg y(a) \wedge n(b, u) \wedge \neg y(b) \wedge (a \neq b) \\ &\text{by the precondition to this case} \\ \mathit{HeapShared}'(u) &= \exists a, b.n(a, u) \wedge n(b, u) \wedge (a \neq b) \\ &= \mathit{HeapShared}(u) \end{aligned}$$

since $PtByVar(u)$ does not change under the action `y.n=null`, it follows that $\mathit{Interruption}'(u) = \mathit{Interruption}(u)$.

Table A.2: Derivation of the transformer for y.n=null for case 3.

$\mathit{UList}_1[z_1, z_2]'$	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge n(v_1, v_2) \wedge \neg y(v_1)$	$[*]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge n(v_1, v_2) \wedge \exists v_3.z_1(v_3) \wedge \neg y(v_3)$	$[*]$
	$\mathit{UList}_1[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y]$	
$\mathit{UList}_1[z_1, \text{null}]'$	$\exists v.z_1(v) \wedge \forall u.\neg(n(v, u) \wedge \neg y(v))$	$[*]$
	$\exists v.z_1(v) \wedge \forall u.(\neg n(v, u) \vee y(v))$	$[*]$
	$\exists v.y(v) \vee z_1(v) \wedge \forall u.\neg n(v, u)$	$[*]$
	$\mathit{UList}_1[z_1, \text{null}] \vee \mathit{Aliased}[z_1, y]$	
$\mathit{UList}_2[z_1, z_2]'$	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}'(m) \wedge$	
	$\wedge (n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))$	[III]
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}(m) \wedge$	
	$\wedge (n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))$	[II]
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}(m) \wedge$	
	$\wedge n(v_1, m) \wedge \neg y(v_1) \wedge n(m, v_2)$	$[*]$
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \exists m.\neg \mathit{Interruption}(m) \wedge$	
	$\wedge n(v_1, m) \wedge n(m, v_2) \wedge z_1(v_1) \wedge \neg y(v_1)$	$[*]$
	$\mathit{UList}_2[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y]$	
$\mathit{UList}[z_1, z_2]'$	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \mathit{UList}'(v_1, v_2)$	[IV]
	$\exists v_1, v_2.z_1(v_1) \wedge z_2(v_2) \wedge \neg y(v_1) \wedge \mathit{UList}(v_1, v_2)$	$[*]$
	$\mathit{UList}[z_1, z_2] \wedge \neg \mathit{Aliased}[z_1, y]$	
$\mathit{UList}[z_1, \text{null}]'$	$\exists v_1.z_1(v_1) \wedge \mathit{UListNULL}(v_1)$	[V]
	$\exists v_1.z_1(v_1) \wedge \mathit{UListNULL}(v_1) \vee y(v_1)$	$[*]$
	$\mathit{UList}[z_1, \text{null}] \vee \mathit{Aliased}[z_1, y]$	

(IV)

$$
\begin{array}{ll}
\textit{UList}'(v_1, v_2) & (n(v_1, v_2) \wedge \neg y(v_1) \vee \\
& \quad (\exists m. \neg \textit{Interruption}'(m) \wedge (((n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))) \vee \\
& \quad (\exists m_1, m_2. (n(v_1, m_1) \wedge \neg y(v_1) \wedge \neg y(v_1)) \wedge (n(m_2, v_2) \wedge \neg y(m_2)) \wedge \\
& \quad (\text{TC } a, b : n(a, b) \wedge \neg y(a) \wedge \neg \textit{Interruption}'(a) \wedge \neg \textit{Interruption}'(b))(m_1, m_2))) \quad [\text{III}] \\
& \exists v_1, v_2. z_1(v_1) \wedge z_2(v_2) \wedge (n(v_1, v_2) \wedge \neg y(v_1) \vee \\
& \quad (\exists m. \neg \textit{Interruption}(m) \wedge (n(v_1, m) \wedge \neg y(v_1)) \wedge (n(m, v_2) \wedge \neg y(m))) \vee \\
& \quad (\exists m_1, m_2. n(v_1, m_1) \wedge \neg y(v_1) \wedge n(m_2, v_2) \wedge \\
& \quad (\text{TC } a, b : n(a, b) \wedge \neg y(a) \wedge \neg \textit{Interruption}(a) \wedge \neg \textit{Interruption}(b))(m_1, m_2))) \quad [\text{II}] \\
& \exists v_1, v_2. z_1(v_1) \wedge z_2(v_2) \wedge (n(v_1, v_2) \wedge \neg y(v_1) \vee \\
& \quad (\exists m. \neg \textit{Interruption}(m) \wedge n(v_1, m) \wedge \neg y(v_1) \wedge n(m, v_2)) \vee \\
& \quad (\exists m_1, m_2. n(v_1, m_1) \wedge \neg y(v_1) \wedge n(m_2, v_2) \wedge \\
& \quad (\text{TC } a, b : n(a, b) \wedge \neg \textit{Interruption}(a) \wedge \neg \textit{Interruption}(b))(m_1, m_2))) \quad [*] \\
& \exists v_1, v_2. z_1(v_1) \wedge z_2(v_2) \wedge \neg y(v_1) \wedge (n(v_1, v_2) \vee \\
& \quad (\exists m. \neg \textit{Interruption}(m) \wedge n(v_1, m) \wedge n(m, v_2)) \vee \\
& \quad (\exists m_1, m_2. n(v_1, m_1) \wedge n(m_2, v_2) \wedge \\
& \quad (\text{TC } a, b : n(a, b) \wedge \neg \textit{Interruption}(a) \wedge \neg \textit{Interruption}(b))(m_1, m_2))) \quad [*] \\
& \textit{UList}'(v_1, v_2) \wedge \neg y(v_1)
\end{array}
$$

(V)

$$
\begin{array}{ll}
\textit{UListNULL}'(u) & \forall v.\neg(n(u,v) \wedge \neg y(u)) \vee \\
& \exists m.(n(u,m) \wedge \neg y(u)) \wedge \neg \textit{Interruption}'(m) \wedge \textit{UListNULL}'_1(m) \vee \\
& \exists m_1, m_2 : (n(u,m_1) \wedge \neg y(u)) \wedge \textit{UListNULL}'_1(m_2) \wedge \\
& \quad (\text{TC } a,b : (n(a,b) \wedge \neg y(a)) \wedge \neg \textit{Interruption}'(a) \wedge \\
& \quad \neg \textit{Interruption}'(b))(m_1, m_2) \qquad [\text{III}] \\
& \forall v.\neg(n(u,v) \wedge \neg y(u)) \vee \\
& \exists m.(n(u,m) \wedge \neg y(u)) \wedge \neg \textit{Interruption}(m) \\
& \wedge (\textit{UListNULL}_1(m) \vee y(m)) \vee \\
& \exists m_1, m_2 : (n(u,m_1) \wedge \neg y(u)) \wedge (\textit{UListNULL}_1(m_2) \vee y(m_2)) \wedge \\
& \quad (\text{TC } a,b : (n(a,b) \wedge \neg y(a)) \wedge \neg \textit{Interruption}(a) \wedge \\
& \quad \neg \textit{Interruption}(b))(m_1, m_2) \qquad [\text{II}] + [*] \\
& \forall v.\neg(n(u,v) \wedge \neg y(u)) \vee \\
& \exists m.(n(u,m) \wedge \neg y(u)) \wedge \neg \textit{Interruption}(m) \\
& \wedge (\textit{UListNULL}_1(m) \vee y(m)) \vee \\
& \exists m_1, m_2 : (n(u,m_1) \wedge \neg y(u)) \wedge (\textit{UListNULL}_1(m_2) \vee y(m_2)) \wedge \\
& \quad (\text{TC } a,b : (n(a,b) \wedge \neg y(a)) \wedge \neg \textit{Interruption}(a) \wedge \\
& \quad \neg \textit{Interruption}(b))(m_1, m_2) \qquad [\text{II}] \\
& \forall v.(\neg n(u,v) \vee y(u)) \vee \\
& \exists m.(n(u,m) \wedge \neg y(u)) \wedge \neg \textit{Interruption}(m) \wedge \textit{UListNULL}_1(m) \vee \\
& \exists m_1, m_2 : (n(u,m_1) \wedge \neg y(u)) \wedge \textit{UListNULL}_1(m_2) \wedge \\
& \quad (\text{TC } a,b : n(a,b) \wedge \neg \textit{Interruption}(a) \wedge \\
& \quad \neg \textit{Interruption}(b))(m_1, m_2) \qquad [*] \\
& \forall v.(\neg n(u,v)) \vee y(u) \vee \\
& \neg y(u) \wedge \exists m.n(u,m) \wedge \neg \textit{Interruption}(m) \wedge \textit{UListNULL}_1(m) \vee \\
& \neg y(u) \wedge \exists m_1, m_2 : n(u,m_1) \wedge \textit{UListNULL}_1(m_2) \wedge \\
& \quad (\text{TC } a,b : n(a,b) \wedge \neg \textit{Interruption}(a) \wedge \\
& \quad \neg \textit{Interruption}(b))(m_1, m_2) \qquad [*](resolution) \\
& \forall v.(\neg n(u,v)) \vee y(u) \vee \\
& \exists m.n(u,m) \wedge \neg \textit{Interruption}(m) \wedge \textit{UListNULL}_1(m) \vee \\
& \exists m_1, m_2 : n(u,m_1) \wedge \textit{UListNULL}_1(m_2) \wedge \\
& \quad (\text{TC } a,b : n(a,b) \wedge \neg \textit{Interruption}(a) \wedge \\
& \quad \neg \textit{Interruption}(b))(m_1, m_2) \qquad [*] \\
& \textit{UListNULL}'(u) \vee y(u)
\end{array}
$$

□

A.2 Proving Theorem 3.6.1

We want to prove that the Predicate Abstraction, $\beta_{PredAbs}$, presented in Section 3.4 and the Canonical Abstraction, $\beta_{Canonic}$, presented in Section 3.5 are equivalent. Before delving into the details, we make the claim more precise.

Recall that both abstractions are parameterized by an index k ranging from 1 to n (the number of program variables). The proof here is for $k = n$ (i.e., the full set of auxiliary variables is used).

By equivalence of abstractions, we mean that, for any two concrete heaps C_1 and C_2 (2-valued structures), the following holds:

$$\beta_{PredAbs}(C_1) = \beta_{PredAbs}(C_2) \text{ when } \beta_{Canonic}(C_1) = \beta_{Canonic}(C_2) \ .$$

We will use the following shorthand notations

$$\begin{array}{rcl} A_1^P = & \beta_{PredAbs}(C_1) & , \\ A_2^P = & \beta_{PredAbs}(C_2) & , \\ A_1^C = & \beta_{Canonic}(C_1) & , \\ A_2^C = & \beta_{Canonic}(C_2) & , \end{array}$$

and make use of the embedding functions f and g such that $C_1 \sqsubseteq^f A_1^C$ and $C_2 \sqsubseteq^g A_2^C$. With these notations we rephrase the equivalence claim: $A_1^P = A_2^P$ when $A_1^C = A_2^C$. Note that by $A_1^C = A_2^C$ we mean that structures A_1^C and A_2^C are isomorphic, i.e., $A_1^C \sqsubseteq A_2^C$ and $A_2^C \sqsubseteq A_1^C$. The semantics of formulae for 3-valued structures is explained in [SRW02].

We will sometimes write the name of a predicate from Table 3.6 as shorthand for its defining formula. The exact meaning, however, should be clear from the context, depending on whether the structure referred to is concrete (2-valued) or abstract (3-valued).

Since the join operator used in both kinds of abstractions is the same—set union—the equivalence of the abstractions carries over from single concrete heaps to sets of concrete heaps.

Proof Structure.

We want to show that both abstractions are able to make exactly the same distinctions about any two concrete heaps. We start by showing that whenever C_1 and C_2 assign different interpretations to a predicate in P^A (indicating that their Predicate Abstraction is different), A_1^C is different from A_2^C. This is shown by a case analysis according the 7 predicate types that appear in Table 3.6 in order of appearance. In each case we assume that the predicates considered in previous cases have the same interpretation in both C_1 and C_2. Finally, we consider the case where all predicates in P^A have the same interpretation in C_1 and C_2 (indicating that their Predicate Abstraction is the same), and show that $A_1^C = A_2^C$.

We use the following lemmas to show that two concrete heaps are different under Canonical Abstraction[1]. In the lemmas, we use the shorthand notations introduced above.

In the proofs of the lemma we will use use the fact that, by the Embedding Theorem [SRW02], if two 3-valued structures are isomorphic then the value of every closed formula evaluates to the value in both structures.

Lemma A.2.1 *Let C_1 and C_2 be a pair of 2-valued structures, and let $\varphi(v)$ be a conjunction of unary predicates and negations of unary predicates.*

If $[\![\exists v : \varphi(v)]\!]^{C_1} = 1$ and $[\![\exists v : \varphi(v)]\!]^{C_2} = 0$ then $A_1^C \neq A_2^C$.

Proof: Let v be a node in U^{C_1} for which $\varphi(v)$ holds. Since Canonical Abstraction preserves the definite values of unary predicates, $\varphi(v)$ evaluates to a definite value for $f(v)$ (the same

[1]The lemmas are stated for the Canonical Abstraction from Section 3.5, but they are actually true for Canonical Abstraction with any set of predicates.

value as in C_1). Therefore, $[\![\exists v : \varphi(v)]\!]^{A_1^C} = 1$. Since there is no node in U^{C_1} for which $\varphi(v)$ holds, there is also node node in A_2^C for which $\varphi(v)$ holds, and therefore $[\![\exists v : \varphi(v)]\!]^{A_2^C} = 0$.

We conclude that $A_1^C \neq A_2^C$.

□

Lemma A.2.2 *Let C_1 and C_2 be a pair of 2-valued structures, and let $\varphi(v)$ be a conjunction of unary predicates and negations of unary predicates.*

If $\varphi(v)$ holds for exactly one individual u in C_1, and $\varphi(v)$ holds for more than one individual in C_2 then $A_1^C \neq A_2^C$.

Proof: Since $\varphi(v)$ holds for exactly one individual u in C_1, we have that $\varphi(v)$ holds for exactly one individual $v = f(u)$ in A_1^C. Therefore, $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_1^C} = 1$.

Let V_2 be the set of nodes in U^{C_2} for which $\varphi(v)$ holds. If $w = g(u) = g(v)$ for some pairs of nodes $u, v \in V_2$ then $eq(w, w) = 1/2$ and $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_2^C} = 1/2$. Otherwise, there $eq^{A_2^C}(g(u), g(v)) = 0$ for every distinct nodes $u, v \in V_2$, and therefore $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_2^C} = 0$.

In both cases $[\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_1^C} \neq [\![\forall a, b : \varphi(a) \wedge \varphi(b) \implies eq(a, b)]\!]^{A_2^C}$ and we conclude that $A_1^C \neq A_2^C$.

□

To give some intuition, Figure A.1 shows the different cases of concrete lists and their Canonical Abstraction, along with the values of the predicates from Table 3.6.

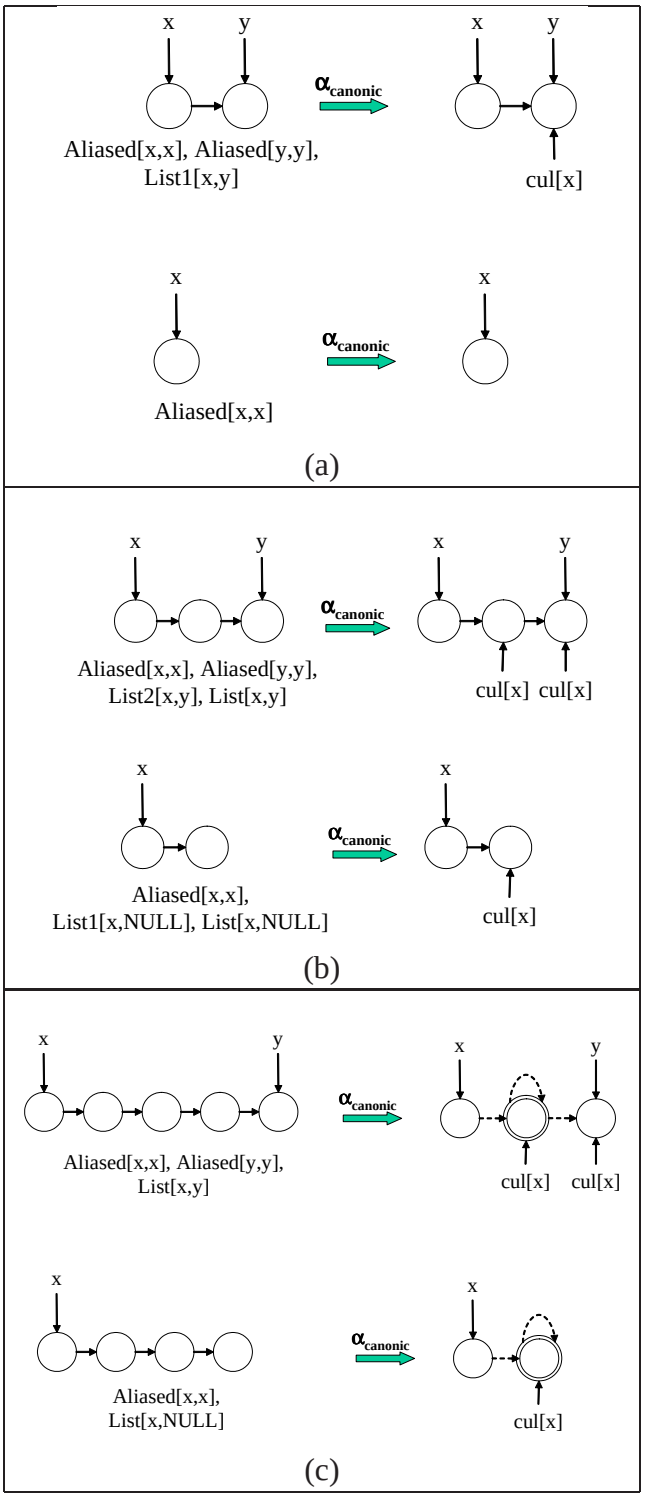


Figure A.1: Applying Canonical Abstraction to lists of different lengths: (a) lists of length 1, (b) lists of length 2, and (c) lists of length greater than 2

Proof:[of Theorem 3.6.1]

Case 1 : Distinction by *Aliased*$[x, y]$ predicates. Assume that for two variables $x, y \in$ *Var* we have $[\![\mathit{Aliased}[x, y]]\!]^{C_1} = 1$ and $[\![\mathit{Aliased}[x, y]]\!]^{C_2} = 0$. Substituting the predicate *Aliased*$[x, y]$ with its defining formula from Table 3.6, we get $[\![\exists v : x(v) \wedge y(v)]\!]^{C_1} = 1$ and $[\![\exists v : x(v) \wedge y(v)]\!]^{C_2} = 0$. Therefore, by Lemma A.2.1, $A_1^C \neq A_2^C$.

Case 2 : Distinction by *UList*$_1[x, y]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, and that for some $x, y \in$ *Var* we have $[\![\mathit{UList}_1[x, y]]\!]^{C_1} = 1$ and $[\![\mathit{UList}_1[x, y]]\!]^{C_2} = 0$.

Substituting the predicate *UList*$_1[x, y]$ with its defining formula from Table 3.6, we get $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{C_1} = 1$ and $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{C_2} = 0$. Let $u_x, y_u \in U^{C_1}$ be the unique (Proposition 3.3.4) nodes such that $x^{C_1}(u_x) = 1$ and $y^{C_1}(u_y) = 1$. From the assumption that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, we have that there exist unique nodes $v_x, v_y \in\in U^{C_2}$ such that $x^{C_2}(v_x) = 1$ and $y^{C_2}(v_y) = 1$.

We now have that there exists unique nodes $u'_x = f(u_x) \in A_1^C$ and $u'_y = f(u_y) \in A_1^C$ such that $x^{A_1^C}(u'_x) = 1$ and $x^{A_1^C}(u'_y) = 1$. Therefore, $n^{A_1^C}(u'_x, u'_y) = n^{C_1}(u_x, u_y) = 1$ and $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{A_1^C} = 1$.

Furthermore, there exists unique nodes $v'_x = g(v_x) \in A_2^C$ and $v'_y = g(v_y) \in A_2^C$ such that $x^{A_2^C}(v'_x) = 1$ and $x^{A_2^C}(v'_y) = 1$. Therefore, $n^{A_2^C}(v'_x, v'_y) = n^{C_2}(u_x, u_y) = 0$ and $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge n(v_x, v_y)]\!]^{A_2^C} = 0$.

We conclude that $A_1^C \neq A_2^C$.

Case 3 : Distinction by *UList*$_2[x, y]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ and *UList*$_1[x, y]$, and that for some $x, y \in$ *Var* we have $[\![\mathit{UList}_2[x, y]]\!]^{C_1} = 1$ and $[\![\mathit{UList}_2[x, y]]\!]^{C_2} = 0$.

The meaning of $[\![\mathit{UList}_2[x, y]]\!]^{C_1} = 1$ is that there exist two nodes v_x and v_y in U^{C_1}, which are pointed-to by variables x and y, respectively, and a third node v_m, such that v_x, v_m, v_y is a maximal uninterrupted list in C_1. Therefore, *cul*$[x](v) \wedge \neg y(v)$ holds uniquely for v_m in C_1. In addition, $[\![\mathit{UList}_2[x, y]]\!]^{C_1} = 1$ implies $[\![\mathit{UList}_1[x, y]]\!]^{C_1} = 0$, since a maximal uninterrupted list has a determined integer length. Now, since C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ then there exist two nodes u_x and u_y in U^{C_2} that are pointed-to by variables x and y, respectively.

We consider the following three sub-cases: (i) There is no uninterrupted list between u_x and u_y. Therefore, $[\![\exists v : \mathit{cul}[x](v) \wedge \neg y(v)]\!]^{C_2} = 0$, and by Lemma A.2.1, $A_1^C \neq A_2^C$; (ii) There exists a maximal uninterrupted list between u_x and u_y of length 1. This possibility is ruled out since it contradicts the fact that $[\![\mathit{UList}_1[x, y]]\!]^{C_1} = 0$ with our assumption that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ and *UList*$_1[x, y]$; and (iii) There exists a maximal uninterrupted list between u_x and u_y of length > 2. This means that *cul*$[x](v) \wedge \neg y(v)$ holds for more than one node in C_2 (but only for v_m in C_1, and so by Lemma A.2.2, $A_1^C \neq A_2^C$.

Case 4 : Distinction by *UList*$[x, y]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, *UList*$_1[x, y]$, and *UList*$_2[x, y]$; and that for some $x, y \in$ *Var* we have $[\![\mathit{UList}[x, y]]\!]^{C_1} = 1$ and $[\![\mathit{UList}[x, y]]\!]^{C_2} = 0$.

Since $[\![\mathit{UList}[x, y]]\!]^{C_1} = 1$ we can substitute the definition of *cul*$[x](v)$ in the definition of *UList*$[x, y]$ and get $[\![\exists v : y(v) \wedge \mathit{cul}[x](v)]\!]^{C_1} = 1$. Applying this substitution For $[\![\mathit{UList}[x, y]]\!]^{C_2} = 0$ gives us $[\![\exists v : y(v) \wedge \mathit{cul}[x](v)]\!]^{C_2} = 0$. Therefore, by Lemma A.2.1, $A_1^C \neq A_2^C$.

Case 5 : Distinction by *UList*$_1[x, \mathbf{null}]$ predicates. Assume that C_1 and C_2 identify on

all predicates of the form *Aliased*$[x, y]$, *UList*$_1[x, y]$, *UList*$_2[x, y]$, and *UList*$[x, y]$; and that for some $x \in$ *Var* we have $[\![$*UList*$_1[x, \text{null}]]\!]^{C_1} = 1$ and $[\![$*UList*$_1[x, \text{null}]]\!]^{C_2} = 0$.

Since $[\![$*UList*$_1[x, \text{null}]]\!]^{C_1} = 1$, we have that there is no list emanating from the node pointed-to by x in C_1, and $[\![\exists v : \textit{cul}[x](v)]\!]^{C_1} = 0$. Since $[\![$*UList*$_1[x, \text{null}]]\!]^{C_2} = 0$, we have that there is a non-empty list emanating from the node pointed-to by x in C_2, and $[\![\exists v : \textit{cul}[x](v)]\!]^{C_2} = 1$. Therefore, by Lemma A.2.1, $A_1^C \neq A_2^C$.

Case 6 : Distinction by *UList*$_2[x, \text{null}]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, *UList*$_1[x, y]$, *UList*$_2[x, y]$, *UList*$[x, y]$, and *UList*$_1[x, \text{null}]$; and that for some $x \in$ *Var* we have $[\![$*UList*$_2[x, \text{null}]]\!]^{C_1} = 1$ and $[\![$*UList*$_2[x, \text{null}]]\!]^{C_2} = 0$.

We consider the following sub-cases: (i) There exists a maximal uninterrupted list of length 1 from the node pointed-to by x to null, in C_1, i.e., $[\![$*UList*$_2[x, \text{null}]]\!]^{C_2} = 1$. This case is ruled out, since by the assumption that C_1 and C_2 identify on all predicates of the form *UList*$_1[x, \text{null}]$ this would mean that $[\![$*UList*$_1[x, \text{null}]]\!]^{C_1} = 1$, which is not possible since there exists a maximal uninterrupted list of length 2 from that node to null and any maximal uninterrupted list has a determined integer length; (ii) There exists a maximal uninterrupted list of length > 2 from the node pointed-to by x to null, in C_1. This means that in C_1 the predicate *cul*$[x](v)$ holds for exactly one node (the one following the node pointed-to by x), and in C_2 the predicate *cul*$[x](v)$ holds for more than one node (all of the nodes following the node pointed-to by x). Therefore, by Lemma A.2.2, $A_1^C \neq A_2^C$; and (iii) There is no maximal uninterrupted list from x to null in C_2, which means that there exists a maximal uninterrupted list from x to a (possible the same) variable y, i.e., $[\![\exists v : \textit{cul}[x](v) \wedge y(v)]\!]^{C_2} = 1$. However, since in C_1 there is no maximal uninterrupted list from x to any variable, $[\![\exists v : \textit{cul}[x](v) \wedge y(v)]\!]^{C_1} = 0$, and therefore, by Lemma A.2.1, $A_1^C \neq A_2^C$.

Case 7 : Distinction by *UList*$[x, \text{null}]$ predicates. Assume that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, *UList*$_1[x, y]$, *UList*$_2[x, y]$, *UList*$[x, y]$, *UList*$_1[x, \text{null}]$, and *UList*$_2[x, \text{null}]$; and that for some $x \in$ *Var* we have $[\![$*UList*$[x, \text{null}]]\!]^{C_1} = 1$ and $[\![$*UList*$[x, \text{null}]]\!]^{C_2} = 0$. (This reasoning here is the same as the third sub-case in the previous case.)

This means that in C_2 there exists a maximal uninterrupted list from x to a (possible the same) variable y, i.e., $[\![\exists v : \textit{cul}[x](v) \wedge y(v)]\!]^{C_2} = 1$. However, since in C_1 there is no maximal uninterrupted list from x to any variable, $[\![\exists v : \textit{cul}[x](v) \wedge y(v)]\!]^{C_1} = 0$, and therefore, by Lemma A.2.1, $A_1^C \neq A_2^C$.

Case 8 : No distinctions by predicates from Table 3.6. Assume that C_1 and C_2 identify on all predicates from Table 3.6.

We show that A_1^C is isomorphic to A_2^C by showing that: (i) for every node $u_1 \in U^{A_1^C}$ there exists a unique corresponding node $u_2 \in U^{A_2^C}$ such that for every unary predicate $p(v)$ from Table 3.7 $p(u_1)^{A_1^C} = p(u_2)^{A_2^C}$ (i.e., A_1^C and A_2^C have the same set of canonic names); and (ii) for every pair of nodes $u_1, v_1 \in U^{A_1^C}$ and corresponding pair of nodes (with respect to the values of unary predicates) $u_2, v_2 \in U^{A_2^C}$, the equalities $n(u_1, v_1)^{A_1^C} = n(u_2, v_2)^{A_2^C}$ and $eq(u_1, v_1)^{A_1^C} = eq(u_2, v_2)^{A_2^C}$ hold.

Universe to universe bijection and preservation of unary predicates. Let u_1 be a node in $U^{A_1^C}$, and let X and L be subsets of *Var* such that the unary predicates that hold for u_1 in A_1^C are $x(u_1)$ for every $x \in X$ and *cul*$[x](v)$ for every $x \in L$.

We consider two cases separately according to the emptiness of X.

X **is non-empty.** From Proposition 3.3.4 we have that $f^{-1}(u_1) = \{v_1\}$, and $x^{C_1}(v_1) = 1$

for every $x \in X$. Thus, $[\![\exists v : x(v) \wedge y(v)]\!]^{C_1} = 1$ for every $x, y \in X$. From our assumption that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$ we get that $[\![\exists v : x(v) \wedge y(v)]\!]^{C_2} = 1$ for every $x, y \in X$. Using Proposition 3.3.4 we get that there exists a unique node $v_2 \in U^{C_2}$ such that $x^{C_2}(v_2) = 1$ for every $x \in X$. We denote by u_2 the node $g(v_2)$, which is designated as the corresponding node for u_1 in the isomorphism map, and using the definition of Canonical Abstraction we get that $x^{A_2^C}(u_2) = x^{A_1^C}(u_1)$ for every $x \in X$.

From the definition of the predicates *cul*$[x](v)$, we have that $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge$ *UList*$(v_y, v_x)]\!]^{C_1}$ for every $y \in L$ and $x \in X$. From our assumption that C_1 and C_2 identify on all predicates of the form *UList*$[x, y]$ we get that $[\![\exists v_x, v_y : x(v_x) \wedge y(v_y) \wedge$ *UList*$(v_y, v_x)]\!]^{C_2}$ for every $y \in L$ and $x \in X$. Using the definition of Canonical Abstraction we get that *cul*$[y]^{A_2^C}(u_2) =$ *cul*$[y]^{A_1^C}(u_1)$ for every $y \in L$.

X **is empty.** Let $f^{-1}(u_1) = V_1$ be the set of nodes mapped by f to u_1. Since X is empty we have that for every node $v_1 \in V_1$: $x^{C_1}(v_1) = 0$ for every $x \in Var$ and *cul*$[x]^{C_1}(v_1) = 1$ for every $x \in L$. Either V_1 is part of a maximal uninterrupted list from x to null, or V_1 is part of a maximal uninterrupted list from x to some variable y. In either case, from our assumption that C_1 and C_2 identify on all predicates of the form *Aliased*$[x, y]$, *UList*$[x, y]$, and *UListNULL*$[x]$, we have that there exists a non-empty set of nodes $V_2 \subseteq U^{C_2}$ such that for every $v_2 \in V_2$: $x^{C_2}(v_2) = 0$ for every $x \in Var$ and *cul*$[x]^{C_2}(v_2) = 1$ for every $x \in L$. Therefore, if we denote by u_2 the image of V_2 under g, we get from the definition of Canonical Abstraction that $x^{A_2^C}(u_2) = x^{A_1^C}(u_1)$ for every $x \in X$ and *cul*$[y]^{A_2^C}(u_2) =$ *cul*$[y]^{A_1^C}(u_1)$ for every $y \in L$. The uniqueness of u_2 is determined by the fact that the values of all unary predicates are considered for the nodes of V_2.

The correspondence by values of unary predicates defines a bijection $h : U^{A_1^C} \rightarrow U^{A_2^C}$ such that $h(u) = v$ when $p(u_1)^{A_1^C} = p(u_2)^{A_2^C}$ for every unary predicate $p(v)$ from Table 3.7.

Preservation of the binary predicate $eq(u, v)$**.** Since $eq(u, v)$ is interpreted as 0 in every 3-valued structure for distinct u and v, we are only interested in $eq(u, u)$.

Recall that by the meaning of the predicate $eq(u, v)$ its interpretation can either be 1 or $1/2$ (but never 0).

Let u_1 be a node in $U^{A_1^C}$ and let u_2 be $h(u_1)$. Assume that $eq(u_1, u_1)^{A_1^C} = 1/2$ (i.e., u_1 is a summary node). Let X be the set variables such that $x^{A_1^C}(u_1) = 1$ for every $x \in X$ and L be the set of variables such that *cul*$[y]^{A_1^C}(u_1) = 1$ for every $y \in L$. From Proposition 3.3.4 we get that $X = \emptyset$.

Denote by V_1 the set $f^{-1}(u_1)$. We have that $|V_1| > 1$, which means that V_1 are part of an uninterrupted list in C_1 containing more than two elements, which emanates from the node pointed-to by the variables in L.

Denote by V_2 the set $g^{-1}(u_2)$. Since we assumed that C_1 and C_2 identify on all predicates from Table 3.6, we get that from the node pointed-to by the variables in L emanates an uninterrupted list containing more than two elements in C_1. Hence, $|V_2| > 1$. Therefore, $eq^{A_2^C}(u_2, u_2) = 1/2$.

Preservation of the binary predicate $n(u, v)$**.** We will show that, for a structure in the image of Canonical Abstraction with the predicates from Table 3.7, the values of unary predicates together with the value of the predicate $eq(u, v)$, determine the value of the predicate $n(u, v)$. Since A_1^C and A_2^C are isomorphic with respect to those predicates, this completes the proof.

Let S be a structure in the image of Canonical Abstraction with the predicates from Table 3.7 and let u_1 and u_2 be two nodes in U^A. Furthermore, let X_1 and L_1 be the sets of variables such that the unary predicates that hold for u_1 are $x(v)$ for every $x \in X_1$ and *cul*$[x](v)$ for every $x \in L_1$, and let X_2 and L_2 be the sets of variables such that the unary predicates that hold for u_2 are $x(v)$ for every $x \in X_2$ and *cul*$[x](v)$ for every $x \in L_2$.

We consider the following sub-cases (the symmetric cases are not discussed):

X_1 **and** X_2 **are non-empty.** If $X_1 \subseteq L_2$ it means that u_1 and u_2 represent the end-points of a maximal uninterrupted list. If there is no node in U^A such that *cul*$[x](v)$ holds for some $x \in X_1$ then the list is of length 1 and therefore $n^S(u_1, u_2) = 1$. Otherwise, the length of the list is greater than 1 and $n^S(u_1, u_2) = 0$.

X_1 **is empty,** X_2 **is non-empty, and** $eq^S(u_1, u_1) = 0$**.** If $L_1 \subseteq L_2$ it means that u_2 represents the last node of a maximal uninterrupted list containing the nodes represented by u_1. Therefore, $n^S(u_1, u_2) = eq^S(u_1, u_1)$ and $n^S(u_2, u_1) = 0$.. If $L_2 \subseteq L_1$ it means that u_2 represents the first node of a maximal uninterrupted list containing the nodes represented by u_1. Therefore, $n^S(u_2, u_1) = eq^S(u_1, u_1)$ and $n^S(u_1, u_2) = 0$. Otherwise, u_1 and u_2 represent nodes belonging to distinct uninterrupted list and so $n^S(u_1, u_2) = n^S(u_2, u_1) = 0$.

X_1 **and** X_2 **are both empty.** If $u_1 = u_2$ then $n^S(u_1, u_1) = eq^S(u_1, u_1)$. Otherwise, this means that u_1 and u_2 represent distinct uninterrupted lists and therefore $n^S(u_1, u_2) = n^S(u_2, u_1) = 0$.

□

A.3 Proofs for Section 3.6

Proof:[of Proposition 3.3.2] A program variable points to at most 1 element, and therefore the number of list elements pointed by all program variables is at most n. The proof that the number of heap-shared elements is at most n is done by induction on the number of non-null variables.

Basis: Suppose the only non-null program variable is x. The proof is split into the following cases.

Case 1: The path from the element pointed by x reaches null. In this case, there are no heap-shared elements.

Case 2: The path from the element pointed by x reaches the element pointed by x, thereby forming a cycle. In this case, there are no heap-shared elements.

Case 3: The path from the element pointed by x reaches an element other than the one pointed by x. In this case, there is exactly 1 heap-shared element.

Induction hypothesis: Assume that the proposition holds for $k \geq 0$ non-null program variables.

Induction step: Suppose there are $k + 1$ non-null program variables $x_1, \ldots, x_{k+1}$. Let H_k be the sub-heap consisting of only the elements reachable from $x_1, \ldots, x_k$ and the links between them. The proof is split into the following cases, according to the interaction between variable x_{k+1} and H_k.

Case 1: The set of elements in H_k and the set of elements reachable from variables x_{k+1} do not intersect. By the induction hypothesis, the sub-heap H_k contains at most k heap-shared elements, and the sub-heap containing elements reachable from x_{k+1} contains at most a single heap-shared element. Therefore, the entire heap contains at most $k+1$ heap-shared elements.

Case 2: Variable x_{k+1} points to an element in H_k. Since the variable x_{k+1} in itself does not contribute to the in-degree of the element it points to, setting x_{k+1} to null does no change the number of heap-shared elements. Therefore, by the induction hypothesis, the heap contains at most k heap-shared elements.

Case 3: Variable x_{k+1} connects to the sub-heap H_k via the path $[u_1, \ldots, u_m]$ (none of $u_1, \ldots, u_m$ is heap-shared). By the induction hypothesis, H_k contains at most k heap-shared elements. The link from u_m to an element in H_k contributes at most a single heap-shared element to the entire heap, and therefore the entire heap contains at most $k+1$ heap-shared elements.

Proof:[of Proposition 3.3.4] To prove the first part of the claim, suppose u is heap-shared. If u is pointed-to by a program variable then the claim trivially holds. Since we assume that the heap is garbage-free, node u is reachable from some program variable. Let x be the program variable that reaches u on the shortest path. Obviously no node on the path is pointed-to by a program variable (otherwise there would be a shorter path from a different variable). By Corollary 3.3.3, the path from x to u consists of k maximal uninterrupted lists, for some $k < n$. Therefore, by definition, auxiliary variable $x_{s,k}(v)$ points to u.

The second part of the claim is proved by induction on the sharing-depth k.

Basis: The term $\mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v)$ means that $x_{s,1}(v)$ can hold only for a subset of interruptions that are heap shared but not pointed by any (regular) program variable. The term $\exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v)$ further restricts the set of nodes to only ones that are reachable by an uninterrupted list from a node pointed by the variable x. Since x is a reference variable, it can point to at most one node, which means that $\exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v)$ holds for at most one interruption. Therefore, the entire conjunction $\exists v_x.x(v_x) \wedge \mathit{UList}(v_x, v) \wedge \mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v)$ holds for at most one node.

Induction hypothesis: Assume that the proposition holds for every reference variables and sharing-depth $i \leq k$.

Induction step: By the induction hypothesis $x_{s,k}(v)$ holds for at most one node. Therefore, the arguments that were used to prove the basis hold (with x replaced by $x_{s,k}$) for the sub-formula

$$\exists v_k.x_{s,k}(v_k) \wedge \mathit{UList}(v_k, v) \wedge \mathit{HeapShared}(v) \wedge \neg \mathit{PtByVar}(v) \ .$$

The conjunction $\neg(\bigvee_{m=1\ldots k} x_{s,m}(v))$ can only further restrict the set of nodes for which the sub-formula above holds, and therefore the claim holds for the entire formula.

Proof:[of Proposition 3.6.2]

Figure A.2 shows a representative case of a concrete heap where the heap-sharing depth reaches the upper bound.

We will use the simple fact that, since the out-degree of any node in the heap is at most 1, every connected component of the heap (considering the undirected version of the relation $n(u, v)$) contains at most one simple cycle.

Let u be a heap-shared node of depth $k > 1$. There are two cases:

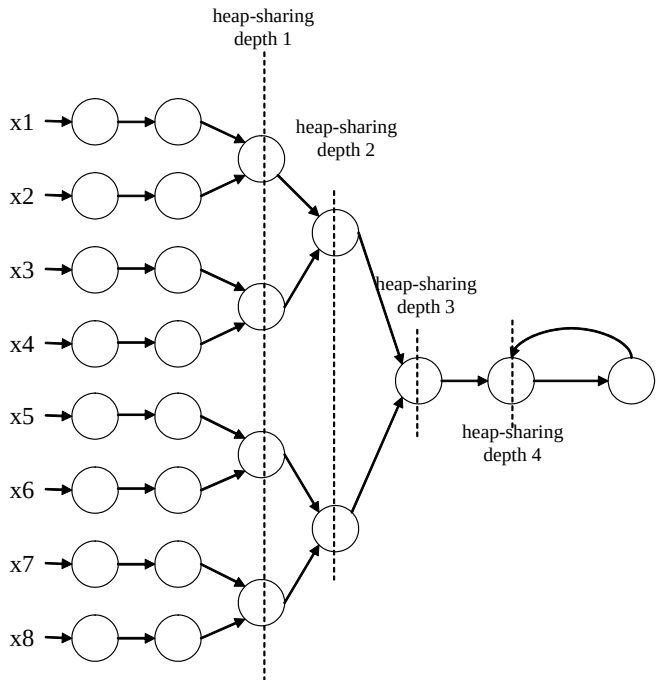


Figure A.2: A representative case for a heap sharing depth that reaches the upper bound. The vertical dashed lines are used to show the different levels of heap-sharing depth

Node u does not reside on a cycle. Consider the part of heap containing nodes that reach u. These nodes, along with u form a connected component without cycles where the out-degree of every node is at most 1. This is a tree with program variables at the leaves and u as a root.

The fact that u has heap-sharing depth k means that u is reachable from at least two distinct nodes a and b of heap-sharing depth $k-1$. In addition, a and b do not reside on a cycle.

The same reasoning can now be applied to a and b, obtaining the fact that u is reachable from at least 4 nodes of heap-sharing depth $k-2$. The reasoning goes on until we get to the leaves of the tree, and have that u is reachable from 2^k nodes that are pointed by program variables. This means that $2^k \leq n$ and therefore $k \leq \lfloor \log n \rfloor$.

Node u resides on a cycle. Since node u is heap-shared and found on a cycle, there are two distinct interrupting nodes a and b such that the lists from a to u and from b to u are maximal uninterrupted lists, and a is on the same that u is on and b is outside of that cycle.

Since b does not reside on a cycle, we have already shown that b can have a heap-sharing depth of at most $\lfloor \log n \rfloor$. Therefore, node u has heap-sharing depth of at most $\lfloor \log n \rfloor + 1$.

Appendix B

Additional Details for Chapter 5

B.1 The Code of queue_2_stacks

```
// A procedure that tests an implementation of a queue
// via two stacks.
class List {
    public List n;
    public Object data;

    public List(Object data) {
        this.data = data;
    }
}

class Queue {
    List stack1;
    List stack2;

    public void qneueue(Object elem) {
        // Push into stack1.
        List cell = new List(elem);
        cell.n = stack1;
        stack1 = cell;
    }

    public Object dequeue() {
        if (isEmpty())
            throw IllegalOperationException();

        List cell;
        if (stack2 != null) {
            // Pop from stack2
            cell = stack2;
            stack2 = cell.n;
            // In C we would also free cell here.
            return cell.data;
        }
        else {
            // Pop contents of stack1 and push it to stack2.
            while (stack1 != null) {
```

```
                cell = stack1;
                stack1 = cell.n;
                cell.n = stack2;
                stack2 = cell;
            }
            // Now pop from stack2.
            cell = stack2;
            stack2 = cell.n;
            // In C we would also free cell here.
            return cell.data;
        }
    }

    public boolean isEmpty() {
        return stack1 == null && stack2 == null;
    }

    public static void main(String[] args) {
        Queue q = new Queue();
        if (...) {
            q.enqueue(1);
            // Now stack1 is not empty and stack2 is empty.
        }
        else {
            q.enqueue(2);
            q.enqueue(3);
            q.dequeue();
            // Now stack1 is empty and stack2 is not empty.
        }
        // At this point the partially disjunctive abstraction
        // represents a state where both stacks are empty,
        // which causes a false alarm.
        q.dequeue();
    }
}
```

Appendix C

Proofs and Additional Details for Chapter 6

C.1 Proofs for Section 6.4

Proof:[of Theorem 6.4.5] Let j be any index in $\{1, \ldots, k\}$, and let TS_j be the corresponding combiner set.

$$I_1 \otimes \ldots \otimes I_k \sqsubseteq \bigotimes_{r \in \mathit{TS}_j} I_r \tag{C.1}$$

$\triangleright$ since $\otimes$ is monotone, i.e., $X \subseteq Y \implies \bigotimes_{r \in X} X \sqsubseteq \bigotimes_{r \in Y} Y$

$$\gamma(I) \sqsubseteq \gamma_{\mathit{TS}_j}(I) \tag{C.2}$$

$\triangleright$ by (C.1), (6.13), and (6.3)

$$\tau(\gamma(I)) \sqsubseteq \tau(\gamma_{\mathit{TS}_j}(I)) \tag{C.3}$$

$\triangleright$ since τ is monotone

$$\tau(\gamma(I)) \sqsubseteq \hat{\eta}_j(\tau(\gamma_{\mathit{TS}_j}(I))) \tag{C.4}$$

$\triangleright$ by (C.3) and since $\hat{\eta}_j$ is extensive

$$\tau(\gamma(I)) \sqsubseteq \tau_1[\mathit{TS}_j, j](I) \tag{C.5}$$

$\triangleright$ by (C.4) and (6.14)

$$\tau(\gamma(I)) \sqsubseteq \tau_1[\mathit{TS}_1, 1](I) \otimes \ldots \otimes \tau_1[\mathit{TS}_k, k](I) \tag{C.6}$$

$\triangleright$ since $\otimes$ is a meet operator

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\mathit{TS}_1, 1](I), \ldots, \tau_1[\mathit{TS}_k, k](I)) \tag{C.7}$$

$\triangleright$ by (6.3) and (C.6)

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\mathit{TS}](I))$$

$\triangleright$ by (6.15) and (C.7)

$\square$

Proof:[of Theorem 6.4.7] Let j be any index in $\{1, \ldots, k\}$, let $\mathit{TS}_j \subseteq \mathcal{P}(\{1, \cdots, k\})$ be the corresponding set of combiner sets, and let $Y \subseteq \{1, \cdots, k\}$ be a combiner set in TS_j.

$$\tau(\gamma(I)) \sqsubseteq \tau_1[R, j](I) \tag{C.8}$$

by (C.5) in $Theorem$6.4.5

$$\tau(\gamma(I)) \sqsubseteq \bigsqcap_{R \in TS_j} \tau_1[R, j](I) \tag{C.9}$$

by (C.8) and the properties of $\sqcap$

$$\tau(\gamma(I)) \sqsubseteq \tau_2[TS_j, j](I) \tag{C.10}$$

by (C.9) and (6.17)

$$\tau(\gamma(I)) \sqsubseteq \tau_2[TS_k, k](I) \otimes \ldots \otimes \tau_2[TS_k, k](I) \tag{C.11}$$

by (C.10) and since $\otimes$ is a meet operator

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[TS_1, 1](I), \ldots, \tau_2[TS_k, k](I)) \tag{C.12}$$

by (C.12) and (6.3)

$$\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[TS](I))$$

by (6.18) and (C.12)

□

C.2 HeDec System Optimizations

In this section we explain some of the important implementation details of the HeDec system.

HeDec implements standard fixed point iteration techniques where the abstract elements are tuples of sets of substates, one set per location selection predicate.

C.2.1 Incremental Transformers

We optimize the fixed point iteration by reusing the results from previous iterations. Without composition, the transformers are distributive and thus they are trivially incremental. The challenge is handling changes to sets from different tuples when they are combined. Combining sets is defined as $X_1 \otimes X_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in X_1, \sigma_2 \in X_2\}$ where $\sigma_1 \otimes \sigma_2$ is an operation that combines individual substates.

For two sets of substates X and Y, let ΔX and ΔY be new substates for each set, respectively. Now, we would like to compute $\tau((X \sqcup \Delta X) \otimes (Y \sqcup \Delta Y))$ by reusing $\tau(X \otimes Y)$. We use a known technique in computing differential fixpoint iterations (see, e.g., [EY04]), and use the transformer

$$\begin{aligned} \tau((X \sqcup \Delta X) \otimes (Y \sqcup \Delta Y)) \;=\; & \tau(X \otimes Y) \sqcup \\ & \tau(X \otimes \Delta Y) \sqcup \\ & \tau(Y \otimes \Delta X) \sqcup \\ & \tau(\Delta X \otimes \Delta Y) \end{aligned}$$

where the first joined element is taken from the previous iteration.

The use of incremental transformer is very important for efficiency. For example, on the non-blocking stack of Section 6.2.1, the incremental transformers improve the running times of 5 threads from 206 seconds to 36 seconds and of 10 threads from 2612 seconds to 211

```
[1]  #define EMPTY -1
[2]  typedef struct queue_t {
[3]    struct element_t *Head;
[4]    struct element_t *Tail;
[5]    lock_type HLock;
[6]    lock_type TLock;
[7]  } Queue;

//   @pre Queue->Head!=NULL &&
//        Queue->Tail!=NULL
[8]  void enqueue(Queue *Q, data_type v){
[9]    Node *x = alloc(sizeof(Node));
[10]   x->d = v;
[11]   lock(&Q->TLock);
[12]     Node *t = Q->Tail;
[13]     t->n = x;
[14]     Q->Tail = x;
[15]   unlock(&Q->TLock);
[16] }
```

```
     // @pre Queue->Head!=NULL &&
     //    Queue->Tail!=NULL
[17] data_type dequeue(Queue *Q){
[18]   lock(&Q->HLock);
[19]     Node *h = Q->Head;
[20]     Node *s = h->n;
[21]     if (s == NULL)
[22]       unlock(&Q->HLock);
[23]       return EMPTY;
[24]     data_type r = s->d;
[25]     Q->Head = s;
[26]   unlock(&Q->HLock);
[27]   return r;
[28] }
```

Figure C.1: Two-lock queue implementation

seconds. More than 10-fold improvement that increases as the complexity of the problem and the number of threads increase.

C.2.2 Optimized Composition for Sets of Substates

One of the costly operations in our framework is the combination operator on sets $X \otimes Y$ (which is implemented using the algorithm from [AMSS06]). The number of substates that need to be combined grows exponentially with the number of sets. In our benchmarks, we usually compose at most 3 sets but this is still very costly, in practice.

However, many of the pairs of substates that are combined are inconsistent, and thus do not contribute substates in the output. We therefore use pruning techniques to avoid combining many inconsistent substates unnecessarily.

For a state $\sigma \in X$, we say that $\mathit{signature}_X(\sigma)$ is a signature of σ in X, if for every $\sigma' \in X$, we have the property that if $\mathit{signature}_X(\sigma) \neq \mathit{signature}_X(\sigma')$ then σ and σ' are inconsistent. We use signatures based on unary predicates to combine sets of substates by:

$$X \otimes Y = \{\sigma_1 \otimes \sigma_2 \mid \mathit{signature}_{X \cup Y}(\sigma_1) = \mathit{signature}_{X \cup Y}(\sigma_2)\} \ .$$

We have observed, in our experiments, that using the optimized combination for sets reduces the amount of useless combinations operations by up to a factor of 100.

C.3 Case Study: Proving Linearizability for a Two-Lock Queue

Figure C.1 shows the two-lock queue implementation described in [MS96]. The queue has Head and Tail pointers, each protected with its own lock. Note that although the implementation uses locks, the algorithm allows benign data-races in case the queue is empty, i.e., the Head and Tail pointers are aliased.

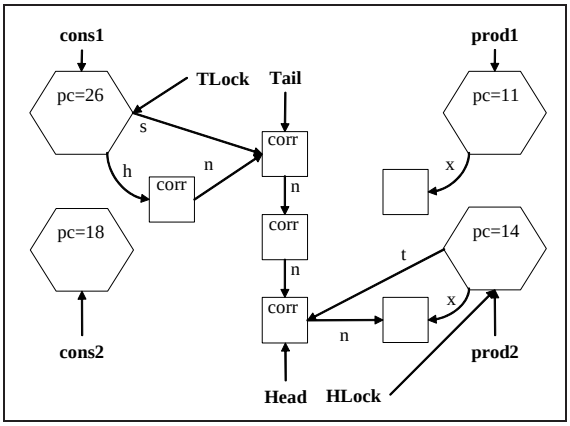


Figure C.2: A concrete memory in the two-lock queue implementation shown in Figure C.1

C.3.1 Concrete Execution

Figure C.2 shows one example of a store occurring in the two-lock queue implementation shown in Figure C.1. The figure shows two consumer threads and two producer threads. The elements of the heap already correlated with the sequential execution are marked with *corr*. Locks are depicted by arrows to the locking thread.

prod1 and **cons2** are waiting in the corresponding lock acquire point, waiting for the lock. **cons1** finished dequeuing an element from the queue and is about to release the lock. Finally, **prod2** has already added an element to the tail of the queue, but has not yet updated the Tail pointer. The source of exponential explosion in the state space exploration of the two-lock queue algorithm is the correlation between the program locations of the different threads as in the coarse-grained concurrency.

C.3.2 The Decomposition Scheme

We refine the decomposition scheme of Section 6.2.1 by adding a subdomain to represent the locks. The subheap contains the objects pointed-to by global variables and for each lock, the thread object acquiring it. Figure C.3 shows the the effect of applying this decomposition to the full state in Figure C.2.

The important thing to notice is that all the exponential explosion in the state space that existed in the full heap is eliminated by this decomposition. The possible subheaps of each thread become independent of the number of threads in the system (for more than 2 threads). The subheaps of the locks subdomain ($\{T_3\}$) only contain the thread information of 2 threads at most at a time.

C.3.3 Transformers

The compositions described in Section 6.2.1 work here as well. In the added operations of acquiring and releasing a lock, the subdomain of the currently executing thread is combined with the locks subdomain and each of the other components.

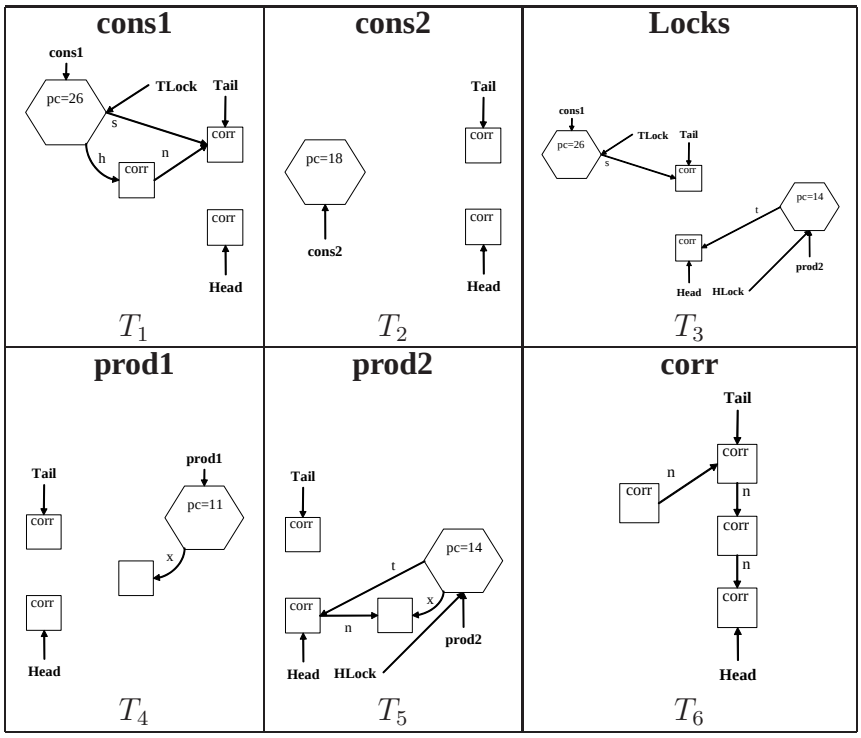


Figure C.3: The decomposed states abstracting the full state in Figure C.2. The names of the sub-domains appear above each substate