Symbolic Shape Analysis

Dissertation

zur Erlangung des Doktorgrades
der Fakultät für Angewandte Wissenschaften
der Albert-Ludwigs-Universität Freiburg im Breisgau
von

Thomas Wies

Dekan:	Prof. Dr. Bernhard Nebel
Referenten:	Prof. Dr. Andreas Podelski
	Prof. Dr. Amir Pnueli
	?
Tag der Einreichung:	?

Acknowledgments

First and foremost I would like to thank my supervisor Andreas Podelski for accepting me as a PhD student and for his support over all the years. I very much enjoyed our endless discussions while developing our symbolic shape analysis. Andreas' pursuit for simplicity and beauty is a great guiding principle for any young researcher.

I am grateful to Viktor Kuncak whose visit to the Max-Planck Institute for Computer Science in 2003 initiated my interest in shape analysis and marks the commence of a very fruitful collaboration. Viktor has always been a source for inspirations and insightful discussions. I also thank Viktor for having the courage to make the Jahob system available as a playground for my own tool development.

I am thankful to Patrick Lam for his work on the Hob system in which I implemented the first version of the Bohne tool and our field constraint analysis. Many of the later improvements have been based on our initial experience with Hob. I further express my gratitude to all contributers of the Jahob systems. Special thanks go to Charles Bouillaguet and Karen Zee for incorporating various external theorem provers into Jahob, which I subsequently used in Bohne.

I thank all members of the former Programming Logics group at the Max-Planck Institute for Computer Science and the members of the Software Engineering group at the University of Freiburg, including Berit Brauer, Harald Fecher, Jochen Hoenicke, Marlis Jost, Martin Preen, Bernd Westphal as well as my fellow students Stephan Arlt, Evren Ermis, Alexander Malkis, Stefan Maus, Martin Mehlmann, Corina Mitrohin, Martin Schäf, Nassim Seghir, Silke Wagner, and Martin Wehrle. They provided me with an excellent environment for my PhD research and made these years an unforgettable experience.

The members of the AVACS subproject "S2" gave valueable feedback on my research for which I am very grateful. In particular, I would like to thank Reinhard Wilhelm for giving me a hard time in holding my viewpoint on shape analysis. The discussions with him helped me to clearly define my research goals.

I thank Josh Berdine and Byron Cook as well as Shuvendu Lahiri and Shaz Qadeer for being my mentors during two internships at Microsoft Research in fall 2006 and fall 2007. Both internships have been a great experience.

I would like to thank my colleagues around the world for interesting discussions, including Cristiano Calcagno, Dino Distefano, Peter O'Hearn, Martin Rinard, Andrey Rybalchenko, Mooly Sagiv, and Hongseok Yang.

My PhD research has been sponsored by the DFG graduate schools "Quality Guarantees for Computer Systems" and "Mathematical Logic and Applications" as well as the Microsoft European PhD Scholarship Program. I thank the DFG and Microsoft for their financial support.

Finally, I thank my family and, in particular, my parents Heribert and Christine Wies, my brother Stefan, and Ružica Piskač for their love and support over all the years.

Meinen Eltern

Abstract

Program verification takes the most rigorous approach to ensure software reliability. In program verification one establishes a mathematical correctness proof which guarantees that the software behaves according to its specification. Program analysis tools can assist the developer in the verification process. Ideally a program analysis should be applicable to a wide range of verification problems without imposing a high burden on its users, i.e., without requiring deep mathematical knowledge and experience in formal verification.

A big step forward towards this ideal has been achieved by combining *abstract interpretation* with *automated reasoning*. An abstract interpretation automatically transforms a *concrete program* into an *abstract program*. The abstract program enables the program analysis to statically collect information over all possible executions of the concrete program. The collected information is used to automatically verify the correctness of the concrete program. Abstract interpretation shifts the burden of formally reasoning about programs from the developer to the designer of the program analysis tool, who has to find the right abstraction. Automated reasoning pushes the degree of automation even further. The use of theorem provers and decision procedures enables the automatic construction of the abstraction for a specific program and a specific correctness property and to automatically refine this abstraction if necessary. We refer to this approach of combining abstract interpretation with automated reasoning as *symbolic program analysis*

A problem that has recently seen much attention in program analysis and verification is the question of how to effectively deal with linked heap-allocated data structures. Program analyses that target properties of these data structures are commonly referred to as *shape analyses*. A symbolic shape analysis promises to handle a spectrum of different linked heap-allocated data structures, and a spectrum of properties to verify, without requiring the user to manually adjust the analysis to the specific problem instance. It was open what a symbolic shape analysis would look like, and whether it could fulfill its promise. In this thesis, we develop such a shape analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software is often the most unreliable part of today's computer systems. Yet, computers continue to pervade all aspects of modern technology. Therefore, finding effective methods that increase software reliability remains an important objective in programming language research.

Formal program verification takes the most rigorous approach to ensure software reliability. The goal of program verification is to formally prove that a program behaves according to its specification. Traditionally these correctness proofs have been established manually by the programmer in a formal calculus such as Hoare logic [46,56]. The main obstacles to a wide-spread use of this technique in practical software development are two-fold. First, only few programmers possess the knowledge and experience to formally reason about programs and second, the increasing complexity of software makes it hard to manually prove even simple correctness properties for entire software systems.

Since many years research in program verification has therefore been driven by the desire to develop program analysis tools that assist the programmer in the task of proving program correctness, i.e., to develop software that automatically verifies software. Since most problems related to program verification are undecidable, such tools can in general only provide approximative solutions. A formal framework for the design of approximative program analyses is formulated in *abstract interpretation* [37,38]. An abstract interpretation transforms the *concrete program* into an *abstract program* for which the verification problem is decidable. The abstraction guarantees that the concrete program is correct if the abstract program is correct. The analysis is approximative, because the concrete program might be correct even though the abstract program is incorrect, i.e., the analysis can produce counterexamples that do not exist in the concrete program. We call such counterexamples *spurious*. Abstract interpretation shifts the burden of formally reasoning about programs from the programmer to the designer of the program analysis tool, i.e., the designer has to find the right abstraction for a specific verification problem. While there will always be programs and properties that are too difficult to verify automatically and require manual proofs, abstract interpretation has been a great success. Today, abstract interpretation is the foundation of many tools that are used to automatically verify properties such as absence of runtime errors for industrial-scale programs [20, 113].

Recently, researchers started to investigate whether it is possible to push the degree of automation even further. Over the last years, we have seen significant advances in

theorem proving technology [12,41,108,109] and powerful decision procedures have become available [62]. The progress in automated reasoning makes it possible to automate the process of reasoning about programs. The goal of this research direction is to reduce program verification entirely to automated reasoning in expressive logics, i.e., one uses software to automatically construct software that automatically verifies software.

We refer to program analyses that combine abstract interpretation and automated reasoning technology as *symbolic*. Symbolic program analyses are interesting for several reasons. First, the use of automated reasoning procedures allows one not only to automate the transformation of the concrete program into an abstract program and the subsequent analysis of the abstract program, but also to automate the construction of the abstraction itself. Abstraction refinement techniques [35,52] apply automated reasoning procedures to decide whether a counterexample that is produced by the analysis is spurious. The detected spurious counterexamples are then used to automatically refine the abstraction. Second, the deployment of logics separates the problem of reasoning about the semantics of the program from the actual analysis of the program, i.e., one separates the problem of generating facts that imply program correctness from the problem of proving these facts. This separation of concerns allows one to formulate the analysis as an algorithmic problem that is independent of the programs and properties to verify. The analysis is then specialized for a specific verification task by choosing a logic and reasoning procedure that fits the given task. A further consequence of this separation of concerns is that establishing the correctness of a symbolic program analysis is easier than for a non-symbolic analysis; the most tedious parts in the correctness proof, i.e., those parts that are concerned with the semantics of the program, follow from the correctness of the underlying reasoning procedures. Finally, logics provide a natural language for specifying the behavior of code fragments. Therefore, symbolic program analysis can be easily combined with techniques that enable modular verification [11,44,65].

While there will always remain verification tasks that require the ingenuity of a program analysis designer who tailors an abstraction specifically for a given problem, symbolic program analysis can help to reduce his burden. The idea of symbolic program analysis has brought forth a new generation of program verification tools [9, 30, 54] that offer an unmatched degree of automation. These tools are already used by industry, e.g., as part of Microsoft's Windows device driver development kid [88].

A problem that has recently seen much attention in program analysis and verification is the question of how to effectively deal with linked heap-allocated data structures. The ability of linked data structures to dynamically grow and change their shape during program execution makes them a powerful programming concept in imperative programming languages. It is therefore not surprising that linked data structures are at the heart of many efficient algorithms and software design patterns. However, the flexibility and diversity of linked data structures also make it difficult to reason about programs that manipulate them. The importance and difficulty of data structure verification explains the increased interest in finding solutions to this problem.

Program analyses that target properties of linked data structures are commonly referred to as *shape analyses* [61]. A symbolic shape analysis promises to handle a spectrum of different linked heap-allocated data structures, and a spectrum of properties to verify, without requiring the user to manually adjust the analysis to the specific problem instance. It was

open what a symbolic shape analysis would look like, and whether it could fulfill its promise. In the present dissertation, we are concerned with these questions.

1.1 Symbolic Shape Analysis

The goal of a shape analysis is to verify complex consistency properties of linked data structures. By consistency properties we mean invariants on the shape of a data structure that are required to hold at specific points during program execution, e.g., at entry and exit points of library functions that implement the data structure. As an example, consider the Java program fragment shown in Figure 1.1. This program fragment shows parts of a data structure implementing containers that store an unbounded set of elements. The data structure supports various operations such as adding and removing elements from the set as well as more complex operations such as filtering the stored elements according to a given predicate. The actual set is implemented using a doubly-linked list. One of the consistency properties of this data structure therefore states that the list pointed to by field root forms a doubly-linked list. One uses shape analysis to verify that these invariants are preserved by all data structure operations.

The verification of consistency properties is important in itself because the correct execution of the program often requires data structure consistency, e.g., if the doubly-linked list property is violated at entry to method filter then the method will behave unexpectedly or even crash. In addition, such consistency properties are important for verifying other program properties. For instance, a termination proof for the while loop in method filter relies on the assumption that the list pointed to by field root is acyclic. One can use shape analysis to verify such assumptions.

In this thesis we investigate a new symbolic shape analysis. This shape analysis uses automated reasoning procedures to abstract a heap-manipulating program by a program that manipulates logical formulae. Our approach generalizes predicate abstraction [49], an existing symbolic program analysis technique, by incorporating the key ideas of three-valued shape analysis [103], an existing non-symbolic shape analysis. The fruitful combination of these techniques results in a shape analysis that exhibits a unique combination of qualities. First, our analysis is not *a priori* restricted to specific data structures and properties to verify, yet, it offers a high degree of automation. In particular, we used our analysis to verify complex user-specified consistency properties of data structure implementations. For instance, we were able to verify preservation of data structure invariants for operations on threaded binary trees [107] (including sortedness and the in-order traversal invariant) without manually adjusting the analysis to this specific problem and without providing user assistance beyond stating the properties to verify. We are not aware of any other shape analysis that can verify such properties with a comparable degree of automation.

Second, our shape analysis naturally fits into the Jahob approach of modular data structure verification [65,121]. This approach exploits user-provided procedure contracts to separate the verification of libraries (that implement data structures) from the verification of clients (that use these data structures). The library interfaces use abstract sets and relations to characterize the behavior of a data structure but hide the complexity of the underlying data structure implementation. For instance, the interface of class DLLSet in Figure 1.1 declares an abstract set content that denotes the set of objects stored in a given

```
public interface Predicate {
   //: public specvar pred :: objset;

   public boolean contains(Object o);
   //: ensures "result = (o ∈ pred)"
}

public class DLLSet {
   class Node {
      Node next;
      Node prev;
      Object data;
   }

   private Node root;

   /*: public specvar content :: objset;
       private vardefs "content == {x. <root reaches a node y via next such that y.data=x>}";
       invariant "<the list starting from root is acyclic>";
       invariant "<the list starting from root is doubly-linked>"; */

   public void add(Object o)
   /*: requires "o ∉ content"
       modifies content
       ensures "content = old content ∪ {o}" */
   {
      Node n = new Node();
      n.next = root;
      n.data = o;
      root.prev = n;
      root = n;
   }
    ...
   public void filter (Predicate p)
   /*: requires "p ≠ null"
       modifies content
       ensures "content = old content ∩ (pred p)" */
   {
      Node e = root;
      while (e != null) {
         Node c = e;
         e = e.next;
         if (!p.contains(c.data)) {
            if (c.prev == null) {
               e.prev = null;
               root = e;
            } else {
               c.prev.next = e;
               e.prev = c;
            }
         }
      }
   }
}
```

Figure 1.1: A set container implemented by a doubly-linked list

instance of the class. The set content is used in the pre and postconditions of public methods of class DLLSet to describe the methods' effects that are observable by the client. We used our shape analysis to verify such procedure contracts. While the analysis of the libraries requires the high precision of a shape analysis, the analysis of the clients calls for more scalable (but perhaps less precise) techniques. We believe that such a modular verification approach may be the key to make precise shape analysis applicable to large programs.

1.2 Technical Contributions

Our new symbolic shape analysis builds upon a series of technical contributions. These contributions are summarized as follows:

- We propose *Domain Predicate Abstraction*, a new parameterized abstract domain for symbolic shape analysis that can express detailed properties of different regions in the program's unbounded memory.
- We provide means to automate the transformation of a heap-manipulating program into an abstract program using automated reasoning procedures.
- We propose *Nested Lazy Abstraction Refinement*, an abstraction refinement technique for domain predicate abstraction. This technique eliminates the need for the user to manually adjust the abstraction for the analysis of a specific program or property.
- We present *Field Constraint Analysis*, a new technique for reasoning about data structures that enables the application of decidable logics to data structures which were originally beyond the scope of these logics.

We now discuss these contributions in detail.

Domain Predicate Abstraction. We show that the key idea of three-valued shape analysis [103], the partitioning of the heap according to predicates on heap objects, can be cast in the framework of predicate abstraction [49]. The symbiosis of these ideas results in a new analysis which we call *domain predicate abstraction*. Domain predicate abstraction enables the inference of invariants in the form of disjunctions of universally quantified facts over the program's unbounded memory. The building blocks of these quantified facts are predicates on heap objects. Our construction of the abstract post operator is analogous to the corresponding construction for classical predicate abstraction, except that predicates over objects take the place of state predicates, and Boolean heaps (sets of bit-vectors) take the place of Boolean states (bit-vectors). A program is abstracted to a program over Boolean heaps. For each command of the program, the corresponding abstract command is effectively constructed by use of automated reasoning. domain predicate abstraction thus provides a parametric framework for symbolic shape analysis.

Lazy Nested Abstraction Refinement. We develop a lazy nested abstraction refinement technique for symbolic shape analysis. Our technique uses spurious counterexamples to refine both the abstract domain and the abstract post operator of our symbolic shape

analysis (for each of the abstract domains). The two abstraction refinement phases are nested within a lazy abstraction refinement loop. The second refinement phase is crucial for the practical success of the shape analysis. In many benchmarks, the verification does not succeed *without*. The practical results are in line with the theoretical findings about the so-called *progress property*. Progress means that every spurious counterexample encountered during the analysis is eventually eliminated by a refinement step. We show that with the second refinement phase our lazy nested abstraction refinement loop has the progress property and it does not without. We further provide experimental evidence that the increased degree of automation obtained by abstraction refinement also results in targeted precision. This targeted precision is reflected by lower space consumption; the nested refinement loop seems to achieve the local fine-tuning of the abstraction at the required precision.

Field Constraint Analysis. One of the compelling characteristics of our symbolic shape analysis is that at its core one can plug in existing decision procedures and theorem provers as black boxes. However, in practice the capabilities of existing decision procedures often do not quite fit the requirements of the analysis. Therefore, it can be necessary to introduce an additional layer between the analysis and the actual decision procedure. We present one such technique in this thesis.

We introduce *field constraint analysis*, a new technique for reasoning about data structures. A field constraint for a reference field in a data structure is a formula specifying a set of objects to which the field can point. Field constraints enable the application of decidable logics to data structures which were originally beyond the scope of these logics, by dividing the fields of a data structure into backbone fields and derived fields that cross-cut the backbone in arbitrary ways. Reasoning about the data structure is reduced to reasoning about its backbone by exploiting constraints given on the derived fields. Previously, such derived fields could only be handled when they were uniquely determined by these field constraints. This significantly limited the range of supported data structures.

Our field constraint analysis permits *nondeterministic* field constraints on cross-cutting fields, which allows reasoning about data structures such as skip lists. Nondeterministic field constraints also enable the verification of invariants between data structures, yielding an expressive generalization of static type declarations.

The generality of our field constraints requires new techniques, which are orthogonal to the traditional use of structure simulation [57, 59]. We present one such technique and prove its soundness, as well as, completeness in interesting, but important cases. Using our technique we were able to verify data structures that were previously beyond the reach of similar techniques.

1.3 Proof of Concept

All the techniques presented in this thesis have been implemented and evaluated in a tool called Bohne. Bohne is implemented on top of the Jahob system for verifying data structure consistency [66]. Bohne analyzes Java programs annotated with special comments that specify procedure contracts and representation invariants of data structures such as the program shown in Figure 1.1. Our tool verifies that all methods comply to their procedure contracts and that all representation invariants are preserved by data structure operations.

We used Bohne to verify complex user-specified consistency properties for a range of data structures implementations and data structure clients without manually specified loop invariants or manually provided abstractions. This proves that symbolic shape analysis is able to fullfil its promise, namely to verify a diverse set of data structures and properties with a high degree of automation.

1.4 Outline

In Chapter 2 we give the preliminaries of our thesis. We briefly introduce higher-order logic, our notion of programs, and recall the foundations of abstract interpretation. Chapter 3 presents domain predicate abstraction. A less general version of this result has been presented at SAS'05 [98] and in [110]. In the fourth chapter we introduce our technique of abstraction refinement for domain predicate abstraction. Chapter 5 describes our field constraint analysis for automated reasoning about data structures. This material has been presented at VMCAI'06 [111]. Finally, Chapter 6 describes some implementation details of the Bohne tool and provides an overview of our case studies. Some of the techniques described in this chapter have been previously presented at VMCAI'07 [26] and HAV'07 [112].

Chapter 2

Preliminaries

2.1 Higher-Order Logic

We follow the approach taken in [65] and formalize programs, their semantics, and properties in terms of higher-order logic (HOL). The advantage of such an approach is that we can express both programs and their abstractions in a unique formalism. Futhermore, HOL does not impose any artificial restrictions on the properties that we are able to express.

The core of HOL is the simply typed lambda-calculus. We briefly sketch the foundations of this calculus for later reference. A more comprehensive discussion of typed lambda-calculi can be found, e.g., in [10, 55].

Types. Let B be a finite set of type constants. The set of Σ-types Types_B over B is defined as follows:

$$\begin{array}{rrll} t \in \mathit{Types}_B ::= & b & \text{where } b \in B & \text{(type constant)} \\ \mid & t_1 \Rightarrow t_2 & & \text{(total functions)} \end{array}$$

Following common convention, we consider function types to be right associative, i.e.

$$t_1 \Rightarrow t_2 \Rightarrow t_3 \;\equiv\; t_1 \Rightarrow (t_2 \Rightarrow t_3) \;.$$

Terms. We assume a countable infinite set V of variables with typical elements $v, w \in V$. A signature Σ is a tuple (B, C, ty) where B is a finite set of type constants, C a set of constant symbols disjoint from B, and ty a function $C \to \mathit{Types}_B$ mapping each constant symbol to a Σ-type over B. The set of Σ-terms Terms_Σ over a signature $\Sigma = (B, C, ty)$ is defined as follows:

$$\begin{array}{rrll} F \in \mathit{Terms}_\Sigma ::= & v & & \text{(variable)} \\ \mid & c & \text{where } c \in C & \text{(constant)} \\ \mid & \lambda v :: t.\, F & \text{where } t \in \mathit{Types}_B & \text{(lambda abstraction)} \\ \mid & F_1\, F_2 & & \text{(function application)} \end{array}$$

For nested lambda abstractions of the form $(\lambda v_1 :: t_1. \ldots \lambda v_n :: t_n.\, F)$ we use the abbreviation $\lambda (v_1 :: t_1) \ldots (v_n :: t_n).\, F$. Given a lambda abstraction of this form, we call F the *scope*

$$\frac{ty(c) = t}{\Gamma \vdash c :: t} \qquad \frac{\Gamma(v) = t}{\Gamma \vdash v :: t}$$

$$\frac{\Gamma[v \mapsto t_1] \vdash F :: t_2}{\Gamma \vdash \lambda v :: t_1 . F :: t_1 \Rightarrow t_2} \qquad \frac{\Gamma \vdash F_1 :: t_1 \Rightarrow t_2 \qquad \Gamma \vdash F_2 :: t_1}{\Gamma \vdash F_1\, F_2 :: t_2}$$

Figure 2.1: Typing relation.

of $\lambda(v_1 :: t) \ldots (v_n :: t_n)$. An occurrence of a variable v is *bound* if it is inside the scope of some $\lambda(v_1 :: t) \ldots (v_n :: t_n)$ and $v \in \{v_1, \ldots, v_n\}$. Other occurrences are called *free*. We denote by $\mathsf{FV}(F)$ the set of all free variables of F. A term F is *closed* if $\mathsf{FV}(F) = \emptyset$. We write $F(v_1, \ldots, v_n)$ to indicate that $\mathsf{FV}(F) \subseteq \{v_1, \ldots, v_n\}$.

A substitution σ is a partial mapping from variables to terms. We write $F\sigma$ to denote the term that results from simultaneously substituting every free occurrence of variables $v \in \mathsf{dom}(\sigma)$ in F_1 by $\sigma(v)$.

Typing relation. A typing context Γ is an assignment from variables to Σ-types. Let F be a Σ-term and t a Σ-type. The typing relation $\Gamma \vdash F :: t$ is defined in Figure 2.1. We say term F has type t under typing context Γ if $\Gamma \vdash F :: t$. We call F well-typed under Γ if there exists a type t such that $\Gamma \vdash F :: t$. We will often omit type annotations in lambda abstractions if these types can be inferred from the typing context.

Structures. A structure $\mathcal{A}$ for signature $\Sigma = (B, C, ty)$ is a function with $\mathsf{dom}(\mathcal{A}) = B \cup C$ and the following properties: $\mathcal{A}$ maps each type constant $b \in B$ to some nonempty set. We call $\mathcal{A}(b)$ the domain of type constant b. We extend $\mathcal{A}$ to a function on Σ-types as follows:

$$\begin{aligned} [\![b]\!]_{\mathcal{A}} &= \mathcal{A}(b) \\ [\![t_1 \Rightarrow t_2]\!]_{\mathcal{A}} &= [\![t_1]\!]_{\mathcal{A}} \to [\![t_2]\!]_{\mathcal{A}} \end{aligned}$$

Finally, $\mathcal{A}$ maps each constant symbol $c \in C$ to a value in $[\![ty(c)]\!]_{\mathcal{A}}$.

Let Γ be a typing context and let β be an assignment from variables $v \in V$ to values in $[\![\Gamma(v)]\!]_{\mathcal{A}}$. Further, let F be a Σ-term that is well-typed under Γ. The interpretation of F in a structure $\mathcal{A}$ under variable assignment β, written $[\![F]\!]_{\mathcal{A},\beta}$, is defined recursively on the structure of terms as follows:

$$\begin{aligned} [\![c]\!]_{\mathcal{A},\beta} &= \mathcal{A}(c) \\ [\![v]\!]_{\mathcal{A},\beta} &= \beta(v) \\ [\![\lambda v :: t . F]\!]_{\mathcal{A},\beta} &= \left\{ o_1 \mapsto o_2 \mid o_1 \in [\![t]\!]_{\mathcal{A}} \text{ and } o_2 = [\![F]\!]_{\mathcal{A},\beta[v \mapsto o_1]} \right\} \\ [\![F_1\, F_2]\!]_{\mathcal{A},\beta} &= ([\![F_1]\!]_{\mathcal{A},\beta})([\![F_2]\!]_{\mathcal{A},\beta}) \ . \end{aligned}$$

Formulae. In the rest of this thesis we will only consider signatures that contain at least the type constant bool for Booleans. We expect that structures interpret bool as the set $\mathbb{B} = \{0, 1\}$. A well-typed term of type bool is called a *formula*. We further assume that all

c	$ty(c)$	$\mathcal{A}(c)$
true	bool	1
$\neg$	$\mathsf{bool} \Rightarrow \mathsf{bool}$	$\lambda o.\, 1 - o$
$\wedge$	$\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$	$\lambda o_1\, o_2.\, \min\{o_1, o_2\}$
$\forall_t$	$(t \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$	$\lambda p.\, \min\{\, p(o) \mid o \in [\![t]\!]_{\mathcal{A}} \,\}$
$=_t$	$t \Rightarrow t \Rightarrow \mathsf{bool}$	$\lambda o_1\, o_2.\, \mathsf{if}\ o_1 = o_2\ \mathsf{then}\ 1\ \mathsf{else}\ 0$
ite_t	$\mathsf{bool} \Rightarrow t \Rightarrow t \Rightarrow t$	$\lambda o_1\, o_2\, o_3.\, \mathsf{if}\ o_1 = 1\ \mathsf{then}\ o_2\ \mathsf{else}\ o_3$

Table 2.1: Standard constant symbols in a signature $\Sigma = (B, C, ty)$ and their interpretation in Σ-structures $\mathcal{A}$.

Notation	Term	Typing constraints
false	$\neg\mathsf{true}$	
$F_1 \wedge F_2$	$\wedge\, F_1\, F_2$	
$F_1 \vee F_2$	$\neg(\neg F_1 \wedge \neg F_2)$	
$F_1 \rightarrow F_2$	$\neg F_1 \vee F_2$	
$\forall v :: t.\, F$	$\forall(\lambda v :: t.\, F)$	
$\exists v :: t.\, F$	$\neg(\forall v :: t.\, \neg F)$	
$F_1 = F_2$	$=\, F_1\, F_2$	
$\mathsf{if}\ F_1\ \mathsf{then}\ F_2\ \mathsf{else}\ F_3$	$\mathsf{ite}\, F_1\, F_2\, F_3$	
$\{\, v :: t.\, F \,\}$	$\lambda v :: t.\, F$	
$\{F\}$	$\lambda v :: t.\, v = F$	
$\emptyset$	$\lambda v :: t.\, \mathsf{false}$	
$F_1 \in S_1$	$S_1\, F_1$	$\Gamma \vdash S_1 :: t \Rightarrow \mathsf{bool}$
$S_1 \cap S_2$	$\lambda v :: t.\, v \in S_1 \wedge v \in S_2$	$\Gamma \vdash S_{1,2} :: t \Rightarrow \mathsf{bool}$
$S_2 \cup S_2$	$\lambda v :: t.\, v \in S_1 \vee v \in S_2$	$\Gamma \vdash S_{1,2} :: t \Rightarrow \mathsf{bool}$
$S_1 - S_2$	$\lambda v :: t.\, \mathsf{if}\ v \in S_2\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ v \in S_1$	$\Gamma \vdash S_{1,2} :: t \Rightarrow \mathsf{bool}$

Table 2.2: Syntactic sugar for terms constructed from standard constant symbols.

signatures provide at least the set of standard constant symbols listed in Table 2.1 and that structures respect the provided interpretations. For instance, the symbol $\neg$ denotes Boolean negation and $=_t$ denotes the equality predicate for type t. We will omit type subscripts from constant symbols whenever the type is uniquely determined by the subterms and the typing context.

For notational convenience we will use syntactic sugar for some formulae constructed from constant symbols in Figure 2.1. A list of short-hand definitions is shown in Table 2.2.

Given a formula F and a structure $\mathcal{A}$, we say that F is satisfiable in $\mathcal{A}$ under variable assignment β, written $\mathcal{A}, \beta \models F$, if $[\![F]\!]_{\mathcal{A},\beta} = 1$. Let D be a mapping from type constants in B to nonemepty sets. We write $[\![F]\!]_{D,\beta}$ to denote the set of all Σ-structures that are compatible with D and in which F is satisfiable under variable assignment β, i.e.:

$$[\![F]\!]_{D,\beta} \stackrel{\text{def}}{=} \{\, \mathcal{A} \mid \mathcal{A}|_B = D \ \wedge\ \mathcal{A}, \beta \models F \,\} \quad .$$

We say that F is valid in $\mathcal{A}$, written $\mathcal{A} \models F$, if F is satisfiable in $\mathcal{A}$ under all variable

assignments. If formula F is valid in structure $\mathcal{A}$ then we call $\mathcal{A}$ a model of F. We denote by $[\![F]\!]$ the set of all models of F. Formula F is called valid, written $\models F$, if it is valid in all Σ-structures. Finally, we say that formula F entails formula G, written $F \models G$, if the formula $F \to G$ is valid.

2.2 Programs

We now formalize programs. A program $P = (\Sigma, D, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ consists of:

- Σ: a signature $\Sigma = (B, C, ty)$.
- D: a domain mapping that maps each type constant $b \in B$ to a nonempty set.
- X: a finite set of program variables such that $X \subseteq C$.[1]
- $\mathcal{L}$: a finite set of control locations of the program.
- ℓ_0: an initial control location.
- ℓ_E: an error control location with $\ell_E \neq \ell_0$.
- $\mathcal{T}$: a finite set of program transitions. Each transition $\tau = (\ell, c, \ell')$ consists of an entry and exit location ℓ and ℓ', and a command c. Commands are defined by the following grammar, where x is a program variable of type t, E a closed Σ-term of type t and F a closed Σ-formula:

$$\begin{array}{rll} c \in \mathit{Com} ::= & x := E & \text{(assignment to } x\text{)} \\ \mid & \texttt{havoc}(x) & \text{(nondeterministic assignment to } x\text{)} \\ \mid & \texttt{assume}(F) & \text{(assume statement)} \\ \mid & c; c & \text{(sequential composition)} \end{array}$$

States. A program *state* s is a tuple $(\ell, \mathcal{A})$ where ℓ is a program location and $\mathcal{A}$ a Σ-structure such that $\mathcal{A}(b) = D(b)$ for all type constants $b \in B$. We use $s(pc)$ to denote program location ℓ of state s and $s(x)$ to denote $\mathcal{A}(x)$ for a program variable x. We call a state s with $s(pc) = \ell_E$ an *error state*. For a program variable $x \in X$ we denote by $s[x \mapsto o]$ the state that is obtained by updating the interpretation of x in s to o. The set of all states is given by *States*. We will often identify a state with the contained structure, e.g., we extend the satisfaction relation $\models$ from Σ-structures to states as expected: let $(\ell, \mathcal{A})$ be a state, β a variable assignment and F a formula then we define

$$(\ell, \mathcal{A}), \beta \models F \overset{\text{def}}{\iff} \mathcal{A}, \beta \models F \ .$$

We proceed similarly for validity and entailment.

[1] Program variables are logical constants and not to be confused with logical variables.

Transition Relations. Each command c represents a relation $[\![c]\!]$ that contains pairs of logical structures $(\mathcal{A}, \mathcal{A}')$ such that $\mathcal{A}$ and $\mathcal{A}'$ satisfy conditions given below for each kind of commands.

- If c updates a program variable $x := E$, we have $\mathcal{A}' = \mathcal{A}[[\![E]\!]_s]$,
- if c is a havoc command $\texttt{havoc}(x)$, we have $\mathcal{A}' = \mathcal{A}[x \mapsto o]$ where o is some value in $[\![ty(x)]\!]_s$,
- if c is an assume command $\texttt{assume}(F)$, we require that $\mathcal{A} \models F$ and $\mathcal{A}' = \mathcal{A}$
- and if c is a sequential composition $c_1; c_2$, there exists a structure $\mathcal{A}_0$ such that $(\mathcal{A}, \mathcal{A}_0) \in [\![c_1]\!]$ and $(\mathcal{A}_0, \mathcal{A}') \in [\![c_2]\!]$.

Finally, the *transition relation* $[\![\tau]\!]$ of a transition $\tau = (\ell, c, \ell')$ is the relation on states defined as follows:

$$[\![\tau]\!] \stackrel{\text{def}}{=} \left\{ ((\ell, \mathcal{A}), (\ell', \mathcal{A}')) \mid (\mathcal{A}, \mathcal{A}') \in [\![c]\!] \right\} \ .$$

Computations. A program *computation* is a (possibly infinite) sequence $\delta = s_0 \stackrel{c_0}{\rightarrow} s_1 \stackrel{c_1}{\rightarrow} \ldots$ of states and commands such that $s_0(pc) = \ell_0$ and for each pair of consecutive states s_i and s_{i+1} we have $(s_i, s_{i+1}) \in [\![(\ell, c_i, \ell')]\!]$ where (ℓ, c_i, ℓ') is a transition in $\mathcal{T}$. If δ is finite then for its final state, say s, and for each transitions $\tau \in \mathcal{T}$ there is no state s' such that $(s, s') \in [\![\tau]\!]$. We call any prefix of a computation a *trace* and we call command π that corresponds to the sequential composition of the commands in a trace a *path*. An *error trace* is a trace that reaches an error state and an *error path* is a path associated with an error trace. A program is called *safe* if it does not exhibit any error traces.

Predicate Transformers. Given a binary relation R on states and a set of states S, we define *strongest postcondition* post and *weakest liberal precondition* wlp as usual:

$$\begin{aligned} \mathsf{post}, \mathsf{wlp} &\in 2^{(States \times States)} \rightarrow 2^{States} \rightarrow 2^{States} \\ \mathsf{post}(R)(S) &\stackrel{\text{def}}{=} \left\{ s' \mid \exists s.\, (s, s') \in R \wedge s \in S \right\} \\ \mathsf{wlp}(R)(S) &\stackrel{\text{def}}{=} \left\{ s \mid \forall s'.\, (s, s') \in R \Rightarrow s' \in S \right\} \ . \end{aligned}$$

A closed formula F is a symbolic representation of a set of states, namely, the set of its models $[\![F]\!]$. It is therefore convenient to overload the predicate transformers post and wlp to *symbolic predicate transformers* that manuipulate formulae according to the seamtics of commands. Figure 2.2 provides the corresponding definitions. Note that the definitions in Figure 2.2 are not restricted to closed formulae. The correctness of these definitions is stated by the following proposition.

Proposition 1 *Let c be a command, F a formula, and β an assignment to the free variables of F then:*

$$\begin{aligned} \mathsf{post}([\![c]\!])([\![F]\!]_{D,\beta}) &= [\![\mathsf{post}(c)(F)]\!]_{D,\beta} \ \textit{and} \\ \mathsf{wlp}([\![c]\!])([\![F]\!]_{D,\beta}) &= [\![\mathsf{wlp}(c)(F)]\!]_{D,\beta} \ . \end{aligned}$$

$$
\begin{aligned}
\mathsf{post}(x{:=}E)(F) &\stackrel{\text{def}}{=} \exists v.\, F[x{:=}v] \wedge x = E[x{:=}v] \quad \text{with } v \notin \mathsf{FV}(F) \\
\mathsf{post}(\mathtt{havoc}(x))(F) &\stackrel{\text{def}}{=} \exists v.\, F[x{:=}v] \quad \text{with } v \notin \mathsf{FV}(F) \\
\mathsf{post}(\mathtt{assume}(G))(F) &\stackrel{\text{def}}{=} G \wedge F \\
\mathsf{post}(c_1; c_2)(F) &\stackrel{\text{def}}{=} \mathsf{post}(c_2)(\mathsf{post}(c_1)(F))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{wlp}(x{:=}E)(F) &\stackrel{\text{def}}{=} F[x{:=}E] \\
\mathsf{wlp}(\mathtt{havoc}(x))(F) &\stackrel{\text{def}}{=} \forall v.\, F[x{:=}v] \quad \text{with } v \notin \mathsf{FV}(F) \\
\mathsf{wlp}(\mathtt{assume}(G))(F) &\stackrel{\text{def}}{=} G \rightarrow F \\
\mathsf{wlp}(c_1; c_2)(F) &\stackrel{\text{def}}{=} \mathsf{wlp}(c_1)(\mathsf{wlp}(c_2)(F)) \ .
\end{aligned}
$$

Figure 2.2: Symbolic predicate transformers.

Proof. The prove goes by induction on the structure of commands. We only show the case for weakest liberal preconditions and assignment commands. The other cases are similar. Let c be the command $(x{:=}E)$ then we have:

$$
\begin{aligned}
\mathsf{wlp}([\![x{:=}E]\!])([\![F]\!]_{D,\beta}) &= \{\, s \mid \forall s'.\, (s, s') \in [\![x{:=}E]\!] \Rightarrow s' \in [\![F]\!]_{D,\beta} \,\} \\
&= \{\, s \mid \forall s'.\, (s' = s[x \mapsto [\![E]\!]_s]) \Rightarrow s' \in [\![F]\!]_{D,\beta} \,\} \\
&= \{\, s \mid s[x \mapsto [\![E]\!]_s] \in [\![F]\!]_{D,\beta} \,\} \\
&= \{\, s \mid s[x \mapsto [\![E]\!]_s], \beta \models F \,\} \\
&= \{\, s \mid s, \beta \models F[x{:=}E] \,\} \\
&= [\![F[x{:=}E]]\!]_{D,\beta} \ .
\end{aligned}
$$

■

2.2.1 Heap-Manipulating Programs

Programs defined in Section 2.2 provide an assembly language that allow us to model a broad range of systems. Heap-manipulating programs are the classical application domain of shape analysis. We will often use them as running examples throughout this thesis. In the following, we sketch how heap-manipulating programs can be modelled in terms of programs defined in Section 2.2.

Developing a memory model for a specific programming language which is both accurate and suitable for program analysis is a difficult research task in itself. For exposition purposes we consider a rather primitive imperative programming language with garbage collected heaps. A more sophisticated memory model for Java-like programs is described, e.g., in [65, 78]. A memory model for C programs that is suitable for verification can be found, e.g., in [33]. A heap manipulating program is a program $(\Sigma_h, D_h, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ with signature $\Sigma_h = (B_h, C_h, ty_h)$ and the following properties. There is a type constant $\mathsf{obj} \in B_h$ that models the set of available heap objects. The domain $D_h(\mathsf{obj})$ of type obj is a nonempty set of unspecified size. The set of program variables X consists of reference variables Var

and reference fields Fld. A reference variable denotes a heap object, i.e., for all $x \in \mathrm{Var}$ we have $ty_h(x) = \mathsf{obj}$. Reference fields model fields in data structures and denote functions on heap objects, i.e., for all $f \in \mathrm{Fld}$ we have $ty_h(f) = (\mathsf{obj} \Rightarrow \mathsf{obj})$.

Commands. The basic commands one would expect a heap-manipulating program to be composed of are:

- writing the content of a reference variable to a reference variable: $x{:=}\,y$,
- writing the content of a reference field at some index to a reference variable: $x{:=}\,y.f$,
- writing to a field at some particular index: $x.f{:=}\,y$,
- and allocation of a new heap object: $\mathtt{new}(x)$.

Read and write commands directly translate into assignments of program variables where a field access of the form $x.f$ simply correspond to a function application $f\,x$. The only interesting case is a field write of the form $(x.f{:=}\,y)$. We model field writes by assignments to function-valued program variables:

$$f{:=}\,(\lambda v :: \mathsf{obj}.\ \mathsf{if}\ v = x\ \mathsf{then}\ y\ \mathsf{else}\ f\,v) \quad .$$

We will use the short notation $f{:=}\,f[x{:=}\,y]$ for such updates.

For modelling allocation of fresh heap objects we need to track the set of allocated objects. For this purpose we assume a special program variable alloc of type $\mathsf{obj} \Rightarrow \mathsf{bool}$. The idea is that in a given state s the predicate $s(\mathsf{alloc})$ describes the subset of all heap objects in $D_h(\mathsf{obj})$ that are currently allocated. Allocation of a fresh heap object $\mathtt{new}(x)$ is then modelled by the following sequence of commands:

$$\begin{array}{l} \mathtt{havoc}(x); \\ \mathtt{assume}(x \notin \mathsf{alloc}); \\ \mathsf{alloc}{:=}\,\mathsf{alloc} \cup \{x\} \quad . \end{array}$$

Null Dereferences. A field in a heap-manipulating program is not always defined, e.g., because a specific object is not allocated. Thus reference fields more closely resemble partial functions on heap objects. Dereferencing a field on an object for which the field is undefined may cause the program to crash. In order to detect such runtime errors we add a special constant symbol null of type obj that models the undefined value. Thus, we can think of null as a special program variable that can never change its value. In order to ensure absence of runtime errors, every transition (ℓ, c, ℓ') where c contains a field access $f.x$ is guarded by a transition to the error location of the form $(\ell, \mathtt{assume}(f(x){=}\mathsf{null}), \ell_E)$. Thus, any potential dereference of null will be reflected by an error trace.

Background Formula. Any additional specifics of the memory model that restrict the set of possible program states can be encoded into a *background formula BG*. We require that all outgoing transitions from the initial location ℓ_0 to a location ℓ' are of the

form $(\ell_0, \texttt{assume}(BG), \ell')$. A reasonable choice for a background formula that encodes the memory model of a garbage-collected language is, e.g., as follows:

$$\begin{aligned} BG \stackrel{\text{def}}{=}\ & \mathsf{null} \in \mathsf{alloc} \\ & \wedge \bigwedge_{f \in \mathrm{Fld}} f\,\mathsf{null} = \mathsf{null} \\ & \wedge \forall v :: \mathsf{obj}, w :: \mathsf{obj}.\, v \notin \mathsf{alloc} \rightarrow \bigwedge_{f \in \mathrm{Fld}} f\,w \neq v \ . \end{aligned}$$

The first conjunct of the background formula guarantees that null is an allocated object. This ensures that after allocating a fresh object with command $\texttt{new}(x)$ program variable x will always point to an object different from null. The second conjunct guarantees that null always points back to itself. The last conjunct ensures that non-allocated objects are isolated in the heap.

Reachability Properties. Reachability properties play an important role in the analysis of heap programs. By reachability properties we mean properties such as:

"Reference variable x is reachable from reference variable y by following field f in the heap."

Such properties can be used for describing the shape of data structures. For instance, the fact that a list pointed to by a reference variable x is acyclic can be expressed by saying that x reaches null. Reachability properties are also important for expressing invariants such as disjointness of heap regions and for defining useful abstractions of recursive data structures. In order to formalize reachability properties, we assume that our signature for heap programs provides a reflexive transitive closure operator rtrancl_pt of type:

$$\mathsf{rtrancl\_pt} :: (\mathsf{obj} \Rightarrow \mathsf{obj} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{obj} \Rightarrow \mathsf{obj} \Rightarrow \mathsf{bool} \ .$$

The semantics of the reflexive transitive closure operator is given by the following equivalence:

$$\mathsf{rtrancl\_pt}\, p\, e\, t \ \equiv\ \forall S.\, e \in S \wedge (\forall v\, w.\, v \in S \wedge p\, v\, w \rightarrow w \in S) \rightarrow t \in S \ .$$

For instance, acyclicity of a list over field *next* that is pointed to by reference variable x is expressed by the following formula:

$$\mathsf{rtrancl\_pt}\, (\lambda v\, w.\, \mathit{next}\, v = w)\, x\, \mathsf{null} \ .$$

For our convenience we will use syntactic sugar for the reflexive transitive closure of reference fields and write "$\mathit{next}^*\, x\, \mathsf{null}$" as a shorthand for the above formula.

2.3 Abstract Interpretation

Most problems related to program verification are undecidable for any interesting class of programs. A possible solution is to analyze approximations of programs for which the

verification problem becomes decidable. Instead of proving that the concrete program behaves according to its specification one proves that an abstract program behaves according to an abstract specification. These approximations should be sound, i.e., correctness of the abstract program implies correctness of the concrete program.

The framework of abstract interpretation developed by Patrick and Radhia Cousot [37, 38] formalizes sound approximations of programs. In abstract interpretation the semantics of programs and their approximation is defined in terms of lattice-theoretic domains. The concrete semantics is given by the least fixed point of a functional on a complete lattice. For instance, for proving that a program is safe, we need to prove that the set of program states reachable by any trace of the program is disjoint from the error states. The set of reachable states of a program is given by the least fixed point of the operator post on the power-set lattice of program states. The abstract semantics is the least fixed point of an abstraction of the concrete functional on an abstract lattice. The abstract fixed point is an approximation of the concrete fixed point. For program safety this means that the abstract fixed point denotes a superset of the reachable program states.

We will formulate our symbolic shape analysis in terms of abstract interpretation. For later reference, we now give an overview of the key notions used in the abstract interpretation framework.

2.3.1 Partially Ordered Sets and Lattices

Definition 2 (Partially Ordered Set) *A set S with a binary relation $\sqsubseteq$ is called a* partially ordered set *if and only if the following three conditions hold:*

1. *$\leq$ is reflexive: $\forall x \in S. x \sqsubseteq x$,*
2. *$\leq$ is transitive: $\forall x, y, z \in S. x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$,*
3. *$\leq$ is antisymmetric: $\forall x, y \in S. x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$.*

A partially ordered set is denoted by $S(\sqsubseteq)$.

An element $x \in S$ is a lower bound *of $X \subseteq S$ if and only if $\forall x' \in X. x \leq x'$. A lower bound x of $X \subseteq S$ is called* greatest lower bound *if and only if for all lower bounds $x' \in S$ of X we have $x' \leq x$. Conversely, $x \in S$ is an* upper bound *of $X \subseteq S$ if and only if $\forall x' \in X. x' \leq x$. An upper bound x of $X \subseteq S$ is called* least upper bound *if and only if for all upper bounds $x' \in S$ of X we have $x \leq x'$.*

Definition 3 (Complete Partially Ordered Set) *Let set $S(\leq)$ be a partially ordered set. A* chain *$(x_i)_{i \in \mathbb{N}}$ is a monotone sequence of elements in S: $x_0 \leq x_1 \leq x_2 \ldots$.*

A partially ordered set $S(\leq)$ is called complete partially ordered set (CPO) *if and only if S has a least element $\bot$ and all chains $C \subseteq S$ have a least upper bound $\bigvee C$.*

Let $S(\leq)$ be a CPO. A function $f \in S \to S$ is called continuous if and only if f is monotone and f distributes over least upper bounds of chains. Formally, for all chains $(x_i)_{i \in \mathbb{N}}$ in S, $f(\bigvee (x_i)_{i \in \mathbb{N}}) = \bigvee (f(x_i))_{i \in \mathbb{N}}$.

Theorem 4 (Kleene's Fixed Point Theorem) *Let $S(\leq)$ be a CPO and $f \in S \to S$ a continuous function. Then f has a least fixed point:*

$$\mathsf{lfp}(f) = \bigvee (f^i(\bot))_{i \in \mathbb{N}} \ .$$

According to Tarski [106] it is sufficient to consider monotone functions rather than continuous functions to guarantee the existence of the least fixed point. However, in this thesis we will only consider continuous fixed point functionals.

Definition 5 (Lattice) *A partially ordered set $L(\leq)$ is called a* lattice *if and only if for all $x, y \in L$ there exists a least upper bound $x \vee y$ and a greeast lower bound $x \wedge y$ of x and y in L. A lattice is denoted by $L(\sqsubseteq, \vee, \wedge)$. We call $\vee$ the* join *and $\wedge$ the* meet operation *of the lattice.*

Definition 6 (Complete Lattice) *A lattice $L(\leq, \vee, \wedge)$ is called* complete *if for all $X \subseteq L$ there exists a least upper bound $\bigvee X$ and a greatest lower bound $\bigwedge X$ of X in L. In particular, L has a greatest element $\top = \bigvee L$ and a least element $\bot = \bigwedge L$. A complete lattice is denoted by $L(\leq, \vee, \wedge, \bot, \top)$.*

Note that every complete lattice is also a CPO.

Definition 7 (Completely Distributive Lattice) *A complete lattice $L(\leq, \vee, \wedge, \bot, \top)$ is called* completely distributive lattice *if and only if arbitrary joins in L distribute over arbitrary meets. Formally, for any doubly-index family $\{ x_{i,j} \in L \mid i \in I, j \in J_i \}$ we have:*

$$\bigwedge_{i \in I} \bigvee_{j \in J_i} x_{i,j} = \bigvee_{f \in F} \bigwedge_{i \in I} x_{i,f(i)}$$

where F is the set of all functions choosing for an index $i \in I$ an index $f(i) \in J_i$.

Proposition 8 *Let S be a set. Then $2^S(\subseteq, \cup, \cap, \emptyset, S)$ is a completely distributive lattice.*

Proposition 9 *Let S be a set and $L(\leq, \vee, \wedge, \bot, \top)$ be a completely distributive lattice. Then functions $S \to L$ form again a completely distributive lattice, where the ordering $\dot{\leq}$ on $S \to L$ is defined point-wise as follows:*

$$f \dot{\leq} f' \iff \forall x \in S.\, f(x) \leq f'(x) \ .$$

Example 10 The set of reachable states *Reach* of a program $P = (\Sigma, D, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ is the least fixed point of operator post under initial states. Formally, we define the set of initial states *Init*, and the transition relation $[\![P]\!] \in \mathit{States} \times \mathit{States}$ associated with a program P as follows:

$$\mathit{Init} \stackrel{\text{def}}{=} \{ s \in \mathit{States} \mid s(pc) = \ell_0 \}$$
$$[\![P]\!] \stackrel{\text{def}}{=} \bigcup_{\tau \in \mathcal{T}} [\![\tau]\!]$$

We then define a fixed point functional f:

$$f \stackrel{\text{def}}{=} \lambda S.\, \mathit{Init} \cup \mathsf{post}([\![P]\!])(S) \ .$$

The operator $\mathsf{post}(R)$ is continuous for any relation R, hence, so is f. Thus, the least fixed point of f exists according to Theorem 4:

$$\mathit{Reach} \stackrel{\text{def}}{=} \mathsf{lfp}(f) \ .$$

♦

2.3.2 Galois Connections

The connection between concrete and abstract lattice is formalized in terms of Galois connections.

Definition 11 (Galois Connection) *Let $L_1(\leq)$ and $L_2(\sqsubseteq)$ be partially ordered sets. The pair (α, γ) is called a* Galois connection*, or a pair of* adjoint functions *if and only if $\alpha \in L_1 \to L_2$, $\gamma \in L_2 \to L_1$ and:*

$$\forall x \in L_1, y \in L_2 . \alpha(x) \sqsubseteq y \iff x \leq \gamma(y) \ .$$

We call α the lower adjoint *and γ the* upper adjoint *of the Galois connection.*

The following Proposition summarizes some alternative characterizations of Galois connections.

Proposition 12 *Let $L_1(\leq, \vee, \wedge, \bot_1, \top_1)$ and $L_2(\sqsubseteq, \sqcup, \sqcap, \bot_1, \top_2)$ be two complete lattices. For functions $\alpha \in L_1 \to L_2$ and $\gamma \in L_2 \to L_1$ the following statements are equivalent:*

1. *(α, γ) is a Galois connection,*
2. *the following three conditions hold:*
   - *α and γ are monotone,*
   - *$\alpha \circ \gamma$ is reductive: $\forall y \in L_2 . \alpha(\gamma(y)) \sqsubseteq y$,*
   - *$\gamma \circ \alpha$ is extensive: $\forall x \in L_1 . x \leq \gamma(\alpha(x))$.*
3. *the following two conditions hold:*
   - *γ is a complete meet-morphism, i.e.*
     $\forall Y \subseteq L_2 . \gamma(\bigsqcap Y) = \bigwedge \{ \gamma(y) \mid y \in Y \}$ and $\gamma(\top_2) = \top_1$
   - *$\alpha = \lambda x \in L_1 . \bigsqcap \{ y \in L_2 \mid x \leq \gamma(y) \}$,*
4. *the following two conditions hold:*
   - *α is a complete join-morphism, i.e.*
     $\forall X \subseteq L_1 . \alpha(\bigvee X) = \bigsqcup \{ \alpha(x) \mid x \in X \}$ and $\alpha(\bot_1) = \bot_2$
   - *$\gamma = \lambda y \in L_2 . \bigvee \{ x \in L_1 \mid \alpha(x) \sqsubseteq y \}$,*

The equivalence of statements 1. and 2. is stated in [38, Theorem 5.3.0.4]. The fact that Statement 2. implies statements 3. and 4. is stated in [38, Corollary 5.3.0.5].

Proposition 13 *Let $L_1(\leq_1)$, $L_2(\leq_2)$, and $L_3(\leq_3)$ be partially ordered sets. Furthermore, let (α, γ) be a Galois connection between $L_1(\leq_1)$ and $L_2(\leq_2)$, and let (α', γ') be a Galois connection between $L_2(\leq_2)$ and $L_3(\leq_3)$. Then $(\alpha' \circ \alpha, \gamma \circ \gamma')$ is a Galois connection between $L_1(\leq_1)$ and $L_3(\leq_3)$.*

Strongest post and weakest liberal precondition are a canonical example of Galois connections.

Proposition 14 *Let P be a program and R a binary relation on states of P. Then $(\mathsf{post}(R), \mathsf{wlp}(R))$ is a Galois connection on sets of states of program P:*

$$\forall S, S' \in 2^{States}.\, \mathsf{post}(R)(S) \subseteq S' \iff S \subseteq \mathsf{wlp}(R)(S') \ .$$

For proof see, e.g., [104].

Let (α, γ) be a Galois connection between a concrete and an abstract lattice and let the concrete semantics of a program be defined by the least fixed point of a functional f on the concrete lattice. Then the abstract semantics is defined by the least fixed point of the abstraction of f which is given by the functional $\alpha \circ f \circ \gamma$ on the abstract lattice. The abstract semantics is guaranteed to be an approximation of the concrete semantics.

Proposition 15 *Let $L_1(\leq, \vee, \wedge, \bot_1, \top_1)$ and $L_2(\sqsubseteq, \sqcup, \sqcap, \bot_2, \top_2)$ be two complete lattices. Let $f \in L_1 \to L_1$ be a continous function, and let (α, γ) be a Galois connection between L_1 and L_2 then:*

$$\mathsf{lfp}(f) \ \leq \ \gamma(\mathsf{lfp}(\alpha \circ f \circ \gamma)) \ .$$

Chapter 3

Domain Predicate Abstraction

The transition graph of a program is formed by its states and the transitions between them. The idea of *predicate abstraction* [49] (used in tools such as SLAM [7] and BLAST [54]) is to abstract a state by its evaluation under a number of given state predicates; each edge between two concrete states in the transition graph gives rise to an edge between the two corresponding abstract states. One thus abstracts the transition graph to a graph over abstract states.

For a program manipulating the heap, each state is represented by a *heap graph*. A heap graph is formed by the allocated objects in the heap and pointer links between them. The idea of *three-valued shape analysis* [103] is to apply to the heap graph the same abstraction that we have applied to the transition graph. One abstracts an object in the heap by its evaluation under a number of predicates on heap objects; edges between concrete objects in the heap graph give rise to edges between the corresponding abstract objects. One thus abstracts a heap graph to a graph over abstract objects.

The analogy between predicate abstraction and the abstraction proposed in three-valued shape analysis is remarkable. It does not seem helpful, however, when it comes to the major challenge: how can one compute the abstraction of the transition graph when states are graphs and the abstraction is defined on nodes of the graph? This chapter answers a refinement of this question, namely whether the abstraction can be defined and computed in the formal setup and with the basic machinery of predicate abstraction.

In compliance with Chapter 2, program states are represented as logical structures rather than graphs. Thus, nodes in a graph correspond to objects in a domain of a logical structure. As in predicate abstraction, the analysis is an abstract interpretation [38] defined in terms of an abstract domain of formulae. These formulae are constructed from a finite set of predicates. However, in contrast to predicate abstraction, the building blocks of formulae are not state predicates, but *domain predicates*, meaning that they range not just over states of a program, but also over objects in the domains of these states. The formulae defining the abstract domain are given by universally quantified facts over objects in the domains of states. The building blocks of these facts are domain predicates. Thus, we do not just propose a generalization of predicate abstraction suitable for shape analysis, but more generally an analysis that enables the inference of universally quantified invariants.

Contributions. The key technical contributions that are described in this chapter are summarized as follows:

- We introduce a new abstract domain for an analysis that infers universally quantified facts about objects in the domains of program states.
- We generalize predicate transformers to *domain predicate transformers* that capture the effect of concrete transitions on objects in the domains of states. As for predicate transformers, domain predicate transformers can be computed symbolically via syntactic transformation of formulae.
- We show that one can implement the abstraction by a simple source-to-source transformation of a program to an abstract finite-state program which we call a Boolean heap program. This transformation is analogous to the corresponding transformation in predicate abstraction, except that domain predicates take the place of state predicates and Boolean heaps (sets of bitvectors) take the place of Boolean states (bitvectors).
- We formally identify the post operator of a Boolean heap program as an abstraction of the best abstract post operator on our abstract domain. For each command of the program, the corresponding abstract command is constructed by the application of a weakest liberal precondition operator on domain predicates and an entailment test (implemented by a syntactic manipulation of formulae, respectively, by a call to a decision procedure or theorem prover).

3.1 Boolean Heap Programs

Before we formally introduce domain predicate abstraction, it is instructive to highlight the main differences and similarities to predicate abstraction [49].

Predicate Abstraction. Following the framework of abstract interpretation [38], a static analysis is defined by lattice-theoretic domains and by fixed point iteration over these domains. For predicate abstraction the analysis computes an invariant (i.e., a superset of the reachable program states); the fixed point operator is an abstraction of the operator post; the concrete domain consists of sets of states. The abstract domain consists of sets of abstract states. The abstract domain is a finite sub-lattice of the concrete domain: each abstract state denotes an equivalence class of states, an element of the abstract domain denotes a union of such equivalence classes. The equivalence classes are induced by evaluating states under a finite set of abstraction predicates (closed formulae). Each equivalence class is represented by a bitvector over abstraction predicates, each element of the abstract domain by a set of such bitvectors.

The abstraction of the post operator corresponds to a finite-state Boolean program, one Boolean program variable per abstraction predicate. Thus, a state of the Boolean program corresponds to an abstract state. Each transition of the concrete program gives rise to transition in the abstract program that corresponds to a simultaneous update of the Boolean variables.

General scheme

Concrete command:
c

State predicates:
$\mathcal{P} = \{p_1, \ldots, p_n\}$

Abstract Boolean program:

var $p_1, \ldots, p_n ::$ bool
for each $p_i \in \mathcal{P}$ **do**
  if $\mathsf{wlp}^{\#}(c)(p_i)$ **then** $p_i{:=}\,\mathsf{true}$
  else if $\mathsf{wlp}^{\#}(c)(\neg p_i)$ **then** $p_i{:=}\,\mathsf{false}$
  else $p_i{:=}\,*$

Example

Concrete command:
var $x ::$ int
$x{:=}\,x+1$

State predicates:
$p_1 \equiv x = 0, \qquad p_2 \equiv x > 0$

Abstract Boolean program:

var $p_1, p_2 ::$ bool
if false **then** $p_1{:=}\,\mathsf{true}$
else if $p_1 \vee p_2$ **then** $p_1{:=}\,\mathsf{false}$
else $p_1{:=}\,*$
if $p_1 \vee p_2$ **then** $p_2{:=}\,\mathsf{true}$
else if $\neg p_1 \wedge \neg p_2$ **then** $p_2{:=}\,\mathsf{false}$
else $p_2{:=}\,*$

Figure 3.1: Construction of a Boolean program from a concrete command via predicate abstraction. All predicates are updated simultaneously. The value '*' stands for nondeterministic choice.

Figure 3.1 shows the transformation of a concrete command to the corresponding predicate updates in the abstract Boolean program. The actual abstraction step lies in the computation of $\mathsf{wlp}^{\#}(c)(p)$ – the best Boolean under-approximation (in terms of abstraction predicates) of the weakest liberal precondition of predicate p and command c. For example false is the best under-approximation of $\mathsf{wlp}(x{:=}\,x+1)(x=0)$ with respect to predicates p_1 and p_2. The abstract weakest liberal preconditions are computed automatically by checking validity of entailments using a decision procedure or automated theorem prover.

The resulting Boolean program is analyzed using finite-state model checking. If the error location is not reachable in the abstract program, then the concrete program is guaranteed to be safe.

Domain Predicate Abstraction. Our analysis proceeds analogously to predicate abstraction: (1) we choose a set of abstraction predicates for the abstraction (defining the abstract domain); (2) we construct an abstract finite-state program (the abstract post operator); and (3) we apply finite-state model checking to the abstract program (the fixed point computation). In the following, we explain in detail how the abstract domain and the construction of the abstract program look like.

In domain predicate abstraction, we define equivalence classes of objects in the domains of program states by evaluating them under a finite set of *domain predicates*. A domain predicate ranges over both program states and objects in the domains of these states. Domain predicates are represented symbolically by *domain formulae*. A domain formula is obtained from a formula by first-order lambda abstraction. As an example of such a domain

formula, consider the term

$$\lambda v :: \mathsf{obj}.\, next\, v = z \ .$$

This term evaluates to true for a given object in a given state, if the *next* field of this object points to program variable z. We call the equivalence classes induced by domain predicates *abstract objects*. Abstract objects are represented by bitvectors over domain predicates. An abstract state is given by a set of abstract objects, i.e., a set of bitvectors. A concrete state s belongs to the equivalence class represented by an abstract state, if every concrete object in the domain of s belongs to the equivalence class represented by one abstract object in the abstract state. The abstract domain of the analysis is given by sets of abstract states, i.e., sets of sets of bitvectors.

Intuitively, one can think of domain predicate abstraction as predicate abstraction being exponentiated. This also means that domain predicate abstraction is exponentially more succinct than standard predicate abstraction assuming the same number of abstraction predicates in both approaches. This additional precision enables domain predicate abstraction to express detailed properties about different regions in the domains of program states by using only a small number of predicates and makes domain predicate abstraction applicable for shape analysis. However, being exponentially more expressive also implies high costs for computing the abstract post operator. For an abstract domain given by abstract states over abstract objects it is exponentially more expensive to compute the best abstract post operator than it is for standard predicate abstraction. In order to avoid this exponential blowup, one would like to approximate the best abstract post operator by decomposing it into *local* updates. Local means that one updates each abstract object in isolation. The problem is: how can one account for the update of the global state by local updates on abstract objects?

We abstract a concrete program by a *Boolean heap program*. The abstraction is illustrated in Figure 3.2. The construction of a Boolean heap program naturally extends the one used in predicate abstraction. The difference is that a state of the abstract program is not given by a single bitvector, but by a set of bitvectors. Transitions in Boolean heap programs change the abstract state via local updates on abstract objects ($\vec{p}.p_i{:=}\mathsf{true}$) rather than global updates on the whole abstract state ($p_i{:=}\mathsf{true}$). Consequently, we replace the abstraction of the weakest liberal precondition operator on state predicates $\mathsf{wlp}^{\#}$ by the abstraction of a weakest liberal precondition operator on domain predicates $\dot{\mathsf{wlp}}^{\#}$. This construction avoids the exponential blowup that occurs in the construction of the best abstract post operator on abstract states. However, the analysis still provides an exponentially more succinct abstract domain than standard predicate abstraction.

In the rest of the chapter we give a formal account of Boolean heap programs. In particular, we make precise what it means to compute the operator $\dot{\mathsf{wlp}}^{\#}$. Furthermore, we identify the post operator of a Boolean heap program as an abstraction of the best abstract post operator on our abstract domain. Thus, we precisely identify the points in the analysis where we can lose precision.

General scheme

Concrete command:
c

Domain predicates:
$\mathcal{P} = \{p_1, \ldots, p_n\}$

Boolean heap program:
var V :: set of bitvectors over $\mathcal{P}$
for each $\vec{p} \in V$ **do**
 for each $p_i \in \mathcal{P}$ **do**
 if $\vec{p} \in \dot{\mathsf{wlp}}^{\#}(c)(p_i)$
 then $\vec{p}.p_i := \mathsf{true}$
 else if $\vec{p} \in \dot{\mathsf{wlp}}^{\#}(c)(\neg p_i)$
 then $\vec{p}.p_i := \mathsf{false}$
 else $\vec{p}.p_i := *$

Example

Concrete command:
var $x, y, z :: \mathsf{list}$
$x.next := y$

Domain predicates:
$p_1 \equiv \lambda v.\, x = v, \quad p_2 \equiv y = z,$
$p_3 \equiv \lambda v.\, next(v) = z$

Boolean heap program:
var V: set of bitvectors over $\{p_1, p_2, p_3\}$
for each $\vec{p} \in V$ **do**
 if $\vec{p}.p_1$ **then** $\vec{p}.p_1 := \mathsf{true}$
 else if $\neg\vec{p}.p_1$ **then** $\vec{p}.p_1 := \mathsf{false}$
 if $\vec{p}.p_2$ **then** $\vec{p}.p_2 := \mathsf{true}$
 else if $\neg\vec{p}.p_2$ **then** $\vec{p}.p_2 := \mathsf{false}$
 if $\neg\vec{p}.p_1 \wedge \vec{p}.p_3 \vee \vec{p}.p_1 \wedge \vec{p}.p_2$ **then**
 $\vec{p}.p_3 := \mathsf{true}$
 else if $\neg(\neg\vec{p}.p_1 \wedge \vec{p}.p_3 \vee \vec{p}.p_1 \wedge \vec{p}.p_2)$
 then $\vec{p}.p_3 := \mathsf{false}$

Figure 3.2: Construction of a Boolean heap program from a concrete command.

3.2 Domain Predicates

In predicate abstraction a state is abstracted by its evaluation under a finite number of predicates. These predicates are state predicates, i.e., sets of program states. As an example, consider the formula

$$F \equiv x > 0 \ .$$

Formula F denotes the predicate $[\![F]\!]$. A state s is in the denotation of F if $s(x)$ is greater than 0. We generalize predicate abstraction by considering predicates that do not just range over states, but also over objects in the domains of these states. As an example, consider the term

$$G \equiv \lambda v :: \mathsf{obj}.\, f\, v = z \ .$$

Given a state s, the term G evaluates to true for an object $\in s(\mathsf{obj})$ if its field f points to program variable z. Intuitively, the denotation $[\![G]\!]$ is a function of a dependent type

$$\Pi s \in \mathit{States}.\, s(\mathsf{obj}) \to \{0, 1\} \ .$$

For technical reasons, we use an equivalent representation of the denotation of such terms. We are not interested in arbitrary logical structures, but in logical structures that correspond to program states. All remaining definitions in this chapter are subject to a particular program P. For convenience we fix a particular program $P = (\Sigma, D, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ for the

rest of this chapter and keep all dependencies on P implicit. Recall that the domains of all program states of program P are fixed by the domain mapping D. Thus, we can define the denotation of term G as follows:

$$\begin{aligned}\llbracket G \rrbracket &\in D(\mathsf{obj}) \to 2^{States} \\ \llbracket G \rrbracket &= \lambda o \in D(\mathsf{obj}).\ \{\, s \in States \mid s, [v \mapsto o] \models f\ v{=}x \,\} \\ &= \lambda o \in D(\mathsf{obj}).\ \llbracket f\ v{=}x \rrbracket_{D,[v \mapsto o]}\ .\end{aligned}$$

These considerations suggest the following formal definition of domain predicates.

Definition 16 (Domain Predicates) *A* domain predicate *p over basic types $b_1, \ldots, b_n$ is a function*

$$p \in (D(b_1) \times \cdots \times D(b_n)) \to 2^{States}\ .$$

We call n the arity of p and we denote by $DomPreds(D(b_1) \times \cdots \times D(b_n))$ the set of all domain predicates over basic types $b_1, \ldots, b_n$. A term F of the form

$$\lambda v_1 :: b_1 \ldots v_n :: b_n.\, G$$

where G is a closed formula, is called a domain formula*. The denotation of a domain formula is a domain predicate:*

$$\llbracket F \rrbracket \stackrel{\text{def}}{=} \lambda \vec{o} \in (D(b_1) \times \ldots \times D(b_n)).\ \llbracket G \rrbracket_{D,[\vec{v} \mapsto \vec{o}]}\ .$$

A domain formula is obtained by lambda abstraction from a formula. Thus, we can think of sets of states as 0-ary domain predicates. This observation suggest that we can lift operations on sets of states to operations on domain predicates. In particular, we can obtain a partial order $\dot{\subseteq}$ on domain predicates by point-wise lifting set inclusion on sets of states as follows: let p and q be domain predicates over the same domain Dom then we define:

$$p \mathrel{\dot{\subseteq}} q \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall \vec{o} \in Dom.\ p(\vec{o}) \subseteq q(\vec{o})\ .$$

Likewise, we lift set union $\cup$ and intersection $\cap$ to operations $\dot{\cup}$ and $\dot{\cap}$ on domain predicates. Let p and q be domain predicates over the same domain Dom then we define:

$$\begin{aligned} p \mathbin{\dot{\cup}} q &\stackrel{\text{def}}{=} \lambda \vec{o} \in Dom.\ p(\vec{o}) \cup q(\vec{o}) \\ p \mathbin{\dot{\cap}} q &\stackrel{\text{def}}{=} \lambda \vec{o} \in Dom.\ p(\vec{o}) \cap q(\vec{o})\ . \end{aligned}$$

The order $\dot{\subseteq}$ together with the operations $\dot{\cup}$ and $\dot{\cap}$ induce a lattice structure.

Proposition 17 *The domain predicates over common domain Dom form a completely distributive lattice with partial order $\dot{\subseteq}$, join $\dot{\cup}$, meet $\dot{\cap}$, least element $\lambda \vec{o} \in Dom.\, \emptyset$, and greatest element $\lambda \vec{o} \in Dom.\, States$.*

Proof. Follows from Proposition 8 and Proposition 9. ∎

Not only can we lift the lattice structure of sets of states to domain predicates, we can further lift predicate tranformers (which are operations on sets of states) to *domain predicate transformers*.

Definition 18 (Domain Predicate Transformers) *The* domain predicate transformers $\dot{\mathsf{post}}$ *and* $\dot{\mathsf{wlp}}$ *for domain predicates over common domain Dom are defined as follows:*

$$\begin{aligned}\dot{\mathsf{post}}, \dot{\mathsf{wlp}} &\in 2^{(States \times States)} \to DomPreds(Dom) \to DomPreds(Dom)\\ \dot{\mathsf{post}}(R)(p) &\stackrel{\text{def}}{=} \lambda \vec{o} \in Dom.\, \mathsf{post}(R)(p(\vec{o}))\\ \dot{\mathsf{wlp}}(R)(p) &\stackrel{\text{def}}{=} \lambda \vec{o} \in Dom.\, \mathsf{wlp}(R)(p(\vec{o}))\ .\end{aligned}$$

For a given domain predicate p, we call $\dot{\mathsf{post}}(R)(p)$ the strongest domain postcondition and $\dot{\mathsf{wlp}}(R)(p)$ the weakest domain precondition of p with respect to relation R.

Domain predicate transformers are one of the key incrediences that allow us to charactierize Boolean heap programs. Since the domain predicate transformers are obtained from the standard predicate transformers via a simple lifting, their characteristic properties are preserved. In particular, $\dot{\mathsf{post}}$ and $\dot{\mathsf{wlp}}$ form a Galois connection on the complete lattice of domain predicates.

The intuition behind domain predicate transformers becomes more clear when one represents domain predicates in terms of domain formulae. As in the case of predicate transformers post and wlp, we can characterize domain predicate transformers in terms of syntactic transformations of domain formulae. For instance, consider the domain formula $G \equiv (\lambda v :: \mathsf{obj}.\, f\, v = z)$ then the weakest domain precondition of domain predicate $[\![G]\!]$ and command $c = (x.f := y)$ is given by:

$$\begin{aligned}\dot{\mathsf{wlp}}([\![c]\!])([\![G]\!]) &= \lambda o \in D(\mathsf{obj}).\, \{\, s \mid \forall s'.\, (s, s') \in [\![c]\!] \Rightarrow s' \in ([\![G]\!](o)) \,\}\\ &= [\![G[f := \lambda v :: \mathsf{obj}.\, \mathsf{if}\ x = v\ \mathsf{then}\ y\ \mathsf{else}\ f\, v]]\!]\\ &= [\![\lambda v :: \mathsf{obj}.\, (\lambda v :: \mathsf{obj}.\, \mathsf{if}\ x = v\ \mathsf{then}\ y\ \mathsf{else}\ f\, v)\, v = z]\!]\\ &= [\![\lambda v :: \mathsf{obj}.\, x = v \wedge y = z \vee x \neq v \wedge f\, v = z]\!]\ .\end{aligned}$$

The resulting formula denotes the domain predicate that given an object o contains all states s where the f-successor of o is pointed to by z in the successor state of s under c.

The correctness of the transformation performed in the above example is justified by the following proposition.

Proposition 19 *Let F be a domain formula over domain Dom. The weakest domain precondition of the domain predicate $[\![F]\!]$ is characterized as follows:*

$$\begin{aligned}\dot{\mathsf{wlp}}([\![x := E]\!])([\![F]\!]) &= [\![F[x := E]]\!]\\ \dot{\mathsf{wlp}}([\![\mathtt{havoc}(x)]\!])([\![F]\!]) &= [\![\lambda \vec{v}.\, \forall w.\, (F\, \vec{v})[x := w]]\!] && \text{with } w \notin \mathsf{FV}(F\, \vec{v})\\ \dot{\mathsf{wlp}}([\![\mathtt{assume}(E)]\!])([\![F]\!]) &= [\![\lambda \vec{v}.\, E \to F\, \vec{v}]\!]\\ \dot{\mathsf{wlp}}([\![c_1; c_2]\!])([\![F]\!]) &= [\![\lambda \vec{v}.\, \mathsf{wlp}(c_1)(\mathsf{wlp}(c_2)(F\, \vec{v}))]\!]\ .\end{aligned}$$

Proof. Let $t(c, F)$ be the syntactic transformation on domain formulae F that is defined by the right-hand sides of the equations given for each kind of command c. Then for all

commands c and domain formulae F of form $\lambda\vec{v}.\, G$:

$$\begin{aligned}
\dot{\mathsf{wlp}}(\llbracket c\rrbracket)(\llbracket F\rrbracket) &= \lambda\vec{o} \in \mathit{Dom}.\, \mathsf{wlp}(\llbracket c\rrbracket)(\llbracket F\rrbracket(\vec{o})) && \text{(Def. of } \dot{\mathsf{wlp}}\text{)}\\
&= \lambda\vec{o} \in \mathit{Dom}.\, \mathsf{wlp}(\llbracket c\rrbracket)(\llbracket G\rrbracket_{D,[\vec{v}\mapsto\vec{o}]})\\
&= \lambda\vec{o} \in \mathit{Dom}.\, \llbracket t(c,G)\rrbracket_{D,[\vec{v}\mapsto\vec{o}]} && \text{(Prop. 1)}\\
&= \lambda\vec{o} \in \mathit{Dom}.\, \llbracket t(c,F)\rrbracket(\vec{o})\\
&= \llbracket t(c,F)\rrbracket\ .
\end{aligned}$$

■

Given the characterization of $\dot{\mathsf{wlp}}$ in Proposition 19, it is convenient to overload $\dot{\mathsf{wlp}}$ to an operator on domain formulae in the same way we extended wlp to an operator on formulae. Thus, for a domain formula F and command c we denote by $\dot{\mathsf{wlp}}(c)(F)$ the corresponding domain formula that is given on the right-hand side of the corresponding equation in Proposition 19.

3.3 Domain Predicate Abstraction

We now formally introduce domain predicate abstraction. Following the framework of abstract interpretation [37], we formalize the abstraction in terms of a Galois connection between a concrete and an abstract lattice. In domain predicate abstraction, we have two concrete and two abstract lattices that are defined in terms of each other and two Galois connections, each connecting one pair of abstract and concrete lattice. The concrete lattices are sets of states, respectively, domain predicates and the abstract lattices are sets of abstract states, respectively, sets of abstract objects.

3.3.1 Abstract Domains

The abstract domains of our analysis is parameterized by a finite set of domain predicates. For the rest of this chapter we fix a particular finite set of domain predicates $\mathcal{P}$. We assume that all predicate in $\mathcal{P}$ range over the same domain which we denote by $\mathsf{dom}(\mathcal{P})$.

Definition 20 (Abstract Objects) *We call a function $o^\#$ mapping abstraction predicates in $\mathcal{P}$ to values in $\{\{0\},\{1\},\{0,1\}\}$ an* abstract object. *We denote by $\mathit{AbsObjs}$ the powerset over abstract objects, i.e.:*

$$\mathit{AbsObjs} \stackrel{\text{def}}{=} 2^{\mathcal{P}\to\{\{0\},\{1\},\{0,1\}\}}.$$

We define a relation $\dot{\sqsubseteq}$ on sets of abstract objects as follows: let $O_1^\#, O_2^\# \in \mathit{AbsObjs}$ then

$$O_1^\# \,\dot{\sqsubseteq}\, O_2^\# \iff \forall o_1^\# \in O_1^\#.\, \exists o_2^\# \in O_2^\#.\, \forall p \in \mathcal{P}.\, o_1^\#(p) \subseteq o_2^\#(p)\ .$$

The relation $\dot{\sqsubseteq}$ forms a preorder on sets of abstract objects. For convenience we identify $\mathit{AbsObjs}$ with the quotient $(\mathit{AbsObjs}/(\dot{\sqsubseteq} \cap \dot{\sqsubseteq}^{-1}))$ and consider elements of $\mathit{AbsObjs}$ as representatives of their equivalence class in the quotient. Then $\dot{\sqsubseteq}$ forms a partial order on sets of abstract objects.

We can define a function expand mapping sets of abstract objects to sets of total functions $\mathcal{P} \to \{\{0\},\{1\}\}$ as follows:

$$\mathsf{expand} \in \mathit{AbsObjs} \to 2^{\mathcal{P}\to\{\{0\},\{1\}\}}$$
$$\mathsf{expand}(O^\#) \stackrel{\text{def}}{=} \bigcup_{o^\# \in O^\#} \left\{ f \mid \forall p \in \mathcal{P}.\, f(p) \subseteq o^\#(p) \right\}$$

The function expand is bijective. In fact, expand is an order isomorphism between sets of abstract objects ordered by $\dot{\sqsubseteq}$ and the powerset over functions $\mathcal{P} \to \{\{0\},\{1\}\}$ ordered by set inclusion. We can define join and meet operations on sets of abstract objects, as follows: let $O_1^\#$ and $O_2^\#$ be sets of abstract objects then:

$$O_1^\# \mathbin{\dot{\sqcup}} O_2^\# \stackrel{\text{def}}{=} \mathsf{expand}(O_1^\#) \cup \mathsf{expand}(O_2^\#)$$
$$O_1^\# \mathbin{\dot{\sqcap}} O_2^\# \stackrel{\text{def}}{=} \mathsf{expand}(O_1^\#) \cap \mathsf{expand}(O_2^\#) \ .$$

Thus, by construction expand forms a lattice isomorphism between sets of abstract objects and the free lattice over functions $\mathcal{P} \to \{\{0\},\{1\}\}$.

Proposition 21 *Sets of abstract objects AbsObjs form a complete lattice with partial order $\dot{\sqsubseteq}$, join $\dot{\sqcup}$, meet $\dot{\sqcap}$, least element $\emptyset$, and greatest element $\{\lambda p.\{0,1\}\}$.*

The motivation for defining sets of abstract objects as a lattice over functions of type $\mathcal{P} \to \{\{0\},\{1\},\{0,1\}\}$ rather than the free lattice over functions $\mathcal{P} \to \{\{0\},\{1\}\}$ (or even $\mathcal{P} \to \{0,1\}$) will become clear in Section 3.4.2.

Definition 22 (Abstract States) *We call a set of abstract objects* abstract state *and denote by AbsStates the powerset over abstract states, i.e.:*

$$\mathit{AbsStates} \stackrel{\text{def}}{=} 2^{\mathit{AbsObjs}} \ .$$

We extend the partial order $\dot{\sqsubseteq}$ on sets of abstract objects to a preorder $\sqsubseteq$ on sets of abstract states as follows: let $S_1^\#, S_2^\#$ be sets of abstract states then

$$S_1^\# \sqsubseteq S_2^\# \stackrel{\text{def}}{\iff} \forall s_1^\# \in S_1^\#.\, \exists s_2^\# \in S_2^\#.\, s_1^\# \mathbin{\dot{\sqsubseteq}} s_2^\# \ .$$

Again, we identify elements in *AbsStates* with their equivalence classes in the quotient $(\mathit{AbsStates}/(\sqsubseteq \cap \sqsubseteq^{-1}))$ which gives us a partial order $\sqsubseteq$ on sets of abstract states. The partial order $\sqsubseteq$ again induces join $\sqcup$ and meet $\sqcap$ operations on sets of abstract states:

$$S_1^\# \sqcup S_2^\# \stackrel{\text{def}}{=} \mathsf{expand}(S_1^\#) \cup \mathsf{expand}(S_2^\#)$$
$$S_1^\# \sqcap S_2^\# \stackrel{\text{def}}{=} \mathsf{expand}(S_1^\#) \cap \mathsf{expand}(S_2^\#) \ .$$

Proposition 23 *Sets of abstract states AbsStates form a complete lattice with partial order $\sqsubseteq$, join $\sqcup$, meet $\sqcap$, least element $\emptyset$, and greatest element $\{\{\lambda p.\{0,1\}\}\}$.*

3.3.2 Concretization

We now define functions that assign meaning to elements in the abstract domains by mapping them to elements in the concrete domains.

A set of abstract objects denotes a Boolean combination of domain predicates in $\mathcal{P}$. Formally, we define a function $\dot{\gamma}$ that maps sets of abstract objects to domain predicates as follows:

$$\dot{\gamma} \in \mathit{AbsObjs} \to \mathit{DomPreds}(\mathsf{dom}(\mathcal{P}))$$

$$\dot{\gamma}(O^\#) \stackrel{\text{def}}{=} \dot{\bigcup_{o^\# \in O^\#}} \dot{\bigcap_{p \in \mathcal{P}}} p^{o^\#(p)}$$

$$\text{where} \quad p^i = \begin{cases} p & \text{if } i = \{1\} \\ \lambda \vec{o} \in \mathsf{dom}(\mathcal{P}).\, \overline{p(\vec{o})} & \text{if } i = \{0\} \\ \lambda \vec{o} \in \mathsf{dom}(\mathcal{P}).\, \mathit{States} & \text{otherwise} \end{cases} .$$

For the concretization of a single abstract object $o^\#$ we will write $\dot{\gamma}(o^\#)$ instead of $\dot{\gamma}(\{o^\#\})$.

A concrete state s is represeted by an abstract state $s^\#$ if every tuple of domain objects in s is represented by some abstract object in $s^\#$. Formally, the meaning function γ that maps sets of abstract states to sets of states is defined in terms of $\dot{\gamma}$ as follows:

$$\gamma \in \mathit{AbsStates} \to 2^{\mathit{States}}$$

$$\gamma(S^\#) \stackrel{\text{def}}{=} \bigcup_{s^\# \in S^\#} \bigcap_{\vec{o} \in \mathsf{dom}(\mathcal{P})} \dot{\gamma}(s^\#)(\vec{o}) \ .$$

Again, for the concretization of a single abstract state $s^\#$ we will write $\gamma(s^\#)$ instead of $\gamma(\{s^\#\})$.

Proposition 24 *The following properties hold:*

1. *$\dot{\gamma}$ is a complete meet morphism between sets of abstract objects and domain predicates*
2. *$\dot{\gamma}$ is a complete join morphism between sets of abstract objects and domain predicates*
3. *γ is a complete meet morphism between sets of abstract states and sets of states*
4. *γ is a complete join morphism between sets of abstract states and sets of states.*

Proof. Properties 2 and 4 immediately follow from the definition of $\dot{\gamma}$, respectively, γ. For proving property 1, let $\mathcal{O}^\# \subseteq \mathit{AbsObjs}$. Let further I be an index set for the elements in $\mathcal{O}^\#$, i.e., $\mathcal{O}^\# = \{ O_i^\# \mid i \in I \}$. In addition, let for each $i \in I$, J_i be an index set for the elements in $O_i^\# \in \mathcal{O}^\#$, i.e., for all $i \in I$, $O_i^\# = \{ o_{i,j}^\# \mid j \in J_i \}$. Finally, let F be the set of all functions mapping an index $i \in I$ to some index $j \in J_i$. Now, define for $f \in F$:

$$o_f^\# \stackrel{\text{def}}{=} \lambda p \in \mathcal{P}. \bigcap_{i \in I} o_{i,f(i)}^\#(p)$$

$$O_f^\# \stackrel{\text{def}}{=} \begin{cases} \{o_f^\#\} & \text{if for all } p \in \mathcal{P},\, o_f^\#(p) \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

Then we can characterize the greatest lower bound of $\mathcal{O}^\#$ as follows:

$$\dot{\bigsqcap}\mathcal{O}^\# = \dot{\bigsqcup_{f\in F}} O_f^\# \ .$$

We then have:

$$\begin{aligned}
\dot\gamma(\dot{\bigsqcap}\mathcal{O}^\#) &= \dot\gamma(\dot{\bigsqcup_{f\in F}} O_f^\#) \ = \ \dot{\bigcup_{f\in F}} \dot\gamma(O_f^\#) && \text{(by Property 2)}\\
&= \dot{\bigcup_{f\in F}} \begin{cases} \dot\bigcap_{p\in\mathcal{P}} p^{o_f^\#(p)} & \text{if } O_f^\# \neq \emptyset \\ \lambda\vec{o}\in \mathsf{dom}(\mathcal{P}).\,\emptyset & \text{otherwise} \end{cases}\\
&= \dot{\bigcup_{f\in F}} \dot{\bigcap_{p\in\mathcal{P}}} \dot{\bigcap_{i\in I}} p^{o^\#_{i,f(i)}(p)} \ = \ \dot{\bigcup_{f\in F}} \dot{\bigcap_{i\in I}} \dot{\bigcap_{p\in\mathcal{P}}} p^{o^\#_{i,f(i)}(p)}\\
&= \dot{\bigcap_{i\in I}} \dot{\bigcup_{j\in J_i}} \dot{\bigcap_{p\in\mathcal{P}}} p^{o^\#_{i,j}(p)} && \text{(by Proposition 17)}\\
&= \dot{\bigcap_{i\in I}} \dot\gamma(O_i^\#) \ = \ \dot\bigcap\Big\{ \dot\gamma(O^\#) \mid O^\# \in \mathcal{O}^\# \Big\}
\end{aligned}$$

Furthermore, we have:

$$\begin{aligned}
\dot\gamma(\{\lambda p\in\mathcal{P}.\ \{0,1\}\}) \ &= \ \dot{\bigcap_{p\in\mathcal{P}}} p^{\{0,1\}} \ = \ \dot{\bigcap_{p\in\mathcal{P}}} (\lambda\vec{o}\in\mathsf{dom}(\mathcal{P}).\ \mathit{States})\\
&= \ \lambda\vec{o}\in\mathsf{dom}(\mathcal{P}).\ \mathit{States} \ .
\end{aligned}$$

Thus, $\dot\gamma$ is a complete meet morphism.

The proof of Property 3 is similar. It uses Property 1, Property 4, and the fact that sets of states form a completely distributive lattice. ∎

3.3.3 Abstraction

By Proposition 24 functions $\dot\gamma$ and γ are complete meet morphisms on there respective domains. This implies that they are the upper adjoints of Galois connections between domain predicates and sets of abstract objects, respectively, between sets of states and sets of abstract states. The lower adjoints of these Galois connections define the most precise over-approximation of a given set of states (domain predicate) in terms of abstract states (abstract objects). Formally, we define abstraction functions $\dot\alpha^+$ and α^+ by these lower adjoints as follows:

$$\begin{aligned}
\dot\alpha^+ \ &\in \ \mathit{DomPreds}(\mathsf{dom}(\mathcal{P})) \to \mathit{AbsObjs}\\
\dot\alpha^+(p) &\stackrel{\text{def}}{=} \dot\bigcap\Big\{ O^\# \in \mathit{AbsObjs} \mid p \mathrel{\dot\subseteq} \dot\gamma(O^\#) \Big\}\\
\alpha^+ \ &\in \ 2^{\mathit{States}} \to \mathit{AbsStates}\\
\alpha^+(S) &\stackrel{\text{def}}{=} \bigsqcap\Big\{ S^\# \in \mathit{AbsStates} \mid S \sqsubseteq \gamma(S^\#) \Big\} \ .
\end{aligned}$$

Since $\dot{\gamma}$ and γ are also complete join morphisms, they also form the lower adjoints of Galois connections. The corresponding upper adjoints define the most precise under-approximations of sets of states, respectively, domain predicates. These under-approximating abstraction functions $\dot{\alpha}^-$ and α^- are defined as follows:

$$\begin{aligned}
\dot{\alpha}^- &\in \mathit{DomPreds}(\mathsf{dom}(\mathcal{P})) \to \mathit{AbsObjs} \\
\dot{\alpha}^-(p) &\stackrel{\text{def}}{=} \dot{\bigsqcup} \left\{ O^\# \in \mathit{AbsObjs} \mid \dot{\gamma}(O^\#) \mathrel{\dot{\subseteq}} p \right\} \\
\alpha^- &\in 2^{\mathit{States}} \to \mathit{AbsStates} \\
\alpha^-(S) &\stackrel{\text{def}}{=} \bigsqcup \left\{ S^\# \in \mathit{AbsStates} \mid \gamma(S^\#) \sqsubseteq S \right\} \quad .
\end{aligned}$$

Proposition 25 *The following properties hold:*

1. $(\dot{\alpha}^+, \dot{\gamma})$ *and* $(\dot{\gamma}, \dot{\alpha}^-)$ *form Galois connections between domain predicates and sets of abstract objects.*

2. (α^+, γ) *and* (γ, α^-) *form Galois connections between sets of states and sets of abstract states.*

Proof. The statement follows from Proposition 12, Proposition 24, and the definitions of the abstraction functions. ∎

It is instructive to give a more intuitive characterization of the abstraction functions. If we consider a concrete state s then the abstraction function α^+ maps the singleton $\{s\}$ to the smallest abstract state that contains state s. This abstract state describes the Boolean covering of domain objects $\vec{o} \in \mathsf{dom}(\mathcal{P})$ with respect to the domain predicates in $\mathcal{P}$. In order to describe these smallest Boolean coverings, we assign an abstract object $\dot{\alpha}^+(s, \vec{o})$ to every tuple $\vec{o} \in \mathsf{dom}(\mathcal{P})$ and state s. This abstract object represents the equivalence class of all tuples of objects that satisfy the same domain predicates as $\vec{o}$ in s:

$$\begin{aligned}
\dot{\alpha}^+(s, \vec{o}) &\stackrel{\text{def}}{=} \dot{\alpha}^+(\lambda \vec{o}_0 \in \mathsf{dom}(\mathcal{P}).\, \mathsf{if}\ \vec{o} = \vec{o}_0\ \mathsf{then}\ \{s\}\ \mathsf{else}\ \emptyset) \\
&= \{\lambda p \in \mathcal{P}.\, \mathsf{if}\ s \in p(\vec{o})\ \mathsf{then}\ \{1\}\ \mathsf{else}\ \{0\}\} \quad .
\end{aligned}$$

The smallest abstract state that contains s consists of all equivalence classes $\dot{\alpha}^+(s, \vec{o})$ for $\vec{o} \in \mathsf{dom}(\mathcal{P})$. Figure 3.3 visualizes this fact. Formally, the abstraction of a set of states S is characterized by the following proposition.

Proposition 26 *Let S be a set of states. Then $\alpha(S)$ is characterized as follows:*

$$\alpha^+(S) = \bigsqcup_{s \in S} \left\{ \dot{\bigsqcup_{\vec{o} \in \mathsf{dom}(\mathcal{P})}} \dot{\alpha}^+(s, \vec{o}) \right\} \quad .$$

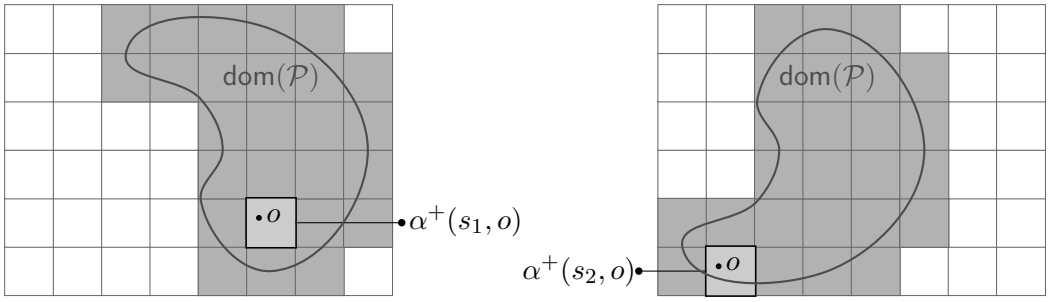


Figure 3.3: The abstract states for two states s_1 and s_2. The same object $o \in \mathsf{dom}(\mathcal{P})$ falls into different equivalence classes $\dot{\alpha}^+(s_1, o)$ and $\dot{\alpha}^+(s_2, o)$ for each of the states s_1 and s_2. This leads to a different Boolean covering of the set $\mathsf{dom}(\mathcal{P})$ in the two states and hence to different abstract states.

3.3.4 Symbolic Representation of Abstract States

We can symbolically represent the concretization of abstract states in terms of closed formulae. Assume that abstraction predicates in $\mathcal{P}$ have arity n and assume that each abstraction predicate $p \in \mathcal{P}$ is given by the denotation of a domain formula F_p. For a variable vector $\vec{v} = (v_1, \ldots, v_n)$ we write $F(\vec{v})$ for the formula that is obtained by applying domain formulae F to the variables $v_1, \ldots, v_n$:

$$(\ldots (F\, v_1) \ldots)\, v_n \ .$$

The concretization of a set of abstract states is the denotation of a disjunction of universally quantified Boolean combinations of domain formulae:

$$\gamma(S^\#) = [\![\bigvee_{s^\# \in S^\#} \forall \vec{v}. \bigvee_{o^\# \in s^\#} \bigwedge_{p \in \mathcal{P}} F_p^{o^\#(p)}(\vec{v})]\!]$$

$$\text{where } F^i = \begin{cases} F & \text{if } i = \{1\} \\ \lambda \vec{v}. \neg F(\vec{v}) & \text{if } i = \{0\} \\ \lambda \vec{v}. \mathsf{true} & \text{if } i = \{0, 1\} \end{cases} .$$

Consequently, an abstract interpretation based on domain predicate abstraction can be used to infer invariants that express universally quantified properties over domain objects.

Example 27 We now give a concrete example of an abstract state and its concretization. Assume a heap program that manipulates objects with a single reference field *next*. Given a program variable x in such a heap program, we define two kinds of domain predicates in terms of the following domain formulae:

$$p(x) \stackrel{\text{def}}{=} \lambda v.\, x = v$$
$$r(x) \stackrel{\text{def}}{=} \lambda v.\, next^*\, x\, v \ .$$

Domain formula $p(x)$ denotes the singleton object pointed to by program variable x, while domain formula $r(x)$ denotes the set of all objects that are reachable from x by following *next* fields in the heap. We will use these domain formulae throughout the next examples. Thereby, we will take the notational liberty and identify domain formulae with the domain predicates that they denote.

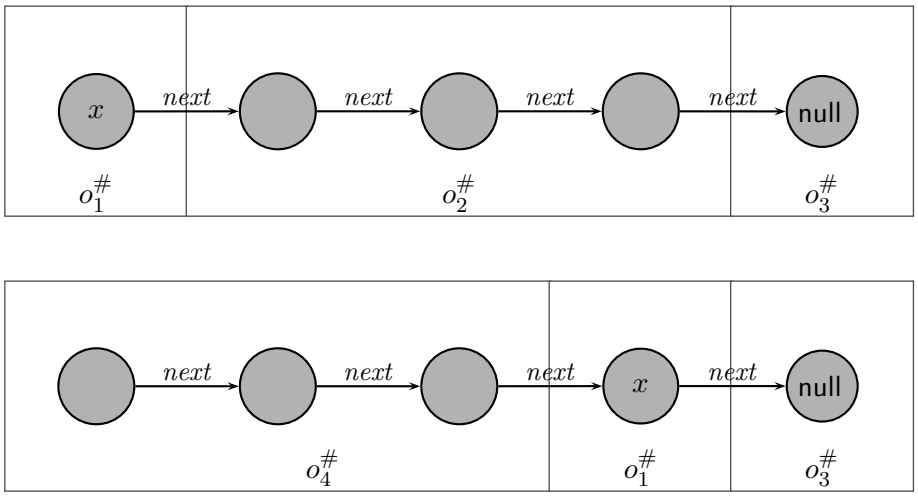


Figure 3.4: Two concrete program states represented by Boolean heap $\{o_1^\#, o_2^\#, o_3^\#, o_4^\#\}$ in Example 27.

Assume that the set of abstraction predicates is given by $\mathcal{P} = \{p(x), p(\mathsf{null}), r(x)\}$. Now consider the following 4 abstract objects over $\mathcal{P}$:

$$\begin{aligned} o_1^\# &= [p(x) \mapsto \{1\}, p(\mathsf{null}) \mapsto \{0\}, r(x) \mapsto \{1\}] \\ o_2^\# &= [p(x) \mapsto \{0\}, p(\mathsf{null}) \mapsto \{0\}, r(x) \mapsto \{1\}] \\ o_3^\# &= [p(x) \mapsto \{0\}, p(\mathsf{null}) \mapsto \{1\}, r(x) \mapsto \{1\}] \\ o_4^\# &= [p(x) \mapsto \{0\}, p(\mathsf{null}) \mapsto \{0\}, r(x) \mapsto \{0\}] \end{aligned}$$

The abstract state $s^\# = \{o_1^\#, o_2^\#, o_3^\#, o_4^\#\}$ represents all states that contain at least one non-empty, null-terminated list that is pointed to by program variable x. Figure 3.4 shows two concrete states that are in the concretization $\gamma(\{s^\#\})$ of abstract state $s^\#$. The framed boxes indicate the equivalence classes of heap objects that are described by the abstract objects in $s^\#$. Note that the concretization function γ does not enforce that these equivalence classes are non-empty, i.e., the first state is also represented by the Boolean heap $\{o_1^\#, o_2^\#, o_3^\#\}$ and the second state by the Boolean heap $\{o_1^\#, o_3^\#, o_4^\#\}$.

♦

3.4 Abstract Post Operator

We now formally characterize the abstract post operator on sets of abstract states that corresponds to the post operator of a Boolean heap program. We characterize the abstract

post operator associated with individual commands rather than the whole program. The extension from commands to programs is straightforward. In the following, we fix a command c and consider all applications of predicate transformers with respect to this particular command.

According to [38] for a Galois connection with upper and lower adjoints (α^+, γ) that connects the concrete and abstract domain, the best abstract post operator $\mathsf{post}^\#$ that is an abstraction of the concrete post operator post is given by the composition of α^+, post and γ. Thus, for domain predicate abstraction the image of a set of abstract states $S^\#$ under $\mathsf{post}^\#$ is given by:

$$\begin{aligned}\mathsf{post}^\#(S^\#) &\stackrel{\text{def}}{=} \alpha^+ \circ \mathsf{post} \circ \gamma(S^\#) \\ &= \bigsqcup_{s' \in \mathsf{post}(\gamma S^\#)} \alpha(s') \ .\end{aligned}$$

Since the concretization of a set of abstract states is in general a set of infinite cardinality, we can only compute $\mathsf{post}^\#(S^\#)$ indirectly. If we represent the concretization of abstract states symbolically in terms of formulae then we can check for each abstract state over abstraction predicates $\mathcal{P}$ whether it is contained in $\mathsf{post}^\#(S^\#)$. This check can be encoded into a decision procedure query. Assuming that n is the number of domain predicates in $\mathcal{P}$, considering all 2^{2^n} abstracts states explicitly results in a doubly-exponential number of decision procedure calls for the computation of $\mathsf{post}^\#(S^\#)$. In the following, we develop an approximation of the best abstract post operator $\mathsf{post}^\#$ that in theory requires worst-case exponentially many decision procedure calls and in practice can be implemented using only polynomially many calls to a decision procedure. We formally characterize this abstract post operator in terms of an abstraction of post by composition of $\mathsf{post}^\#$ with upper closure operators.

3.4.1 Context-sensitive Abstraction

Note that the best abstract post operator distributes over joins in the abstract domain, i.e., we can compute $\mathsf{post}^\#(S^\#)$ by computing the join of $\mathsf{post}^\#(\{s^\#\})$ for all abstract states $s^\# \in S^\#$. We therefore characterize the abstraction of $\mathsf{post}^\#$ on abstract states rather than sets of abstract states. As illustrated in Fig. 3.5, the problem is that even if we apply $\mathsf{post}^\#$ to a single abstract state $s^\#$, its image under $\mathsf{post}^\#$ will in general be a set of abstract states. Our first approximation is to abstract the resulting set of abstract states by a single abstract state. We can think of this abstraction as merging all Boolean coverings of domain objects represented by abstract states in $\mathsf{post}^\#(\{s^\#\})$ into a single one, or in other words, by pushing the universal quantifiers in the concretization of $\mathsf{post}^\#(\{s^\#\})$ over the outer disjunction. The resulting single abstract state represents a covering of all domain objects for all states that are represented by $\mathsf{post}^\#(\{s^\#\})$. Technically this abstraction is accomplished by restricting the best abstract post operator to singleton sets of abstract states. We denote by *AbsState* the set of all singletons of abstract states. If we restrict the order $\sqsubseteq$ to *AbsState* then we obtain again a complete lattice. We use the same symbols for join and meet operations on *AbsState* that we use for the lattice *AbsStates*. It will always be clear from the context which lattice we are referring to. We write $\alpha^+|_{AbsState}$ for the

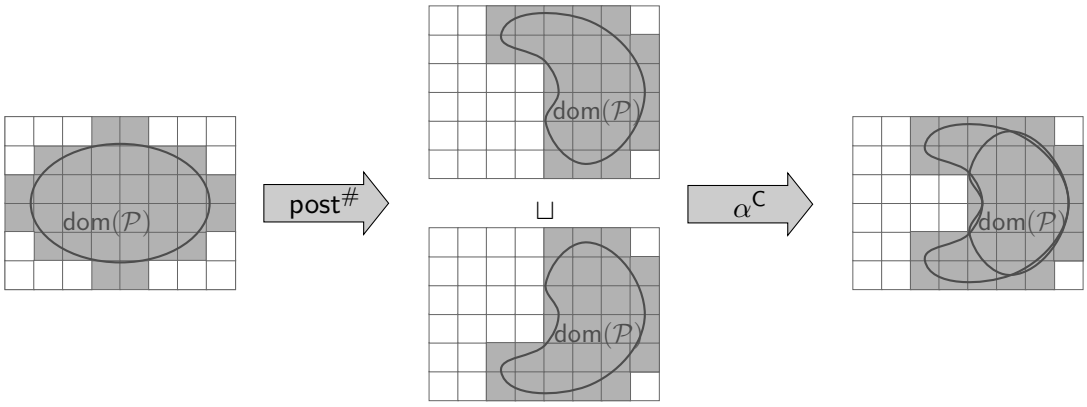


Figure 3.5: Application of $\mathsf{post}^\#$ to a single abstract state $s^\#$ and the abstraction of the resulting set of abstract states by restricting $\mathsf{post}^\#$ to singleton sets of abstract states.

abstraction function α^+ restricted to singletons of abstract states. Our first abstraction of $\mathsf{post}^\#$ is then described by the following abstract post operator:

$$\begin{aligned}\mathsf{post}^\#|_{AbsState} &\in AbsState \to AbsState\\ \mathsf{post}^\#|_{AbsState} &\stackrel{\text{def}}{=} \alpha^+|_{AbsState} \circ \mathsf{post} \circ \gamma \ .\end{aligned}$$

We can think of the operator $\mathsf{post}^\#|_{AbsState}$ as the abstraction of post under a new Galois connection. This new Galois connection is obtained by composing the Galois connection (α^+, γ) with a Galois connection $(\alpha^\mathsf{C}, \gamma^\mathsf{C})$ that connects *AbsStates* and *AbsState*:

$$\begin{aligned}\alpha^\mathsf{C} &\in AbsStates \to AbsState\\ \alpha^\mathsf{C} &\stackrel{\text{def}}{=} \lambda S^\#. \bigsqcap \left\{ \{s^\#\} \in AbsState \mid S^\# \sqsubseteq \{s^\#\} \right\}\\ \gamma^\mathsf{C} &\in AbsState \to AbsStates\\ \gamma^\mathsf{C} &\stackrel{\text{def}}{=} \mathsf{id} \ .\end{aligned}$$

Then we get the following characterization of operator $\mathsf{post}^\#|_{AbsState}$:

$$\mathsf{post}^\#|_{AbsState} = \alpha^\mathsf{C} \circ \mathsf{post}^\# \circ \gamma^\mathsf{C} \ .$$

It follows that $\mathsf{post}^\#|_{AbsState}$ is an abstraction of $\mathsf{post}^\#$.

Proposition 28 *The abstract post operator* $\mathsf{post}^\#|_{AbsState}$ *is an abstraction of the best abstract post operator* $\mathsf{post}^\#$*, formally for all abstract states* $s^\#$*:*

$$\mathsf{post}^\#(\{s^\#\}) \sqsubseteq \mathsf{post}^\#|_{AbsState}(\{s^\#\}) \ .$$

The operator $\mathsf{post}^\#|_{AbsState}$ computes the new Boolean covering for all domain objects in the post states of states represented by an abstract state $s^\#$. We can therefore characterize this operator in terms of an abstract domain predicate transformer. For this purpose, we

may define the best abstractions of the strongest domain postcondition $\dot{\mathsf{post}}$ with respect to the given abstraction predicates as follows:

$$\begin{aligned}\dot{\mathsf{post}}^{\#} &\in \mathit{AbsObjs} \to \mathit{AbsObjs}\\ \dot{\mathsf{post}}^{\#} &\stackrel{\text{def}}{=} \dot{\alpha}^{+} \circ \dot{\mathsf{post}} \circ \dot{\gamma}\end{aligned}$$

Given a set of abstract objects $O^{\#}$ the operator $\dot{\mathsf{post}}^{\#}$ computes the smallest covering of objects represented by $O^{\#}$ in arbitrary post states under the given command. Can we compute $\mathsf{post}^{\#}|_{\mathit{AbsState}}(\{s^{\#}\})$ by applying operator $\dot{\mathsf{post}}^{\#}$ to the set of abstract objects given by $s^{\#}$? Unfortunately, operator $\dot{\mathsf{post}}^{\#}$ is not quite sufficient in order to characterize $\mathsf{post}^{\#}|_{\mathit{AbsState}}(\{s^{\#}\})$ precisely: we are not interested in Boolean coverings of domain objects in arbitrary post states, but in Boolean coverings of domain objects in post states of states that are represented by $\{s^{\#}\}$. We need to take into account the *context* and restrict the operator $\dot{\mathsf{post}}$ to states that are represented by $\{s^{\#}\}$. For this purpose we introduce a family of *context-sensitive domain predicate transformers.*

Definition 29 (Context-sensitive Domain Predicate Transformers) *Let S be a set of states. The* context-sensitive domain predicate transformers *with respect to S are defined as follows:*

$$\begin{aligned}\dot{\mathsf{post}}_S &\in \mathit{DomPreds}(\mathsf{dom}(\mathcal{P})) \to \mathit{DomPreds}(\mathsf{dom}(\mathcal{P}))\\ \dot{\mathsf{post}}_S(p) &\stackrel{\text{def}}{=} \lambda\vec{o} \in \mathsf{dom}(\mathcal{P}).\, \mathsf{post}(S \cap p(\vec{o}))\\ \dot{\mathsf{wlp}}_S &\in \mathit{DomPreds}(\mathsf{dom}(\mathcal{P})) \to \mathit{DomPreds}(\mathsf{dom}(\mathcal{P}))\\ \dot{\mathsf{wlp}}_S(p) &\stackrel{\text{def}}{=} \lambda\vec{o} \in \mathsf{dom}(\mathcal{P}).\, \overline{S} \cup \mathsf{wlp}(p(\vec{o}))\ .\end{aligned}$$

Proposition 30 *Let S be a set of states. The context-sensitive domain predicate transformers $\dot{\mathsf{post}}_S$ and $\dot{\mathsf{wlp}}_S$ form a Galois connection on domain predicates, i.e., for any domain predicates p and q:*

$$\dot{\mathsf{post}}_S(p) \;\dot{\subseteq}\; q \quad\Longleftrightarrow\quad p \;\dot{\subseteq}\; \dot{\mathsf{wlp}}_S(q)\ .$$

Proof. We have for all sets of states S and domain predicates p and q:

$$\begin{aligned}&\dot{\mathsf{post}}_S(p) \;\dot{\subseteq}\; q\\ \Longleftrightarrow\;& (\lambda\vec{o} \in \mathsf{dom}(\mathcal{P}).\, \mathsf{post}(S \cap p(\vec{o}))) \;\dot{\subseteq}\; q\\ \Longleftrightarrow\;& \forall\vec{o} \in \mathsf{dom}(\mathcal{P}).\, \mathsf{post}(S \cap p(\vec{o})) \;\subseteq\; q(\vec{o})\\ \Longleftrightarrow\;& \forall\vec{o} \in \mathsf{dom}(\mathcal{P}).\, S \cap p(\vec{o}) \;\subseteq\; \mathsf{wlp}(q(\vec{o}))\\ \Longleftrightarrow\;& \forall\vec{o} \in \mathsf{dom}(\mathcal{P}).\, p(\vec{o}) \;\subseteq\; \overline{S} \cup \mathsf{wlp}(q(\vec{o}))\\ \Longleftrightarrow\;& p \;\dot{\subseteq}\; \lambda\vec{o} \in \mathsf{dom}(\mathcal{P}).\, \overline{S} \cup \mathsf{wlp}(q(\vec{o}))\\ \Longleftrightarrow\;& p \;\dot{\subseteq}\; \dot{\mathsf{wlp}}_S(q)\ .\end{aligned}$$

■

The context-sensitive domain predicate transformers allow us to compute the abstract post on abstract states by computing an abstract post on abstract objects. The abstract

post on abstract objects takes into account the context of the given abstract state. We would like to control how much context information is taken into account. For this purpose we introduce an additional parameter to our analysis. This parameter allows us to adjust the trade-off between precision and efficiency of the analysis.

Definition 31 (Context Operator) *A* context operator κ *is a function mapping sets of abstract states to sets of states such that* κ *is monotone and extensive with respect to* γ*, i.e., for all sets of states* $S^\#$*,* $\gamma(S^\#) \quad \subseteq \quad \kappa(S^\#)$.

The most precise context operator is the concretization function γ. The least precise context operator is the trivial one that maps any set of abstract states $S^\#$ to the full set of states.

Given a context operator κ, we define the *context-sensitive abstract domain predicate transformers* as follows:

$$\begin{aligned} \dot{\mathsf{post}}^\#_\kappa &\in \mathit{AbsStates} \to \mathit{AbsObjs} \to \mathit{AbsObjs} \\ \dot{\mathsf{post}}^\#_\kappa(S^\#) &\stackrel{\text{def}}{=} \dot{\alpha}^+ \circ \dot{\mathsf{post}}_{\kappa(S^\#)} \circ \dot{\gamma} \\ \dot{\mathsf{wlp}}^\#_\kappa &\in \mathit{AbsStates} \to \mathit{AbsObjs} \to \mathit{AbsObjs} \\ \dot{\mathsf{wlp}}^\#_\kappa(S^\#) &\stackrel{\text{def}}{=} \dot{\alpha}^- \circ \dot{\mathsf{wlp}}_{\kappa(S^\#)} \circ \dot{\gamma} \quad . \end{aligned}$$

Note that the context-sensitive abstract domain predicate transformers are obtained by composition of Galois connections. Therefore they form themselves a Galois connection.

Proposition 32 *Let* $S^\#$ *be a set of abstract states, then the pair* $(\dot{\mathsf{post}}^\#_\kappa(S^\#), \dot{\mathsf{wlp}}^\#_\kappa(S^\#))$ *is a Galois connection on sets of abstract objects.*

Proof. By Proposition 25 the pairs $(\dot{\alpha}^+, \dot{\gamma})$ and $(\dot{\gamma}, \dot{\alpha}^-)$ are Galois connections between domain predicates and sets of abstract objects. Furthermore, by Proposition 30 the pair $(\mathsf{post}_{\kappa(s^\#)}, \mathsf{wlp}_{\kappa(s^\#)})$ is a Galois connection on domain predicates. Thus, by Proposition 13 the pair $(\dot{\alpha}^+ \circ \mathsf{post}_{\kappa(s^\#)}, \mathsf{wlp}_{\kappa(s^\#)} \circ \dot{\gamma})$ is a Galois connection between domain predicates and sets of abstract objects. Hence, again by Proposition 13 the pair $(\dot{\alpha}^+ \circ \mathsf{post}_{\kappa(s^\#)} \circ \dot{\gamma}, \dot{\alpha}^- \circ \mathsf{wlp}_{\kappa(s^\#)} \circ \dot{\gamma})$ is a Galois connection on sets of abstract objects. ∎

Now we can approximate the image of an abstract state $s^\#$ under abstract post operator $\mathsf{post}^\#|_{\mathit{AbsState}}$ by applying the context-sensitive domain post operator to the abstract objects represented by $s^\#$.

Proposition 33 *Let* $s^\#$ *be an abstract state and let* κ *be a context operator. Applying* $\dot{\mathsf{post}}^\#_\kappa$ *to* $s^\#$ *results in an abstraction of* $\mathsf{post}^\#|_{\mathit{AbsState}}(\{s^\#\})$*:*

$$\mathsf{post}^\#|_{\mathit{AbsState}}(\{s^\#\}) \sqsubseteq \{\dot{\mathsf{post}}^\#_\kappa(\{s^\#\})(s^\#)\} \quad .$$

Proof.

$$
\begin{aligned}
&\mathsf{post}^{\#}|_{AbsState}(\{s^{\#}\}) \\
&= \bigsqcap \left\{ \{s_1^{\#}\} \in AbsState \mid \mathsf{post}(\gamma(\{s^{\#}\})) \subseteq \gamma(\{s_1^{\#}\}) \right\} \\
&= \bigsqcap \left\{ \{s_1^{\#}\} \in AbsState \mid \gamma(\{s^{\#}\}) \subseteq \mathsf{wlp}(\gamma\{s_1^{\#}\}) \right\} && (1) \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \gamma(\{s^{\#}\}) \subseteq \mathsf{wlp}(\gamma\{O_1^{\#}\}) \right\} \right\} \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \gamma(\{s^{\#}\}) \subseteq \mathsf{wlp} \left(\bigcap_{\vec{o} \in \mathsf{dom}(\mathcal{P})} \dot{\gamma}(O_1^{\#})(\vec{o}) \right) \right\} \right\} \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \gamma(\{s^{\#}\}) \subseteq \bigcap_{\vec{o} \in \mathsf{dom}(\mathcal{P})} \mathsf{wlp}(\dot{\gamma}(O_1^{\#})(\vec{o})) \right\} \right\} && (2) \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \forall \vec{o} \in \mathsf{dom}(\mathcal{P}).\, \gamma(\{s^{\#}\}) \cap \dot{\gamma}(s^{\#})(\vec{o}) \subseteq \mathsf{wlp}(\dot{\gamma}(O_1^{\#})(\vec{o})) \right\} \right\} && (3) \\
&\sqsubseteq \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \forall \vec{o} \in \mathsf{dom}(\mathcal{P}).\, \kappa(\{s^{\#}\}) \cap \dot{\gamma}(s^{\#})(\vec{o}) \subseteq \mathsf{wlp}(\dot{\gamma}(O_1^{\#})(\vec{o})) \right\} \right\} && (4) \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \forall \vec{o} \in \mathsf{dom}(\mathcal{P}).\, \dot{\gamma}(s^{\#})(\vec{o}) \subseteq \overline{\kappa(\{s^{\#}\})} \cup \mathsf{wlp}(\dot{\gamma}(O_1^{\#})(\vec{o})) \right\} \right\} \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \dot{\gamma}(s^{\#}) \subseteq \dot{\mathsf{wlp}}_{\kappa(\{S^{\#}\})}(\dot{\gamma}(O_1^{\#})) \right\} \right\} \\
&= \left\{ \dot{\bigsqcap} \left\{ O_1^{\#} \in AbsObjs \mid \dot{\mathsf{post}}_{\kappa(\{s^{\#}\})}(\dot{\gamma}(s^{\#})) \subseteq \dot{\gamma}(O_1^{\#}) \right\} \right\} && (5) \\
&= \{\dot{\alpha}^{+} \circ \dot{\mathsf{post}}_{\kappa(\{s^{\#}\})} \circ \dot{\gamma}(s^{\#})\} \\
&= \{\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(s^{\#})\}
\end{aligned}
$$

(1) follows from Proposition 14.

(2) follows from Proposition 14 and Proposition 12, Statement 3.

(3) follows from the tautology:

$$(\forall \vec{v}.\, B(\vec{v})) \rightarrow \forall \vec{v}.\, A(\vec{v}) \;\equiv\; \forall \vec{w}.\, (\forall \vec{v}.\, B(\vec{v})) \wedge B(\vec{w}) \rightarrow A(\vec{w})$$

(4) follows from the fact that κ is extensive.

(5) follows from Proposition 30

■

The proof of Proposition 33 indicates that there is only one reason for potential loss of precision when one uses the context-sensitive abstract post to compute the abstract post on singletons of abstract states, namely, the choice of the context operator. If the context operator takes into account the full context in form of the abstract state for which the post is computed then the two sides of the set inclusion in Proposition 33 become equal.

Corollary 34 *The best abstract post operator* $\mathsf{post}^{\#}|_{AbsState}$ *on singleton sets of abstract states is characterized as follows:*

$$\mathsf{post}^{\#}|_{AbsState} = \lambda\{s^{\#}\}.\,\{\dot{\mathsf{post}}^{\#}_{\gamma}(\{s^{\#}\})(s^{\#})\} \ .$$

Since the operator $\dot{\mathsf{post}}^{\#}_{\kappa}$ is the upper adjoint of a Galois connection on sets of abstract objects, it distributes over joins. Thus, we have:

$$\begin{aligned}\mathsf{post}^{\#}|_{AbsState}(\{s^{\#}\}) \ &\sqsubseteq \{\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(s^{\#})\} \\ &= \left\{ \dot{\bigsqcup_{o^{\#}\in\mathsf{expand}(s^{\#})}} \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\}) \right\} \ .\end{aligned}$$

Consequently, we can construct $\mathsf{post}^{\#}|_{AbsState}(\{s^{\#}\})$ by mapping *locally* each abstract object $o^{\#}$ in $s^{\#}$ to the new Boolean covering $\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\})$ that represents all domain objects in $o^{\#}$ in the post states of $s^{\#}$. However, $\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\})$ will in general be a set of abstract objects. Essentially, we face the same problem as in the case of computing $\mathsf{post}^{\#}$: we would have to consider all 2^n abstract objects over abstraction predicates, in order to compute the precise image of a single abstract object under operator $\dot{\mathsf{post}}^{\#}_{\kappa}$. Therefore, we apply yet another abstraction.

3.4.2 Cartesian Abstraction

Analogously to the abstraction of $\mathsf{post}^{\#}$ that is obtained by restricting $\mathsf{post}^{\#}$ to singletons of abstract states, we abstract operator $\dot{\mathsf{post}}^{\#}_{\kappa}$ by restricting it to singletons of abstract objects. We denote by *AbsObj* the subset of *AbsObjs* that consists of singleton sets of abstract objects. As in the case of singletons of abstract states, *AbsObj* forms a complete lattice if we restrict the partial order $\dot{\sqsubseteq}$ appropriately. Again we overload the symbols for joins and meets on *AbsObjs* to joins and meets on *AbsObj*. Now we define a Galois connection $(\dot{\alpha}^{\mathsf{C}}, \dot{\gamma}^{\mathsf{C}})$ that connects *AbsObjs* and *AbsObj* as follows:

$$\begin{aligned}\dot{\alpha}^{\mathsf{C}} \ &\in \mathit{AbsObjs} \to \mathit{AbsObjs} \\ \dot{\alpha}^{\mathsf{C}} \ &\stackrel{\text{def}}{=} \lambda O^{\#}.\, \dot{\bigsqcap} \left\{ \{o^{\#}\} \in \mathit{AbsObj} \mid O^{\#} \sqsubseteq \{o^{\#}\} \right\} \\ \dot{\gamma}^{\mathsf{C}} \ &\in \mathit{AbsObj} \to \mathit{AbsObjs} \\ \dot{\gamma}^{\mathsf{C}} \ &\stackrel{\text{def}}{=} \mathsf{id} \ .\end{aligned}$$

A more constructive characterization of abstraction function $\dot{\alpha}^{\mathsf{C}}$ is given as follows:

$$\begin{aligned}\dot{\alpha}^{\mathsf{C}}(O^{\#}) \ &= \left\{ \lambda p \in \mathcal{P}.\, O^{\#}(p) \right\} \\ \text{where } O^{\#}(p) \ &\stackrel{\text{def}}{=} \left\{ o^{\#}(p) \mid o^{\#} \in O^{\#} \right\} \ .\end{aligned}$$

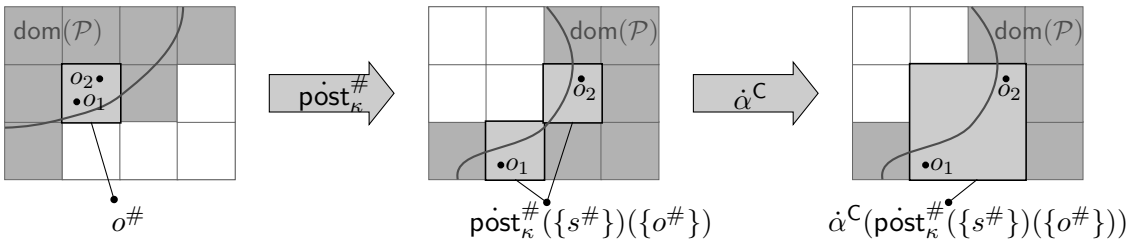


Figure 3.6: Application of $\dot{\mathsf{post}}^{\#}_{\kappa}$ to a single abstract object $o^{\#}$ and its approximationn under Cartesian abstraction.

Thus, a set of abstract objects $O^{\#}$ is mapped to a single abstract objects by projecting all abstract objects in $O^{\#}$ to the individual abstraction predicates. This abstraction principle is also known as Independant Attribute Abstraction or Cartesian Abstraction [8]. When applied to the abstract post operator the effect of Cartesian abstraction is that one can update each predicate in isolation.

Figure 3.6 sketches the effect of Cartesian abstraction in our context. It abstracts all abstract objects in the image under the operator $\dot{\mathsf{post}}^{\#}_{K}$ by a single abstract object. Composing the operator $\dot{\mathsf{post}}^{\#}_{K}$ with Cartesian abstraction gives us our final abstraction of the best abstract post operator.

Definition 35 (Context-sensitive Cartesian Post) *Let κ be a context operator. The* context-sensitive Cartesian post *for κ is defined as follows:*

$$\mathsf{post}^{\mathsf{C}}_{\kappa} \in \mathit{AbsStates} \to \mathit{AbsStates}$$

$$\mathsf{post}^{\mathsf{C}}_{\kappa}(S^{\#}) \stackrel{\text{def}}{=} \bigsqcup_{s^{\#} \in S^{\#}} \left\{ \dot{\bigsqcup_{o^{\#} \in \mathsf{expand}(s^{\#})}} \dot{\alpha}^{\mathsf{C}} \circ \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\}) \circ \dot{\gamma}^{\mathsf{C}}(\{o^{\#}\}) \right\} \ .$$

There is a small technical inconvenience that is caused by Cartesian abstraction. Note that $\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(s^{\#})$ will always be a nonempty set of abstract objects, as long as $s^{\#}$ represents some concrete state that has a successor under the given command. This is due to the fact that all domains of logical structures are guaranteed to be nonempty. However, if the given command is, e.g., an assume command and none of the states represented by $s^{\#}$ satisfy the guard then no concrete post state exists. In this case $\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(s^{\#})$ may become empty. Unfortunately, there is no abstract object that represents the domain predicate $(\lambda \vec{o} \in \mathsf{dom}(\mathcal{P}).\, \emptyset)$ which corresponds to the denotation of the empty set of abstract objects, unless there are contradictory predicates in $\mathcal{P}$. Therefore, whenever $\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(s^{\#})$ becomes empty then Cartesian abstraction may lose precision. As we will show in Chapter 4, one can explicitly account for this situation in order to avoid this loss of precision.

The following theorem states soundness of the context-sensitive Cartesian post operator.

Theorem 36 (Soundness of Cartesian Post) *The context-sensitive Cartesian post is an abstraction of* $\mathsf{post}^{\#}$. *Formally, let κ be a context operator. Then For all sets of abstract states $S^{\#}$ we have*

$$\mathsf{post}^{\#}(S^{\#}) \sqsubseteq \mathsf{post}^{\mathsf{C}}_{\kappa}(S^{\#}) \ .$$

Proof. The statement follows immediately from Proposition 28, Proposition 33, and the definition of $\dot{\alpha}^{\mathsf{C}}$. ∎

We can now use abstract weakest domain preconditions to characterize the context-sensitive Cartesian post in terms of independent updates of individual predicates in abstract objects. This shows that the context-sensitive Cartesian post operator for a given command corresponds to a Boolean heap program; cf. Fig. 3.2.

Theorem 37 (Characterization of Cartesian Post) *The context-sensitive Cartesian post for context operator κ is characterized as follows: for all sets of abstract states $S^{\#}$:*

$$\mathsf{post}^{\mathsf{C}}_{\kappa}(S^{\#}) = \bigsqcup_{s^{\#}\in S^{\#}} \left\{ \dot{\bigsqcup_{o^{\#}\in \mathsf{expand}(s^{\#})}} \left\{ \lambda p \in \mathcal{P}.\, upd(p, \{s^{\#}\}, \{o^{\#}\}) \right\} \right\}$$

$$\text{where} \quad upd(p, S^{\#}, O^{\#}) \stackrel{\text{def}}{=} \begin{cases} \{1\} & \text{if } O^{\#} \mathrel{\dot{\sqsubseteq}} \dot{\mathsf{wlp}}^{\#}_{\kappa}(S^{\#})(\{o^{\#}_{p,1}\}) \\ \{0\} & \text{if } O^{\#} \mathrel{\dot{\sqsubseteq}} \dot{\mathsf{wlp}}^{\#}_{\kappa}(S^{\#})(\{o^{\#}_{p,0}\}) \\ \{0,1\} & \text{otherwise} \end{cases}$$

$$o^{\#}_{p,i} \stackrel{\text{def}}{=} \lambda p' \in \mathcal{P}.\, \mathsf{if}\ p = p'\ \mathsf{then}\ \{i\}\ \mathsf{else}\ \{0,1\} \quad .$$

Proof. We need to show that the following equality holds for any abstract state $s^{\#}$ and abstract object $o^{\#} \in \mathsf{expand}(s^{\#})$:

$$\dot{\alpha}^{\mathsf{C}} \circ \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\}) \circ \dot{\gamma}^{\mathsf{C}}(\{o^{\#}\}) = \left\{ \lambda p \in \mathcal{P}.\, upd(p, \{s^{\#}\}, \{o^{\#}\}) \right\} \quad .$$

We have by definition of $\dot{\alpha}^{\mathsf{C}}$ and $\dot{\gamma}^{\mathsf{C}}$:

$$\begin{aligned} & \dot{\alpha}^{\mathsf{C}} \circ \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\}) \circ \dot{\gamma}^{\mathsf{C}}(\{o^{\#}\}) \\ = & \dot{\bigsqcap} \left\{ \{o^{\#}_1\} \in AbsObj \mid \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\}) \mathrel{\dot{\sqsubseteq}} \{o^{\#}_1\} \right\} \\ = & \left\{ \lambda p \in \mathcal{P}.\, \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\})(p) \right\} \quad . \end{aligned}$$

Now it is easy to see that the following equality holds:

$$\dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\})(p) = \begin{cases} \{1\} & \text{if } \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\}) \mathrel{\dot{\sqsubseteq}} \{o^{\#}_{p,1}\} \\ \{0\} & \text{if } \dot{\mathsf{post}}^{\#}_{\kappa}(\{s^{\#}\})(\{o^{\#}\}) \mathrel{\dot{\sqsubseteq}} \{o^{\#}_{p,0}\} \\ \{0,1\} & \text{otherwise} \quad . \end{cases}$$

Then the theorem follows from the fact that for any set of abstract states $S^{\#}$ we have that $\dot{\mathsf{post}}^{\#}_{\kappa}(S^{\#})$ and $\dot{\mathsf{wlp}}^{\#}_{\kappa}(S^{\#})$ from a Galois connection on sets of abstract objects. ∎

3.4.3 Symbolic Computation of Abstract Post

The characterization of the context-sensitive Cartesian post given in Theorem 37 focusses the abstraction of a concrete command to the computation of abstract weakest domain

preconditions of abstraction predicates. We can automate the computation of these abstract weakest domain preconditions using theorem provers. For instance, in order to compute $\dot{\mathsf{wlp}}_{\kappa}(S^{\#})(\{o^{\#}_{p,1}\})$, the best under-approximation of $\dot{\mathsf{wlp}}_{\kappa(S^{\#})}(p)$ with respect to abstraction predicates $\mathcal{P}$ and context $S^{\#}$, one needs to compute the largest set of abstract objects $O^{\#}$ that satisfies:

$$\forall \vec{o} \in \mathsf{dom}(\mathcal{P}).\ \dot{\gamma}(O^{\#})(\vec{o}) \subseteq \overline{\kappa(S^{\#})} \cup \dot{\mathsf{wlp}}(\dot{\gamma}(\{o^{\#}_{p,1}\}))(\vec{o}) \tag{3.1}$$

We can compute the set $O^{\#}$ by computing the union of all abstract objects that satisfy condition 3.1 and are minimal with respect to partial order $\dot{\sqsubseteq}$. Assume again that every abstraction predicate p is represented by a domain formula F_p. Further, assume that context operator κ maps abstract states to closed formulae and concretization function $\dot{\gamma}$ maps sets of abstract objects to domain formulae. In order to check whether some minimal abstract object $o^{\#}$ satisfies condition 3.1, we check validity of the entailment:

$$\kappa(S^{\#}) \wedge \dot{\gamma}(\{o^{\#}\})(\vec{v}) \models \dot{\mathsf{wlp}}(F_p)(\vec{v}) \tag{3.2}$$

Hereby $\dot{\gamma}(\{o^{\#}\})(\vec{v})$ is a complete conjunction of literals over the domain formulae F_p that represent abstraction predicates.

Complexity of Analysis. The theoretical worst-case complexity of the analysis is dominated by the maximal number of iterations for computing the least fixed point of the abstract post operator. This number is bounded by the height of the abstract domain, which is doubly-exponential in the number of abstraction predicates. However, in practice the running time is dominated by the number of decision procedure calls that are needed for computing abstract weakest domain preconditions. The number of decision procedure calls is worst-case exponential in the number of abstraction predicates. In practice, one can restrict the entailment checks of the form 3.2 to abstract objects that denote conjunctions of a fixed length rather than complete conjunctions. This gives a polynomial bound on the number of decision procedure calls. In Chapter 6, we describe an implementation of the context-sensitive Cartesian post and additional optimizations that further reduce the number of decision procedure calls.

Example 38 We want to conclude this section with a concrete example that illustrates the computation of the context-sensitive Cartesian post. Consider the following set of domain predicates:

$$\mathcal{P} = \{p(x), p(y), p(z), r(x), r(y), r(z)\}$$

and the following abstract objects over $\mathcal{P}$:

$$\begin{aligned}
o^{\#}_1 &= [p(x) \mapsto \{0\}, p(y) \mapsto \{0\}, p(z) \mapsto \{1\}, r(x) \mapsto \{0\}, r(y) \mapsto \{0\}, r(z) \mapsto \{1\}] \\
o^{\#}_2 &= [p(x) \mapsto \{0\}, p(y) \mapsto \{0\}, p(z) \mapsto \{0\}, r(x) \mapsto \{0\}, r(y) \mapsto \{0\}, r(z) \mapsto \{1\}] \\
o^{\#}_3 &= [p(x) \mapsto \{1\}, p(y) \mapsto \{0\}, p(z) \mapsto \{0\}, r(x) \mapsto \{1\}, r(y) \mapsto \{0\}, r(z) \mapsto \{1\}] \\
o^{\#}_4 &= [p(x) \mapsto \{0\}, p(y) \mapsto \{1\}, p(z) \mapsto \{0\}, r(x) \mapsto \{0\}, r(y) \mapsto \{1\}, r(z) \mapsto \{0\}] \\
o^{\#}_5 &= [p(x) \mapsto \{0\}, p(y) \mapsto \{0\}, p(z) \mapsto \{0\}, r(x) \mapsto \{1\}, r(y) \mapsto \{1\}, r(z) \mapsto \{1\}]
\end{aligned}$$

Figure 3.7 shows a concrete program state that is represented by abstract state

$$s^{\#} = \{o_1^{\#}, o_2^{\#}, o_3^{\#}, o_4^{\#}, o_5^{\#}\} \ .$$

We compute the abstract post for the singleton $\{s^{\#}\}$ and command $x.next{:=}y$ that updates the *next* field of object x to object y. The operator $\mathsf{post}_{\kappa}^{\mathsf{C}}$ maps abstract state $s^{\#}$ to a new abstract state

$$s^{\#\prime} = \{{o^{\#}}'_1, {o^{\#}}'_2, {o^{\#}}'_3, {o^{\#}}'_4, {o^{\#}}'_5\} \ .$$

Each abstract object $o_i^{\#} \in s^{\#}$ is updated to a new abstract object ${o^{\#}}'_i \in s^{\#\prime}$ by computing new values for each domain predicate in isolation. For instance, in order to determine whether $o_1^{\#}(r(z))$ should be set to $\{1\}$, the following entailment is checked for validity:

$$\gamma(s^{\#}) \wedge \gamma(o_1^{\#}) \models next[x \mapsto y]^* z\, v \ .$$

This entailment holds because $o_1^{\#}(r(z)) = \{1\}$ and $o_1^{\#}(r(x)) = \{0\}$. Thus, ${o_1^{\#}}'(r(z))$ is set to $\{1\}$. The resulting abstract objects in $s^{\#\prime}$ are as follows:

$$\begin{aligned}
{o_1^{\#}}' &= [p(x){\mapsto}\{0\}, p(y){\mapsto}\{0\}, p(z){\mapsto}\{1\}, r(x){\mapsto}\{0\}, r(y){\mapsto}\{0\}, r(z){\mapsto}\{1\}] \\
{o_2^{\#}}' &= [p(x){\mapsto}\{0\}, p(y){\mapsto}\{0\}, p(z){\mapsto}\{0\}, r(x){\mapsto}\{0\}, r(y){\mapsto}\{0\}, r(z){\mapsto}\{1\}] \\
{o_3^{\#}}' &= [p(x){\mapsto}\{1\}, p(y){\mapsto}\{0\}, p(z){\mapsto}\{0\}, r(x){\mapsto}\{1\}, r(y){\mapsto}\{0\}, r(z){\mapsto}\{1\}] \\
{o_4^{\#}}' &= [p(x){\mapsto}\{0\}, p(y){\mapsto}\{1\}, p(z){\mapsto}\{0\}, r(x){\mapsto}\{1\}, r(y){\mapsto}\{1\}, r(z){\mapsto}\{1\}] \\
{o_5^{\#}}' &= [p(x){\mapsto}\{0\}, p(y){\mapsto}\{0\}, p(z){\mapsto}\{0\}, r(x){\mapsto}\{1\}, r(y){\mapsto}\{1\}, r(z){\mapsto}\{1\}]
\end{aligned}$$

Figure 3.7 shows a concrete state that is represented by abstract state $s^{\#\prime}$.

This example also nicely demonstrates the importance of the context for the precision of predicate updates. For instance, consider the entailments that are checked for determining the value of $o_4^{\#}(r(z))$. The entailment

$$\gamma(s^{\#}) \wedge \dot{\gamma}(o_4^{\#}) \models next[x{\mapsto}y]^* z\, v$$

is valid because $\gamma(s^{\#})$ entails $next^*\, z\, x$ and $o_4^{\#}(r(x)){=}\{1\}$. However, the entailment

$$\dot{\gamma}(o_4^{\#}) \models next[x{\mapsto}y]^* z\, v$$

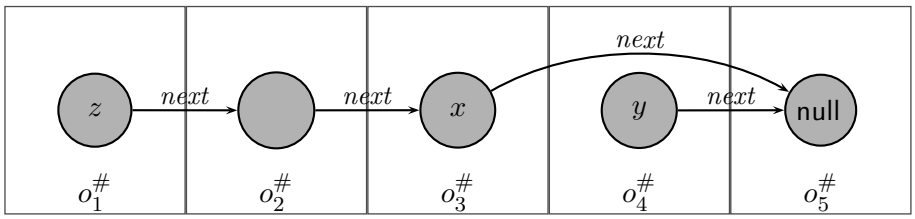
is not valid. If the abstract post would not take into account the context of abstract object $o_4^{\#}$ then ${o_4^{\#}}'(r(z))$ would be set to $\{0,1\}$, i.e., the analysis would lose precision.

♦

3.5 Further Related Work

We have shown that domain predicate abstraction generalizes predicate abstraction by using the key idea of three-valued shape analysis *à la* Sagiv, Reps, and Wilhelm [103]. In the following, we provide a more detailed comparison with these approaches and other shape analyses.

A state represented by abstract state $s^{\#} = \left\{o_1^{\#}, o_2^{\#}, o_3^{\#}, o_4^{\#}, o_5^{\#}\right\}$:



A state represented by abstract state $s^{\#\prime}$:

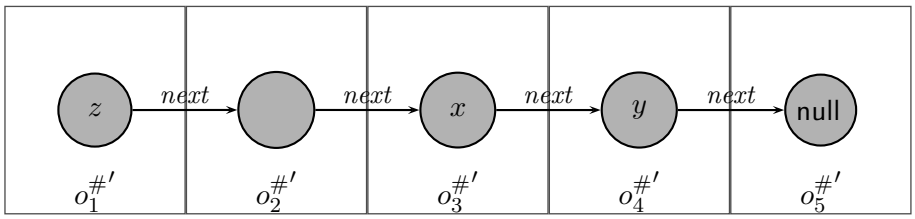


Figure 3.7: Application of the context-sensitive Cartesian post to the abstract state $\{o_1^{\#}, o_2^{\#}, o_3^{\#}, o_4^{\#}, o_5^{\#}\}$ in Example 38.

Three-valued Shape Analysis. In [103] Sagiv, Reps and Wilhelm describe a parametric framework to shape analysis based on three-valued logic. They abstract sets of states by three-valued logical structures. This *canonical abstraction* is defined in terms of equivalence classes of objects in the heap that are induced by a finite set of predicates on heap objects. Adapting three-valued shape analysis to the analysis of specific data structures requires the user to provide predicates and precomputed transfer functions for these predicates. Recent approaches enable the automatic computation of transfer functions [81,100] some of which are using decision procedures [118,120]. Domain predicate abstraction is inspired by three-valued shape analysis. In fact, there is a close connection between the abstract domain in [103] and ours: a translation from three-valued logical structures, as they arise under canonical abstraction, into formulae in first-order logic is given in [119]. Shape analysis constraints [69] characterizes this class in terms of a Boolean algebra of formulae that is isomorphic to the class of three-valued logical structures obtained under canonical abstraction; our abstract domain subsumes the universal fragment of shape analysis constraints.

Predicate Abstraction. Domain predicate abstraction is a proper generalization of predicate abstraction [49] that enables the inference of universally quantified invariants. Quantified invariants are required for the verification of quantified properties which naturally occur in programs with dynamic memory allocation. Our analysis incorporated ideas such as Cartesian abstraction [8] that have been previously applied in the context of predicate abstraction.

In [98, 110] we presented the special case of Boolean heap programs where the arity of domain predicates is restricted to one and their domain to heap objects. The gener-

alization presented in this thesis is interesting for two reasons. First, it allows abstract states in Boolean heap programs to quantify over relations between different objects in the heap. Such relations naturally occur in many applications, e.g., implementations of maps, instantiatable data structures, and concurrent data structures where an additional variable can be used to quantify over thread objects; see e.g. [16, 114]. Second, this generalization allows abstract states to quantify not just over heap objects, but over objects of arbitrary domains, e.g., integers. Therefore, domain predicate abstraction can be used for the verification of programs that are beyond the classical application domain of shape analysis such as programs manipulating arrays.

Among the main approaches for dealing with quantified invariants in predicate abstraction is the use of Skolem constants [45], indexed predicates [71], range predicates [60], and the use of abstraction predicates that contain quantifiers.

The key difficulty in using Skolem constants for shape analysis is that the properties of individual objects depend on the "context", given by the properties of surrounding objects. A typical example of such *non-local* properties are reachability properties. In order to automatically verify such properties it is not enough to use a fixed Skolem constant throughout the analysis; it is instead necessary to instantiate universal quantifiers from previous loop iterations, in some cases multiple times. Our analysis attempts to find a balance between these extremes: it computes the abstract post locally on abstract objects, but it still takes into account the context of surrounding objects.

Compared to indexed predicates [71] our abstract domain is more general because it contains disjunctions of universally quantified statements. The presence of disjunctions is not only more expressive in principle, but allows the analysis to keep formulae under the universal quantifiers more specific. This enables the use of less precise, but more efficient algorithms such as Cartesian abstraction for computing changes to properties of objects, without losing too much precision in the overall analysis. Disjunctions also play an important role in the context of counterexample-guided abstraction refinement; cf. Chapter 4.

Range predicates [60] are able to express quantified properties over arrays. In principle, range predicates could also be used for shape analysis. However, the technique only applies to linear data structures such as lists.

Predicate abstraction has also been used directly for shape analysis; see e.g. [3, 19, 39]. The advantage of using abstraction tailored to shape analysis compared to using global predicates is that the parameters to shape-analysis-oriented abstraction are properties of objects in a state, as opposed to global properties of a state, and the number of global predicates that is needed to emulate shape analysis domains is exponential in the number of properties [86].

In template-based techniques [17, 50] the user specifies templates for quantified invariants. The analysis finds an invariant by automatically instantiating the template parameters. While [17] is specific to the analysis of programs with arrays, [50] enables the lifting of a given abstract interpretation to an abstract interpretation over a quantified abstract domain. In particular, this technique has been used to infer quantified invariants for heap-manipulating programs.

Other Shape Analyses. Our symbolic shape analysis computes the abstract post locally on abstract objects rather than globally on the whole abstract state. The idea of local

reasoning has also been exploited by other shape analyses [15, 28, 31, 34, 42, 51, 83] some of which are based on separation logic [93, 94, 101]. These analyses take a less general approach than domain predicate abstraction; their abstract domains are tailored towards specific programs and properties such as memory safety of list-manipulating programs. However, these analysis can deliver impressive results for the programs they are designed for [29, 51, 115].

Some shape analysis are based on automata [22, 23, 25]. The most general automata-based approach so far is described in [25]. This method encodes heap programs into tree transducers. This encoding reduces shape analysis to abstract regular tree model checking [24]. The encoding into tree transducers loses precision if the structures observed in the heap program do not exhibit some regularity. It seems that the translation is precise for structures that are described by graph types [64] or some extension similar to what is captured by our field constraint analysis. We believe that our abstraction is conceptually simpler. It is defined in terms of a composition of the concrete post operator with upper closure operators. Thus, it is sound by construction and we know precisely where it loses precision. Also, since our abstract domain is parameterized, there is no intrinsic restriction to specific data structures.

3.6 Conclusion

In this chapter we proposed Domain Predicate Abstraction. Domain predicate abstraction generalizes predicate abstraction to the point where it becomes suitable for shape analysis. We showed how the abstraction originally proposed in three-valued shape analysis can be cast in the framework of predicate abstraction. The consequences of our results are:

- a different view on the underlying concepts of three-valued shape analysis.
- a framework of *symbolic* shape analysis. Symbolic means that the abstract post operator is an operation over formulae and is itself constructed solely by automated reasoning.
- a clear phase separation between the computation of the abstraction and the computation of the fixed point. Among other potential advantages this allows the offline computation of the abstract post operator.
- the possibility to use efficient symbolic methods such as BDDs or SAT solvers. In particular, the abstract post operator itself can be represented as a BDD.

We formally characterized the abstract post operator of our analysis in terms of an abstraction of the best abstract post operator on sets of abstract states. In the course of this characterization we identified three sources for potential loss of precision:

- the restriction of the best abstract post on sets of abstract states to singleton sets of abstract states,
- Cartesian abstraction on sets of abstract objects,
- and the choice of the context operator.

In Chapter 4 we address the problem how to regain the precision that is lost due to the first and second item.

The context operator determines the trade-off between efficiency and precision of the analysis. A less precise context operator will lose precision if the analysis attempts to keep track of non-local properties such as reachability. The most precise context operator is given by the concretization function on sets of abstract states. However, choosing a more precise context operator might require recomputation of the abstraction for each individual application of the abstract post. This defeats the purpose of separating the computation of the abstraction from the fixed point computation and increases the number of decision procedure calls. In Chapter 6 we describe a context operator that provides a good balance between precision and efficiency.

Domain predicate abstraction does not *a priori* impose any restrictions on the data structures and properties to verify. The capabilities of our analysis are determined by the underlying decision procedur that is used for checking the entailments generated for computing the abstraction. There is ongoing research on how to adapt or extend existing theorem provers and decision procedures to the theories that are needed in the context of shape analysis. We present one such technique in Chapter 5.

Chapter 4

Lazy Nested Abstraction Refinement

In the previous chapter we have developed domain predicate abstraction, a new analysis that generalizes predicate abstraction to the point where it becomes effectively applicable as a shape analysis. Domain predicate abstraction provides a parameterized abstract domain and techniques to automatically compute the abstraction for a given instance of the abstract domain. However, the user of the analysis still needs to manually provide the right abstraction predicates in order to instantiate the analysis for the verification of a particular program and property. Can we push the degree of automation even further. In a wide range of existing program analyses [9,30,53], counterexample-guided abstraction refinement [35] provides an unmatched degree of automation by, essentially, instantiating a parameterized abstract domain automatically for a specific program and a specific correctness property. In this chapter we investigate the question whether it is possible to obtain the same automation in a shape analysis.

We develop a lazy nested abstraction refinement technique for symbolic shape analysis. Our abstraction refinement technique uses the notion of a *spurious error trace* which is also used in [9,30,35,52,53]. A spurious error trace is an error trace in the abstract system that has no correspondence in the concrete system. We extract new domain predicates from the proof of its spuriousness (in the spirit of [52] who, however, extract state predicates). We thus use a spurious error trace in order to automatically refine the abstract domain.

However, our new *nested* abstraction refinement loop uses spurious error traces to refine not only the abstract domain but also the abstract post operator on the abstract domain. I.e., two refinement phases are nested within a lazy abstraction refinement loop. The first phase refines the abstract domain as described above. If a spurious error trace is not eliminated by merely refining the abstract domain, a second phase called *Cartesian refinement* starts. Cartesian refinement uses the spurious error trace to increase the precision of the abstract post operator (its name refers to Cartesian abstraction; see Chapter 3.4).

As we will show in our experimental evaluation, the second refinement phase is crucial for the practical success of our symbolic shape analysis. In many benchmarks, the verification does not succeed *without*. The practical results are in line with the theoretical findings about the so-called *progress property* [53]. Progress means that every spurious error trace encountered during the analysis is eventually eliminated by a refinement step. The shape analysis with Cartesian refinement has the progress property and it does not without. In fact, it was this theoretical finding that lead us to the nested refinement and

the experimental success on the above-mentioned benchmarks.

Contributions. The technical contributions of this chapter are summarized as follows:

- We develop a new abstraction refinement technique that refines both the abstract domain of the analysis and the abstract post operator on that abstract domain.
- We show how this abstraction refinement technique can be combined with lazy abstraction [53].
- We prove that the resulting analysis is sound and has the progress property.

4.1 Example

In the following, we discuss our nested abstraction refinement technique on an example program. While this example does not require the full power that the abstract domain of our symbolic shape analysis provides, it illustrates all important aspects.

Consider program ListFilter given in Figure 4.1. The left hand side shows the pseudo code of the program. The program iterates over a list pointed to by program variable *first* and removes all nodes from the list whose *data* field is set to true. Our goal is to verify absence of null dereferences. All but one dereference are guarded by conditionals that imply that the dereferenced variable is not null. The critical statement is the dereference of variable *tmp* in the loop body. It is guarded by an assert statement. If we propagate this assertion back to location ℓ_1, we get the following formula:

$$\mathit{NullCheck} \equiv\ e{\neq}\mathsf{null} \wedge \mathit{data}\, e \wedge \mathit{prev}{\neq}\mathsf{null} \rightarrow (\mathit{next}\, \mathit{prev}){\neq}\mathsf{null}$$

Thus, our goal is to verify that *NullCheck* is an invariant at location ℓ_1. The formula *NullCheck* is implied by the following non-trivial inductive invariant *Inv* at location ℓ_1:

$$\mathit{Inv} \equiv\quad \mathit{prev}{\neq}\mathsf{null} \rightarrow (\mathit{next}\, \mathit{prev}){=}e\ .$$

We can express invariant *Inv* in the abstract domain of our symbolic shape analysis as a set of abstract states over unary domain predicates that denote sets of heap objects in a given program state. The denotation of such an abstract domain element is given by the following formula:

$$(\forall v.\, p_0\, v \leftrightarrow p_1\, v) \vee (\forall v.\, p_2\, v \leftrightarrow p_3\, v)$$

where the unary domain predicates p_0 to p_3 are given by:

$$\begin{aligned} p_0 &= (\lambda v.\, \mathit{prev}{=}v), & p_1 &= (\lambda v.\, \mathsf{null}{=}v),\\ p_2 &= (\lambda v.\, (\mathit{next}\, \mathit{prev}){=}v), & p_3 &= (\lambda v.\, e{=}v)\ . \end{aligned}$$

Our algorithm infers these predicates and synthesizes an invariant that implies the correctness of program ListFilter. We will now give a detailed presentation of our nested abstraction refinement algorithm. In Section 4.3 we will come back to the above example and explain in more details how program ListFilter is proved correct.

```
ℓ0: e:=first;
    prev:=null;
ℓ1: while e≠null do
      if e.data then
        if prev=null then
          prev:=e;
          e:=e.next;
          first:=e;
          prev.next:=null;
          prev:=null;
        else
          e:=e.next;
          tmp:=prev.next;
          assert(tmp≠null);
          tmp.next:=null;
          prev.next:=e;
      else
        prev:=e;
        e:=e.next;
ℓ2: done
```

$$\text{ListFilter} = (\Sigma_h, D_h, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$$
$$X = \{next, data, first, prev, tmp, e\}$$
$$\mathcal{L} = \{\ell_0, \ell_1, \ell_2, \ell_E\}$$
$$\mathcal{T} = \{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$$

τ_0: $(\ell_0, e{:=}first;$
$\quad prev{:=}\mathsf{null},\ \ell_1)$

τ_1: $(\ell_1, \mathtt{assume}(e{\neq}\mathsf{null});$
$\quad \mathtt{assume}(data(e));$
$\quad \mathtt{assume}(prev{=}\mathsf{null});$
$\quad prev{:=}e;$
$\quad e{:=}next\,e;$
$\quad first{:=}e;$
$\quad next{:=}next[prev{:=}\mathsf{null}];$
$\quad prev{:=}\mathsf{null},\ \ell_1)$

τ_2: $(\ell_1, \mathtt{assume}(e{\neq}\mathsf{null});$
$\quad \mathtt{assume}(data\ e);$
$\quad \mathtt{assume}(prev{\neq}\mathsf{null});$
$\quad e{:=}next\,e;$
$\quad tmp{:=}next\ prev$
$\quad \mathtt{assert}(tmp{\neq}\mathsf{null});$
$\quad next{:=}next[tmp{:=}\mathsf{null}];$
$\quad next{:=}next[prev{:=}e],\ \ell_1)$

τ_3: $(\ell_1, \mathtt{assume}(e{\neq}\mathsf{null});$
$\quad \mathtt{assume}(\neg(data\ e));$
$\quad prev{:=}e;$
$\quad e{:=}next\,e,\ \ell_1)$

τ_4: $(\ell_1, \mathtt{assume}(e{=}\mathsf{null}),\ \ell_2)$

τ_5: $(\ell_1, \mathtt{assume}(\neg NullCheck), \ell_E)$

Figure 4.1: Program ListFilter

4.2 Lazy Nested Abstraction Refinement

We now present our lazy nested abstraction refinement algorithm[1] in detail. We present the algorithm in a more abstract setting and then identify necessary conditions on the underlying analysis that guarantee soundness and progress of abstraction refinement. In the following sections we will prove that these conditions are satisified by domain predicate abstraction. Note, however, that the algorithm is also applicable for abstraction refinement of an analysis that uses Cartesian abstraction on top of classical predicate abstraction.

We assume a parametric abstract domain $AbsDom[\mathcal{P}]$ over a set of abstraction predicates $\mathcal{P}$. The abstraction predicates $\mathcal{P}$ can either be state predicates or domain predicates. We assume that the abstract domain is equipped with a partial order $\sqsubseteq$, join $\sqcup$ and meet $\sqcup$ operations, least element $\bot$, and greatest element $\top$ such that $AbsDom[\mathcal{P}](\sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is

[1] In general there is no guarantee that our method terminates. However, we stick to the term "algorihm" instead of "semi-algorithm".

a complete lattice. Furthermore, we assume functions

$$\alpha[\mathcal{P}] \in 2^{States} \to AbsDom[\mathcal{P}] \quad \text{and}$$
$$\gamma[\mathcal{P}] \in AbsDom[\mathcal{P}] \to 2^{States}$$

such that $(\alpha[\mathcal{P}], \gamma[\mathcal{P}])$ is a Galois connection between the lattices $AbsDom[\mathcal{P}]$ and 2^{States}. We assume that the concrete domain is given by sets of states. However, for notational convenience, we will often identify sets of state with formulae. In particular, we assume that for every concretization $\gamma[\mathcal{P}](S^\#)$ of some abstract domain element $S^\# \in AbsDom[\mathcal{P}]$ there exists a formula whose models are given by the set $\gamma[\mathcal{P}](S^\#)$. For sets of abstraction predicates $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1 \subseteq \mathcal{P}_2$ we require that $AbsDom[\mathcal{P}_1]$ is a sublattice of $AbsDom[\mathcal{P}_2]$ and for all sets of states S we have $\alpha[P_2](S) \sqsubseteq \alpha[P_1](S)$. We further require that $\gamma[\mathcal{P}]$ is a join morphism. Finally, we assume an abstract post operator $\mathsf{post}^\mathsf{A}[\mathcal{P}_1, \mathcal{P}_2]$ that maps elements of the abstract domain $AbsDom[\mathcal{P}_1]$ to elements of $AbsDom[\mathcal{P}_2]$ under a given command. We require that post^A is monotone and an approximation of the concrete post operator, i.e., we assume that for all commands c and $s^\# \in AbsDom[\mathcal{P}_1]$ the following condition holds:

$$\mathsf{post}([\![c]\!])(\gamma[\mathcal{P}_1](S^\#)) \subseteq \gamma[\mathcal{P}_2](\mathsf{post}^\mathsf{A}[\mathcal{P}_1, \mathcal{P}_2](c)(S^\#)) \ .$$

Our lazy nested abstraction refinement algorithm is shown in Figure 4.2. The procedure LazyNestedRefine takes a program $P = (\mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ as input and constructs an abstract reachability tree (ART) in the spirit of lazy abstraction [53]. An ART is a tree where each node r is labeled by a location $r.loc$ in $\mathcal{L}$, a set of abstraction predicates $r.preds$, and abstract states $r.states$ in $AbsDom[r.preds]$. The root node r_0 of the ART is labeled by the initial location $r_0.loc = \ell_0$. Edges in the ART are labeled by commands of transitions in program P. We write $r \xrightarrow{c} r'$ to denote that there is an edge in the ART from node r to node r' which is labeled by command c and we write $r \xrightarrow{\pi}{}^* r'$ to indicate that there is a (possibly empty) path from r to r' in the ART that is labeled by the sequence of commands π. Each path $r_0 \xrightarrow{\pi}{}^* r$ that starts in the root node of the ART induced an *abstract trace*. The abstract trace consists of the consecutive sequence of abstract states labeling the nodes on the path and the commands in π. We call an abstract trace $\sigma^\# = r_0 \xrightarrow{c_0} \ldots \xrightarrow{c_{n-1}} r_n$ with $r_n.loc = \ell_E$ an abstract error trace and we call the path $\pi = (c_0; \ldots; c_{n-1})$ the abstract error path associated with σ. We say that $\sigma^\#$ is *feasible* if there is some concrete trace $\sigma = s_0 \xrightarrow{c_0} \ldots \xrightarrow{c_{n-1}} s_n$ of P that is represented by $\sigma^\#$, i.e., if for all i with $0 \leq i \leq n$ we have $s_i \in \gamma[r_i.preds](r_i.states)$. An infeasible abstract trace is called *spurious*.

The lazy nested abstraction refinement algorithm iteratively constructs an ART until either a fixed point is reached and every trace of the program is contained in some abstract trace in the ART, or until the ART contains a feasible abstract error trace. If a spurious abstract error trace is encountered during the fixed point computation then this trace is removed from the ART and the corresponding abstract error path is used to refine the abstraction. We now describe the algorithm in detail.

The algorithm maintains a set of unprocessed ART edges. In each iteration one unprocessed ART edge (r_1, c, r_2) is selected. Then the abstract post for the corresponding command c is computed and the resulting abstract states are stored in $r_2.states$. If the computed abstract states are already subsumed by other ART nodes then the node r_2 is

proc LazyNestedRefine
  input $(\Sigma, D, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$: program
  begin
    let $r_0 = \langle loc\colon \ell_0, preds\colon \emptyset, states\colon \top, covered\colon \mathsf{false}\rangle$
    let $succ(r) =$
      let $Succ = \emptyset$
      for all $(r.loc, c, \ell') \in \mathcal{T}$ **do**
        let $r' = \langle loc\colon \ell', preds\colon \emptyset, states\colon \bot, covered\colon \mathsf{false}\rangle$
        add an edge $r \xrightarrow{c} r'$
        $Succ := Succ \cup \{(r, c, r')\}$
      done
      return $Succ$
    let $Unprocessed = succ(r_0)$
    while $Unprocessed \neq \emptyset$ **do**
      choose and remove $(r_1, c, r_2) \in Unprocessed$
      $r_2.states := \mathsf{post}^{\mathsf{A}}[r_1.preds, r_2.preds](c)(r_1.states)$
      if $r_2.states \sqsubseteq \bigsqcup_r \{\, r.states \mid r \neq r_2 \wedge r.loc = r_2.loc \,\}$ **then** $r.covered := \mathsf{true}$
      else if $r_2.loc \neq \ell_E$ **then** $Unprocessed := Unprocessed \cup succ(r')$
      else
        let r_s, π **such that** π is maximal path with $r_s \xrightarrow{\pi}{}^{*} r_2 \wedge$
                $\gamma[r_s.preds](r_s.states) \not\models \mathsf{wlp}(\pi)(\mathsf{false})$
        if $r_s.loc = \ell_0$ **then return** $counterexample(\pi)$
        else
          let r_p, c **such that** $r_p \xrightarrow{c} r_s$
          let $\mathcal{P}_\pi = \mathsf{extrPreds}(\mathsf{wlp}(\pi)(\mathsf{false}))$
          if $\mathcal{P}_\pi \not\subseteq r_p.preds$ **then** $r_s.preds := r_s.preds \cup \mathcal{P}_\pi$
          else
            let $\mathcal{P}_{c;\pi} = \mathsf{extrPreds}(\mathsf{wlp}(c;\pi)(\mathsf{false}))$
            $r_p.preds := r_p.preds \cup \mathcal{P}_{c;\pi}$
            $r_p.states := \alpha[r_p.preds](\mathsf{wlp}(c;\pi)(\mathsf{false})) \sqcap r_p.states$
          remove subtrees starting from r
          $r_s.states := \bot$
          $r_s.covered := \mathsf{false}$
          $Unprocessed := Unprocessed \cup \{(r_p, c, r_s)\}$
          for all r_2 **such that** $r_2.covered \wedge r_2.states \not\sqsubseteq \bot \wedge r_s.states$ older than $r_2.states$ **do**
            let r_1, c **such that** $r_1 \xrightarrow{c} r_2$
            $r_2.covered := \mathsf{false}$
            $r_2.states := \bot$
            $Unprocessed := Unprocessed \cup \{(r_1, c, r_2)\}$
          done
    done
    return "program is safe"
  end

Figure 4.2: Lazy nested abstraction refinement algorithm

marked as covered. Otherwise if $r_2.loc$ is the error location then there is an abstract error trace going from r_0 to r_2. In this case, the analysis determines whether the error trace is spurious. For this purpose it performs a symbolic backward analysis of the error trace. This backward analysis tries to find the oldest ancestor node r_s of r_2 with $r_s \xrightarrow{\pi}{}^* r_2$ such that $r_s.states$ represents some concrete state that can reach an error state by executing the sequence of commands π, i.e., formally r_s is the oldest node in the abstract error trace such that:

$$\gamma[r_s.preds](r_s.states) \not\models \mathsf{wlp}(\pi)(\mathsf{false}) \ .$$

If r_s is the root node of the ART then there exists a concrete error trace, i.e., the abstract error trace is feasible and the procedure returns the corresponding error path. If, however, r_s is not the root node then the abstract error trace is spurious. In this case we call r_s the *spurious node* of the abstract error trace. The algorithm then determines the immediate predecessor node r_p of the spurious node. We call r_p the *pivot node* of the abstract error trace. The pivot node is the youngest node that does not represent any concrete states that can reach an error state by following the commands in the abstract error trace. Depending on the refinement phase either r_s or r_p is refined and the spurious subtree below r_s is removed from the ART. The ART edge between r_p and r_s which was spurious is then scheduled for reprocessing. Finally, in order to ensure soundness, ART nodes that have potentially been marked as covered due to subsumption by nodes in the removed subtree are uncovered and also scheduled for reprocessing.

If the set of unprocessed ART edges becomes empty then all outgoing edges of inner ART nodes have been processed and all leaf nodes are covered, i.e., the least fixed point has been computed. For each program location ℓ in the input program an invariant can be computed from the ART by taking the join of the abstract states of all ART nodes labeled with location ℓ. The algorithm guarantees that the computed invariant implies that the program is safe (Theorem 40).

We now explain the two nested abstraction refinement phases. The spurious part of the error trace starts from the spurious node r_s. Assume that the ART edge from r_s to r_p is labeled by c and the path from r_s to r_2 by π. Our abstraction refinement procedure first attempts to refine the abstract domain of node r_s by adding new abstraction predicates $\mathcal{P}_\pi$ that are extracted from the spurious part π of the abstract error path associated with the abstract error trace. The predicate extraction function $\mathsf{extrPreds}$ should guarantee that the weakest precondition $\mathsf{wlp}(\pi)(\mathsf{false})$ of the path is expressible in the abstract domain of the refined node r_s, i.e., formally the following entailment should hold:

$$\gamma[\mathcal{P}_\pi](\alpha[\mathcal{P}_\pi](\mathsf{wlp}(\pi)(\mathsf{false}))) \models \mathsf{wlp}(\pi)(\mathsf{false}) \ .$$

If the underlying analysis was to compute the most precise abstract post operator then we would have:

$$\gamma[\mathcal{P}_\pi](\mathsf{post}^{\mathsf{A}}[r_p.preds, \mathcal{P}_\pi](c)(r_p.states)) \models \mathsf{wlp}(\pi)(\mathsf{false}) \ .$$

Thus, we were guaranteed that after refining the predicate set of node r_s and reprocessing the ART edge (r_p, c, r_s), the node r_s would no longer be spurious for this abstract error trace. This would give us progress of abstraction refinement. However, if the abstract post is not optimal then the same spurious error trace might be reproduced after the refinement. Thus, the refinement procedure might fail to derive new predicates for node r_s. In this

case, the nested refinement phase refines the abstract post for command c and node r_p. The refinement of the abstract post is performed indirectly by computing the meet of the abstract states that label node r_p with an abstraction of the weakest liberal precondition $\mathsf{wlp}(c;\pi)(\mathsf{false})$. Note that in practice computing $\alpha[\mathcal{P}_{c;\pi}](\mathsf{wlp}(c;\pi)(\mathsf{false}))$ is cheap, because $\mathcal{P}_\pi$ consists of the predicates extracted from the formula $\mathsf{wlp}(c;\pi)(\mathsf{false})$.

The second refinement phase counteracts any Cartesian abstraction that is performed during the computation of the abstract post. We therefore refer to this refinement step as *Cartesian refinement*. Note that Cartesian refinement does not ensure that the best abstract post is computed for the spurious ART edge. However, Cartesian refinement still ensures progress of the abstraction refinement loop (Theorem 41).

4.2.1 Soundness

The soundness of our lazy nested abstraction refinement algorithm is formally stated in Theorem 40. The soundness proof depends on various invariants of the refinement loop in procedure LazyNestedRefine. These invariants are stated in the following lemma. The proof of this lemma goes by induction on the runs of LazyNestedRefine.

Lemma 39 *Let $P = (\Sigma, D, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ be a program. In any run of* LazyNestedRefine(P) *the following properties hold at each entry to the outer* ***while*** *loop of* LazyNestedRefine*:*

1. $r_0.\mathit{covered} = \mathsf{false}$
2. $\gamma[r_0.\mathit{preds}](r_0.\mathit{states}) \equiv \mathsf{true}$
3. $\bigsqcup_r \{\, r.\mathit{states} \mid r.\mathit{loc} = \ell_E \,\} = \bot$
4. *for all ART nodes r with $r.\mathit{covered} = \mathsf{false}$ and $r.\mathit{states} \neq \bot$, and for all $(r.\mathit{loc}, c, \ell') \in \mathcal{T}$ there exists an ART node r' with $r'.\mathit{loc} = \ell'$ and either:*
   (a) $(r, c, r') \in \mathit{Unprocessed}$ *or*
   (b) $\mathsf{post}([\![c]\!])(\gamma[r.\mathit{preds}](r.\mathit{states})) \subseteq \gamma[r'.\mathit{preds}](r'.\mathit{states})$
5. *for all ART nodes r with $r.\mathit{covered} = \mathsf{true}$*

$$r.\mathit{states} \sqsubseteq \bigsqcup_{r'} \{\, r'.\mathit{states} \mid r'.\mathit{covered} = \mathsf{false} \wedge r'.\mathit{loc} = r.\mathit{loc} \,\} \quad .$$

Theorem 40 (Soundness) *Procedure* LazyNestedRefine *is sound, i.e., for any program P if* LazyNestedRefine(P) *terminates with "program is safe" then program P is safe.*

Proof. Let $P = (\Sigma, D, X, \mathcal{L}, \ell_0, \ell_E, \mathcal{T})$ and let $\sigma = s_0 \overset{c_0}{\rightarrow} \ldots \overset{c_{n-1}}{\rightarrow} s_n$ be a trace of program P. We first prove the following property by induction on i: for all $0 \leq i \leq n$ there exists an uncovered ART node r_i such that $r_i.\mathit{loc} = s_i(pc)$ and $s_i \in \gamma[r_i.\mathit{preds}](r_i.\mathit{states})$. Let $i = 0$ then $r_0.\mathit{loc} = s_0(pc)$. Furthermore, from Properties 1 and 2 of Lemma 39 follows that r_0 is not covered and $s_0 \in \gamma[r_0.\mathit{preds}](r_0.\mathit{states})$. Now, let $i > 0$ then by induction hypothesis there exists an uncovered ART node r_{i-1} with $s_{i-1}(pc) = r_{i-1}.\mathit{loc}$ and $s_{i-1} \in \gamma[r_{i-1}.\mathit{preds}](r_{i-1}.\mathit{states})$, i.e., $r_{i-1}.\mathit{states} \neq \bot$. Let $\tau_{i-1} = (s_{i-1}(pc), c_{i-1}, s_i(pc))$.

Then $\tau_{i-1} \in \mathcal{T}$ since σ is a trace of P. Since $\mathit{Unprocessed} = \emptyset$, it follows from Property 4 of Lemma 39 that there exists some ART node r' such that $r'.\mathit{loc} = s_i(\mathit{pc})$ and

$$\mathsf{post}([\![\mathit{comm}]\!])(\gamma[r_{i-1}.\mathit{preds}](r_{i-1}.\mathit{states})) \subseteq \gamma[r'.\mathit{preds}](r'.\mathit{states}) \ .$$

Since $(s_{i-1}, s_i) \in [\![\tau_{i-1}]\!]$, we have $r'.\mathit{loc} = s_i(\mathit{pc})$ and $s_i \in \gamma[r'.\mathit{preds}](r'.\mathit{states})$. If r' is uncovered choose $r_i = r'$. Otherwise, from Property 5 of Lemma 39 follows:

$$r'.\mathit{states} \sqsubseteq \bigsqcup_{r''} \{\, r''.\mathit{states} \mid r''.\mathit{covered} = \mathsf{false} \wedge r''.\mathit{loc} = r'.\mathit{loc} \,\} \ .$$

Thus, from monotonicity of γ and the fact that γ is a join morphism follows:

$$\gamma[r'.\mathit{preds}](r'.\mathit{states}) \models \bigvee_{r''} \{\, \gamma[r''.\mathit{preds}](r''.\mathit{states}) \mid r''.\mathit{covered} = \mathsf{false} \wedge r''.\mathit{loc} = r'.\mathit{loc} \,\} \ .$$

Hence there is at least one uncovered ART node r'' with $r''.\mathit{loc} = r'.\mathit{loc} = s_i(\mathit{pc})$ and $s_i \in \gamma[r''.\mathit{preds}](r''.\mathit{states})$. Then choose $r_i = r''$ for one such r'', which concludes the induction proof.

From Property 3 of Lemma 39 follows that $\bigsqcup_r \{\, r.\mathit{states} \mid r.\mathit{loc} = \ell_E \,\} = \bot$ holds whenever $\mathsf{LazyNestedRefine}(P)$ terminates with "program is safe". Thus, if for any $0 \leq i \leq n$ we had $s_i(\mathit{pc}) = \ell_E$ then for all ART nodes r with $r.\mathit{loc} = \ell_E$ we would have $s_i \notin \gamma[r.\mathit{preds}](r.\mathit{states})$. This would contradict the property proved above. It follows that σ is not an error trace. Since σ was chosen arbitrarily, we conclude that P is safe. ∎

4.2.2 Progress

We now identify sufficient conditions on the predicate extraction function $\mathsf{extrPreds}$ and abstract post operator $\mathsf{post}^{\mathsf{A}}$ that guarantee progress of abstraction refinement.

Note that we cannot prove that a given spurious error path can only occur finitely often in a run of procedure $\mathsf{LazyNestedRefine}$. The reason is that whenever we refine an ART node r we dispose the already explored subtrees of r. It is therefore possible that a spurious error trace is rediscovered infinitely often because the same subtree is repeatedly removed and reconstructed due to refinement steps that are triggered by other spurious error traces. In principle, one can modify procedure $\mathsf{LazyNestedRefine}$ and impose restrictions on how the ART is explored, or remember already explored subtrees, such that a given error trace only occurs finitely often in any run. However, this would not make the procedure terminate more often: we prove that any non-terminating run of procedure $\mathsf{LazyNestedRefine}$ that involves infinitely many refinement steps must involve infinitely many spurious error paths. Thus, it can never happen that procedure $\mathsf{LazyNestedRefine}$ does not terminate because it gets stuck on refining a specific spurious error path over and over again.

Theorem 41 (Progress) *Assume that for all closed formulae F, commands c, and $\mathcal{P}_1, \mathcal{P}_2$ such that $\mathcal{P}_1 = \mathsf{extrPreds}(\mathsf{wlp}(c)(F))$ and $\mathcal{P}_2 = \mathsf{extrPreds}(F)$ the following entailment holds:*

$$\gamma[\mathcal{P}_2](\mathsf{post}^{\mathsf{A}}[\mathcal{P}_1, \mathcal{P}_2](c)(\alpha[\mathcal{P}_1](\mathsf{wlp}(c)(F)))) \models F \ .$$

Then a run Δ of procedure $\mathsf{LazyNestedRefine}$ terminates, unless the set of spurious error paths encountered in Δ is infinite.

Proof. Let P be a program. Assume that there is a non-terminating run Δ of $\mathsf{LazyNestedRefine}(P)$ that only encounters finitely many spurious error paths. For the i-th refinement step in Δ, let π_i be the spurious error path in this refinement step, i.e., the sequence of commands labeling the path from the root node r_0 of the ART to the error node in the i-th refinement step. Furthermore, let $r_{p,i}$ be the pivot node in this refinement step, $r_{s,i}$ the spurious node, $\pi_{p,i}$ the prefix of π_i labeling the path to $r_{p,i}$, $\pi_{s,i}$ the suffix of π_i labeling the path from $r_{s,i}$ to the error node, and c_i the command labeling the edge between $r_{p,i}$ and $r_{s,i}$, i.e., $\pi_i = \pi_{p,i}; c_i; \pi_{s,i}$. Furthermore, let states_i (preds_i) be the function associating abstract states (abstraction predicates) with ART nodes before the i-th refinement step in Δ. For $i \in \mathbb{N}$ let $\mathit{ext}(i)$ be the set of all spurious error paths encountered in some refinement step j with $i \leq j$ that are extensions of $\pi_{p,i}$, i.e.

$$\mathit{ext}(i) \stackrel{\text{def}}{=} \{\, \pi_j \mid i \leq j \text{ and } \pi_{p,j} = \pi_{p,i} \,\} \quad .$$

Finally, let $\mathit{inf}(\Delta)$ be the set of paths $\pi_{i,p}$ that are encountered infinitely often in Δ, i.e.

$$\mathit{inf}(\Delta) \stackrel{\text{def}}{=} \{\, \pi_p \mid |\{\, i \in \mathbb{N} \mid \pi_{i,p} = \pi_p \,\}| = \infty \,\} \quad .$$

Since there are only finitely many spurious error paths encountered in Δ and each such path itself is finite, $\mathit{inf}(\Delta)$ is nonempty. Moreover, there exists some $\pi_p \in \mathit{inf}(\Delta)$ such that for all $\pi'_p \in \mathit{inf}(\Delta)$, π'_p is not a proper prefix of π_p. Choose one such path $\pi_p \in \mathit{inf}(\Delta)$. We can conclude that there is some $n \in \mathbb{N}$ such that $\pi_{p,n} = \pi_p$ and for all $j > n$, $\pi_{p,j}$ is not a proper prefix of π_p. Thus, after the n-th refinement step, Δ will at most add additional successors to the ART node $r_{p,n}$. Neither $r_{p,n}$ nor any of its immediate successor nodes will be removed from the ART after the n-th refinement step in Δ. Furthermore, for all $j \geq n$ we have:

1. $r_{p,n}.\mathit{states}_{j+1} \sqsubseteq r_{p,n}.\mathit{states}_j$.
2. for all nodes r with $r = r_{p,n}$ or $r_{p,n} \xrightarrow{c} r$ for some command c: $r.\mathit{preds}_j \subseteq r.\mathit{preds}_{j+1}$

Now choose some spurious error path $\pi \in \mathit{ext}(n)$. Then there exists some refinement step m in Δ with $n \leq m$ such that $\pi_m = \pi$, $\pi_{p,m} = \pi_{p,n}$ and for all $j > m$:

$$\mathsf{extrPreds}(\mathsf{wlp}(\pi_{s,m})(\mathsf{false})) \subseteq r_{s,m}.\mathit{preds}_j$$

holds. Now assume there exists yet another refinement step $k > m$ with $\pi_k = \pi$ and $\pi_{p,k} = \pi_{p,n}$. Then k is a Cartesian refinement step. Thus, from the assumption in Theorem 41 and the monotonicty of α and $\mathsf{post}^{\mathsf{A}}$, we conclude that for all $j > k$:

$$\gamma[r_{s,k}.\mathit{preds}_j](\mathsf{post}^{\mathsf{A}}[r_{p,k}.\mathit{preds}_j, r_{s,k}.\mathit{preds}_j](c_k)(r_{p,k}.\mathit{states}_j)) \subseteq \mathsf{wlp}(\pi_{s,k})(\mathsf{false}) \quad .$$

From this we conclude that for all $j > k$:

$$\gamma[r_{s,k}.\mathit{preds}_j](r_{s,k}.\mathit{states}_j) \subseteq \mathsf{wlp}(\pi_{s,k})(\mathsf{false}) \quad .$$

This means that for all $j > k$ with $\pi = \pi_j$ we have $r_{p,k} \neq r_{p,j}$ and thus $r_{p,n} \neq r_{p,j}$. Since $r_{p,n}$ is never removed from the ART and there is at most one path in the ART labeled by the commands in $\pi_{p,n}$, we conclude that for all $j > k$ we have $\pi_{p,j} \neq \pi_{p,n}$ and hence $\pi_{p,j} \neq \pi_p$. Since π was chosen arbitrarily in $\mathit{ext}(n)$, it follows that for all $\pi \in \mathit{ext}(n)$ there exists some $k \geq n$ such that for all $j > k$ with $\pi = \pi_j$ we have $\pi_{p,j} \neq \pi_p$. Let k_{max} be the maximum of all these k. Then for all $j > k_{max}$ we have $\pi_{p,j} \neq \pi_p$. This contradicts the fact that $\pi_p \in \mathit{inf}(\Delta)$. ∎

4.3 Example Run of Nested Abstraction Refinement

We now come back to our motivating example, program ListFilter shown in Figure 4.1. The right-hand side of Figure 4.1 represents program ListFilter in a form that fits our notion of programs defined in Section 2.2. In order to make the presentation of the example more concise, the program only consists of four locations, the initial location ℓ_0, the loop cut point ℓ_1, an exit location ℓ_2, and an error location ℓ_E. The basic blocks in the left-hand-side version of the program have been contracted to single commands. Note that we further transformed the assert command in the loop body into a control-flow edge from location ℓ_1 to the error location.

We will now apply procedure LazyNestedRefine to this example program using domain predicate abstraction as the underlying analysis. In our example, all domain predicates are unary predicates that range over heap objects. To increase readability we will represent abstract objects as sets of (potentially complemented) abstraction predicates. For instance, the abstract object $[p_0 \mapsto \{0\}, p_1 \mapsto \{1\}, p_2 \mapsto \{0,1\}]$ over domain predicates p_0, p_1, and p_2 is represented by the set $\{\overline{p_0}, p_1\}$.

We construct the ART starting from the root node $r_0\colon (\ell_0, \{\emptyset\}, \emptyset)$ labeled by location ℓ_0, abstract state $\{\emptyset\}$, and an empty set of abstraction predicates. The abstract state $\{\emptyset\}$ denotes the set of all concrete states of program ListFilter. Two executions of the loop in procedure LazyNestedRefine produce an ART that consists of the following abstract error trace σ_0:

$$r_0\colon (\ell_0, \{\emptyset\}, \emptyset) \xrightarrow{c_0} r_1\colon (\ell_1, \{\emptyset\}, \emptyset) \xrightarrow{c_5} r_2\colon (\ell_E, \{\emptyset\}, \emptyset) \ .$$

The trace reaches the error location ℓ_E, because the assertion *NullCheck* fails when location ℓ_1 is reached for the first time. Now the algorithm determines whether σ_0 corresponds to a concrete error trace or whether it is an artefact of the abstraction. The trace σ_0 is spurious. The smallest suffix of σ_0 that proves its spuriousness is σ_0 itself: the weakest precondition $\mathsf{wlp}(c_0; c_5)(\mathsf{false})$ is given by the formula:

$$\mathit{first} \neq \mathsf{null} \wedge \mathit{data}\ \mathit{first} \wedge \mathsf{null} \neq \mathsf{null} \rightarrow (\mathit{next}\ \mathsf{null}) \neq \mathsf{null} \ .$$

This formula is valid, since one of the conjuncts in the antecedent of the implication is unsatisfiable. The spurious node of this abstract error trace is the ART node r_1. We refine the abstract domain of r_1 by extracting new domain predicates that express atomic formulae in the weakest liberal precondition $\mathsf{wlp}(c_5)(\mathsf{false})$ which is equivalent to the assertion *NullCheck*. From this formula we extract the set of abstraction predicates $\mathcal{P}_0$ consisting of the following new domain predicates:

$$\begin{aligned} p_0 &\equiv (\lambda v.\, \mathit{prev} = v) \\ p_1 &\equiv (\lambda v.\, \mathsf{null} = v) \\ p_2 &\equiv (\lambda v.\, (\mathit{next}\ \mathit{prev}) = v) \ . \end{aligned}$$

Continuing the algorithm with the new predicates produces yet another abstract error trace σ_1:

$$\begin{aligned} &r_0\colon (\ell_0, \{\emptyset\}, \emptyset) \xrightarrow{c_0} r_1\colon (\ell_1, \{\{p_0, p_1\}, \{\overline{p_0}, \overline{p_1}\}\}, \mathcal{P}_0) \xrightarrow{c_3} \\ &r_3\colon (\ell_1, \{\{\overline{p_0}, p_1\}, \{\overline{p_1}\}\}, \mathcal{P}_0) \xrightarrow{c_5} r_4\colon (\ell_E, \{\emptyset\}, \emptyset) \end{aligned}$$

The trace σ_1 starts at location ℓ_o, iterates once through the while loop by executing the else branch of the first conditional in the body of the while loop, and goes back to location ℓ_1 where assertion *NullCheck* fails. Trace σ_1 is again spurious. The first spurious node in the trace is r_3. From the spurious part of the trace we infer one new domain predicate:

$$p_3 \equiv (\lambda v.\ e{=}v) \ .$$

Note that at this point our abstract domain is able to express the inductive invariant *Inv* at location ℓ_1 which guarantees that the error location is not reachable. A set of abstract states whose concretization corresponds to the models of invariant *Inv* is, e.g., given by:

$$\{\{\{p_0, p_1\}, \{\overline{p_0}, \overline{p_1}\}\}, \{\{p_2, p_3\}, \{\overline{p_2}, \overline{p_3}\}\}\} \ .$$

If our analysis was to compute the best abstract post operator for our abstract domain then it would guarantee that, after adding predicate p_3 to r_3, the abstract error trace σ_1 is eliminated.

However, our analysis is based on the context-sensitive Cartesian post rather than the best abstract post. The price that we pay for the loss of precision under this abstract post is that progress of abstraction refinement is no longer guaranteed. In fact, in our example the refinement algorithm produces the same spurious error trace σ_1 even after predicate p_3 has been added, i.e., the abstraction of command c_3 remains spurious. The loss of precision under Cartesian abstraction is caused by the fact that the best abstract post operator for command c_3 behaves nondeterministically. We can think of this nondeterminism as a form of materialization [103]. Cartesian abstraction counteracts materialization. In order to better understand this problem, we take a closer look at the abstract post for command c_3. Consider the abstract state

$$s^\# = \{\{p_0, p_1\}, \{\overline{p_0}, \overline{p_1}\}\}$$

that labels ART node r_1 and consider abstract object $o^\# = \{\overline{p_0}, \overline{p_1}, p_2, p_3\}$ in the expansion of abstract state $s^\#$. Figure 4.3 shows three concrete states that are represented by $s^\#$ and their post states under command c_3. There is a concrete object in state s_3 that is represented by $o^\#$ and this object is pointed to by reference variable e after execution of command c_3. However, in state s_1 there is a concrete object that is also represented by abstract object $o^\#$, but which is not pointed to by e after execution of c_3. If we want to keep track of the correlation between predicates p_2 and p_3, we need to split $s^\#$ and abstract objects in $s^\#$ according to the fact whether some object is pointed to by reference variable e after execution of c_3 or not. In shape analysis this process of splitting is commonly referred to as materialization. If the analysis is not able to perform materialization then it loses precision and most certainly fails to verify many interesting programs and properties [32]. The best abstract post operator performs materialization implicitly and we say that it behaves nondeterministically if materialization actually occurs. Computing the best abstract post for command c_3 and $s^\#$ results in the following three abstract states:

$$\begin{aligned} & \{\{\overline{p_1}, \overline{p_2}, \overline{p_3}\}, \{\overline{p_0}, \overline{p_1}, p_2, p_3\}, \{\overline{p_0}, \overline{p_2}, \overline{p_3}\}\}, \\ & \{\{\overline{p_1}, \overline{p_2}, \overline{p_3}\}, \{\overline{p_0}, p_1, p_2, p_3\}\}, \\ \text{and} \quad & \{\{p_0, \overline{p_1}, p_2, p_3\}, \{\overline{p_0}, \overline{p_2}, \overline{p_3}\}\} \ . \end{aligned}$$

Abstracting these three abstract states by a single abstract state results in:

$$\{\{p_1, p_2, p_3\}, \{\overline{p_0}, \overline{p_2}, \overline{p_3}\}, \{\overline{p_0}, p_2, p_3\}, \{\overline{p_1}, \overline{p_2}, \overline{p_3}\}\} \quad .$$

This abstract state corresponds to the join of the following two sets of abstract objects:

$$\{\{\overline{p_0}, p_1, p_2, p_3\}, \{\overline{p_0}, p_1, \overline{p_2}, \overline{p_3}\}\},$$
$$\{\{\overline{p_0}, \overline{p_1}, p_2, p_3\}, \{p_0, \overline{p_1}, \overline{p_2}, \overline{p_3}\}, \{p_0, \overline{p_1}, p_2, p_3\}\} \quad .$$

The first set is the result of applying the context-sensitive abstract post operator on abstract objects to abstract object $\{p_0, p_1\}$ in $s^\#$, i.e., this set of abstract objects is the new Boolean covering in the post states of $s^\#$ of objects represented by abstract object $\{p_0, p_1\} \in s^\#$. The second set is the result of applying the context-sensitve Cartesian post operator to abstract object $\{\overline{p_0}, \overline{p_1}\}$. If we apply Cartesian abstraction to each of these two sets (i.e., compute the union of all abstract objects in the set and remove predicates that occur with both polarities) and join the results, we obtain the Cartesian post for command c_3 and abstract state $s^\#$:

$$\{\{\overline{p_0}, p_1\}, \{\overline{p_1}\}\} \quad .$$

The Cartesian post loses the correlation between domain predicates p_2 and p_3, because the best abstract post behaves nondeterministically.

After refining ART node r_3 using spurious error trace σ_1 we still get the same spurious error path. Thus, we are not able to infer any new abstraction predicates and our analysis would be stuck if we used spurious error traces to refine only the abstract domain. At this point Cartesian refinement comes into play. Cartesian refinement refines the Cartesian post indirectly by splitting the abstract state $s^\#$ labelling the pivot node r_1 into a set of abstract states and individual abstract objects in $s^\#$ into sets of abstract objects, such that the best abstract post operator behaves deterministically with respect to our target property *NullCheck*.

Thus, Cartesian refinement performs materialization on-demand and guided by the property to verify. For the purpose of splitting we compute the weakest liberal precondition $\mathsf{wlp}(c_3; c_5)(\mathsf{false})$, extract domain predicates from this formula so that it can be precisely expressed in our abstract domain, and compute its abstraction. The splitting of $s^\#$ is accomplished by computing the meet of $s^\#$ with the abstracted weakest liberal precondition. In our example we extract one new domain predicate:

$$p_4 \equiv (\lambda v.\, (next\, e){=}v) \quad .$$

The meet of $s^\#$ with the abstracted weakest liberal precondition is given by the following two abstract states:

$$\{\{p_0, p_1, p_4\}, \{\overline{p_0}, \overline{p_1}, \overline{p_4}\}\},$$
$$\{\{p_0, p_1, \overline{p_4}\}, \{\overline{p_0}, \overline{p_1}, p_4\}, \{\overline{p_0}, \overline{p_1}, \overline{p_4}\}\} \quad .$$

This refined set of abstract states distinguishes between concrete states where null will be pointed to by e after execution of command c_3 and states where null will not be pointed to by e. Also, in each abstract state there is no abstract object whose covering in the post states contains, both, abstract objects with p_3 and abstract objects with $\overline{p_3}$. The Cartesian

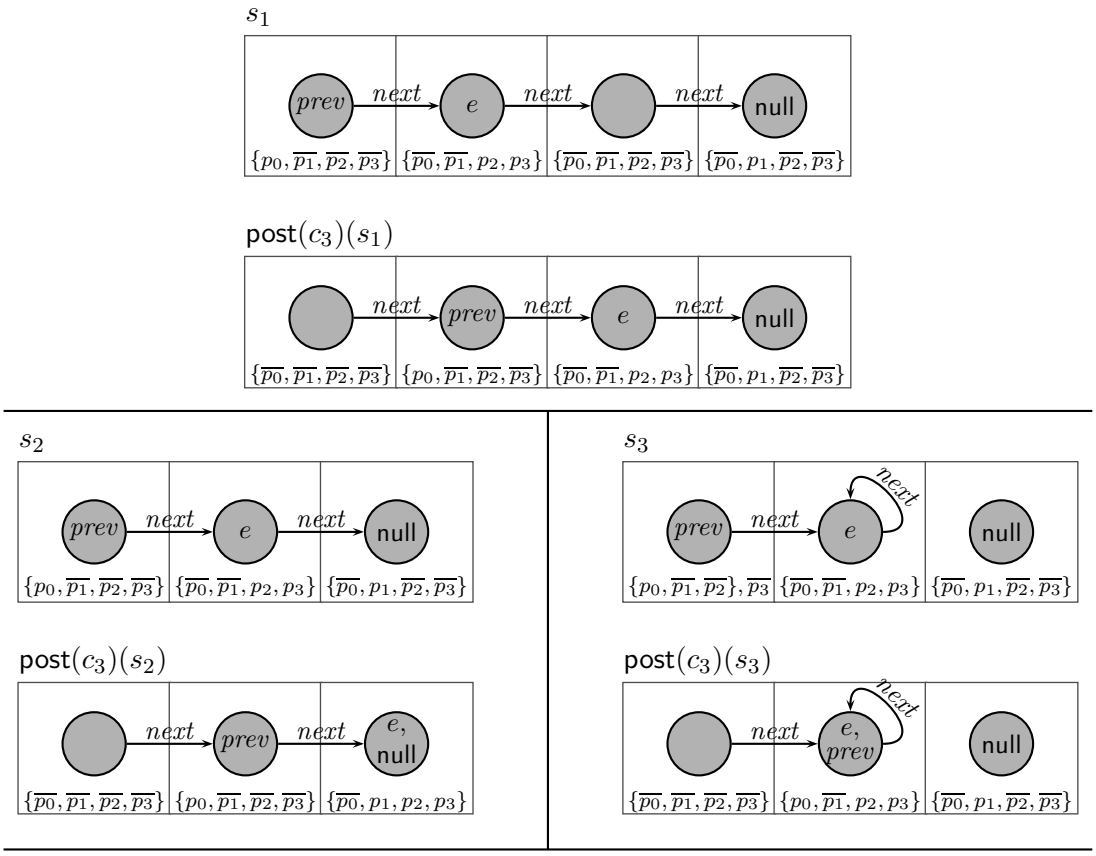


Figure 4.3: Three concrete states represented by abstract state $\{\{p_0, p_1\}, \{\overline{p_0}, \overline{p_1}\}\}$ and their post states under command c_3.

post for the refined set of abstract states and command c_3 results in the following two abstract states:

$$\{\{\overline{p_0}, p_1, p_2, p_3\}, \{\overline{p_1}, \overline{p_2}, \overline{p_3}\}\},$$
$$\{\{\overline{p_0}, p_1, \overline{p_2}, \overline{p_3}\}\, \{\overline{p_1}, p_2, p_3\}, \{\overline{p_1}, \overline{p_2}, \overline{p_3}\}\} \quad .$$

The concretization of these abstract states implies invariant *Inv*, i.e., the outgoing ART edge of r_1 labeled by command c_3 is no longer spurious. After three more iterations of the abstraction refinement loop the analysis produces an invariant that proves that the error location ℓ_E is not reachable.

4.4 Progress for Domain Predicate Abstraction

We now show that domain predicate abstraction fulfils the requirements for soundness and progress of lazy nested abstraction refinement. We require that the domain of all domain predicates that are used as abstraction predicates is fixed *a priori*, i.e., all abstraction predicates that are inferred during the analysis range over the same fixed domain. One can determine an appropriate domain, e.g., by looking at the formulae that occur in the

analyzed program. In the following, the chosen domain of abstraction predicates is denoted by *Dom*.

The abstract domain $AbsDom[\mathcal{P}]$ over a set of domain predicates $\mathcal{P}$ is given by sets of abstract states $AbsStates[\mathcal{P}]$ as defined in Chapter 3. The Galois connection between concrete and abstract domain is given by the functions α^+ and γ. The abstract post operator $\mathsf{post}^{\mathsf{A}}$ is given by the context-sensitive Cartesian post operator $\mathsf{post}^{\mathsf{C}}_{\kappa}$, where κ is some context operator. We make two minor modifications to the definition of $\mathsf{post}^{\mathsf{C}}_{\kappa}$ given in Section 3.4. First, we allow that abstract states in the domain and range of $\mathsf{post}^{\mathsf{C}}_{\kappa}$ are defined with respect to separate sets of abstraction predicates. Second, recall from the discussion in Section 3.4 that the Cartesian post operator loses precision in the case where the image of the concrete post is the empty set of states. We handle this case explicitly in order to avoid such loss of precision. The abstract post operator of our analysis is then defined as follows: let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two sets of abstraction predicates then:

$$\mathsf{post}^{\mathsf{A}}[\mathcal{P}_1,\mathcal{P}_2] \in Com \to AbsStates[\mathcal{P}_1] \to AbsStates[\mathcal{P}_2]$$

$$\mathsf{post}^{\mathsf{A}}[\mathcal{P}_1,\mathcal{P}_2](c)(S^{\#}) \stackrel{\text{def}}{=} \bigsqcup_{s^{\#}\in S^{\#}} \mathsf{if}\ \gamma(s^{\#}) \models \mathsf{wlp}(c)(\mathsf{false})\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \mathsf{post}^{\mathsf{C}}_{\kappa}[\mathcal{P}_1,\mathcal{P}_2](c)(\{s^{\#}\})\ .$$

From the definitions and properties of domain predicate abstraction that are given in Chapter 3, it is easy to see that this setup fulfils all the requirements for soundness of nested lazy abstraction refinement. It remains to show that it also fulfils the requirements for the progress property.

4.4.1 Progress for Havoc-free Programs

We first prove that progress of nested lazy abstraction refinement is guaranteed for programs that do not contain any `havoc` commands. We need to prove the assumption made in Theorem 41, i.e., for any deterministic command c and formulae F the following property holds: if $\mathcal{P}_1 = \mathsf{extrPreds}(\mathsf{wlp}(c)(F))$ and $\mathcal{P}_2 = \mathsf{extrPreds}(F)$ then

$$\gamma[\mathcal{P}_2](\mathsf{post}^{\mathsf{A}}[\mathcal{P}_1,\mathcal{P}_2](c)(\alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(F)))) \models F\ . \tag{4.1}$$

We cannot prove this property without imposing any restrictions on the predicate extraction function $\mathsf{extrPreds}$. First, we expect that function $\mathsf{extrPreds}$ extracts sufficiently many predicates from a given formula F such that there exists some set of abstract states whose concretization corresponds to the models of F. This restriction ensures that Condition 4.1 would hold, if $\mathsf{post}^{\mathsf{A}}$ was the best abstract post operator. We need to impose an additional restriction in order to prove Condition 4.1 for the context-sensitive Cartesian post. For the progress property to hold, Cartesian refinement needs to ensure that the best abstract post behaves deterministically with respect to abstract states that imply the weakest liberal precondition of formula F. However, this is not possible if the predicate extraction function decreases the *granularity* of the abstract domain when it extracts predicates from $\mathsf{wlp}(c)(F)$. This means that if we have some abstract object $o^{\#}$ over predicates extracted from F then the weakest domain precondition of the domain predicate represented by $o^{\#}$ should be precisely representable in terms of abstract objects over domain predicates extracted from $\mathsf{wlp}(c)(F)$. The two restrictions on the predicate extraction function $\mathsf{extrPreds}$ are formalized in the following definition.

Definition 42 *We say that a predicate extraction function* $\mathsf{extrPreds}$ *is* admissible *if the following two conditions hold:*

1. *for any closed formula* F*, if* $\mathcal{P} = \mathsf{extrPreds}(F)$ *then*

$$\gamma[\mathcal{P}](\alpha^+[\mathcal{P}](F)) \equiv F \ .$$

2. *for any closed formula* F *and deterministic command* c*, if* $\mathcal{P}_1 = \mathsf{extrPreds}(\mathsf{wlp}(c)(F))$ *and* $\mathcal{P}_2 = \mathsf{extrPreds}(F)$ *then for all domain predicates* p *such that either* p *or its complement is in* $\mathcal{P}_2$*, and for all abstract states* $s^\#$ *with* $\{s^\#\} = \alpha^+[\mathcal{P}_1](\{s\})$ *for some state* s*:*

$$\dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](\dot{\mathsf{wlp}}(c)(p))) \mathbin{\dot{\cap}} (\lambda\vec{o}.\, \kappa[\mathcal{P}_1](\{s^\#\})) \mathbin{\dot{\subseteq}} \dot{\mathsf{wlp}}(c)(p) \ .$$

Definition 42 suggests a simple rule for constructing admissible predicate extraction functions. If we start from a set of domain predicates $\mathcal{P}_2$ that is able to express some formula F, e.g., the atomic formulae occuring in F, then the set of predicates $\mathcal{P}_1$ for $\mathsf{wlp}(c)(F)$ may consist of all predicates $\dot{\mathsf{wlp}}(c)(p)$ for $p \in \mathcal{P}_2$. However, in practice one would like to avoid adding each $\dot{\mathsf{wlp}}(c)(p)$ as a single monolithic predicate. One would rather like to split these predicates into simpler ones. For instance, if $\dot{\mathsf{wlp}}(c)(p)$ is of the form $(\lambda\vec{v}.\, G \to p'(\vec{v}))$ where G is a guard from an assume command (i.e., a closed formula) then one can split this domain formula into simpler domain formulae that represent G and domain formulae that represent p'. In particular, the predicate extraction function can split a closed formulae G into simpler domain predicates that are not *nullary* predicates and still satisfy the second condition of Definition 42. For instance, in Section 4.1 we split guards of the form $e = \mathsf{null}$ into domain predicates $(\lambda v.\, e = v)$ and $(\lambda v.\, \mathsf{null} = v)$. If the abstract domain is able to express both G and its negation then the abstraction of a single concrete state s will preserve the information whether s satisfies G. The second condition of admissibility is still satisified if this information is also preserved by the context operator.

The following lemma states that for deterministic commands weakest liberal preconditions distribute over joins. The lemma can be easily proved from the semantics of deterministic commands.

Lemma 43 *Let* c *be a deterministic command. Then* $\mathsf{wlp}(c)$ *and* $\dot{\mathsf{wlp}}(c)$ *are complete join morphisms.*

The next lemma states that, given an admissible predicate extraction function, if abstract domains are constructed from a formula F, respectively its weakest liberal precondition for some deterministic command, then the granularity of abstract states representing F is preserved under wlp.

Lemma 44 *Let* $\mathsf{extrPreds}$ *be an admissible predicate extraction function,* F *a closed formula,* c *a deterministic command, and* $\mathcal{P}_1, \mathcal{P}_2$ *such that* $\mathsf{extrPreds}(\mathsf{wlp}(c)(F))$ *and* $\mathcal{P}_2 = \mathsf{extrPreds}(F)$*. Then for all abstract states* $s^\# \in \alpha^+[\mathcal{P}_2](F)$*:*

$$\gamma[\mathcal{P}_1](\alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#)))) \subseteq \mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#)) \ .$$

Proof. We show that for all $s_1^\# \in \alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#)))$ we have:

$$\gamma[\mathcal{P}_1](s_1^\#) \subseteq \mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#)) \ .$$

Let $s_1^\# \in \alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#)))$. From Proposition 26 we know that there is a state s_1 such that:

$$s_1^\# = \alpha^+[\mathcal{P}_1](\{s_1\}) = \dot{\bigsqcup_{\vec{o} \in Dom}} \dot{\alpha}^+(s_1, \vec{o}).$$

Thus, we have by definition of γ:

$$\gamma[\mathcal{P}_1](s_1^\#) = \bigcap_{\vec{o}_2 \in Dom} \bigcup_{\vec{o}_1 \in Dom} \dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1))(\vec{o}_2) \ .$$

Furthermore, since $\mathsf{wlp}(c)$ distributes over both joins and meets, we have:

$$\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#)) = \bigcap_{\vec{o} \in Dom} \bigcup_{o^\# \in s^\#} \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#))(\vec{o}) \ .$$

Thus, we need to show the following set inclusion:

$$\bigcap_{\vec{o}_2 \in Dom} \bigcup_{\vec{o}_1 \in Dom} \dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1))(\vec{o}_2) \ \subseteq \ \bigcap_{\vec{o} \in Dom} \bigcup_{o^\# \in s^\#} \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#))(\vec{o}) \ .$$

Instead, we show that the following stronger property holds:

$$(\lambda \vec{o} \in Dom.\, \kappa[\mathcal{P}_1](s_1^\#)) \mathbin{\dot{\cap}} \dot{\bigcup_{\vec{o}_1 \in Dom}} \dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1)) \ \dot{\subseteq} \ \dot{\bigcup_{o^\# \in s^\#}} \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#))$$

Let $\vec{o}_1 \in Dom$. From $s_1 \in \mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s^\#))$ we conclude:

$$s_1 \in \bigcup_{o^\# \in s^\#} \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#))(\vec{o}_1) \ .$$

Thus, there is some $o^\# \in s^\#$ such that

$$s_1 \in \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#))(\vec{o}_1) \ .$$

Thus, we have by definition of $\dot{\alpha}^+(s_1, \vec{o}_1)$ and monotonicity of $\dot{\gamma}$:

$$\dot{\gamma}(\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1)) \ \dot{\subseteq} \ \dot{\gamma}(\dot{\alpha}^+[\mathcal{P}_1](\dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#)))) \ . \tag{1}$$

Furthermore, we have by definition of $\dot{\gamma}$, the fact that for all joins $\bigcap \mathcal{P}$ we have $\dot{\alpha}^+(\dot{\bigcap} \mathcal{P}) \mathrel{\dot{\sqsubseteq}} \dot{\prod} \{ \dot{\alpha}^+(p) \mid p \in \mathcal{P} \}$, and the fact that $\dot{\gamma}$ and $\dot{\mathsf{wlp}}$ distribute over meets:

$$\dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](\mathsf{wlp}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#)))) \ \dot{\subseteq} \ \dot{\bigcap_{p \in \mathcal{P}_2}} \dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](\dot{\mathsf{wlp}}(c)(p^{o^\#(p)}))) \ . \tag{2}$$

From the fact that $\mathsf{extrPreds}$ is admissible follows that for all $p \in \mathcal{P}_2$:

$$\dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](\dot{\mathsf{wlp}}(c)(p^{o^\#(p)}))) \mathbin{\dot{\cap}} (\lambda \vec{o}.\, (\kappa[\mathcal{P}_1])(s_1^\#)) \ \dot{\subseteq} \ \dot{\mathsf{wlp}}(c)(p^{o^\#(p)}) \ . \tag{3}$$

Now, from (1)-(3) follows that for all $p \in \mathcal{P}_2$:

$$\dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1)) \dot{\cap} (\lambda \vec{o}.\, \kappa[\mathcal{P}_1](s_1^\#)) \;\dot{\subseteq}\; \dot{\mathsf{wlp}}(c)(p^{o^\#(p)}) \ .$$

From this follows:

$$\dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1)) \dot{\cap} (\lambda \vec{o}.\, \kappa[\mathcal{P}_1](s_1^\#)) \;\dot{\subseteq}\; \dot{\bigcap_{p \in \mathcal{P}_2}} \dot{\mathsf{wlp}}(c)(p^{o^\#(p)}) \ .$$

Now, from the fact that $\dot{\mathsf{wlp}}$ distributes over meets and the definition of $\dot{\gamma}$ we finally conclude:

$$\dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](s_1, \vec{o}_1)) \dot{\cap} (\lambda \vec{o}.\, \kappa[\mathcal{P}_1](s_1^\#)) \;\dot{\subseteq}\; \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o^\#)$$

which proves our goal. ■

The following proposition together with Theorem 41 states that admissible predicate extraction functions and the abstract post operator post^A guarantee progress of abstraction refinement for havoc-free programs.

Proposition 45 *Let* $\mathsf{extrPreds}$ *be an admissible predicate extraction function. Furthermore, let* F *be a closed formula,* c *a deterministic command, and* $\mathcal{P}_1, \mathcal{P}_2$ *such that* $\mathcal{P}_1 = \mathsf{extrPreds}(\mathsf{wlp}(c)(F))$ *and* $\mathcal{P}_2 = \mathsf{extrPreds}(F)$*. Then*

$$\gamma[\mathcal{P}_2](\mathsf{post}^\mathsf{A}[\mathcal{P}_1, \mathcal{P}_2](c)(\alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(F)))) \models F \ .$$

Proof. It suffices to show that for all $s_1^\# \in \alpha^+(\mathsf{wlp}(c)(F))$ we have:

$$\mathsf{post}^\mathsf{A}[\mathcal{P}_1, \mathcal{P}_2](c)(s_1^\#) \sqsubseteq \alpha^+[\mathcal{P}_2](F) \ .$$

The claim then follows from the fact that post^A distributes over joins, monotonicity of γ, and the fact that $\mathsf{extrPreds}$ is admissible.

Let $s_1^\# \in \alpha^+(\mathsf{wlp}(c)(F))$. If $\gamma[\mathcal{P}_1](s_1^\#) \models \gamma(\mathsf{wlp}(c)(\mathsf{false}))$ then the goal immediately follows from the definition of post^A. Thus, assume $\gamma[\mathcal{P}_1](s_1^\#) \not\models \gamma(\mathsf{wlp}(c)(\mathsf{false}))$. Since $\gamma[\mathcal{P}_1](s_1^\#) \subseteq \mathsf{wlp}(c)(F)$, it follows that there is at least one post state of $\mathsf{wlp}(c)(F)$ under command c that satisfies F. Hence, we know that there exists at least one abstract state $s_2^\#$ in $\alpha^+[\mathcal{P}_2](F)$.

By Lemma 43, and the fact that α^+ is the upper adjoint of a Galois connection we know that α^+ and $\mathsf{wlp}(c)$ distribute over joins. We therefore conclude from the definition of γ:

$$\alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(F)) = \bigsqcup_{s_2^\# \in \alpha^+[\mathcal{P}_2](F)} \alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s_2^\#))) \ .$$

Thus there exists some $s_2^\# \in \alpha^+[\mathcal{P}_2](F)$ such that

$$s_1^\# \sqsubseteq \alpha^+[\mathcal{P}_1](\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s_2^\#))) \ .$$

From monotonicity of γ and Lemma 44 it follows:

$$\gamma[\mathcal{P}_1](s_1^\#) \subseteq \mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s_2^\#)) \ . \tag{1}$$

Now let $o_1^\# \in \mathsf{expand}(s_1^\#)$. By the characterization of α^+ in Proposition 26 there is some s_1 and $\vec{o}_1 \in Dom$ such that $s_1 \models \mathsf{wlp}(c)(F)$, $\alpha[\mathcal{P}_1](\{s_1\}) = s_1^\#$ and $\dot{\alpha}^+(s_1, \vec{o}_1) = o_1^\#$. Furthermore, from (1) and the fact that $\gamma[\mathcal{P}_1] \circ \alpha[\mathcal{P}_1]$ is extensive follows that $s_1 \in \mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s_2^\#))$. By definition of γ and the fact that $\mathsf{wlp}(c)$ distributes over both joins and meets we have:

$$\mathsf{wlp}(c)(\gamma[\mathcal{P}_2](s_2^\#)) = \bigcap_{\vec{o} \in Dom} \bigcup_{o_2^\# \in s_2^\#} \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o_2^\#))(\vec{o}) \ .$$

It follows that $s_1 \in \bigcup_{o_2^\# \in s_2^\#} \dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o_2^\#))(\vec{o}_1)$. Hence, there is some $o_2^\# \in s_2^\#$ such that:

$$\{o_1^\#\} \mathrel{\dot{\sqsubseteq}} \dot{\alpha}^+[\mathcal{P}_1](\dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o_2^\#))) \ .$$

From monotonicity of $\dot{\gamma}$ we can conclude:

$$\dot{\gamma}[\mathcal{P}_1](o_1^\#) \mathrel{\dot{\subseteq}} \dot{\gamma}[\mathcal{P}_1](\dot{\alpha}^+[\mathcal{P}_1](\dot{\mathsf{wlp}}(c)(\dot{\gamma}[\mathcal{P}_2](o_2^\#)))) \ .$$

Following the same chain of reasoning as in the proof of Lemma 44 we conclude that for all $p \in \mathcal{P}_2$:

$$\dot{\gamma}[\mathcal{P}_1](o_1^\#) \mathbin{\dot{\cap}} (\lambda \vec{o}.\, \kappa[\mathcal{P}_1](s_1^\#)) \ \mathrel{\dot{\subseteq}} \ \dot{\mathsf{wlp}}(c)(p^{o_2^\#(p)})$$

which is equivalent to the statement that for all $p \in \mathcal{P}_2$:

$$\dot{\gamma}[\mathcal{P}_1](o_1^\#) \ \mathrel{\dot{\subseteq}} \ \lambda \vec{o}.\, \kappa[\mathcal{P}_1](s_1^\#) \cup \mathsf{wlp}(c)(p^{o_2^\#(p)}(\vec{o}))$$

From the definition of $\dot{\mathsf{wlp}}_\kappa^\#$ we can therefore conclude that for all $p \in \mathcal{P}_2$ with $o_2^\#(p) = \{i\}$:

$$\{o_1^\#\} \ \mathrel{\dot{\sqsubseteq}} \ \dot{\mathsf{wlp}}_\kappa^\#(c)(s_1^\#)(o_{p,i}^\#) \ .$$

From Theorem 37 follows:

$$\dot{\alpha}^{\mathsf{C}} \circ \dot{\mathsf{post}}_\kappa^\#[\mathcal{P}_1, \mathcal{P}_2](c)(s_1^\#) \circ \dot{\gamma}^{\mathsf{C}}(o^\#) \mathrel{\dot{\sqsubseteq}} \{o_2^\#\} \ .$$

Since $o_1^\#$ was chosen arbitrarily in $s_1^\#$, we have by definition of the context-sensitive Cartesian post:

$$\mathsf{post}_\kappa^{\mathsf{C}}[\mathcal{P}_1, \mathcal{P}_2](c)(s_1^\#) \sqsubseteq \{s_2^\#\}$$

from which we finally conclude:

$$\mathsf{post}_\kappa^{\mathsf{C}}[\mathcal{P}_1, \mathcal{P}_2](c)(s_1^\#) \sqsubseteq \alpha^+[\mathcal{P}_2](F)$$

The goal then follows from the definition of $\mathsf{post}^{\mathsf{A}}$. ∎

4.4.2 Progress for General Programs

We proved the progress property for the analysis of programs that do not contain any `havoc` commands. Can we generalize this result to arbitrary programs? The problem with `havoc` commands is that they introduce unbounded nondeterminism in the concrete system. This unbounded nondeterminism is reflected by the weakest liberal preconditions of `havoc` commands:

$$\mathsf{wlp}(\texttt{havoc}(x))(F) = \forall v.\, F[x{:=}v] \quad \text{where } v \notin \mathsf{FV}(F)$$

One way to obtain progress in the presence of unbounded nondeterminism would be to dynamically change the arity of abstraction predicates such that elements in the abstract domain can quantify over additional variables. However, this would unnecessarily complicate our analysis. Instead, we use a simple trick that sidesteps this problem.

Whenever we add a new edge to the ART that is labeled by a command c of the form $\texttt{havoc}(x)$ then we replace c by a deterministic update $x{:=}x'$ where x' is a fresh program variable. Effectively this transformation moves the nondeterminism to the choice of the initial value of x' in the initial states of the corresponding ART path. However, all commands that label edges in the ART are deterministic and we get progress of nested lazy abstraction refinement for general programs.

4.5 Costs and Gains of Automation

We implemented our nested abstraction refinement algorithm in our tool Bohne. We were able to verify complex properties of programs manipulating data structures without manual specification of abstract transformers or abstraction predicates. A detailed presentation of our tool and an overview of our experiments is given in Chapter 6. It is instructive to measure the costs and gains of automation by comparing Bohne to other shape analysis tools. Due to the similarities between three-valued shape analysis [103] and domain predicate abstraction it seems appropriate to compare Bohne with TVLA [79], the implementation of three-valued shape analysis. A detailed overview and analysis of our comparison can be found in Section 6.5.3.

Our experiments indicate that the running time of our analysis is approximately one order of magnitude higher than the running time of TVLA. Almost all running time of the analysis is spent in the underlying decision procedures. Thus, the increased running time is the price we pay for automated computation of abstract transformers and automated abstraction refinement.

On the other hand, the increased degree of automation reduces the burden that is imposed on the user of the analysis. However, this is not the only benefit of an increased automation. Our lazy nested refinement loop seems to achieve the local fine-tuning of the abstraction at the required precision. This targeted precision results in a smaller space consumption of our analysis. The space consumption of Bohne (measured in number of abstract states in the least fixed point) can be significantly smaller than the space consumption of TVLA. One of the main reasons for the lower space consumption is that Cartesian refinement serves as a property-driven focus operation. Our analysis performs materialization of abstract objects and abstract states only when the additional precision is needed to verify a

particular property. In contrast, TVLA's focus operation performs materialization eagerly which potentially leads to an increased number of explored abstract states.

4.6 Further Related Work

The advantages of combining predicate abstraction, abstraction refinement, and shape analysis are clearly demonstrated in lazy shape analysis [18]. Lazy shape analysis performs independent runs of a shape analysis algorithm, whose results are then used to improve the precision of predicate abstraction. In contrast, domain predicate abstraction generalizes predicate abstraction to the point where it itself becomes effective as a shape analysis. This approach makes the benefits of lazy abstraction [53] immediately accessible to shape analysis.

In the previous section we already made a detailed comparison with three-valued shape analysis [103]. We now summarize additional related work. A method for automated generation of predicates using inductive learning has been presented in [82]. Recent techniques also attempt to decrease the space consumption [84,85,102]. However, none of these methods uses counterexample-guided abstraction refinement. Thus, it is up to the user of the analysis to determine when the application of such a technique is appropriate.

Shape analyses based on separation logic [93,94,101], such as [31,42] are typically tailored towards specific data structures and properties. This makes them scale to programs of impressive size [115], but also limits their application domain. Recent techniques introduce some degree of parametricity and allow the analysis to automatically adapt to specific classes of data structures [13]. A similar technique, while not based on separation logic, is described in [77]. None of these methods is based on abstraction refinement.

Shape analysis based on abstract regular tree model checking [25] can take advantage of abstraction refinement techniques that have been developed in this context [24]. In particular, there is an automata-based version of predicate abstraction that can be combined with abstraction refinement and provide progress guarantees. However, these refinement techniques cannot prevent any loss of precision that is caused by the initial encoding of a heap program into tree transducers. Also, this approach focuses on shape invariants of data structures and does not apply to properties such as sortedness.

Indexed predicates [71] use predicates with free variables to infer quantified invariants, similarly to domain predicate abstraction. Heuristics for automatic discovery of indexed predicates are described in [72]. Unlike indexed predicate abstraction, our abstract domain contains disjunctions of universally quantified statements. The presence of disjunctions avoids loss of precision on join points of the control flow graph. This is important in the context of abstraction refinement because it allows to precisely identify spurious error traces in the abstract system.

The SLAM tool [9] uses Cartesian abstraction [8] on top of predicate abstraction. In [5] Ball *et al.* present a technique based on [40] that guarantees progress of abstraction refinement in this context. This technique gradually refines the abstract post towards the most precise abstract post for the current abstract domain, if adding new predicates alone does not rule out a particular spurious error trace. Thus, the spurious error trace is eventually eliminated. Our lazy nested abstraction refinement does not implement the best abstract post. Remarkably it still guarantees the progress property. Our nested refinement

is inspired by materialization in shape analysis, i.e., it implements a property-driven focus operation. In fact, one can think of nested abstraction refinement as an improvement of the focus operator that is used in classical predicate abstraction [8]. The focus operator in [8] eliminates loss of precision under Cartesian abstraction in cases where the best abstract post behaves deterministically. Our nested abstraction refinement also prevents loss of precision in cases where the best abstract post is nondeterministic.

4.7 Conclusion

In this chapter we presented a nested abstraction refinement technique for symbolic shape analysis. This technique enabled us to verify complex properties of a variety of data structure implementations without providing any user assistance other than stating the properties to verify.

Our technique uses spurious error traces to refine both the abstract domain of the analysis and the abstract post operator for that abstract domain. We showed that our nested refinement loop can be viewed as a solution to the problem of materialization in shape analysis and that the refinement of the abstract post operator implements a property-driven focus operation. We further showed that nested refinement guarantees the progress property, i.e., every spurious error trace is eventually eliminated. While at first glance the progress property seems to be merrily of theoretical importance, it was the key for making the analysis practical.

Chapter 5

Field Constraint Analysis

In the last two chapters we have developed techniques that use decision procedures as black boxes in order to obtain a fully automated shape analysis. In this chapter we are concerned with decision procedures for reasoning about data structures in heap programs. These decision procedures provide the missing link to make our shape analysis applicable in practice.

For recursive data structures such as lists and trees it is important to be able to reason about reachability. Reachability properties are useful for expressing constraints on the shapes of data structures as well as for defining precise abstractions. Unfortunately most logics for reasoning about reachability are either undecidable [58], or restrict the class of considered structures. This makes any analysis that depends on such a logic inapplicable to many useful data structures. Among the most striking examples is the restriction on pointer fields in the Pointer Assertion Logic Engine [89]. This restriction states that all fields of the data structure that are not part of the data structure's tree backbone must be functionally determined by the backbone; that is, such fields must be specified by a formula that uniquely determines where they point to. Formally, we have

$$\forall v\ w.\, f(v) = v \ \leftrightarrow\ F(v, w) \tag{5.1}$$

where f is a function representing the field, and F is the defining formula for f. The restriction that F is functional means that, although data structures such as doubly-linked lists with backward pointers can be verified, many other data structures remain beyond the scope of the analysis. This includes data structures where the exact value of pointer fields depends on the history of data structure operations, and data structures that use randomness to achieve good average-case performance, such as skip lists [99]. In such cases, the invariant on the pointer field does not uniquely determine where the field points to, but merely gives a constraint on the field, of the form

$$\forall v\ w.\, f(v){=}w \ \rightarrow\ F(v, w) \tag{5.2}$$

This constraint is equivalent to $\forall v.\, F(v, f(v))$, which states that the function f is a solution of a given binary predicate. The motivation of this chapter is to find a technique that supports reasoning about constraints of this, more general, form. In a search for existing approaches, we have considered structure simulation [57,59], which, intuitively, allows richer

logics to be embedded into existing logics that are known to be decidable, and of which [89] can be viewed as a specific instance. Unfortunately, even the general structure simulation requires definitions of the form

$$\forall v\ w.\, r(v,w) \ \leftrightarrow\ F(v,w)$$

where $r(v,w)$ is the relation being simulated. When the relation $r(v,w)$ is a function, which is the case with most reference fields in programming languages, structure simulation implies the same restriction on the functionality of the defining relation. To handle the general case, an alternative approach therefore appears to be necessary.

Field constraint analysis. This chapter presents field constraint analysis, our approach for analyzing data structures with general constraints of the form (5.2). Field constraint analysis is a proper generalization of the existing approach and reduces to it when the constraint formula F is functional. It is based on approximating the occurrences of field f with its constraint formula F, taking into account the polarity of f, and is always sound. It is expressive enough to verify constraints on pointers in data structures such as two-level skip lists. The applicability of our field constraint analysis to nondeterministic field constraints is important because many complex properties have useful nondeterministic approximations. Yet despite this fundamentally approximate nature of field constraints, we were able to prove its completeness for some important special cases. Field constraint analysis naturally combines with structure simulation, as well as with our symbolic approach to shape analysis presented in Chapter 3 and 4. Our presentation and current implementation are in the context of the monadic second-order logic (MSOL) over trees [63], but our results extend to other logics. We therefore view field constraint analysis as a useful component of shape analysis approaches that makes shape analysis applicable to a wider range of data structures.

Contributions. In this chapter we make the following contributions:

- We introduce an **algorithm** (Figure 5.7) that uses field constraints to eliminate derived fields from verification conditions.
- We prove that the algorithm is both **sound** (Theorem 48) and, in certain cases, **complete**. The completeness applies not only to deterministic fields (Theorem 50), but also to the preservation of field constraints themselves over loop-free code (Theorem 56). The last result implies a complete technique for checking that field constraints hold, if the programmer adheres to a discipline of maintaining them, e.g., at the beginning of each loop.
- We describe how to combine our algorithm with our symbolic shape analysis to **infer loop invariants**.

5.1 Examples

We next explain our field constraint analysis with a set of examples. The doubly-linked list example shows that our analysis handles, as a special case, the ubiquitous back pointers of data structures. The skip list example shows how field constraint analysis handles

nondeterministic field constraints on derived fields, and how it can infer loop invariants. Finally, the students example illustrates inter-data-structure constraints, which are simple but useful for high-level application properties.

5.1.1 Doubly-Linked Lists with Iterators

This section presents a class implementing doubly-linked lists with built-in iterators. This example illustrates the usefulness of field constraints for specifying pointers that form doubly-linked structures.

Our doubly-linked list implementation is a data structure with operations add and remove that insert and remove elements from the list, as well as the operations initIter, nextIter, and lastIter for manipulating the iterator built into the list. We have verified all these operations using our system; we here present only the remove operation. Figure 5.2 depicts an instance of the data structure. It consists of a list of Node objects which are connected via fields next and prev and the list's *head object* with two fields first and current. Field first points to the first Node object in the list and field current indicates the current position of the iterator in the list.

The behavior of method remove is given by a procedure contract. The contract uses two sets: content, which contains the set of Node objects in the list, and iter, which specifies the set of elements that remain to be iterated over. These two sets abstractly characterize the behavior of operations, allowing the clients to soundly reason about the hidden implementation of the data structure. The system verifies that the implementation conforms to the specification, using the definitions of sets content and iter. These definitions are expressed in a subset of Isabelle/HOL [92] formulae that can be translated into monadic second-order logic [63]. The set content is defined as the set of all objects reachable from the first field of the head object followed by arbitrary many next fields. The set iter is defined correspondingly by taking field current instead of field first.

In addition to the procedure contract there are four data structure invariants specified. The definitions of sets content and iter are hidden from the data structure clients. Thus, without exposing additional information clients would be unable to relate these two sets. Therefore, the first invariant is a public invariant that expresses that the set of nodes that remain to be iterated over is always a subset of the content of the list. The remaining representation invariants are private to the data structure implementation. The invariant tree [first, next] expresses that fields first from class List and field next from class Node are *backbone fields* that form a forest in the heap. The third invariant is a field constraint on field prev. It expresses that field prev is the inverse of field next. The last invariant indicates that all objects which are not contained in the any instance of the data structure are isolated: they have no outgoing next pointers.

Our system verifies that the remove procedure implementation conforms to its specification as follows. The system expands the modifies clause into a frame condition, which it conjoins with the ensures clause. This frame condition expresses that only the content and iter sets of other instances of the doubly-linked list data structure are not modified. Next, it conjoins the data structure invariants to both the requires and ensures clause. The resulting pre- and postcondition are expressed in terms of the sets Content and Iter, so the system applies the definitions of the sets to obtain pre and postcondition expressed only

```
public final class Node {
    public /*: claimedby DLLIter */ Node next;
    public /*: claimedby DLLIter */ Node prev;
}

public class DLLIter
{
   private Node first, current;

   /*: public specvar content :: objset;
       public specvar iter :: objset;

       private vardefs "content == {x. x ≠ null ∧ next* first x}";

       private vardefs "iter == {x. x ≠ null ∧ next* current x}";

       public invariant "iter ⊆ content";

       invariant "tree [first, next]";

       invariant "∀ x y. prev x = y →
                          (y ≠ null → next y = x) ∧
                          (y = null ∧ x ≠ null → (∀ z. next z ≠ x))";

       invariant "∀ n (∀ I. I ∈ DLLIter → n ∉ content I) → next n = null";
   */

   public void remove(Node n)
   /*: requires "n ∈ content"
       modifies content, iter
       ensures "content = old content − {n} ∧ iter = old iter − {n}"
   */
   {
      if (n==current) current = current.next;
      if (n==first) first = first.next;
      else n.prev.next = n.next;
      if (n.next != null) n.next.prev = n.prev;
      n.next = null;
      n.prev = null;
   }
}
```

Figure 5.1: Iterable lists implementation and specification

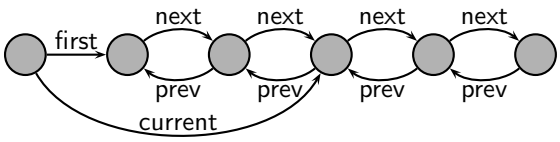


Figure 5.2: An instance of a doubly-linked list with iterator

in terms of fields first, current, next, and prev. It then uses standard weakest precondition computation [1] to generate a verification condition that captures the correctness of remove.

To decide the resulting verification condition, our system analyzes the verification condition and recognizes assumptions of the form:

1. tree [...] and
2. $\forall$ x y. f x = y $\rightarrow$ F(x,y).

These assumptions enable an encoding of the verification condition into monadic second-order logic over trees [63]. The details of this encoding are described in Section 6.3. The first form of assumptions determines the backbone fields of the data structure. The second form determines the derived fields and their constraints expressed in terms of backbone fields. The system uses the constraints on the derived fields to reduce the verification condition to a formula expressible only in terms of the backbone fields. (This elimination is given by the algorithm in Figure 5.7.) Because the backbone fields form a tree, the system can decide the resulting formula using monadic second-order logic over trees. In our case, fields first and next are the backbone fields and field prev is a derived field. The invariants in our example do not determine whether field current is a backbone field or a derived field. In this case the system handles field current as a derived field with a trivial field constraint of the form

$$\forall \text{ x y. current x} = \text{y} \rightarrow (\text{x} = \text{null} \rightarrow \text{y} = \text{null}).$$

While reasoning over such fields which are completely unconstrained is generally incomplete, our system is still able to prove the verification conditions generated for our example.

We note that a first implementation of the doubly-linked list with an iterator was in the context of the Hob system [75]. This implementation was verified using a Hob plugin that relies on the Pointer Assertion Logic Engine tool [89]. What distinguishes field constraint analysis from the previous Hob analysis based on PALE is the ability to handle the cases where the field constraints are nondeterministic, such as the trivial field constraint generated for field current. In the examples that follow, we illustrate the usefulness of nondeterministic field constraints for data structure specification. Additionally, we show how field constraints are combined with our symbolic shape analysis to synthesize loop invariants.

5.1.2 Skip List

We next present the analysis of a two-level skip list. Skip lists [99] support logarithmic average-time access to elements by augmenting a linked list with sublists that skip over some of the elements in the list. The two-level skip list is a simplified implementation of a skip list, which has only two levels: the list containing all elements, and a sublist of this list. Figure 5.3 presents an example two-level skip list. Our implementation uses the next field to represent the main list, which forms the backbone of the data structure, and uses the derived nextSub field to represent a sublist of the main list. We focus on the add procedure, which inserts an element into an appropriate position in the skip list. Figure 5.4 presents the implementation of add, which first searches through nextSub links to get an estimate of the position of the entry, then finds the entry by searching through next links,

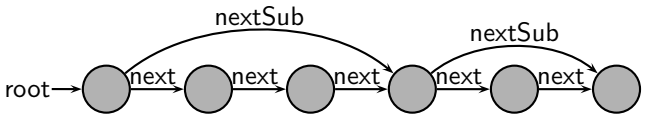


Figure 5.3: An instance of a two-level skip list

and inserts the element into the main next-linked list. Optionally, the procedure also inserts the element into the nextSub list, which is modelled using a nondeterministic choice and is an abstraction of the insertion with certain probability in the original implementation. The contract for add indicates that add always inserts the element into the set of elements stored in the list. The abstract set content is defined as the set of nodes reachable from root.

The skip list implementation has three representation invariants. The first invariant defines next as the backbone field of the data structure. The second invariant is a field constraint on the field nextSub, which defines it as a derived field. It expresses that nextSub points from x to some object y reachable via next fields starting from the next successor of x. Note that the constraint for this derived field is nondeterministic, because it only states that if x.nextSub=y, then there exists a path of length at least one from x to y along next fields, without indicating where nextSub points. Indeed, the simplicity of the skip list implementation stems from the fact that the position of nextSub is not uniquely given by next; it depends not only on the history of invocations, but also on the random number generator used to decide when to introduce new nextSub links. The ability to support such nondeterministic constraints is what distinguishes our approach from approaches that can only handle deterministic fields.

Our analysis successfully verifies that add preserves all invariants, including the nondeterministic field constraint on nextSub. While doing so, the analysis takes advantage of these invariants as well, as is usual in assume/guarantee reasoning. In this example, the analysis is able to infer the loop invariants in add using our symbolic shape analysis.

5.1.3 Students and Schools

Our next example illustrates the power of nondeterministic field constraints. This example contains two linked lists: one containing students and one containing schools. Each Elem object may represent either a student or a school; students have a pointer to the school which they attend. Both students and schools use the next backbone pointer to indicate the next student or school in the relevant linked list. Figure 5.5 presents an example of the data structure.

The specification and implementation of the data structure is given in Figure 5.6. The method addStudent adds a student to the student list and associates it with a school that is supposed to be already contained in the school data structure. The specification uses the abstract sets ST and SC. The specification variable ST denotes all students, that is, all Elem objects reachable from the root reference students through next fields. The specification variable SC denotes all schools, that is, all Elem objects reachable from schools. The class further defines several representation invariants. The first two invariants state disjointness properties: no objects are shared between ST and SC (if an object is reachable from schools through next fields, then it is not reachable from students through next fields, and vice-

```
public final class Node {
   public /*: claimedby Skiplist */ Node next;
   public /*: claimedby Skiplist */ Node nextSub;
   public /*: claimedby Skiplist */ int value;
}

public class Skiplist
{
    private static Node root;

    /*:
      public static specvar content :: objset;

      private vardefs "content == {x. x ≠ null ∧ next* root x}";

      invariant "tree [next]";
      invariant "∀ x y. nextSub x = y → next* (next x) y";
      invariant "∀ x. x ≠ null ∧ next* root x →
                      next x = null ∧ (∀ y. y ≠ null → next y ≠ x)";
    */

   public static void add(Node e)
   /*: requires "e ≠ null ∧ e ∉ content"
       modifies content
       ensures "content = old content ∪ {e}"
   */
    {
        if (root == null) {
            root = e;
            return;
        }
        int v = e.value;
        Node sprev = root;
        Node scurrent = root.nextSub;
        while ((scurrent != null) && (scurrent.value < v)) {
           sprev = scurrent; scurrent = scurrent.nextSub;
        }
        Node prev = sprev;
        Node current = sprev.next;
        while ((current != scurrent) && (current.value < v)) {
           prev = current; current = current.next;
        }
        e.next = current; prev.next = e;
        boolean nondet;
        if (nondet) {
          sprev.nextSub = e; e.nextSub = scurrent;
        } else {
          e.nextSub = null;
        }
    }
}
```

Figure 5.4: Skip list example

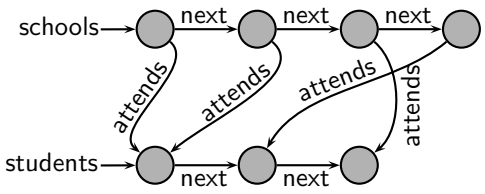


Figure 5.5: Students data structure instance

versa). The third invariant defines field next as the backbone field of the data structure. The fourth invariant states that if an object x is not in either ST or SC, then its next field is set to null, and no object points to x.

The last invariant is a field constraint on the attends field: it states that for any student, field attends points to some (undetermined) element of the SC set of schools. Note that this goes beyond the power of previous analyses, which required the identity of the school pointed to by the student be functionally determined by the identity of the student. The example therefore illustrates how our analysis eliminates a key restriction of previous approaches—certain data structures exhibit properties that the logics in previous approaches were not expressive enough to capture. In general, previous approaches could express and verify properties that were, in some sense, more restrictive than the properties of many data structures that we would like to implement. Because our analysis supports properties that express the correct level of partial information (for example, that a field points to some undetermined object within a set of objects), it is able to successfully analyze these kinds of data structures.

5.2 Field Constraint Analysis

This section presents the field constraint analysis algorithm and proves its soundness as well as, for some important cases, completeness.

5.2.1 Field Constraints

We consider a logic $\mathcal{L}$ over a signature $\Sigma = (B, C, ty)$ as described in Section 2.2.1 where Σ consists of unary function symbols $f \in \mathrm{Fld}$ corresponding to fields in data structures and constant symbols $x \in \mathrm{Var}$ corresponding to reference variables. For simplicity we assume that there is only one other type constant than bool and that D is the domain of that type constant. The extension to multisorted logics is straightforward.

We use monadic second-order logic (MSOL) over trees as our working example, but in general we only require $\mathcal{L}$ to support conjunction, implication and equality reasoning. For a formula F in $\mathcal{L}$, we denote by $\mathsf{Fields}(F) \subseteq C$ the set of all fields occurring in F.

We assume that $\mathcal{L}$ is decidable over some set of well-formed structures and we assume that this set of structures is expressible by a closed formula I in $\mathcal{L}$. We call I the *simulation invariant* [59]. For simplicity, we consider the simulation itself to be given by the restriction of a structure to the fields in $\mathsf{Fields}(I)$, i.e., we assume that there exists a decision procedure for checking validity of implications of the form $I \rightarrow F$ where F is a formula such that

```
class Elem {
    public /*: claimedby Students */ Elem attends;
    public /*: claimedby Students */ Elem next;
}
class Students {
    private static Elem students;
    private static Elem schools;

    /*:
      public static specvar ST :: objset;
      vardefs "ST == {x. x ≠ null ∧ next* students x}";
      public static specvar SC :: objset;
      vardefs "SC == {x. x ≠ null ∧ next* schools x}";

      public invariant "null ∉ (ST ∪ SC)";
      public invariant "ST ∩ SC = ∅";

      invariant "tree [next]";

      invariant "∀ x y. attends x = y → (x ∈ ST → y ∈ SC) ∧
                                        (x ∉ ST → y = null)";

      invariant "∀ x. x ∉ (ST ∪ SC ∪ {null}) →
                      (∀ y. y ∉ null → next y ∉ x) ∧ next x = null";
     */

   public static void addStudent(Elem st, Elem sc)
   /*: requires "st ∉ (ST ∪ SC ∪ {null}) ∧ sc ∈ SC"
       modifies ST
       ensures "ST = old ST ∪ {st}"
   */
   {
      st.attends = sc;
      st.next = students;
      students = st;
   }
}
```

Figure 5.6: Students and schools example

$\mathsf{Fields}(F) \subseteq \mathsf{Fields}(I)$. In our running example, MSOL, the simulation invariant I states that the fields in $\mathsf{Fields}(I)$ span a forest.

We call a field $f \in \mathsf{Fields}(I)$ a *backbone field*, and call a field $f \in \mathsf{Fld} \setminus \mathsf{Fields}(I)$ a *derived field*. We refer to the decision procedure for formulae with fields in $\mathsf{Fields}(I)$ over the set of structures defined by the simulation invariant I as *the underlying decision procedure*. Field constraint analysis enables the use of the underlying decision procedure to reason about nondeterministically constrained derived fields. We state invariants on the derived fields using field constraints.

Definition 46 (Field constraints on derived fields) *A* field constraint FC_f *for a sim-*

ulation invariant I and a derived field f is a formula of the form

$$\mathsf{FC}_f \;\equiv\; \forall v\, w.\, f(v) = w \rightarrow \mathsf{F}_f(v, w)$$

where F_f is a formula with two free variables such that (1) $\mathsf{Fields}(\mathsf{F}_f) \subseteq \mathsf{Fields}(I)$, and (2) F_f is total with respect to I, i.e., $I \models \forall v.\, \exists y.\, \mathsf{F}_f(v, w)$.

We call the constraint FC_f deterministic *if F_f is deterministic with respect to I, i.e.,*

$$I \models \forall v\, w\, u.\, \mathsf{F}_f(v, w) \wedge \mathsf{F}_f(v, u) \rightarrow w = u \;\;.$$

We write FC for the conjunction of FC_f for all derived fields f.

Note that Definition 46 covers arbitrary constraints on a field because FC_f is equivalent to $\forall v.\, \mathsf{F}_f(v, f(v))$.

The totality condition (2) is not required for the soundness of our approach, only for its completeness, and rules out invariants equivalent to "false". The condition (2) does not involve derived fields and can therefore be checked automatically using a single call to the underlying decision procedure.

Our goal is to check validity of formulae of the form $I \wedge FC \rightarrow G$, where G is a formula with possible occurrences of derived fields. If G does not contain any derived fields then there is nothing to do because in that case checking validity immediately reduces to the validity problem without field constraints, as given by the following lemma.

Lemma 47 *Let G be a formula such that $\mathsf{Fields}(G) \subseteq \mathsf{Fields}(I)$.*
Then $I \models G$ iff $I \wedge FC \models G$.

Proof. The left-to-right direction follows immediately. For the right-to-left direction assume that $I \wedge FC \rightarrow G$ is valid. Let $\mathcal{A}$ be a structure such that $\mathcal{A} \models I$. By totality of all field constraints in FC there exists a structure $\mathcal{A}'$ such that $\mathcal{A}' \models I \wedge FC$ and $\mathcal{A}'$ differs from $\mathcal{A}$ only in the interpretation of derived fields. Since $\mathsf{Fields}(G) \subseteq \mathsf{Fields}(I)$ and I contains no derived fields we have for any assignment β that $\mathcal{A}', \beta \models G$ implies $\mathcal{A}, \beta \models G$. ∎

To check validity of $I \wedge FC \rightarrow G$, we therefore proceed as follows. We first obtain a formula G' from G by eliminating all occurrences of derived fields in G. Next, we check validity of G' with respect to I. In the case of a derived field f that is defined by a deterministic field constraint, occurrences of f in G can be eliminated by flattening the formula and substituting each term $f(x) = y$ by $\mathsf{F}_f(x, y)$. However, in the general case of nondeterministic field constraints such a substitution is only sound for negative occurrences of derived fields, since the field constraint gives an over-approximation of the derived field. Therefore, a more sophisticated elimination algorithm is needed.

5.2.2 Eliminating Derived Fields

Figure 5.7 presents our algorithm Elim for elimination of derived fields. Consider a derived field f and let $F \equiv \mathsf{F}_f$. The basic idea of Elim is that we can replace an occurrence $G(f(v))$ of f by a new variable w that satisfies $F(v, w)$, yielding a stronger formula $\forall w.\, F(v, w) \rightarrow G(w)$. As an improvement, if G contains two occurrences $f(v_1)$ and $f(v_2)$, and if v_1 and v_2 evaluate to the same value, then we attempt to replace $f(v_1)$ and $f(v_2)$ with the same

$$
\begin{array}{rcl}
S & - & \text{a term or a formula} \\
\mathsf{Terms}(S) & - & \text{terms occurring in } S \\
\mathsf{FV}(S) & - & \text{variables free in } S \\
\mathsf{Ground}(S) & = & \{t \in \mathsf{Terms}(S).\ \mathsf{FV}(t) \subseteq \mathsf{FV}(S)\} \\
\mathsf{Derived}(S) & - & \text{derived function symbols in } S
\end{array}
$$

proc $\mathsf{Elim}(G) = \mathsf{elim}(G, \emptyset)$

proc $\mathsf{elim}(G$: formula in negation normal form;
K: set of (variable,field,variable) triples)**:**

let $T = \{f(t) \in \mathsf{Ground}(G).\ f \in \mathsf{Derived}(G) \wedge \mathsf{Derived}(t) = \emptyset\}$
if $T \neq \emptyset$ **do**
choose $f(t) \in T$
choose v, w fresh first-order variables
let $F = \mathsf{F}_f$
let $F_1 = F(v, w) \wedge \bigwedge_{(v_i, f, w_i) \in K} (v = v_i \rightarrow w = w_i)$
let $G_1 = G[f(t) := w]$
return $\forall v.\, v = t \rightarrow \forall w.\, (F_1 \rightarrow \mathsf{elim}(G_1, K \cup \{(v, f, w)\}))$
else case G **of**
| $Qv.\ G_1$ **where** $Q \in \{\forall, \exists\}$**:**
return $Qv.\ \mathsf{elim}(G_1, K)$
| $G_1\ op\ G_2$ **where** $op \in \{\wedge, \vee\}$**:**
return $\mathsf{elim}(G_1, K)\ op\ \mathsf{elim}(G_2, K)$
| **else return** G

Figure 5.7: Derived-field elimination algorithm

value. Elim implements this idea using the set K of triples (v, f, w) to record previously assigned values for $f(v)$. Elim runs in time $O(n^2)$ where n is the size of the formula and produces an at most quadratically larger formula. Elim accepts formulae in negation normal form, where all negation signs apply to atomic formulae. We generally assume that each quantifier $Q\,u$ binds a variable u that is distinct from other bound variables and distinct from the free variables of the entire formula. The algorithm Elim is presented as acting on first-order formulae, but is also applicable to checking validity of quantifier-free formulae because it only introduces universal quantifiers which can be replaced by Skolem constants. The algorithm is also applicable to multisorted logics, and, by treating sets of elements as a new sort, to MSOL. To make the discussion simpler, we consider a deterministic version of Elim where the nondeterministic choices of variables and terms are resolved by some arbitrary, but fixed, linear ordering on terms. We write $\mathsf{Elim}(G)$ to denote the result of applying Elim to a formula G.

The correctness of Elim is given by Theorem 48. The proof of Theorem 48 relies on the

monotonicity of logical operations and quantifiers in negation normal form of a formula.

Theorem 48 (Soundness) *The algorithm* Elim *is sound: if* $I \wedge FC \models \mathsf{Elim}(G)$, *then* $I \wedge FC \models G$. *What is more,* $I \wedge FC \wedge \mathsf{Elim}(G) \models G$.

Proof. By induction on the first argument G of elim we prove that, for all finite K,

$$I \wedge FC \wedge \mathsf{elim}(G, K) \wedge \bigwedge_{(v_i, f_i, w_i) \in K} \mathsf{F}_{f_i}(v_i, w_i) \models G$$

For $K = \emptyset$ we obtain $I \wedge FC \wedge \mathsf{Elim}(G) \models G$, as desired. In the inductive proof, the cases when $T = \emptyset$ are straightforward. The case $f(t) \in T$ uses the fact that if $\mathcal{A}, \beta \models G[f(t) := w]$ and $\mathcal{A}, \beta \models f(t) = w$, then $\mathcal{A}, \beta \models G$. ∎

5.2.3 Completeness

We now analyze the classes of formulae G for which Elim is *complete*.

Definition 49 *We say that algorithm* Elim *is complete for* (FC, G) *if and only if*

$$I \wedge FC \models G \text{ implies } I \wedge FC \models \mathsf{Elim}(G) \ .$$

Note that we cannot hope to achieve completeness for arbitrary constraints FC. Indeed, if we let $FC \equiv \mathsf{true}$, then FC imposes no constraint whatsoever on the derived fields, and reasoning about the derived fields becomes reasoning about uninterpreted function symbols, that is, reasoning in unconstrained predicate logic. Such reasoning is undecidable not only for monadic second-order logic, but also for much weaker fragments of first-order logic [47]. Despite these general observations, we have identified two cases important in practice for which Elim is complete (Theorem 50 and Theorem 56).

Theorem 50 expresses the fact that, in the case where all field constraints are deterministic, Elim is complete (and then it reduces to previous approaches [59,89] that are restricted to the deterministic case). The proof of Theorem 50 uses the assumption that F is total and functional to conclude $\forall v\ w.\, F(v, w) \rightarrow f(v) = w$, and then uses an inductive argument similar to the proof of Theorem 48.

Theorem 50 (Completeness for deterministic field constraints) *Algorithm* Elim *is complete for* (FC, G) *when each field constraint in* FC *is deterministic.*
What is more, $I \wedge FC \wedge G \models \mathsf{Elim}(G)$.

Proof. Consider a field constraint $F \equiv \mathsf{F}_f$. Let $\mathcal{A}$ be a structure and β an assignment such that $\mathcal{A}, \beta \models I \wedge FC \wedge F(v, w)$. Because $\mathcal{A}, \beta \models F(v, f(v))$ and F is deterministic by assumption, we have $\mathcal{A}, \beta \models f(v) = w$. It follows that $I \wedge FC \wedge F(v, w) \models f(v) = w$. We then prove by induction on the argument G of elim that, for all finite K,

$$I \wedge FC \wedge G \wedge \bigwedge_{(v_i, f_i, w_i) \in K} f_i(v_i) = w_i \models \mathsf{elim}(G, K)$$

For $K = \emptyset$ we obtain $I \wedge FC \wedge G \models \mathsf{Elim}(G)$, as desired. The inductive proof is similar to the proof of Theorem 48. In the case $f(t) \in T$, we consider a structure $\mathcal{A}$ and assignment β such

that $\mathcal{A}, \beta \models I \wedge FC \wedge G \wedge \bigwedge_{(v_i, f_i, w_i) \in K} f_i(v_i) = w_i$. Consider any $\bar{v}, \bar{w} \in D$ and assignment β' such that: 1) $\beta' = \beta[v \mapsto \bar{v}, w \mapsto \bar{w}]$, 2) $\mathcal{A}, \beta' \models v = t$, 3) $\mathcal{A}, \beta' \models F(v, w)$ and 4) $\mathcal{A}, \beta' \models v = v_i \rightarrow w = w_i$ for all $(v_i, f, w_i) \in K$. To show $\mathcal{A}, \beta' \models \mathsf{elim}(G_1, K \cup \{(v, f, w)\})$, we consider a modified structure $\mathcal{A}_1 = \mathcal{A}[f\bar{v} := \bar{w}]$ which is like $\mathcal{A}$ except that the interpretation of f at $\bar{v}$ is $\bar{w}$. By $\mathcal{A}, \beta' \models F(v, w)$ we conclude $\mathcal{A}_1 \models I \wedge FC$. By $\mathcal{A}, \beta' \models v = w_i \rightarrow w = w_i$, we conclude $\mathcal{A}_1, \beta' \models \bigwedge_{(v_i, f_i, w_i) \in K} f_i(v_i) = w_i$ as well. Because $I \wedge FC \wedge F(v, w) \models f(v) = w$, we conclude $\mathcal{A}_1, \beta' \models f(v) = w$. Because $\mathcal{A}, \beta' \models v = t$ and $\mathsf{Derived}(t) = \emptyset$, we have $\mathcal{A}_1, \beta' \models v = t$ so from $\mathcal{A}, \beta' \models G$ we conclude $\mathcal{A}_1, \beta' \models G_1$ where $G_1 = G[f(t) := w]$. By induction hypothesis we then conclude $\mathcal{A}_1, \beta' \models \mathsf{elim}(G_1, K \cup \{(v, f, w)\}$. Then also $\mathcal{A}, \beta' \models \mathsf{elim}(G_1, K \cup \{(v, f, w)\}$ because the result of elim does not contain f. Because $\bar{v}, \bar{w}$ were arbitrary, we conclude $\mathcal{A}, \beta \models \mathsf{elim}(G, K)$. ∎

We next turn to completeness in the cases that admit nondeterminism of derived fields. Theorem 56 states that our algorithm is complete for derived fields introduced by the weakest precondition operator to a class of postconditions that includes field constraints. This result is very important in practice. For example, when we used a previous version of an elimination algorithm that was incomplete, we were not able to verify the skip list example in Section 5.1.2. To formalize our completeness result, we introduce two classes of well-behaved formulae: *nice formulae* and *pretty nice formulae*.

Definition 51 (Nice Formulae) *A formula G is a* nice formula *if each occurrence of each derived field f in G is of the form $f(t)$, where $t \in \mathsf{Ground}(G)$.*

Nice formulae generalize the notion of quantifier-free formulae by disallowing quantifiers only for variables that are used as arguments to derived fields. Lemma 52 shows that the elimination of derived fields from nice formulae is complete. The intuition behind Lemma 52 is that if $I \wedge FC \models G$, then for the choice of w_i such that $F(v_i, w_i)$ we can find an interpretation of the function symbol f such that $f(v_i) = w_i$, and $I \wedge FC$ holds, so G holds as well, and $\mathsf{Elim}(G)$ evaluates to the same truth value as G.

Lemma 52 Elim *is complete for (FC, G) if G is a nice formula.*

Proof. Lemma 52 Let G be a nice formula. To show that $I \wedge FC \models G$ implies $I \wedge FC \models \mathsf{Elim}(G)$, let $I \wedge FC \models G$ and let $f_1(t_1), \ldots, f_n(t_n)$ be the occurrences of derived fields in G. By assumption, $t_1, \ldots, t_n \in \mathsf{Ground}(G)$ and $\mathsf{Elim}(G)$ is of the form

$$\begin{array}{l} \forall v_1\, w_1.\, v_1 = t_1 \rightarrow (F_1^1 \wedge \\ \quad \forall v_2\, w_2.\, v_2 = t_2' \rightarrow (F_1^2 \wedge \\ \qquad \ldots \\ \qquad \forall v_n, w_n.\, v_n = t_n' \rightarrow (F_1^n \wedge G_0) \ldots)) \end{array}$$

where t_i' differs from t_i in that some of its subterms may be replaced by variables w_j for $j < i$. Here $F^i = \mathsf{F}_{f_i}$ and

$$F_1^i = F^i(v_i, w_i) \wedge \bigwedge_{j<i, f_j = f_i} (v_i = v_j \rightarrow w_i = w_j).$$

Consider a model $\mathcal{A}$ of $I \wedge FC$, we show $\mathcal{A}$ is a models of $\mathsf{Elim}(G)$. Consider any assignment β to variables v_i, w_i for $1 \leq i \leq n$. If any of the conditions $v_i = t_i$ or F_1^i are false for this assignment, then $\mathsf{Elim}(G)$ is true because these conditions are on the left-hand side of an implication. Otherwise, conditions $F_1^i(v_i, w_i)$ hold, so by definition of F_1^i, if $\beta(v_i) = \beta(v_j)$, then $\beta(w_i) = \beta(w_j)$. Therefore, for each distinct function symbol f_j there exists a function $\bar{f}_j$ such that $\bar{f}_j(\beta(v_i)) = \beta(w_i)$ for $f_j = f_i$. Because $\mathcal{A}, \beta \models F^i(v_i, w_i)$ holds and each F_f is total, we can define such $\bar{f}_j$ so that FC holds. Let $\mathcal{A}' = \mathcal{A}[f_j \mapsto \bar{f}_j]_j$ be a model that differs from $\mathcal{A}$ only in that all f_j are interpreted as $\bar{f}_j$. Then $\mathcal{A}' \models I$ because I does not mention derived fields and $\mathcal{A}' \models FC$ by construction. We therefore conclude $\mathcal{A}' \models G$. Because $\mathcal{A}, \beta \models v_i = t_i$ and $\mathsf{Derived}(t_i) = \emptyset$ we have $\mathcal{A}', \beta \models v_i = t_i$. Using this fact, as well as $\bar{f}_j(\beta(v_i)) = \beta(w_i)$, by induction on subformulas of G_0 we conclude that G_0 has the same truth value as G for $\mathcal{A}'$ and β, so $\mathcal{A}', \beta \models G_0$. Because G_0 does not contain derived function symbols, $\mathcal{A}, \beta \models G_0$ as well. Because β was arbitrary, we conclude $\mathcal{A} \models \mathsf{Elim}(G)$. This completes the proof.

Remark. Note that it is not the case that a stronger statement $I \wedge FC \wedge G \models \mathsf{Elim}(G)$ holds. For example, take $FC \equiv \mathsf{true}$, and $G \equiv f(a) = b$. Then $\mathsf{Elim}(G)$ is equivalent to $\forall w.\, w = b$ and it is not the case that $I \wedge f(a) = b \models \forall w.\, w = b$. ∎

Definition 53 (Pretty Nice Formulae) *The set of* pretty nice formulae *is defined inductively by 1) a nice formula is pretty nice; 2) if G_1 and G_2 are pretty nice, then $G_1 \wedge G_2$ is pretty nice; 3) if G is pretty nice and v is a first-order variable, then $\forall v.\, G$ is pretty nice.*

Pretty nice formulae therefore additionally admit universal quantification over arguments of derived fields. Define function skolem as follows: 1) $\mathsf{skolem}(\forall v.\, G) = G$; 2) $\mathsf{skolem}(G_1 \wedge G_2) = \mathsf{skolem}(G_1) \wedge \mathsf{skolem}(G_2)$; and 3) $\mathsf{skolem}(G) = G$ if G is not of the form $\forall v.\, G$ or $G_1 \wedge G_2$.

Lemma 54 *The following observations hold:*

1. *each field constraint FC_f is a pretty nice formula;*
2. *if G is a pretty nice formula, then $\mathsf{skolem}(G)$ is a nice formula and $H \models G$ iff $H \models \mathsf{skolem}(G)$ for any set of formulae H.*

The next Lemma 55 shows that pretty nice formulae are closed under wlp; the lemma follows from the conjunctivity of the weakest precondition operator.

Lemma 55 *Let c be a command. If G is a nice formula, then $\mathsf{wlp}(c)(G)$ is a nice formula. If G is a pretty nice formula, then $\mathsf{wlp}(c)(G)$ is equivalent to a pretty nice formula.*

Lemmas 55, 54, 52, and 47 imply our main theorem, Theorem 56. Theorem 56 implies that Elim is a complete technique for checking preservation (over straight-line code) of field constraints, even if they are conjoined with additional pretty nice formulae. Elimination is also complete for data structure operations with loops as long as the necessary loop invariants are pretty nice.

Theorem 56 (Completeness for preservation of field constraints) *Let G be a pretty nice formula, FC a conjunction of field constraints, and c a command. Then*

$$I \wedge FC \models \mathsf{wlp}(c)(G \wedge FC) \quad \textit{iff} \quad I \models \mathsf{Elim}(\mathsf{wlp}(c)(\mathsf{skolem}(G \wedge FC))).$$

$$\mathsf{FC}_{nextSub} \equiv \forall v_1\ v_2.\ nextSub(v_1) = v_2 \rightarrow next^+(v_1, v_2)$$

$$\begin{aligned} G &\equiv \mathsf{wlp}((e.nextSub{:=}\, root.nextSub\ ;\ e.next{:=}\, root), \mathsf{FC}_{nextSub}) \\ &\equiv \forall u_1\ u_2.\ \ nextSub[e{:=}\, nextSub(root)](u_1) = u_2 \ \rightarrow \\ &\qquad\qquad (next[e{:=}\, root])^+(u_1, u_2) \end{aligned}$$

$$\begin{aligned} G' \equiv\ & \mathsf{skolem}(\mathsf{Elim}(G)) \equiv \\ & v_1 = root \rightarrow next^+(v_1, w_1) \rightarrow \\ & \quad v_2 = v_1 \rightarrow next^+[e{:=}\, w_1](v_2, w_2) \wedge (v_2 = v_1 \rightarrow w_2 = w_1) \rightarrow \\ & \quad\quad w_2 = u_2 \rightarrow (next[e{:=}\, root])^+(u_1, u_2) \end{aligned}$$

Figure 5.8: Elimination of derived fields from a pretty nice formula. The notation $next^+$ denotes the irreflexive transitive closure of predicate $next(v) = w$.

Example 57 The example in Figure 5.8 demonstrates the elimination of derived fields using algorithm Elim. It is inspired by the skip list example from Section 5.1.

The formula G expresses the preservation of field constraint $\mathsf{FC}_{nextSub}$ for updates of fields *next* and *nextSub* that insert e in front of *root*. This formula is valid under the assumption that $\forall v.\, next(v) \neq e$ holds. The algorithm Elim first replaces the inner occurrence $nextSub(root)$ and then the outer occurrence of *nextSub*. Theorem 56 implies that the resulting formula $\mathsf{skolem}(\mathsf{Elim}(G))$ is valid under the same assumption as the original formula G.

♦

Limits of completeness. In our implementation, we have successfully used Elim in the context of MSOL, where we encode transitive closure using second-order quantification. Unfortunately, formulae that contain transitive closure of derived fields are often not pretty nice, leading to false alarms after the application of Elim. This behavior is to be expected due to the undecidability of transitive closure logics over general graphs [58]. On the other hand, unlike approaches based on axiomatizations of transitive closure in first-order logic, our use of MSOL enables complete reasoning about reachability over the backbone fields. It is therefore useful to be able to consider a field as part of a backbone whenever possible. For this purpose, it can be helpful to verify conjunctions of constraints using different backbone for different conjuncts.

5.2.4 Discussion

Verifying conjunctions of constraints. In our skip list example, the field nextSub forms an acyclic (sub-)list. It is therefore possible to verify the conjunction of constraints independently, with nextSub a derived field in the first conjunct (as in Section 5.1.2) but a

backbone field in the second conjunct. Therefore, although the reasoning about transitive closure is incomplete in the first conjunct, it is complete in the second conjunct.

Verifying programs with loop invariants. The technique described so far supports the following approach for verifying programs annotated with loop invariants:

1. generate verification conditions using loop invariants, pre-, and postconditions;
2. eliminate derived fields from verification conditions using Elim (and skolem);
3. decide the resulting formula using a decision procedure such as MONA [63].

Field constraints specific to program point. Our completeness results also apply when, instead of having one global field constraint, we introduce different field constraints for each program point. This allows the developer to refine data structure invariants with the information about the data structure specific to particular program points.

Field constraint analysis and loop invariant inference. Field constraint analysis is not limited to verification in the presence of loop invariants. It can also be used to infer loop invariants automatically. Our implementation described in Chapter 6 combines field constraint analysis with domain predicate abstraction.

Recall from Section 3.4.3 that in domain predicate abstraction the abstraction of the concrete post operator is computed by deciding validity of implications of the form:

$$K \wedge C(\vec{v}) \;\rightarrow\; \mathsf{wlp}(c)(p(\vec{v})) \;\; .$$

Here K is a closed formula, $C(\vec{v})$ is a conjunction of literals over abstraction predicates and $p(\vec{v})$ a literal over abstraction predicates. We use field constraint analysis to check validity of these formulae by augmenting them with the appropriate simulation invariant I and field constraints FC that specify the data structure invariants we want to preserve:

$$I \wedge FC \wedge K \wedge C(\vec{v}) \;\rightarrow\; \mathsf{wlp}(c)(p(\vec{v})) \;\; .$$

The only problem arises from the fact that these additional invariants may be temporarily violated during program execution. To ensure applicability of the analysis, we abstract complete loop free paths in the control flow graph of the program at once. That means we only require that simulation invariants are valid at loop cut points and hence part of the loop invariants. This supports the programming model where violations of data structure invariants are confined to the interior of basic blocks [89].

Amortizing invariant checking in loop invariant inference. A straightforward approach to combine field constraint analysis with abstract interpretation would do a well-formedness check for global invariants and field constraints at every step of the fixed-point computation, invoking a decision procedure at iteration of the fixed-point computation. The following insight allows us to use a single well-formedness check per basic block: *the loop invariant synthesized in the presence of well-formedness is identical to the loop invariant synthesized by ignoring the well-formedness check.* We therefore speculatively compute the

abstraction of the system under the assumption that both the simulation invariant and the field constraints are preserved. After the least fixed point $lfp^{\#}$ of the abstract system has been computed, we generate for every loop free path c with start location ℓ a verification condition: $I \wedge FC \wedge \gamma(lfp^{\#}_{\ell}) \rightarrow \mathsf{wlp}(c)(I \wedge FC)$ where $lfp^{\#}_{\ell}$ is the projection of $lfp^{\#}$ to program location ℓ. We then use again our elimination algorithm to eliminate derived fields and check the validity of these verification conditions. If they are all valid then the analysis is sound and the data structure invariants are preserved. Note that this approach succeeds whenever the straightforward approach would have succeeded, so it improves analysis performance without degrading the precision.

5.3 Further Related Work

Some decision procedures are effective at reasoning about local properties in data structures [70, 87], but are not complete for reasoning about reachability. Promising, although still incomplete, approaches include [80] as well as [73, 90]. Some reachability properties can be reduced to first-order properties using hints in the form of ghost fields [67, 87] or by using decidable extensions of first-order logic [48]. Separation logic [93, 94, 101] eliminates the need for reasoning about reachability, but instead requires techniques for reasoning about inductive predicates [14, 91].

In general, completeness can be achieved by representing loop invariants or candidate loop invariants by formulae in a decidable logic that supports transitive closure [4, 69, 89, 98, 110, 116, 118, 120]. These approaches treat decision procedure as a black box and, when applied to MSOL, inherit the limitations of structure simulation [59]. Our work can be viewed as a technique for lifting existing decision procedures into decision procedures that are applicable to a larger class of structures. Therefore, it can be incorporated into all of these previous approaches.

5.4 Conclusion

Historically, the primary challenge in shape analysis was seen to be dealing effectively with the extremely precise and detailed consistency properties that characterize many (but by no means all) data structures. Perhaps for this reason, many formalisms were built on logics that supported *only* data structures with very precisely defined referencing relationships. This chapter presents a technique that supports both the extreme precision of previous approaches and the controlled reduction in the precision required to support a more general class of data structures whose referencing relationships may be random, depend on the history of the data structure, or vary for some other reason that places the referencing relationships inherently beyond the ability of previous logics and analyses to characterize. We have deployed this analysis in the context of the Jahob program analysis and verification system; our results show that it is effective at 1) analyzing individual data structures to 2) verify interfaces that allow other, more scalable analyses to verify larger-grain data structure consistency properties whose scope spans larger regions of the program.

In a broader context, we view our result as taking an important step towards the practical application of shape analysis. By supporting data structures whose backbone func-

tionally determines the referencing relationships as well as data structures with inherently less structured referencing relationships, it promises to be able to successfully analyze the broad range of data structures that arise in practice. Its integration within the Jahob program analysis and verification framework shows how to leverage this analysis capability to obtain more scalable analyses that build on the results of shape analysis to verify important properties that involve larger regions of the program. Ideally, this research will significantly increase our ability to effectively deploy shape analysis and other subsequently enabled analyses on important programs of interest to the practicing software engineer.

Chapter 6

Proof of Concept

In the previous chapters we developed a new symbolic shape analysis. All the presented techniques have been implemented and evaluated in a tool called Bohne. Bohne is implemented on top of the Jahob data structure verification system [65,66,121]. In the following, we give an overview of the tool. We describe details of the analysis that are more implementation specific but important for making the analysis practical.

Contributions. The main contributions described in this chapter are summarized as follows:

- We describe a method for synthesis of Boolean heap programs that improves the efficiency of fixed point computation by precomputing abstract transition relations and that can control the precision/efficiency trade-off by recomputing abstract transition relations on demand during fixed point computation.
- We present a domain-specific quantifier instantiation technique that significantly improves the running time of the analysis. Furthermore, it often eliminates the need for the underlying decision procedures to deal with quantifiers.
- We introduce semantic caching of decision procedure queries across different fixed point iterations and even different analyzed procedures. The caching yields substantial improvements for procedures that exhibit some similarity, which opens up the possibility of using our analysis in an interactive context.
- We describe an implementation of field constraint analysis that enables automated reasoning in higher-order logic by approximating higher-order logic formulae with formulae in monadic second-order logic over trees.

We used Bohne to verify a range of data structures implementations and data structure clients without manually specified loop invariants or manually provided abstractions. Our examples include implementations of lists (with iterators and with back pointers), two-level skip lists, sorted lists, trees (with and without parent pointers), threaded trees as well as combinations of these data structures. An overview of our case studies is found in Section 6.5.

6.1 Deployment in the Jahob System

The goal of the Jahob system [65,66,121] is to verify data structure consistency properties in the context of non-trivial programs. The input language for Jahob is a subset of Java extended with annotations written as special comments. Therefore, Jahob programs can be compiled and executed using existing Java compilers and virtual machines. Jahob's specification language is similar to JML [76]: it supports preconditions, postconditions, invariants, and specification variables. The main difference is that assertions in Jahob are expressed in a subset of Isabelle/HOL [92].

Figure 6.1 gives an overview of Jahob's system architecture. Our symbolic shape analysis is implemented on top of Jahob. Bohne can be either used as a stand-alone program or called from within Jahob. Both versions take as input the source program annotated with the properties to verify. The stand-alone version will analyze the program and either produce an invariant that guarantees the correctness of the annotated properties, or an error trace if the system is incorrect. If called from within Jahob then the output of Bohne is the source program annotated with the inferred loop invariants. The annotated program can be either passed directly to a verification condition generator or used as input to other program analyses. Bohne exploits Jahob's facilities for verification condition generation and its reasoning backend as black boxes.

Jahob's reasoning backend integrates a diverse set of theorem provers and decision procedures which are used to automate reasoning about higher-order logic formulae. The reasoning backend works as follows. Jahob first splits formulae into an equivalent conjunction of independent smaller formulae. Jahob then attempts to prove each of the resulting conjuncts using a (potentially different) specialized reasoning procedure. Each specialized reasoning procedure in Jahob decides a subset of higher-order logic formulae. Such a procedure therefore first approximates a higher-order logic formula using a formula in the subset, and then proves the resulting formula using a specialized algorithm. We will describe this approximation by means of our field constraint analysis that we presented in Chapter 5.

6.2 Implementation of Domain Predicate Abstraction

In this section we discuss some of the algorithmic aspects of the implementation of domain predicate abstraction in Bohne. Bohne implements the lazy nested abstraction refinement algorithm presented in Chapter 4 with domain predicate abstraction as the underlying analysis. The abstract post operator of the analysis is the context-sensitive Cartesian post that we descibed in Section 3.4. Recall that we used the context operator κ as the key tuning parameter for the precision/efficiency trade-off of our abstract post operator. The context operator is a montone function that maps a set of abstract states $S^\#$ to an over-approximation of the concretization of $S^\#$. Its purpose is to provide non-local information for computing precise local updates of abstract objects. In the following, we describe the context operator that is implemented in Bohne and explain how it is incorporated into Bohne's fixed point computation loop.

What makes Bohne's fixed point computation loop special is the fact that the abstract post is computed on-demand in each fixed point iteration. In particular, the context operator that defines the abstract post is not fixed throughout the analysis. Instead, it is pa-

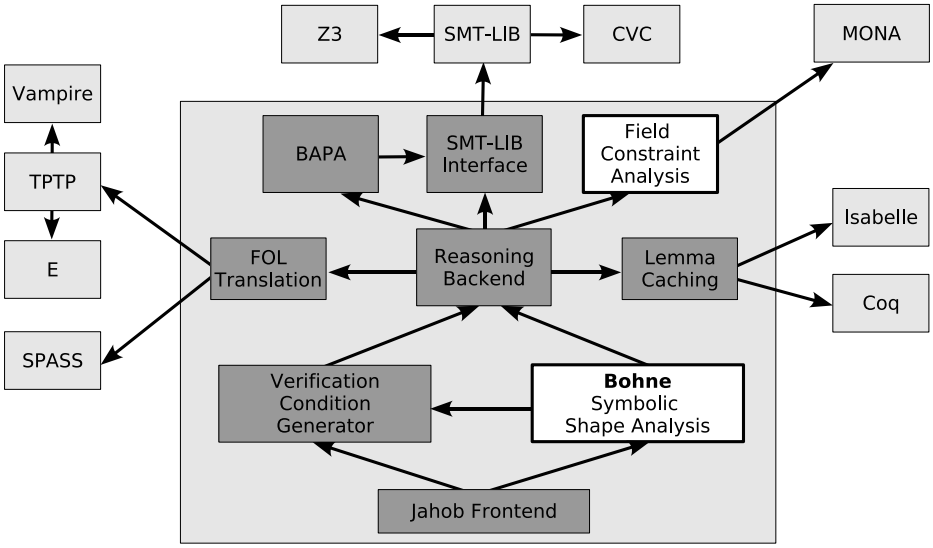


Figure 6.1: Jahob system architecture.

rameterized by the already explored abstract states. Note that this *on-demand abstraction* is directly incorporated into the lazy abstraction refinement loop presented in Chapter 4. However, for exposition purposes we explain the idea of on-demand abstraction by means of a simple fixed point computation loop without abstraction refinement. Therefore, in the following we will fix a global set of abstraction predicates $\mathcal{P}$ that is used throughout the analysis.

The procedure OnDemandAbstraction in Figure 6.2 sketches the fixed point computation loop with on-demand abstraction. The input of this procedure is the program to be analyzed and a set of abstraction predicates. The procedure constructs an abstract reachability tree (ART), similar to the nested lazy abstraction refinement loop presented in Chapter 4. Each node in the ART is labeled by a location and a set of abstract states. Edges between nodes are labeled by commands of the input program. The paths in the ART correspond to traces in the abstract program. Upon termination the procedure returns the root node of the computed ART from which the least fixed point of the abstract post operator can be extracted.

In each iteration of the loop in procedure OnDemandAbstraction, one unprocessed ART edge (r_1, c, r_2) is selected. Then the abstract post for command c and abstract states $r_1.states$ is computed and the result stored in r_2. If new abstract states have been discovered at location $r_2.loc$ then new outgoing ART edges for r_2 are created and added to the set of unprocessed edges. The abstraction of command c is computed on-demand. It uses a context operator κ that depends on the abstract states at location $r_1.loc$ that have been discovered, so far. Thus, the context operator and the abstraction of a particular command can change from one iteration of the loop to the next. However, we make sure that the context operator changes monotonically. This allows our analysis to take advantage of precomputed abstractions from previous fixed point iterations and incrementally recompute the abstraction when the context changes in a significant way. In the following, we show how the context operator and the context-sensitive Cartesian post operator are implemented.

```
proc OnDemandAbstraction
  input
    (Σ, D, X, L, ℓ0, ℓE, T): program
    P: set of abstraction predicates
  begin
    let succ(r) =
      let Succ = ∅
      for all (r.loc, c, ℓ') ∈ T do
        let r' = ⟨loc: ℓ', states: ⊥⟩
        add an edge r -c-> r'
        Succ:= Succ ∪ {(r, c, r')}
      done
      return Succ
    let r0 = ⟨loc = ℓ0; states: ⊤⟩
    let Unprocessed = succ(r0)
    while Unprocessed ≠ ∅ do
      choose (r1, c, r2) ∈ Unprocessed
      Unprocessed:= Unprocessed − {(r1, c, r2)}
      let Context = ⨆r { r.states | r.loc = r1.loc }
      let κ = contextop(Context)
      let New = CSCPost(c, κ, r1.states)
      let Old = ⨆r { r.states | r.loc = r2.loc }
      r2.states:= New
      if New ⋢ Old then
        Unprocessed:= Unprocessed ∪ succ(r2)
    return r0
```

Figure 6.2: On-demand abstraction

6.2.1 Implementation of Context Operator

If we take into account all available context for the abstraction of a command (i.e., the context operator is the concretization function on abstract states) then we need to recompute the abstraction in every iteration of the fixed point computation. Otherwise the analysis would potentially be unsound. In order to avoid unnecessary recomputations we use the operator contextop to compute a new context operator for each iteration of the fixed point computation. This context operator abstracts the context by a formula that less likely changes from one iteration to the next, but still provides enough information to obtain sufficiently precise updates. For this purpose we introduce a domain-specific quantifier instantiation technique. We use this technique not only in connection with the context

$$\mathsf{contextop}(S_0^\#) \stackrel{\text{def}}{=} \lambda S^\#.\, \mathsf{if}\ S^\# \sqsubseteq S_0^\#\ \mathsf{then}\ \mathit{instantiate}(\dot{\bigsqcup} S_0^\#)\ \mathsf{else}\ \mathsf{true}$$

$$\text{where}\quad \mathit{instantiate}(s^\#) \stackrel{\text{def}}{=} \bigwedge_{(\lambda v.\, t=v)\in\mathcal{P}} \dot{\gamma}(s^\# \mathbin{\dot{\sqcap}} o^\#(\lambda v.\, t = v))\, t$$

$$o^\#(p) \stackrel{\text{def}}{=} \lambda p' \in \mathcal{P}.\, \mathsf{if}\ p = p'\ \mathsf{then}\ \{1\}\ \mathsf{else}\ \{0,1\}$$

Figure 6.3: Context instantiation and the context operator

operator, but more generally to eliminate any universal quantifier in a decision procedure query that originates from the concretization of an abstract state. This eliminates the need for the underlying decision procedures to deal with quantifiers.

Our context operator is specifically designed for the analysis of heap programs where abstraction predicates are mostly unary predicates. For the analysis of heap programs, the context operator provides the necessary information for precisely computing the effect of destructive updates on reachability properties. We observed that the most valuable part of the context is the information available over the objects that are involved in the destructive update: if we have a destructive update $s.f{:=}t$, these are the objects denoted by terms s and t. If s and t are in fact relevant for proving some property then our abstraction refinement loop will generate corresponding domain predicates $(\lambda v.\, s = v)$ and $(\lambda v.\, t = v)$. We call domain predicates of this form *singleton predicates*. Our context operator instantiates the universally quantified formulae that result from the concretization of abstract states to the objects that satisfy such singleton predicates.

Figure 6.3 defines the function $\mathsf{contextop}$. Let $S_0^\#$ be the set of already explored abstract states in the current fixed point iteration. Then $\mathsf{contextop}(S_0^\#)$ maps all abstract states that are processed in this iteration to one fixed formula. This fixed formula is computed by the function *instantiate*. Recall Definition 31: the function $\mathsf{contextop}(S_0^\#)$ is a context operator if (1) *instantiate* is monotone and (2) for every abstract state $s^\#$ the formula $\mathit{instantiate}(s^\#)$ is entailed by the concretization of $s^\#$.

Function *instantiate* uses singleton predicates to instantiate an abstract state $s^\#$ to a quantifier free formula (assuming all domain predicates itself are quantifier free). For every singleton predicate p that denotes the object given by the evaluation of some term t it computes the meet of $s^\#$ with abstract object $o^\#(p)$, i.e., the set of all abstract objects in $s^\#$ that have a positive occurrence of predicate p. This set of abstract objects is concretized and the resulting domain formula applied to t. The final formula is represents finitely many instantiations of $\gamma(\{s^\#\})$ with the terms t that occur in singleton predicates. Thus, it is easy to see that *instantiate* is monotone and that for a given abstract state $s^\#$ $\mathit{instantiate}(s^\#)$ is an abstraction of $\gamma(\{s^\#\})$.

Proposition 58 *For any set of abstract states $S^\#$ the function $\mathsf{contextop}(S^\#)$ is a context operator.*

An important property of our fixed point computation loop is that the computed context operators change monotonically from one iteration to the next.

proc CSCPost
  input
    c: *Com*
    κ: context operator
    $S^{\#}$: *AbsStates*
  output
    $S^{\#\prime}$: *AbsStates*
  begin
    let $K = \kappa(S^{\#})$
    let $c^{\#} = \top$
    if $c^{\#}$ is precomputed for (c, K) **then** $c^{\#} := \text{lookup}(c, K)$
    else foreach $p \in \mathcal{P}$ **do**
      $$c^{\#} := c^{\#} \wedge \left(\begin{array}{l} p' \wedge \neg\mathsf{AbstractWLP}(c, K, \neg p) \vee \\ \neg p' \wedge \neg\mathsf{AbstractWLP}(c, K, p) \end{array} \right)$$
    $S^{\#\prime} := \emptyset$
    foreach $s^{\#} \in S^{\#}$ **do**
      let $s^{\#\prime} = \mathsf{RelationalProduct}(s^{\#}, c^{\#})$
      $S^{\#\prime} := S^{\#\prime} \sqcup \{s^{\#\prime}\}$
    return $S^{\#\prime}$
  end

Figure 6.4: Implementation of context-sensitive Cartesian post

Proposition 59 *The function* contextop *is monotone, i.e., let* $S_1^{\#}$ *and* $S_2^{\#}$ *be sets of abstract states with* $S_1^{\#} \sqsubseteq S_2^{\#}$*. Then for all sets of abstract states* $S^{\#}$*:*

$$\mathsf{contextop}(S_1^{\#})(S^{\#}) \models \mathsf{contextop}(S_2^{\#})(S^{\#}) \ .$$

Proposition 59 follows immediately from the monotonicity of joins and monotonicty of function *instantiate*.

6.2.2 Implementation of Abstract Post Operator

We now describe the implementation of the context-sensitive Cartesian post operator. Bohne represents sets of abstract states as sets of ordered binary decision diagrams (OBDDs) [27]. This representation is not canonical. While it is in principle possible to canonically represent sets of abstract states (e.g., by using nondeterministic BDDs [43]), in practice the number of explored abstract states is rather small and there exist many efficient and mature BDD implementations. In the following, we denote by the symbols $\neg$, $\wedge$, and $\vee$ negation, conjunction and disjunction of Boolean functions represented as OBDDs.

Figure 6.4 sketches the implementation of the context-sensitive Cartesian post operator in Bohne. Procedure CSCPost takes a command, a context-operator and a set of abstract

states as input and returns a set of abstract states. The implementation exploits the representation of abstract states as BDDs. First it precomputes an abstract transition relation $c^{\#}$. This abstract transition relation represents the context-sensitive Cartesian post for the given command c in terms of abstract objects over primed and unprimed abstraction predicates. The computation of the abstract transition relation relies on a function AbstractWLP that computes an under-approximation of the weakest domain preconditions of abstraction predicates, as described in Section 3.4.3.

Once the abstract transition relation has been computed, procedure CSCPost computes the relational product of $c^{\#}$ and each given abstract state. The relational product is a standard operation provided by many BDD packages. It conjoins an abstract state with the abstract transition relation, projects the unprimed predicates, and renames primed to unprimed predicates in the resulting abstract state. Note that the abstract transition relation only depends on command c and the context formula K. This allows us to cache abstract transition relations and avoid their recomputation in later fixed point iterations if K is unchanged.

6.2.3 Semantic Caching

Our context operator does not prevent that the abstraction of a given command has to be recomputed occasionally in later fixed point iterations. Whenever we recompute the abstraction, we would like to do this incrementally and reuse the results from previous fixed point iterations. We do this on the level of decision procedure calls by caching the queries and the results of the calls. Syntactic caching of decision procedure queries has been used before (e.g. [6] mentions its use in the SLAM system [7]). The problem with simple syntactic caching of formulae in our approach is that the context formulae are passed to the decision procedure as part of the queries, so a simple syntactic approach is ineffective. However, we know that the context and, thus, the context formulae change monotonically from one iteration to the next. We therefore cache formulae by keeping track of the partial order on the context. Since context formulae occur in the antecedents of the queries, this allows us to reuse negative results of entailment checks from previous fixed point iterations. This method is effective because in practice the number of invalid entailments that are checked for computing the abstraction is significantly higher than the number of valid ones.

Furthermore, formulae are cached up to alpha equivalence. Since the cache is self-contained, this enables caching results of decision procedure calls not only across different fixed point iterations for a single run of the analysis, but even across different runs of the analysis. This yields substantial improvements for the analysis of programs that exhibit some similarity, which opens up the possibility of using our analysis in an interactive context. For example, we verified a procedure inserting an element into a sorted list (see SortedList.add in Figure 6.1) and repeated the analysis without erasing the cache on a modified version of the same procedure where two commuting assignments were exchanged. About 90% of the results to decision procedure calls were found in the cache, causing that running time went down from 10s to 3s.

TODO: this s
timed again.

6.2.4 Further Optimizations

The main challenge for making our symbolic shape analysis practical is to reduce the number of queries to the underlying decision procedure. We have to deal with a general problem of predicate abstraction based approaches, namely that the number of queries grows exponentially with the number of predicates. In the following, we describe the techniques that we use to solve this problem.

Incremental Abstraction and Predicate Abstraction Heuristics. The context-sensitive Cartesian post reduces the problem of abstracting a concrete command c to the problem of checking entailments between conjunctions of domain predicates and weakest liberal preconditions of domain predicates. There are well-known techniques in predicate abstraction (see e.g. [7,45]) that prevent an exponential explosion of the number of decision procedure calls when computing the abstraction of such formulae. We use these techniques in combination with new methods that we developed for our generalized setting. First, Bohne only considers conjunctions up to a fixed length which gives a polynomial bound on the number of decision procedure calls. Second, Bohne incrementally computes the abstraction starting from conjunctions of length one: whenever some conjunction C implies a formula then so do all conjunctions subsumed by C. Finally, we use syntactic heuristics to determine whether a predicate is relevant for the abstraction of a formula, e.g., by comparing the free variables occurring in the predicate and the abstracted formula.

Topological Order on Locations. The number of recomputations of abstract transitions also depends on the strategy used for exploring unprocessed ART edges in the fixed point computation because it influences how often the context changes. For example, if we have a sequence of two loops in the control flow graph then one should first explore all abstract states reachable by following the first loop before one starts the abstraction of the second loop; otherwise the context of the second loop changes more often. To avoid this problem we store unprocessed edges in the reachability graph in a priority queue. The priority is defined by the topological order of the associated program locations in the DAG that results from removing loop back-edges from the control-flow graph. We then choose unprocessed edges with minimal priority to ensure that they are explored in the proper order.

6.3 Implementation of Field Constraint Analysis

This section presents one of the paths through Jahob's reasoning back end to one of the external decision procedures. We describe the translation from an expressive subset of Isabelle/HOL formulae (the input language) to monadic second-order logic over trees (the output language of the translation). Our field constraint analysis is an integral part of this translation. The soundness of the translation is given by the condition that, if the output formula is valid, so is the input formula. Validity of the output formula is automatically checked using MONA [62].

The input language allows constructs such as lambda abstraction, sets and set comprehensions, higher-order quantifiers, conditional expressions, and cardinality operators. The

output language supports atomic formulae build from set expressions over uninterpreted sets, equalities over terms build from unary function symbols that contribute to the tree backbone as well as first and second-order quantification.

6.3.1 Splitting into Sequents

The proof obligations generated by Jahob's verification condition generator can be represented as conjunctions of multiple statements, because they represent all possible paths in the verified procedure, the validity of multiple invariants and postcondition conjuncts, and the absence of run-time errors at multiple program points. The entailment tests generated by Bohne often have a similar conjunctive structure. The first step in the translation splits formulae into these individual conjuncts to prove each of them independently. This process does not lose completeness, yet it improves the effectiveness of the theorem proving process because the resulting formulae are smaller than the starting formula. Moreover, splitting enables Jahob to prove different conjuncts using different techniques, allowing the translation described in this section to be combined with other translations [26, 68, 121]. In particular, Jahob has an inbuilt syntactic prover that can discharge many simple proof obligations immediately. Thus, only a small number of split proof obligations is actually passed on to an external prover.

After splitting, the resulting formulae have the form of sequents $A_1 \wedge \ldots \wedge A_n \implies G$. We call $A_1, \ldots, A_n$ the *assumptions* and G the *goal* of the sequent. The assumptions typically encode a path in the procedure being verified, the precondition, class invariants that hold at procedure entry, as well as properties of our semantic model of memory and the relationships between sets representing Java types.

6.3.2 Translation to Monadic Second-order Logic

After splitting of proof obligations the translation determines the set of backbone fields and derived fields for our field constraint analysis. The backbone fields are the successor functions in the trees that interpret formulae in the output language. The translation searches for the special assumption of the form

$$\mathsf{tree}\,[f_1, \ldots, f_n]$$

in the proof obligation. This assumption states that we are considering structures that form a forest of trees with successor functions f_1 to f_n. Formally, the semantics of tree is defined as follows:

$$\begin{aligned}\mathsf{tree}\,[f_1, \ldots, f_n] \stackrel{\text{def}}{=}\ & \textbf{let}\ f = \lambda v_1\, v_2.\ \textstyle\bigvee_{1 \le i \le n} f_i\, v_1 = v_2\ \textbf{in} \\ & (\forall v\, w.\ f\, v\, w \wedge f^*\, w\, v \rightarrow v = \mathsf{null}) \wedge \\ & (\forall v\, w\, u.\ f\, v\, u \wedge f\, w\, u \wedge v \neq w \rightarrow u = \mathsf{null})\ .\end{aligned}$$

Next the translation searches for assumptions that are field constraints, i.e., assumptions that have the special syntactic form

$$\forall v\, w.\ f\, v = w \rightarrow F$$

Var-Eq
$$\frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge v_1 = v_2 \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v_1{:=}\, v_2]}$$

Var-True
$$\frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge v \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v{:=}\,\mathsf{true}]}$$

Var-False
$$\frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge \neg v \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v{:=}\,\mathsf{false}]}$$

Var-Def
$$\frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge v = e \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v{:=}\, e]} \quad v \notin FV(e)$$

Figure 6.5: Rules for definition substitution

where F only refers to backbone fields. A field constraint of this forms determines that f is a derived field. If some field in the input formula neither occurs in the tree assumption nor in the antecedent of a field constraint then the translation adds a trivial field constraint of the form

$$\forall v\, w.\, f\, v = w \rightarrow \mathsf{true}\ .$$

After each field in the input formula has been determined to be either a backbone or derived field, the translation applies the elimination algorithm for derived fields presented in Chapter 5. The outcome is a formula that is potentially stronger than the input formula but only refers to backbone fields. However, the resulting formula still contains many constructs that are not supported by the output language. Therefore, the next step is to apply a set of rewrite and approximation rules that eliminate these constructs. We describe these rules in the following.

Definition Substitution and Function Unfolding. When one of the assumptions is a variable definition, the translation substitutes its content in the rest of the formula (using rules in Figure 6.5). This approach supports definitions of variables that have complex and higher-order types, but are used simply as shorthands, such as tree and the transitive closure operator rtrancl_pt defined in Section 2.2.1. When the definitions of variables are lambda abstractions, the substitution enables the subsequent beta reduction. In addition to beta reduction, this phase also expands the equality between functions using the extensionality rule (with $f = g$ becoming $\forall x.f\, x = g\, x$).

Function-argument

$$\frac{v_0 = f(e_1, \ldots, e_{i-1}, t, e_{i+1}, \cdots, e_k)}{\exists v.\, v = t \wedge v_0 = f(e_1, \ldots, e_{i-1}, v, e_{i+1}, \cdots, e_k)}$$

Equality-Normalization

$$\frac{t = v}{v = t}$$

Equality-Unfolding

$$\frac{t_1 = t_2}{\exists v.\, v = t_1 \wedge v = t_2}$$

Figure 6.6: Rewriting rules for flattening complex expressions below equalities. Term t denotes a term which is not a variable.

Card-Constraint-Eq

$$\frac{\mathsf{card}(S) = k}{\mathsf{card}(S) \leq k \ \wedge \mathsf{card}(S) \geq k}$$

Card-Constraint-Leq

$$\frac{\mathsf{card}(S) \leq k}{\exists v_1, \ldots, v_k. S \subseteq \{v_1, \ldots, v_k\}}$$

Card-Constraint-Geq

$$\frac{\mathsf{card}(S) \geq k}{\exists v_1, \ldots, v_k.\ \{v_1, \ldots, v_k\} \subseteq S \wedge \bigwedge_{1 \leq i < j \leq k} v_i \neq v_j}$$

Figure 6.7: Rules for constant cardinality constraints

Flattening. To simplify further rewriting the next step is to flatten the formula. The rules for flattening equalities are given in Figure 6.6. Similar rules apply to other predicate symbols. Flattening introduces fresh quantified variables, which could in principle create additional quantifier alternations, making the proof process more difficult. However, each variable can be introduced using either an existential or universal quantifier because $\exists v.v{=}e \wedge F$ is equivalent to $\forall v.v{=}e \rightarrow F$. Our translation therefore chooses the quantifier kind that corresponds to the most recently bound variable in a given scope (taking into account the polarity), preserving the number of quantifier alternations. The starting quantifier kind at the top level of the formula is $\forall$, ensuring that freshly introduced variables for quantifier-free expressions become Skolem constants.

Cardinality Constraints. Constant cardinality constraints express natural generalizations of quantifiers. For example, the statement "there exists at most one element satisfying predicate P" is given by $\mathsf{card}(\{x.\, P\, x\}) \geq 1$. Our translation reduces constant cardinality constraints using the rules in Figure 6.7.

Set Expressions. Both our input and output language support set expressions. However, in the output language set expressions are exclusively built from uninterpreted sets.

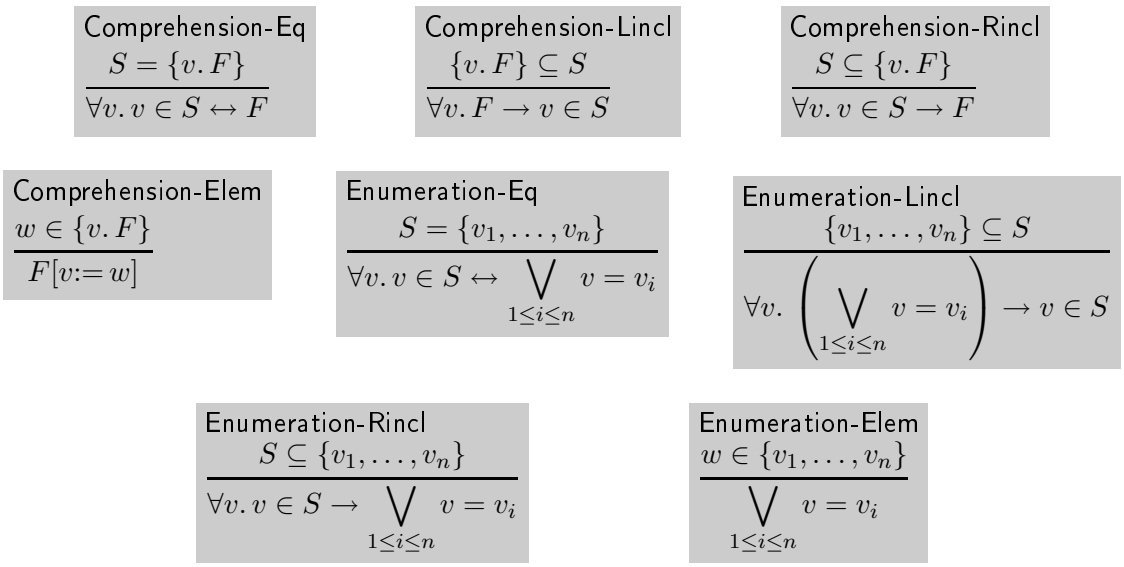


Figure 6.8: Rules for set comprehensions and finite set enumerations

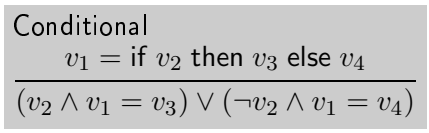


Figure 6.9: Rule for conditional expressions

We therefore need to eliminate set comprehensions and finite set enumerations. Set comprehensions are useful, e.g., for defining specification variables that model the content of data structures. Figure 6.8 shows the rules for eliminating these constructs from a flattened formula.

Conditional Expressions. Conditional expressions are used to express function updates during computation of weakest liberal preconditions. Conditional expressions can be eliminated using case analysis (Figure 6.9). Flattening ensures that this case analysis duplicates only variables and not complex expressions, keeping the translated formula polynomial.

Approximation. Our translation maps higher-order formulae into monadic-second order logic over trees, so there are constructs that it cannot translate exactly. Examples include arithmetic, non-monadic relations, and symbolic cardinality constraints (as in BAPA [68]). Our translation approximates such subformulae in a sound way by replacing them with fresh set or Boolean variables. If the unsupported subformula F contains free variables then the formula is replaced by a formula $v \in S$ where v is the first free variable occuring in F and S is a fresh set variable. The variable S is bound in the same scope as v. These

replacements also ensure that uninterpreted unary predicate symbols in the input formula are translated into monadic sets. If unsupported subformula is closed then it is replaced by a fresh Boolean variable. In both cases, multiple occurrences of a replaced subformula in the same scope are replaced by the same fresh variable.

Another problem is caused by the fact that the input language is multi-sorted while the output language is unsorted. Our translation only keeps subformulae that exclusively use the type constants bool and obj. Any subformula that uses other type constants is approximated using the same technique as described above. The result of the approximation is a stronger formula whose validity implies the validity of the original formula.

6.3.3 Structure Simulation with MONA

After translation of the input formula into monadic second-order logic over trees, we give the translated formula to the MONA decision procedure. MONA supports different modes with different semantic models such as weak monadic second-order logic with one successor (WS1S) and weak monadic second-order logic with two successors (WS2S). Unfortunately none of these modes corresponds one-to-one to the semantic model in Jahob. In particular, the assumption $\mathsf{tree}\,[f_1, \ldots, f_n]$ only states that there acyclicity and sharing-freeness for objects that are different from null. In fact, in Jahob we assume that for all fields f we have $f\,\mathsf{null} = \mathsf{null}$. On the other hand, the interpretations of successor relations in MONA are always acyclic and sharing-free. We therefore use structure simulation [59] to encode formulae interpreted in logical structures that satisfy Jahob's tree assumption in terms of formulae in MONA. For performance reasons we use different modes depending on the number of backbone fields occuring in the proof obligation.

Simulating Lists. If the proof obligation contains a tree assumption $\mathsf{tree}\,[f]$ over one backbone field then we use MONA's `m2l-str` mode. In this mode MONA interprets formulae over finite strings, or equivalently, over the natural numbers from 0 to some constant n; see [62, Section 3.1] for details. Structures that satisfy the assumption $\mathsf{tree}\,[f]$ form a (finite) forest of sharing free lists over field f that are terminated by a unique object null. The object null has a self cycle with respect to field f. We encode such structures by simulating the field f in terms of the successor function on the natural numbers. In order to separate the individual lists we define a subset `$NullSet` of the natural numbers as terminals. This means that the natural numbers between two consecutive elements in the set `$NullSet` form one list in the original structure. All predicate symbols such as equality on individual objects, equality on sets, and set operations correspond to the respective operations in MONA, modulo equivalence of terminals. Figure 6.10 shows an example of a sequent with a tree assumption over one backbone field and the final output of the translation. MONA proves that the formula resulting from the translation is valid.

Simulating Trees. If the proof obligation contains a tree assumptions with more than one backbone field then we use MONA's `ws2s` mode. In this mode MONA interprets formulae over binary trees; see [62, Section 7]. We can encode forests of trees with finitely many successor relations into binary trees. In order to simplify such an encoding MONA provides recursive type declarations. We simulate structures satisfying assumptions of the

Proof obligation:

$$\begin{aligned}&\mathsf{tree}\,[f] \;\wedge\; x \neq \mathsf{null} \;\wedge\; f^* z\, x \;\wedge\; \neg(f^* z\, y) \;\wedge\\ &(\forall v\, w.\, b\, v = w \;\rightarrow\; f w = v \;\vee\; w = \mathsf{null} \wedge (v = \mathsf{null} \vee (\forall u.\, f\, u \neq v)))\\ &\Longrightarrow\; (\lambda v\, w.\, (\mathsf{if}\ v = x\ \mathsf{then}\ y\ \mathsf{else}\ b\, v) = w)^*\, y\, z\end{aligned}$$

MONA input file:

```
m2l-str;
var2 $NullSet where ex1 v : v in $NullSet & (all1 v' : v' ~= v + 1);
pred null(var1 v) = v in $NullSet;
pred $Eq1(var1 v1, v2) = null(v2) & null(v1) | v1 = v2;
pred $Elem(var1 v1, var2 S) = ex1 v2 : v2 in S & $Eq1(v1, v2);
pred $Sub(var2 S1, S2) = all1 v1 : $Elem(v1, S1) => $Elem(v1, S2);
pred $Eq2(var2 S1, S2) = $Sub(S2, S1) & $Sub(S1, S2);
pred $Union(var2 S1, S2, S3) = all1 v1 : $Elem(v1, S2) | $Elem(v1, S1) <=> $Elem(v1, S3);
pred $Inter(var2 S1, S2, S3) = all1 v1 : $Elem(v1, S2) & $Elem(v1, S1) <=> $Elem(v1, S3);
pred $Diff(var2 S1, S2, S3) = all1 v1 : $Elem(v1, S1) & ~ $Elem(v1, S2) <=> $Elem(v1, S3);
var1 x
var1 y
var1 z
pred f(var1 v1, v2) = null(v2) & null(v1) | $Eq1(v1 + 1, v2) & ~ null(v1);
pred b(var1 v, w) = (f(w, v) | (null(v) | ~ (ex1 u : f(u, v))) & null(w));
~ null(x) &
(all2 S : $Elem(z, S) & (all1 v : $Elem(v, S) =>
                                  (ex1 w : $Elem(w, S) & f(v, w))) => $Elem(x, S)) &
~ (all2 S : $Elem(z, S) & (all1 v : $Elem(v, S) =>
                                  (ex1 w : $Elem(w, S) & f(v, w))) => $Elem(y, S)) &
=>
(all2 S : $Elem(y, S) &
      (all1 v, w :
            ex1 c : ($Elem(v, S) & (~ $Eq1(y, v) & $Eq1(c, w) |
                     $Eq1(y, v) & $Eq1(x, w)) => $Elem(w, S)) & b(v, c)) => $Elem(z, S));
```

Figure 6.10: Translation of a valid sequent with a tree assumption over one backbone field and a field constraint over a derived field

form $\mathsf{tree}\,[f_1, \ldots, f_n]$ by declaring a recursive type that represents lists of trees of arity n. Each entry of such a list corresponds to one root object of a tree in the simulated structure. Similar to the encoding of lists we define a unary predicate `null` that holds for all leaf nodes of trees in the list. All predicate symbols are again interpreted modulo equivalence of leave nodes. Figure 6.11 shows an example of a sequent with a tree assumption for binary trees and the result of our translation.

6.3.4 Optimizations

Monadic-second order logic is among the most expressive decidable logics. Thus, the decision problem has high complexity. We developed a set of formula transformations that significantly decreases space consumption and running time of MONA. Without these transformations MONA would often run out of space.

Validity Preserving Transformations. The problem with MONA is not so much that the complexity of monadic second-order logic is non-elementary. In practice, the number

Proof obligation:

$$\begin{array}{l} \mathsf{tree}\,[l,r]\ \wedge\ \mathit{reach}\,(l\,x)\,y\ \wedge \mathit{reach}\,(r\,x)\,y\ \wedge \\ \mathit{reach} = \mathsf{rtrancl\_pt}\ (\lambda v\,w.\,l\,v = w \vee r\,v = w)\ \implies\ y = \mathsf{null} \end{array}$$

MONA input file:

```
ws2s;
type Bb = Bb_null, Bb_node(l: Bb, r: Bb);
type HEAP = Empty, BbTree(bbroot: Bb, $next: HEAP);
universe $U: HEAP;
tree [$U] $Heap where tree_root($Heap) = root ($U);
pred null(var1 v) = variant(v, $Heap, Bb, Bb_null) & v in $Heap;
pred $Eq1(var1 v1, v2) = null(v2) & null(v1) | v1 = v2;
pred $Elem(var1 v1, var2 S) = ex1 v2 : v2 in S & $Eq1(v1, v2);
pred $Sub(var2 S1, S2) = all1 v1 : $Elem(v1, S1) => $Elem(v1, S2);
pred $Eq2(var2 S1, S2) = $Sub(S2, S1) & $Sub(S1, S2);
pred $Union(var2 S1, S2, S3) = all1 v1 : $Elem(v1, S2) | $Elem(v1, S1) <=> $Elem(v1, S3);
pred $Inter(var2 S1, S2, S3) = all1 v1 : $Elem(v1, S2) & $Elem(v1, S1) <=> $Elem(v1, S3);
pred $Diff(var2 S1, S2, S3) = all1 v1 : $Elem(v1, S1) & ~ $Elem(v1, S2) <=> $Elem(v1, S3);
var1 [$U] y where type(y, Bb) & y in $Heap;
var1 [$U] z where type(z, Bb) & z in $Heap;
pred r(var1 v1, v2) =
     type(v1, Bb) & type(v2, Bb) & v1 in $Heap & v2 in $Heap & $Eq1(v2, succ(v1, Bb, Bb_node, r)) |
     null(v1) & null(v2);
pred l(var1 v1, v2) =
     type(v1, Bb) & type(v2, Bb) & v1 in $Heap & v2 in $Heap & $Eq1(v2, succ(v1, Bb, Bb_node, l)) |
     null(v1) & null(v2);
(all2 S : (ex1 v : v in $Heap & type(v, Bb) & $Elem(v, S) & r(x, v)) &
          (all1 w : ((ex1 u : u in $Heap & type(u, Bb) & $Elem(u, S) & l(u, w)) |
                     (ex1 u : u in $Heap & type(u, Bb) & $Elem(u, S) & r(u, w))) &
                    w in $Heap & type(w, Bb) => $Elem(w, S)) & S sub $Heap => $Elem(y, S)) &
(all2 S : (ex1 v : v in $Heap & type(v, Bb) & $Elem(v, S) & l(x, v)) &
          (all1 w : ((ex1 u : u in $Heap & type(u, Bb) & $Elem(u, S) & l(u, w)) |
                     (ex1 u : u in $Heap & type(u, Bb) & $Elem(u, S) & r(u, w))) &
                    w in $Heap & type(w, Bb) => $Elem(w, S)) & S sub $Heap => $Elem(y, S))
=> null(y);
```

Figure 6.11: Translation of a valid sequent with tree assumption over two backbone fields

of quantifier alternations in proof obligations is rather small. Instead, the problem is that the size of the constructed automata can be exponential in the number of variables that occur in the same scope of a quantifier. Unfortunately, MONA itself does not implement any strong optimizations on the formula representation. We therefore implemented several validity preserving transformation in Jahob that optimize the formula representation for automata construction in MONA. The first transformation is to push quantifiers inside the formula as much as possible, so that quantified subformulae are minimized. Next, we try to eliminate quantifiers of the form $\forall v.\, v{=}e \rightarrow F$ and $\exists v.\, v{=}e \wedge F$ by replacing them with $F[v = e]$. This means that the formula is *unflattened* and thus the size of the formula is increased. However, our experience is that the effect of a decreased number of quantified variables far outweighs the effect of increased formula size.

Formula Slicing. We faced the problem that the formulae passed to the decision procedure contain many assumptions coming from the guards of transitions and the background

formula. Often these assumptions are irrelevant for proving a particular goal. However, they introduce additional free variables to the proof obligation. This significantly increases the running time of MONA. Using domain-specific knowledge, we developed heuristics for eliminating irrelevant assumptions from proof obligations. Our experience suggest that this is one of the most valuable techniques that are needed to make the deployment of expensive decision procedures such as MONA practical.

6.4 Implementation of Nested Abstraction Refinement

Our tool uses a few simple heuristics to guess an initial set of domain predicates from the input program and its specification. In particular, Bohne uses a simple syntactic analysis that computes for each program location a set of *singleton* domain predicates that denote object valued program variables that are relevant for this program point. For this purpose the analysis propagates back weakest liberal preconditions from the program's error location and extracts for each program location the set of object valued program variables occuring in the computed weakest liberal precondition. The propagation is continued until a fixed point is reached and the set of generated domain predicates stabelizes at each program location. Additional domain predicates are inferred using our nested abstraction refinement procedure presented in Chapter 4.

6.4.1 Domain Predicate Extraction

We now describe the predicate extraction function extrPreds that is used in the nested abstraction refinement algorithm to extract new predicates from spurious error traces. For a formula F, extrPreds(F) skolemizes top-level universal quantifiers in F and extracts all atomic propositions in the resulting formula. The new heap predicates are obtained by lambda abstraction over the introduced Skolem constants in the atomic propositions.

6.4.2 Reachability Predicates

Our predicate extraction function syntactically extracts new predicates from weakest liberal preconditions of finite paths in the analyzed program. Thus, using this approach we cannot infer new reachability predicates if such predicates do not already occur in the program's specification. We therefore use an additional widening technique to infer new reachability predicates from the domain predicates that are extracted from weakest liberal preconditions. For instance, if the predicate extraction function extracts a domain predicate $(\lambda v.\, f\,(f\,x) = v)$ then Bohne will also add the *widening* of this predicate $(\lambda v.\, f^*\,x\,v)$.

Furthermore, if a reachability predicate occurs in the specification of the program then computing weakest liberal preconditions will often introduce field updates in fields that occur below the transitive closure operator. It is useful to split such predicates into simpler predicates in order to obtain a more fine grained abstraction. For instance Bohne uses the following equivalence to rewrite field updates below reflexive transitive closure of single fields:

$$\begin{aligned}(f[x{:=}y])^*v\,w \equiv\ & \mathsf{rtrancl\_pt}\,(\lambda v_1\ v_2.\, f\ v_1{=}v_2 \wedge v_1{\neq}x)\,v\,w \vee \\ & f^*v\,x \wedge \mathsf{rtrancl\_pt}\,(\lambda v_1\ v_2.\, f\ v_1{=}v_2 \wedge v_1{\neq}x)\,y\,w\ .\end{aligned}$$

benchmark	checked properties	DP	time (in s)	CR
List.traverse	MS, AC, SF	MONA	0.11	no
List.create	MS, AC, SF	MONA	0.78	yes
List.getLast	MS, AC, SF, PC	MONA	0.53	no
List.contains	MS, AC, SF, PC	MONA	0.53	no
List.insertBefore	MS, AC, SF	MONA	2.48	yes
List.append	MS, AC, SF	MONA	8.95	no
List.filter	MS, AC, SF	MONA	5.31	yes
List.partition	MS, AC, SF	MONA	149.16	yes
List.reverse	MS, AC, SF	MONA	5.52	yes
DLL.addLast	MS, AC, SF, DL, PC	MONA	2.05	yes
SortedList.add	MS, AC, SF, SO, PC	MONA, Z3	9.88	no
SkipList.add	MS, AC, SF, PC	MONA	10.82	yes
Tree.add	MS, AC, SF, PC	MONA	18.51	no
ParentTree.add	MS, AC, SF, PL, PC	MONA	20.48	no
ThreadedTree.add	MS, AC, SF, TH, SO, PC	MONA, Z3	445.93	no
Client.move	MS, CS	Z3	3.11	no
Client.createMove	MS, CS, PC	Z3	41.07	yes
Client.partition	MS, CS, FC, PC	Z3	108.15	no

Properties: MS = memory safety, CS = call safety, AC = acyclic, SF = sharing free, DL = doubly linked, PL = parent linked, TH = threaded, SO = sorted, FC = frame condition, PC = post condition

Table 6.1: Summary of experiments. Column DP lists the used decision procedures. Column CR indicates whether Cartesian refinement was required to successfully verify the corresponding program.

Note that this kind of rewriting does not violate the admissibility criterion on the predicate extraction function that is formulated in Definition 42 of Section 4.4.

6.5 Case Studies

We now provide the details of the case studies that we used to evaluate our approach.

6.5.1 Overview

We applied Bohne to verify operations on a diverse set of data structures and properties. Our case studies cover data structures such as (sorted) singly-linked lists, doubly-linked lists, two-level skip lists, trees, trees with parent pointers, and threaded trees. The verified properties include:

- absence of runtime errors such as null dereferences,
- complex data structure consistency properties such as preservation of the tree structure and sortedness

- procedure contracts stating, e.g., how the set of elements stored in a data structure is affected by the procedure, and
- full functional correctness of individual procedures.

In particular, we verified procedure contracts and preservation of representation invariants such as sortedness and the in-order traversal invariant for operations on threaded binary trees.

We further performed modular verification of data structure clients that use the interface for sets with iterators from the `java.util` library [105]. For these benchmarks we annotated procedure contracts to all operations in the interfaces `Set` and `Iterator`. We then used assume-guarantee reasoning inside Bohne to infer invariants for the client. The inferred invariants ensured that all preconditions of data structure operations are satisfied at call sites in the client. Furthermore we verified functional correctness properties of the client code. All benchmarks can be found in the Jahob distribution that is provided on the Jahob project web page [66].

Table 6.1 shows a summary for a collection of benchmarks running on a 2.66 GHz Intel Core2 with 3 GB memory using one core. The system is implemented in Objective Caml and compiled to native code. All the listed properties have been verified in a single run of the analysis. We used different decision procedures for verifying the data structure implementations and the data structure clients. We used mainly MONA [62] for reasoning about data structure implementations and the SMT solver Z3 [41] for proving sortedness properties. For the data structure clients we used Z3 only.

Note that our examples are not limited to stand-alone programs that build and then traverse their own data structures. Instead, our examples verify procedures with non-trivial preconditions, postconditions and representation invariants that can be part of arbitrarily large code.

Further details of the benchmarks are given in Tables 6.2 and 6.3. Table 6.3 gives details on the calls to the validity checker and its underlying decision procedures. One immediately observes that the calls to the validity checker are the main bottleneck of the analysis. On average, 98% of the total running time is spent in the validity checker. The reasons for the high running times are diverse. First, communication with decision procedures is currently implemented via files which is slower than passing data directly. Second, we use expensive decision procedures such as MONA. In some of the examples individual calls to these decision procedures can take up to several seconds. Running times can be improved by incorporating more efficient decision procedures for reasoning about specific data structures [19, 74, 117].

6.5.2 Impact of Context-sensitive Abstraction and Optimizations

We also examined the impact of context-sensitive abstraction and our optimizations such as context instantiation on the analysis. The results are shown in Table 6.4. As expected, running times for context-sensitive abstraction without any of the optimizations enabled are significantly higher (2-21 times) than with our optimizations. In particular, without context instantiation abstract transition relations have to be recomputed many times and caching of decision procedure calls is less effective. If context-sensitive abstraction is disabled

benchmark	#appl. of abs. post	#ref. steps	final ART			#predicates		
			size	depth	st./loc.	total	avrg.	max.
List.traverse	3	0	4	4	1.00	4	2.8	3
List.create	11	6	6	6	1.67	11	6.7	9
List.getLast	7	1	6	6	2.00	7	6.0	7
List.contains	5	1	5	5	2.00	6	5.2	6
List.insertBefore	8	2	5	5	13.50	10	7.4	8
List.append	5	1	4	4	1.50	13	8.2	11
List.filter	31	5	14	5	2.50	12	7.1	10
List.partition	62	21	40	7	3.50	15	10.8	12
List.reverse	9	3	5	5	2.00	11	7.0	9
DLL.addLast	7	3	5	5	1.50	8	7.2	8
SortedList.add	21	3	13	5	1.33	9	6.2	9
Skiplist.add	19	4	16	6	3.67	12	9.6	11
Tree.add	11	0	12	5	3.00	11	10.5	11
ParentTree.add	11	0	12	5	3.00	11	10.5	11
ThreadedTree.add	151	4	82	6	4.33	17	7.8	17
Client.move	8	0	9	9	1.00	16	8.4	11
Client.createMove	46	6	21	18	1.00	33	10.1	14
Client.partition	118	18	24	19	1.00	32	11.9	15

Table 6.2: Analysis details for experiments. The columns list the number of applications of the abstract post, the number of refinement steps, the size and depth of the final ART that represents the computed fixed point, the average number of abstract states per location in the fixed point, the total number of predicates, and the average and maximal number of predicates in a single ART node.

completely the analysis becomes less precise, but also in many cases slower. Most likely the less precise analysis needs to explore a larger part of the abstract state space.

6.5.3 Comparison with TVLA

In order to estimate the costs and gains of an increased degree of automation, we compared Bohne to TVLA [79], the implementation of three-valued shape analysis [103].

We used TVLA version 3.0 alpha [21] for our comparison. We ran both tools on a set of singly-linked list benchmarks. For each example program we used the same precondition in both tools: heaps that form a forest of acyclic, sharing free lists. For TVLA we provided preconditions in the form of sets of three-valued logical structures. Bohne automatically computed the abstraction of preconditions given as logical formulae. We did not use finite differencing [100] to automatically compute predicate updates in TVLA. With finite differencing TVLA was unable to prove preservation of acyclicity of lists in some of the examples. We therefore used the standard predicates and predicate updates for singly-linked lists that are shipped with TVLA. The corresponding abstract domain provides high precision for analyzing list-manipulating programs. We checked for properties that require such high precision, in order to get a meaningful comparison. We checked for absence of null dereferences as well as preservation of acyclicity and absence of sharing. All properties where

benchmark	#VC calls			rel. time spent in VC			time/DP call	
	total	DP	cache	total	abstr.	refine.	avrg.	max.
List.traverse	41	20	51.22%	92.59%	92.59%	0.00%	0.005	0.012
List.create	189	68	64.02%	95.36%	57.22%	38.14%	0.011	0.016
List.getLast	158	56	64.56%	97.74%	54.89%	42.86%	0.009	0.028
List.contains	114	52	54.39%	95.45%	55.30%	40.15%	0.010	0.028
List.insertBefore	246	143	41.87%	97.25%	80.61%	16.64%	0.017	0.052
List.append	311	254	18.33%	99.46%	97.10%	2.37%	0.035	0.080
List.filter	820	273	66.71%	97.36%	87.20%	10.17%	0.019	0.060
List.partition	7650	3027	60.43%	99.17%	95.63%	3.54%	0.049	0.088
List.reverse	615	312	49.27%	98.55%	89.05%	9.50%	0.017	0.048
DLL.addLast	161	89	44.72%	97.86%	62.57%	35.28%	0.023	0.040
SortedList.add	470	190	59.57%	97.89%	65.65%	32.24%	0.051	0.120
Skiplist.add	679	241	64.51%	97.52%	43.84%	53.68%	0.044	0.076
Tree.add	390	124	68.21%	99.52%	67.59%	31.94%	0.149	0.624
ParentTree.add	428	141	67.06%	99.36%	63.41%	35.94%	0.144	0.596
ThreadedTree.add	2882	619	78.52%	99.65%	91.80%	7.85%	0.720	3.816
Client.move	111	82	26.13%	97.17%	85.48%	11.70%	0.037	0.136
Client.createMove	662	393	40.63%	96.35%	33.35%	63.00%	0.101	5.428
Client.partition	2138	896	58.09%	94.92%	27.13%	67.79%	0.115	5.540

Table 6.3: Statistics for validity checker calls. The columns list the total number of calls to the validity checker, the number of actual calls to decision procedures and the corresponding cache hit ration, the time spent in the validity checker relative to the total running time, and the average and maximal time spent for a single call to a decision procedure.

benchmark	List.reverse	List.filter	List.insertBefore	List.append	Skiplist.add	Tree.add
	context-sensitive and with optimizations					
running time (in s)	5.52	5.31	2.48	8.95	10.82	18.51
DP calls (cache hits)	615 (49.27%)	820 (66.71%)	246 (41.87%)	311 (18.33%)	679 (64.51%)	390 (68.21%)
	context-sensitive ($\kappa = \gamma$) and without optimizations					
running time (in s)	43.93	14.38	5.64	83.61	128.8	391.3
DP calls (cache hits)	1555 (19.55%)	1222 (36.74%)	336 (22.62%)	1374 (8.95%)	3049 (26.57%)	1611 (27.56%)
	no context ($\kappa = \lambda S^{\#}.\,\mathsf{true}$) but with optimizations					
running time (in s)	11.26	timeout	timeout	11.75	timeout	15.86
DP calls (cache hits)	2396 (67.86%)	-	-	954 (29.45%)	-	465 (73.98%)

Table 6.4: Effect of context-sensitive abstraction and optimizations

checked in a single run of each analysis. Both tools were able to verify these properties for all our benchmarks.

The results of our experiments are summarized in Table 6.5. The running times for Bohne are between one and two orders of magnitude higher than for TVLA. Observe that almost all time is spent in the decision procedure. Thus, the increase in running time is the price that we pay for automation.

What might be surprising is the fact that the space consumption of Bohne (measured in number of explored abstract states) is smaller than TVLA's, in some examples significantly. We believe that there are three reasons that explain this fact. First, in contrast to summary nodes in three-valued structures, abstract objects in domain predicate abstraction are allowed to be empty. This results in a more compact abstraction. For instance, in order to represent the set of all states containing a list of arbitrary length, one needs at least two three-valued structures, one representing a nonempty list and one representing the empty list. On the other hand, this set of states can be represented by just one Boolean heap.

Next, TVLA uses a fixed set of predicates throughout the analysis. This means that the analysis often tracks information which is irrelevant for proving a specific property. In contrast, our analysis refines the abstract domain by adding predicates on demand and targeted towards specific properties. This can be seen, e.g., at program ListFilter in Section 4.1 where we only need 5 predicates to prove absence of null dereferences. For both analyses the size of the abstract domain is triple exponential in the number of predicates. Thus, a lower number of tracked predicates can make a significant difference in space consumption.

The final reason for lower space consumption is related to materialization. TVLA's focus operator eagerly splits abstract states (and summary nodes) during fixed point computation in order to retain high precision. This potentially leads to an explosion in the number of explored abstract states. Instead, our Cartesian refinement splits abstract states on demand, only if the additional precision is required to rule out some spurious error trace. We can therefore think of Cartesian refinement as a property-driven focus operation.

6.5.4 Limitations

The set of data structures that our implementation can handle is limited by the decision procedures that we have, so far, incorporated into our system. Currently we use monadic second-order logic over trees as our main logic for reasoning about reachability properties. This makes it difficult to verify data structures that admit cycles or sharing. While structure simulation [59] makes it possible to verify some of these data structures with our current implementation, such a simulation needs to be defined by the user.

Furthermore, our widening technique for inferring new reachability predicates only works for flat tree-like structures. It is not appropriate for handling nested data structures such as lists of lists, which may require the inference of nested reachability predicates.

6.6 Conclusion

In this chapter we have presented Bohne, our implementation of symbolic shape analysis. We have deployed a range of techniques that significantly improve the running time of the

benchmark	running time (in s)			avrg. #abs. states		#predicates	
	Bohne	w/o VC	TVLA	Bohne	TVLA	Bohne	TVLA
traverse	0.11	0.008	0.179	1.0	8	4	12
create	0.78	0.036	0.133	1.7	6	11	12
getLast	0.53	0.012	0.214	2.0	10	7	14
insertBefore	2.48	0.068	0.503	13.5	15	10	18
append	8.95	0.048	0.462	1.5	23	13	18
filter	5.31	0.140	0.600	2.5	19	12	18
partition	149.16	1.238	1.508	3.5	72	15	18
reverse	5.52	0.080	0.331	2.0	12	11	14

Table 6.5: Comparison between Bohne and TVLA. The columns list total running times, average number of abstract states per location in the fixed point, and total number of predicates (we refer to the total number of unary predicates used by TVLA.). The third column shows the running time of Bohne without the time spent in the validity checker, i.e., this would be the total running time if we had an oracle for checking validity of formulae that would always return instantaneously.

analysis compared to direct application of the algorithms developed in the previous chapters. These techniques include context-sensitive finite-state abstraction, semantic caching of formulae, and domain-specific quantifier instantiation.

We further compared Bohne to TVLA, the implementation of a non-symbolic shape analysis. In terms of running time, we have to pay the price for the increased degree of automation. In terms of space consumption, however, we even gain from automation; the nested abstraction refinement loop of our symbolic shape analysis seems to achieve the local fine-tuning of the abstraction at the required precision.

Our current experience with Bohne in the context of the Jahob data structure verification system suggests that it is effective for verifying complex properties of a wide range of data structures with a high degree of automation.

Chapter 7

Conclusion

In this thesis we have presented a symbolic shape analysis. Our shape analysis uses logical formulae to symbolically represent sets of states in heap programs. Automated reasoning is used to automatically construct a finitary abstraction from the concrete heap program and to automatically refine the abstraction guided by spurious counterexamples. To our knowledge this is the first shape analysis that incorporates counterexample-guided abstraction refinement from first principles. We used our shape analysis to verify complex user-specified properties of a variety of data structures. Our examples include programs manipulating lists (with iterators and with back pointers), two-level skip lists, sorted lists, trees (with and without parent pointers), threaded trees as well as combinations of these data structures. The analysis offers a high degree of automation: we were able to verify these examples without manually adjusting the analysis to the specific verification problem and without providing user assistance beyond stating the properties to verify.

Our shape analysis is based on a new abstract interpretation called domain predicate abstraction. Domain predicate abstraction provides a new abstract domain that enables the inference of universally quantified invariants over the program's unbounded memory. Our approach incorporates the key idea of three-valued shape analysis [103] into predicate abstraction [49] by replacing predicates on program states by predicates on objects in the heap of program states. Domain predicate abstraction not only provides the foundation for our symbolic shape analysis, but also shades a new light on the underlying concepts of three-valued shape analysis.

Building on top of domain predicate abstraction we developed a new counterexample-guided abstraction refinement technique for shape analysis. Our search for an appropriate refinement procedure was guided by the so-called progress property, i.e., the requirement that every spurious counterexample is eventually eliminated by a refinement step. The resulting procedure uses a lazy nested abstraction refinement loop that refines both the abstract domain of the analysis and the abstract transformer on these abstract domains. The nested refinement guarantees the progress property. In retrospect, it was this search for the progress property that ensured the practical success of our analysis: for many of our example programs the analysis would not succeed without the nested refinement.

Finally, we presented field constraint analysis which provides the missing link to the underlying reasoning procedures that automate our shape analysis. Field constraint analysis relaxes the restrictions on the verified data structures that are imposed by the logics of

the underlying decision procedures. This approach promises to enable our shape analysis successfully analyze the broad range of data structures that arise in practice.

We implemented the presented techniques in our prototype tool Bohne. Our experience with Bohne is in the context of the Jahob system [66] for modular data structure verification. The modular verification exploits user-provided procedure contracts to separate the verification of libraries (that implement data structures) from the verification of clients (that use these data structures). The library interfaces hide the complexity of the underlying data structure implementation. The analysis of the clients calls for more scalable (but perhaps less precise) techniques while the analysis of the libraries requires high precision. Our symbolic shape analysis enables such a modular verification and provides the required precision to verify procedure contracts that express functional correctness properties of data structure operations. We believe that such a modular approach towards data structure verification may be the key to make precise and flexible, but also comparably expensive shape analyses applicable to large programs.

7.1 Future Work

We would like to conclude this thesis with an outlook on possible directions for future work.

Increased Scalability. We have given experimental evidence that the targeted precision that comes with the increased degree of automation in symbolic shape analysis decreases the space consumption compared to non-symbolic shape analyses. Still, automation also comes at the price of increased running times. We have seen that almost the complete running time of our analysis is spent in the underlying decision procedures. We are currently incorporating more efficient decision procedures for specific classes of data structures into our automated reasoning framework. Our symbolic shape analysis can immediately take advantage of these improvements. We are also taking more radical measures for decreasing the running times. One idea is to persistently cache the results of decision procedure calls across multiple runs of the analysis. We are currently investigating how techniques developed in the context of data base systems can be used to increase cache hit ratios by exploiting the partial order on formulae. First experiments with this idea are very promising.

Beyond Safety. In this thesis we investigated techniques for verifying safety properties of heap-manipulating programs, i.e., properties that can be expressed in terms of reachability of an error location. However, safety properties only cover one dimension of the space of temporal properties. Liveness properties cover the other dimension. Like any safety property can be reduced to reachability, any liveness property can be reduced to termination. Automatic termination checkers require different techniques [2,36,95,96] than safety checkers. Still, a termination proof often depends on certain assumptions that are safety properties, e.g., a termination proof for a loop that iterates over a list might depend on the assumption that the list is acyclic. We started to explore how symbolic shape analysis can be used to automatically infer and verify such assumptions. In [97] we presented an algorithm for the inference of preconditions for termination of heap-manipulating programs. The algorithm exploits a unique interplay between a counterexample-producing

abstract termination checker and symbolic shape analysis. The shape analysis produces heap assumptions on demand to eliminate counterexamples, i.e., non-terminating abstract computations. Currently our results only apply to list-manipulating programs. Extending this approach to more complex data structure remains an interesting direction for future research.

Unifying Program Analysis and Automated Reasoning. Many of the innovations that have been made in automated reasoning in recent years have been driven by the desire to apply these techniques in program verification. Among the most striking examples is the development of efficient satisfiability modulo theory solvers and techniques for combining decision procedures for different logical theories. An infesting question is whether it is possible to achieve a more tight integration of these techniques into program analyses. Along these lines we are currently exploring how facts that are synthesized by our program analysis can be used to exchange information between different decision procedures. The result would be a combination of program analysis and decision procedures that is able to prove properties in theories that are beyond the scope of any of the individual decision procedures and that cannot be effectively handled by traditional combination techniques for logical theories.

Beyond Shape Analysis. So far, the experience with domain predicate abstraction is in the context of heap-manipulating programs. However, this approach is clearly not restricted to verification of heap programs. Instead, it can more generally be used to verify systems where properties that involve universal quantification over some unbounded domain play an important role. Programs that manipulate arrays, concurrent programs with unbounded thread creation, parameterized systems, and distributed systems with an unbounded number of participants – to name only a small selection of use cases where our analysis could be beneficial. Exploring the potential applications of domain predicate abstraction is a main goal for our future research.

Zusammenfassung

Software ist häufig der unzuverlässigste Bestandteil von Computersysteme. Dennoch dringen Computer weitherhin in alle technologischen Bereiche vor. Ein zentrales Ziel in der Erforschung von Programmiersprachen besteht daher darin, Methoden zu entwickeln, die die Zuverlässigkeit von Software erhöhen können.

Den ambitioniertesten Weg an dieses Problem heranzugehen beschreitet die formale Programmverifikation. Das Ziel formaler Programmverifikation besteht darin, einen mathematischen Beweis zu erbringen, der sicher stellt, dass ein Programm seine Spezifikation erfüllt. Traditionell werden solche Beweise vom Programmierer selbst in einem formalen Kalkül wie zum Beispiel Hoare-Logik [46, 56] erbracht. Es gibt zwei Gründe, die der praktischen Anwendung dieser Methode in der Softwareentwicklung entgegenstehen. Zum einen gibt es nur wenige Softwareentwickler, die die nötigen Kenntnisse und Erfahrungen besitzen solche formalen Korrektheitsbeweise zu erbringen. Zum anderen führt die zunehmende Komplexität von Software dazu, dass es extrem aufwendig ist selbst einfache Korrektheitseigenschaften für ein vollständiges Softwaresystem manuell zu verifizieren.

Die Forschung in der Programmverifikation ist daher seit vielen Jahren von dem Ideal geleitet, Programmanalysewerkzeuge zu entwickeln, die den Programmierer dabei unterstützen die Korrektheit seiner Software sicherzustellen, d.h. Software zu entwickeln, die in der Lage ist Software automatisch zu verifizieren. Da die meisten Verifikationsprobleme unentscheidbar sind, können solche Verfahren nur approximative Lösungen bieten. Einen formalen Rahmen für den Entwurf approximativer Programmanalyseverfahren bietet die *abstrakte Interpretation* [37, 38]. Eine abstrakte Interpretation transformiert das *konkrete Programm* in ein *abstraktes Programm* für das das Verifikationsproblem entscheidbar ist. Die Abstraktion garantiert, dass das konkrete Programm immer dann korrekt ist, wenn auch das abstrakte Programm korrekt ist. Die Analyse ist approximativ, weil das konkrete Programm auch dann korrekt sein kann, wenn das abstrakte Programm nicht korrekt ist, d.h. die Analyse kann Gegenbeispiele für die zu beweisende Programmeigenschaft generieren, die auf das konkrete Programm nicht zutreffen. Wir nennen solche Gegenbeispiele *Scheingegenbeispiele*. Die Methode der abstrakten Interpretation verschiebt das Problem der formalen Beweisführung über Programme vom Programmierer auf den Designer des Programmanalysewerkzeugs, d.h. es ist die Aufgabe des Designers eine geeignete Abstraktion für ein spezifisches Verifikationsproblem zu finden. Auch wenn es immer Programmeigenschaften geben wird, deren automatische Verifikation schwierig ist und die daher einen manuellen Beweis erfordern, so ist die Methode der abstrakten Interpretation dennoch ein großer Erfolg. Sie bildet die Grundlage vieler moderner Werkzeuge, die Eigenschaften wie zum Beispiel die Abwesenheit von Laufzeitfehlern für Programme industrieller Größe automatisch verifizieren

können [20, 113].

Jüngst haben Forscher damit begonnen die Frage zu untersuchen, ob es möglich ist den Automatisierungsgrad in der Programmverifikation noch weiter zu erhöhen. In den letzten Jahren gab es wesentliche Fortschritte im Bereich des automatischen Theorembeweisens [12,41,108,109] und leistungsfähige Entscheidungsprozeduren wurden entdeckt [62]. Diese Fortschritte haben es ermöglicht das Führen von Beweisen über Programme selbst zu automatisieren. Das Ziel dieser Forschung ist es, Programmverifikation vollständig auf das Problem automatischer Beweisführung in ausdrucksstarken Logiken zu reduzieren, d.h. man verwendet Software, um automatisch Software zu konstruieren, die Software automatisch verifiziert.

Wir nennen die Symbiose aus abstrakter Interpretation und Methoden der automatischen Beweisführung *symbolische Programmanalyse*. Symbolische Programmanalyseverfahren sind aus vielerlei Gründen interessant. Zum ersten ermöglicht die Verwendung von Methoden zur automatischen Beweisführung nicht nur die Automatisierung der Transformation eines konkreten Programms in ein abstraktes Programm und die darauffolgende Analyse des abstrakten Programms, sondern sie ermöglicht sogar die Automatisierung der Konstruktion der Abstraktion. Abstraktionsverfeinerungstechniken [35, 52] verwenden Prozeduren zur automatischen Beweisführung, um Scheingegenbeispiele, die die Analyse des abstrakten Programms generiert, als solche zu erkennen. Die erkannten Scheingegenbeispiele werden dann dazu verwendet die Abstraktion automatisch zu verfeinern. Zum zweiten separiert der Einsatz von Logiken das Problem der Beweisführung über die Semantik des Programms von der eigentlichen Analyse des Programms. Dies erlaube es die Analyse als ein algorithmisches Problem zu formulieren, das unabhängig vom konkreten Programm und der zu beweisenden Eigenschaft ist. Eine Konsequenz dieser Aufgabenteilung ist, dass der Korrektheitsbeweis für eine symbolische Programmanalyse in der Regel einfacher zu führen ist als für eine nichtsymbolische Analyse. Der aufwendigste Teil des Korrektheitsbeweises, d.h. der Teil, der die Semantik des Programms betrifft, folgt aus der Korrektheit der zugrundeliegenden automatischen Beweisführungsprozeduren. Schließlich stellen Logiken auch eine natürliche Sprache zur Verfügung, um das Verhalten von Programmfragmenten zu beschreiben. Dies erlaubt eine einfache Kombination von symbolische Programmanalysen mit Techniken, die die Programmverifikation modularisieren [11,44,65].

Auch wenn es immer Verifikationsprobleme geben wird, die den Einfallsreichtum eines Programmanalysedesigners erfordern, der eine Abstraktion für das gegebene Problem maßschneidert, so kann die symbolische Programmanalyse diesen doch entlasten. In der Tat hat die Idee der symbolischen Programmanalyse eine neue Generation von Verifikationswerkzeugen hervorgebracht [9,30,54], die einen unübertroffen hohen Automatisierungsgrad bieten. Diese Werkzeuge sind bereits im industriellen Einsatz, zum Beispiel als Bestandteil des *Microsoft Windows device driver development kid* [88].

Ein Problem, dem in jüngster Zeit viel Beachtung geschenkt wurde, ist die Fragestellung, wie man in der Verifikation und Programmanalyse effektiv mit dynamisch allozierten Zeigerstrukturen umgehen kann. Die Fähigkeit von Zeigerstrukturen ihre Größe und Form dynamisch zu verändern, macht sie zu einem wichtigen Programmierkonzept in imperativen Programmiersprachen. Daher ist es nicht überraschend, dass Zeigerstrukturen den Kern vieler effizienter Algorithmen bilden und in vielen Software-Entwurfsmustern Verwendung finden. Die Flexibilität und Vielfalt von Zeigerstrukturen erschwert aber auch die

Verifikation von Programmen, die diese manipulieren. Die praktische Relevanz und die Herausforderungen, die die Verifikation von Zeigerstrukturen aufwirft, erklärt das gestiegene Interesse daran dieses Problem zu lösen.

Programmanalysen, deren Hauptaufgabe in der Verifikation von Eigenschaften solcher Zeigerstrukturen besteht, werden im allgemeinen als Shape-Analysen bezeichnet [61]. Eine symbolische Shape-Analyse verspräche ein Spektrum verschiedener dynamisch allozierter Zeigerstrukturen und ein Spektrum an Eigenschaften verifizieren zu können, ohne dass der Benutzer die Analyse manuell an eine spezifische Probleminstanz anpassen müsste. Die Frage, wie solch eine symbolische Shape-Analyse aussieht und ob sie ihre Versprechen einhalten könnte, war offen. In der vorliegenden Dissertation beschäftigen wir uns mit diesen Fragen.

Symbolische Shape-Analyse

Das Ziel einer Shape-Analyse ist die Verifikation komplexer Konsistenzeigenschaften von Zeigerstrukturen. Unter Konsistenzeigenschaften verstehen wir Invarianten, die die Form einer Datenstruktur beschreiben und die an bestimmten Punkten während der Programmausführung gelten müssen (zum Beispiel an Ein- und Austrittspunkten von Bibliotheksfunktionen, die die Datenstruktur implementieren). Als Beispiel betrachten wir das in Abbildung 7.1 dargestellte Fragment eines Java-Programms. Dieses Programmfragment zeigt Teile einer Datenstruktur, die Container zur Speicherung einer unbeschränkten Menge von Objekten implementiert. Die Datenstruktur stellt verschiedene Operationen bereit, wie zum Beispiel das Hinzufügen und Entfernen von Elementen aus einer Menge von Objekten, aber auch komplexere Operationen wie das Filtern der gespeicherten Objekte in Abhängigkeit von einem gegebenen Prädikat. Die eigentliche Menge ist durch eine doppeltverkettete Liste implementiert. Eine der Konsistenzeigenschaften dieser Datenstruktur besagt daher, dass die Liste, auf die das Feld root zeigt, tatsächlich eine doppeltverkettete Liste ist. Man kann eine Shape-Analyse dazu verwenden, um zu verifizieren, dass solche Invarianten unter allen Operationen der Datenstruktur erhalten bleiben.

Die Verifikation von Konsistenzeigenschaften ist für sich betrachtet bereits wichtig, da die korrekte Ausführung eines Programms häufig von der Konsistenz der verwendeten Datenstrukturen abhängt. Wenn zum Beispiel die Liste, auf die das Referenzfeld root zeigt, bei Eintritt in die Methode filter nicht doppeltverkettet ist, dann wird das Verhalten der Methode nicht vorhersagbar sein. Unter Umständen wird die Methode sogar einen Laufzeitfehler verursachen. Darüber hinaus spielen Konsistenzeigenschaften eine wichtige Rolle bei der Verifikation anderer Programmeigenschaften. So lässt sich zum Beispiel die Terminierung der *while*-Schleife in der Methode filter unter der Annahme beweisen, dass die Liste, auf die das Feld root zeigt, azyklisch ist. Man kann eine Shape-Analyse dazu verwenden, solche Annahmen zu verifizieren.

In dieser Dissertation untersuchen wir eine neuartige symbolische Shape-Analyse. Unsere Shape-Analyse verwendet automatische Beweisführungsprozeduren, um ein Programm, dass Zeigerstrukturen manipuliert, automatisch durch ein Programm zu abstrahieren, das logische Formeln manipuliert. Unser Ansatz verallgemeinert das Prinzip der Prädikatenabstraktion [49], ein existierendes symbolisches Programmanalyseverfahren, durch Einbezug der Schlüsselidee in der dreiwertigen Shape-Analyse [103], einem existierenden nichtsym-

```
public interface Predicate {
   //: public specvar pred :: objset;

   public boolean contains(Object o);
   //: ensures "result = (o ∈ pred)"
}

public class DLLSet {
   class Node {
      Node next;
      Node prev;
      Object data;
   }

   private Node root;

   /*: public specvar content :: objset;
       private vardefs "content == {x. <root erreicht ein Node Objekt y via next, so dass y.data=x>}";
       invariant "<die Liste ausgehend von root ist azyklisch>";
       invariant "<die Liste ausgehend von root ist doppeltverkettet>"; */

   public void add(Object o)
   /*: requires "o ∉ content"
       modifies content
       ensures "content = old content ∪ {o}" */
   {
      Node n = new Node();
      n.next = root;
      n.data = o;
      root.prev = n;
      root = n;
   }
   ...
   public void filter (Predicate p)
   /*: requires "p ≠ null"
       modifies content
       ensures "content = old content ∩ (pred p)" */
   {
      Node e = root;
      while (e != null) {
         Node c = e;
         e = e.next;
         if (!p.contains(c.data)) {
            if (c.prev == null) {
               e.prev = null;
               root = e;
            } else {
               c.prev.next = e;
               e.prev = c;
            }
         }
      }
   }
}
```

Abbildung 7.1: Kontäner für Mengen von Objekten, die durch doppeltverkettete Listen implementiert sind

bolischen Shape-Analyseverfahren. Die Verknüpfung dieser Techniken resultiert in einer Shape-Analyse, die über eine einzigartige Kombination von Eigenschaften verfügt. Unsere Shape-Analyse ist nicht *a priori* auf die Verifikation einer bestimmte Klasse von Datenstrukturen und Eigenschaften eingeschränkt. Dennoch bietet sie einen hohen Automatisierungsgrad. Insbesondere waren wir in der Lage die Erhaltung von Konsistenzeigenschaften für Operationen auf geketteten Binärbäumen [107] zu verifizieren (darunter Sortiertheit und die Inorder-Traversierungseigenschaft). Dies gelang ohne unsere Analyse speziell für dieses Problem anzupassen und ohne jegliche Hilfe des Benutzers, die über die bloße Formulierung der zu beweisenden Eigenschaften hinausginge. Uns ist keine andere Shape-Analyse bekannt, die diese Eigenschaften mit einem vergleichbaren Automatisierungsgrad verifizieren könnte.

Schließlich fügt sich unsere Shape-Analyse auf natürliche Weise ein in den Ansatz der modularen Verifikation von Datenstrukturen, den wir im Jahob-System [65,121] beschreiten. Dieser Ansatz verwendet vom Benutzer zur Verfügung gestellte Prozedurkontrakte, um die Verifikation von Bibliotheken (die Datenstrukturen implementieren) von der Verifikation von Klienten (die diese Datenstrukturen verwenden) zu trennen. Die Schnittstellen der Bibliotheken deklarieren abstrakte Mengen und Relationen, die das Verhalten der Datenstruktur charakterisieren, dabei aber die Komplexität der darunterliegenden Implementierung verbergen. Zum Beispiel deklariert die Schnittstelle der Klasse DLLSet in Abbildung 7.1 eine abstrakte Menge content, die die Menge der in einer gegebenen Instanz der Klasse gespeicherten Objekte denotiert. In den Vor- und Nachbedingungen der öffentlichen Methoden der Klasse DLLSet wird die Menge content dazu verwendet, um den vom Klienten der Datenstruktur beobachtbaren Effekt der Methoden zu beschreiben. Wir waren in der Lage solche Prozedurkontrakte von Bibliotheksfunktionen mit unserer Shape-Analyse zu verifizieren. Während die Analyse der Bibliotheken die hohe Präzision einer Shape-Analyse erfordert, so verlangt die Analyse der Klienten den Einsatz einer skalierbaren (aber vielleicht weniger präzisen) Programmanalyse. Wir glauben, dass solch ein modularer Verifikationsansatz den Schlüssel darstellen könnte, der präzise Shape-Analyseverfahren auf große Programme anwendbar macht.

Technische Beiträge

Unsere neue Shape-Analyse basiert auf einer Reihe technischer Beiträge. Diese Beiträge lassen sich wie folgt zusammenfassen:

- Wir entwickeln die *universelle Prädikatenabstraktion*, eine neuartige parametrierte abstrakte Domäne für symbolische Shape-Analyse, die detaillierte Eigenschaften verschiedener Regionen im unbeschränkten Speicher eines Programms ausdrücken kann.

- Wir stellen Mittel bereit, um mit Hilfe von automatischen Beweisführungsprozeduren ein Programm, das Zeigerstrukturen manipuliert, automatisch in ein abstraktes Programm zu transformieren.

- Wir präsentieren eine Abstraktionsverfeinerungsmethode für die universelle Prädikatenabstraktion. Diese Methode beseitigt die Anforderung an den Benutzer, die Abstraktion manuell für die Analyse eines bestimmten Programms oder einer bestimmten Programmeigenschaft anpassen zu müssen.

- Wir entwickeln die *Feldbedingungsanalyse*, eine neue Technik zur Beweisführung über Eigenschaften von Datenstrukturen. Unsere Feldbedingungsanalyse ermöglicht die Verwendung entscheidbarer Logiken für die Verifikation von Datenstrukturen, die ursprünglich außerhalb des Anwendungsbereichs dieser Logiken lagen.

Im folgenden diskutieren wir diese Beiträge im Detail.

Universelle Prädikatenabstraktion. Wir zeigen, dass sich die Schlüsselidee der dreiwertigen Shape-Analyse [103], die Partitionierung des Heaps bezüglich einer Menge von Prädikaten über Heap-Objekten, in die Methode der Prädikatenabstraktion [49] einbetten lässt. Die Symbiose dieser Ideen resultiert in einer neuen Analyse, die wir *universelle Prädikatenabstraktion* (engl. *domain predicate abstraction*) nennen. Die universelle Prädikatenabstraktion ermöglicht die Herleitung von Invarianten in der Form von Disjunktionen universell quantifizierter Aussagen über den unbeschränkten Speicher eines Programms. Die Bausteine dieser quantifizierten Aussagen sind Prädikate über Heap-Objekte. Unsere Konstruktion des abstrakten Post-Operators verläuft analog zur entsprechenden Konstruktion in der klassischen Prädikatenabstraktion, mit dem Unterschied, dass Prädikate über Heap-Objekte den Platz von Zustandsprädikaten einnehmen und *Boolesche Heaps* (Mengen von Bitvektoren) den Platz von Booleschen Zuständen (Bitvektoren). Das konkrete Programm wird abstrahiert durch ein Programm über Boolesche Heaps. Für jedes Kommando des konkreten Programms konstruieren wir das entsprechende abstrakte Kommando effektiv mit Hilfe von Methoden der automatischen Beweisführung. Die universelle Prädikatenabstraktion bietet daher den parametrisierten Unterbau für eine symbolische Shape-Analyse.

Bedarfsgerechte, verschachtelte Abstraktionsverfeinerung. Wir entwickeln eine automatische Abstraktionsverfeinerungsmethode für unsere symbolische Shape-Analyse. Unsere Technik verwendet Scheingegenbeispiele, um sowohl die abstrakte Domäne, als auch den abstrakten Post-Operator unserer symbolischen Shape-Analyse zu verfeinern. Diese beiden Phasen der Abstraktionsverfeinerung sind in einer Schleife verschachtelt, die eine bedarfsgerechte Abstraktionsverfeinerung ermöglicht [53]. Die zweite Phase der Abstraktionsverfeinerung ist entscheidend für den praktischen Erfolg unserer Shape-Analyse. In vielen unserer Fallstudien schlägt die Analyse ohne diese zweite Phase fehl. Dieses praktische Resultat stimmt mit unserem theoretischen Befund überein, der sich mit der so genannten Fortschritteigenschaft befasst. Diese Eigenschaft besagt, dass jedes von der Analyse gefundene Scheingegenbeispiel irgendwann durch einen Verfeinerungsschritt ausgeschlossen wird. Unsere Methode der Abstraktionsverfeinerung besitzt die Fortschrittseigenschaft nur mit der zweiten Verfeinerungsphase.

Wir liefern außerdem einen experimentellen Nachweis dafür, dass der erhöhte Automatisierungsgrad unserer Analyse auch zu zielgerichteter Präzision führt. Diese zielgerichtete Präzision spiegelt sich in einem geringeren Platzverbrauch unserer Analyse wider; die verschachtelte Verfeinerungsschleife scheint die lokale Feinabstimmung der Abstraktion auf das erforderliche Maß an Präzision zu bewirken.

Feldbedingungsanalyse. Eine der faszinierenden Eigenschaften unserer symbolischen Shape-Analyse ist die Tatsache, dass die zugrundeliegende automatische Beweisführungspro-

zedur als Black-Box betrachtet wird. Man kann daher einen beliebigen, existierenden Theorembeweiser oder Entscheidungsprozedur verwenden. In der Praxis erfüllen existierende Entscheidungsprozeduren aber häufig nicht alle Anforderungen, die die Analyse stellt. Daher kann es nötig sein, eine zusätzliche Schicht zwischen Analyse und der tatsächlichen Entscheidungsprozedur einführen zu müssen. Wir präsentieren eine solche Technik in dieser Dissertation.

Wir entwickeln die so genannte *Feldbedingungsanalyse* (engl. *field constraint analysis*), ein neues Verfahren, um Eigenschaften von Datenstrukturen zu beweisen. Eine Feldbedingung für ein Referenzfeld einer Datenstruktur ist eine logische Formel, die eine Menge von Objekten spezifiziert, auf die das Feld zeigen kann. Feldbedingungen ermöglichen die Anwendung entscheidbarer Logiken für die Verifikation von Datenstrukturen, die ursprünglich außerhalb des Anwendungsbereichs dieser Logiken lagen, indem sie die Referenzfelder in zwei Klassen aufteilen: Felder, die das Grundgerüst der Datenstruktur aufspannen und abgeleitete Felder, die dieses Grundgerüst in beliebiger Weise durchkreuzen können. Die Feldbedingungen der abgeleiteten Felder werden ausgenutzt, um das Beweisen von Eigenschaften der Datenstruktur auf das Beweisen von Eigenschaften des Grundgerüsts der Datenstruktur zu reduzieren. Bisher war die Behandlung solcher abgeleiteten Felder nur möglich, wenn sie vollständig durch ihre Feldbedingungen charakterisiert waren. Die Klasse der unterstützten Datenstrukturen wurde dadurch signifikant eingeschränkt.

Unsere Feldbedingungsanalyse erlaubt die Spezifikation nichtdeterministischer Feldbedingungen für abgeleitete Felder. Nichtdeterministische Feldbedingungen ermöglichen die Verifikation von Datenstrukturen wie zum Beispiel Skip-Listen und erlauben außerdem die Verifikation von Invarianten zwischen Datenstrukturen. Damit stellen sie eine ausdrucksstarke Verallgemeinerung statischer Typedeklarationen da.

Die Allgemeinheit unserer Feldbedingungen erfordert neue Techniken, die orthogonal zur traditionellen Methode der Struktursimulation sind [57,59]. Wir präsentieren eine solche Methode und beweisen sowohl ihre Korrektheit, als auch ihre Vollständigkeit in bestimmten, interessanten Fällen. Mit Hilfe unserer neuen Technik waren wir in der Lage Datenstrukturen zu verifizieren, die zuvor außerhalb der Reichweite vergleichbarer Verfahren waren.

Machbarkeitsstudie

Alle in dieser Dissertation präsentierten Techniken wurden in einem Werkzeug namens Bohne implementiert und evaluiert. Bohne baut auf dem Jahob-System [66] für die Verifikation der Konsistenz von Datenstrukturen auf. Bohne analysiert Java-Programme, ähnlich dem in Abbildung 7.1 gezeigten, die mit speziellen Kommentaren annotiert werden, um Prozedurkontrakte und Repräsentationsinvarianten von Datenstrukturen zu spezifizieren. Unser Werkzeug verifiziert, dass alle Methoden ihre Prozedurkontrakte einhalten und alle Repräsentationsinvarianten unter Ausführung von Datenstrukturoperationen erhalten bleiben. Wir haben Bohne dazu verwendet komplexe, benutzerspezifizierte Eigenschaften für eine Reihe von Datenstrukturimplementierungen und deren Klienten zu verifizieren, ohne Schleifeninvarianten manuell zu spezifizieren oder die Abstraktion von Hand zu erstellen. Dies beweist, dass die symbolische Shape-Analyse ihre Erwartungen erfüllen kann, nämlich eine Vielfalt von Datenstrukturen und Eigenschaften mit einem hohen Automatisierungsgrad zu verifizieren.

Bibliography

[1] R.-J. Back, A. Akademi, and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[2] I. Balaban, A. Pnueli, and L. D. Zuck. Ranking abstraction as companion to predicate abstraction. In *Formal Techniques for Networked and Distributed Systems, FORTE'05*, volume 3731 of *LNCS*, pages 1–12. Springer-Verlag, 2005.

[3] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, pages 164–180. Springer-Verlag, 2005.

[4] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis of single-parent heaps. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'07*, volume 4349 of *LNCS*, pages 91–105. Springer-Verlag, 2007.

[5] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'04*, volume 2988 of *LNCS*, pages 388–403. Springer-Verlag, 2004.

[6] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 457–461. Springer-Verlag, 2004.

[7] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *ACM Conference on Programming Language Design and Implementation, PLDI'01*, volume 36 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, 2001.

[8] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, pages 268–283. Springer-Verlag, 2001.

[9] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'02*, pages 1–3. ACM Press, 2002.

[10] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[11] M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects, FMCO'05*, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2005.

[12] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification, CAV'07*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, 2007.

[13] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Computer Aided Verification, CAV'07*, volume 4590 of *LNCS*, pages 178–192. Springer-Verlag, 2007.

[14] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'04*, volume 3328 of *LNCS*, pages 97–109. Springer-Verlag, 2004.

[15] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems, APLAS'05*, volume 3780 of *LNCS*, pages 52–68. Springer-Verlag, 2005.

[16] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *Computer Aided Verification, CAV'08*, volume 5123 of *LNCS*, pages 399–413. Springer-Verlag, 2008.

[17] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'07*, volume 4349 of *LNCS*, pages 378–394. Springer-Verlag, 2007.

[18] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Computer Aided Verification, CAV'06*, volume 4144 of *LNCS*, pages 532–546. Springer-Verlag, 2006.

[19] J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'06*, volume 3855 of *LNCS*, pages 207–221. Springer-Verlag, 2006.

[20] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM Conference on Programming Language Design and Implementation, PLDI'03*, pages 196–207. ACM Press, 2003.

[21] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making Parametric Shape Analysis Competitive. In *Computer Aided Verification, CAV'07*, volume 4590 of *LNCS*, pages 221–225. Springer-Verlag, 2007.

[22] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *Computer Aided Verification, CAV'06*, volume 4144 of *LNCS*, pages 517–531. Springer-Verlag, 2006.

[23] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, volume 3440 of *LNCS*, pages 13–29. Springer-Verlag, 2005.

[24] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. *Electronic Notes in Theoretical Computer Science, ENTCS*, 149(1):37–48, 2006.

[25] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Static Analysis Symposium, SAS'06*, volume 4134 of *LNCS*, pages 52–70. Springer-Verlag, 2006.

[26] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. C. Rinard. Using First-Order Theorem Provers in the Jahob Data Structure Verification System. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'07*, volume 4349 of *LNCS*, pages 74–88. Springer-Verlag, 2007.

[27] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(10):677–691, 1986.

[28] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Footprint Analysis: A Shape Analysis That Discovers Preconditions. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, pages 402–418. Springer-Verlag, 2007.

[29] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by means of Bi-Abduction. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'09*. ACM Press, 2009. To appear.

[30] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering'03*, pages 385–395. IEEE Computer Society, 2003.

[31] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *Static Analysis Symposium, SAS'07*, pages 384–401. Springer-Verlag, 2007.

[32] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Language Design and Implementation, PLDI'90*, pages 296–310. ACM Press, 1990.

[33] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'07*, volume 4424 of *LNCS*, pages 19–33. Springer-Verlag, 2007.

[34] S. Cherem and R. Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'07*, volume 4349 of *LNCS*, pages 234–250. Springer-Verlag, 2007.

[35] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification, CAV'00*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.

[36] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Static Analysis Symposium, SAS'05*, volume 3672 of *LNCS*, pages 87–101. Springer-Verlag, 2005.

[37] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'77*, pages 238–252. ACM Press, 1977.

[38] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'79*, pages 269–282. ACM Press, 1979.

[39] D. Dams and K. S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'03*, volume 2575 of *LNCS*, pages 310–323. Springer-Verlag, 2003.

[40] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Symposium on Logic in Computer Science, LICS'01*, pages 51–60. IEEE Computer Society, 2001.

[41] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[42] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06*, volume 3920 of *LNCS*, pages 287–302. Springer-Verlag, 2006.

[43] B. Finkbeiner. Language containment checking with nondeterministic BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, pages 24–38. Springer-Verlag, 2001.

[44] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation, PLDI'02*, pages 234–245. ACM Press, 2002.

[45] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'02*, pages 191–202. ACM Press, 2002.

[46] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. American Mathematical Society, 1967.

[47] E. Grädel. Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics. In *Proceedings of Kalmár Workshop on Logic and Computer Science*, 2003.

[48] E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Symposium on Logic in Computer Science, LICS'99*, pages 45–54. IEEE Computer Society, 1999.

[49] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, CAV'97*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.

[50] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'08*, pages 235–246. ACM Press, 2008.

[51] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'05*, pages 310–323. ACM Press, 2005.

[52] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'04*, pages 232–244. ACM Press, 2004.

[53] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'02*, pages 58–70. ACM Press, 2002.

[54] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *SPIN 03: Model Checking of Software*, volume 2648 of *LNCS*, pages 235–239. Springer-Verlag, 2003.

[55] R. J. Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[56] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[57] N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.

[58] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The Boundary Between Decidability and Undecidability for Transitive-Closure Logics. In *Computer Science Logic, CSL'04*, volume 3210 of *LNCS*, pages 160–174. Springer-Verlag, 2004.

[59] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 281–294. Springer-Verlag, 2004.

[60] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *Computer Aided Verification, CAV07*, volume 4590 of *LNCS*, pages 193–206. Springer-Verlag, 2007.

[61] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'79*, pages 244–256. ACM Press, 1979.

[62] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[63] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *International Conference on Implementation and Application of Automata'00*, volume 2088 of *LNCS*, pages 182–194. Springer-Verlag, 2000.

[64] N. Klarlund and M. I. Schwartzbach. Graph types. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'93*, pages 196–205. ACM Press, 1993.

[65] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[66] V. Kuncak. The Jahob project web page. `http://lara.epfl.ch/dokuwiki/doku.php?id=jahob_system`, 2008.

[67] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'02*, pages 17–32. ACM Press, 2002.

[68] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, 2006.

[69] V. Kuncak and M. Rinard. Boolean Algebra of Shape Analysis Constraints. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'04*, volume 2937 of *LNCS*, pages 59–72. Springer-Verlag, 2004.

[70] V. Kuncak and M. Rinard. Decision procedures for set-valued fields. In *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005.

[71] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'04*, volume 2937 of *LNCS*, pages 267–281. Springer-Verlag, 2004.

[72] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*. Springer-Verlag, 2004.

[73] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'06*, pages 115–126. ACM Press, 2006.

[74] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'08*, pages 171–182. ACM Press, 2008.

[75] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *International Conference on Compiler Construction, CC'05*, volume 3443 of *LNCS*, pages 237–241. Springer-Verlag, 2005.

[76] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publishers, 1999.

[77] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symposium on Programming, ESOP'05*, volume 3444 of *LNCS*, pages 124–140. Springer-Verlag, 2005.

[78] K. R. M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing*, 5(4):330–360, 1998.

[79] T. Lev-Ami. TVLA: A Framework for Kleene Based Logic Static Analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[80] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conference on Automated Deduction, CADE-20*, volume 3632 of *LNCS*, pages 99–115. Springer-Verlag, 2005.

[81] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Computer Aided Verification, CAV'06*, pages 533–546, 2006.

[82] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, pages 519–533. Springer-Verlag, 2005.

[83] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, pages 419–436. Springer-Verlag, 2007.

[84] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'07*, volume 4424 of *LNCS*, pages 3–18. Springer-Verlag, 2007.

[85] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium, SAS'04*, volume 3148 of *LNCS*, pages 265–279. Springer-Verlag, 2004.

[86] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *LNCS*, pages 181–198. Springer-Verlag, 2005.

[87] S. McPeak and G. C. Necula. Data Structure Specifications via Local Equality Axioms. In *Computer Aided Verification, CAV'05*, volume 3576 of *LNCS*, pages 476–490. Springer-Verlag, 2005.

[88] Static Driver Verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, 2008.

[89] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM Conference on Programming Language Design and Implementation, PLDI'01*, pages 221–231. ACM Press, 2001.

[90] G. Nelson. Verifying reachability invariants of linked structures. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'83*, pages 38–47. ACM Press, 1983.

[91] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *Computer Aided Verification, CAV'08*, volume 5123 of *LNCS*, pages 355–369. Springer-Verlag, 2008.

[92] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[93] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic, BSL*, 5(2):215–244, 1999.

[94] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, CSL'01*, pages 1–19. Springer-Verlag, 2001.

[95] A. Podelski and A. Rybalchenko. Transition Invariants. In *Symposium on Logic in Computer Science, LICS'04*, pages 32–41. IEEE Computer Society, 2004.

[96] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'05*, pages 132–144. ACM Press, 2005.

[97] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *Computer Aided Verification, CAV'08*, volume 5123 of *LNCS*, pages 314–327. Springer-Verlag, 2008.

[98] A. Podelski and T. Wies. Boolean Heaps. In *Static Analysis Symposium, SAS'05*, volume 3672 of *LNCS*, pages 267–282. Springer-Verlag, 2005.

[99] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[100] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming, ESOP'03*, volume 2618 of *LNCS*, pages 380–398. Springer-Verlag, 2003.

[101] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Symposium on Logic in Computer Science, LICS'02*, pages 55–74. IEEE Computer Society, 2002.

[102] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, pages 220–236. Springer-Verlag, 2007.

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

[104] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18:227–258, 1982.

[105] Sun Microsystems. Java 2 Platform, Standard Edition, v 5.0 API Specification. http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html, 2004.

[106] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[107] C. J. Van Wyk. *Data Structures and C Programs*. Addison-Wesley, 1991.

[108] A. Voronkov. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[109] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[110] T. Wies. Symbolic Shape Analysis. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 2004.

[111] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field Constraint Analysis. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'06*, volume 3855 of *LNCS*, pages 157–173. Springer-Verlag, 2006.

[112] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *Heap Analysis and Verification, HAV'07*, 2007.

[113] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems, TECS*, 7(3), 2008.

[114] E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'01*, pages 27–40. ACM Press, 2001.

[115] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification, CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer-Verlag, 2008.

[116] G. Yorsh. Logical Characterizations of Heap Abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003.

[117] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A Logic of Reachable Patterns in Linked Data-Structures. In *Foundations of Software Science and Computation Structures'06*, volume 3921 of *LNCS*, pages 94–110. Springer-Verlag, 2006.

[118] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'04*, volume 2988 of *LNCS*. Springer-Verlag, 2004.

[119] G. Yorsh, T. W. Reps, M. Sagiv, and R. Wilhelm. Logical Characterizations of Heap Abstractions. *ACM Transactions on Computational Logic, TOCL*, 8(1), 2007.

[120] G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *1st AIOOL Workshop*, 2005.

[121] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification for Linked Data Structures. In *ACM Conference on Programming Language Design and Implementation, PLDI'08*. ACM Press, 2008.