Software Verification by Combining Program Analyses of Adjustable Precision

THÈSE N^{O} 4781 (2010)

PRÉSENTÉE LE 24 SEPTEMBRE 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MODÈLES ET THÉORIE DE CALCULS
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Grégory THÉODULOZ

acceptée sur proposition du jury:

Prof. E. Telatar, président du jury
Prof. T. Henzinger, directeur de thèse
Prof. D. T. Beyer, rapporteur
Prof. V. Kuncak, rapporteur
Prof. R. Majumbar, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

A mes grands-parents...

Abstract

In automatic software verification, we have observed a theoretical convergence of model checking and program analysis. In practice, however, model checkers, on one hand, are still mostly concerned with precision, e.g., the removal of spurious counterexamples. Lattice-based program analyzers, on the other hand, are primarily concerned with efficiency. To achieve their respective goal, the former builds and refine reachability tress while the latter annotates location with abstract states and rely on overapproximation to accelerate convergence. In this thesis we focus on capturing within a framework existing approaches as well as new solutions with the objective of enabling a better understanding of the fundamental similitudes and differences between approaches and with a strong accent on implementability.

In a first step, we designed and implemented a framework and a corresponding algorithm for software verification called *configurable program analysis*. The algorithm can be configured to perform not only a purely tree-based or a purely lattice-based analysis, but offers many intermediate settings that have not been evaluated before. An instance of an analysis in the framework consists of one or more program analyses, such as a predicate abstraction or a shape analysis, and their execution and interaction is controlled using several parameters of our generic verification algorithm. Our experiments consider different configurations of combinations of symbolic analyses. By varying the value of parameters we were able to explore a continuous precision-efficiency spectrum and we showed that it can lead to dramatic improvements in efficiency.

In a second step, we improved our framework and algorithm to enable the program analysis to dynamically (on-line) adjust its precision depending on the accumulated results. The framework of configurable program analysis offers flexible, but static, composition of program analyses. Our extension enables composite analyses to adjust the precision of each of their component analyses independently and dynamically. To illustrate, we can allow the explicit tracking

of the values of a variable to be switched off in favor of a predicate abstraction when and where the number of different variable values that have been encountered has exceeded a specified threshold. We evaluated the dynamic precision adjustment mechanism by considering combinations of symbolic and explicit analyses. We analyzed code taken from an SSH client/server software as well as hand-crafted examples. We showed that the new approach offers significant gains compared with a purely symbolic, predicate-abstraction-based approach.

In a third step, we consider the problem of refinement in addition to the dynamic adjustment of the precision. In contrast to precision adjustment, refinement only increases the precision of the analysis. Moreover, when a refinement occurs, states with a lower precision are discarded and replaced by states with a higher precision. Based on our framework, we present a novel refinement approach for shape analysis, a promising technique to prove program properties about recursive data structures. The challenge is to automatically determine the data-structure type, and to supply the shape analysis with the necessary information about the data structure. We present a stepwise approach to the selection of instrumentation predicates for a TVLA-based shape analysis, which takes us a step closer towards the fully automatic verification of data structure implementations. The approach uses two techniques to guide the refinement of shape abstractions. First, during program exploration, an explicit heap analysis collects sample instances of the heap structures. The samples are used to identify the data structures that are manipulated by the program. Second, during abstraction refinement along an infeasible error path, we consider different possible heap abstractions and choose the coarsest one that eliminates the infeasible path. We were able to successfully verify example programs from a data-structure library that manipulate doubly-linked lists and trees.

The techniques presented in this thesis have been implemented as an extension to the BLAST model checker.

Keywords: *software verification; model checking; program analysis framework; composition of program analyses; abstraction refinement; shape analysis.*

Résumé

Une convergence théorique a été observée dans le domaine de la vérification automatique de logiciels entre l'analyse statique de programme et le *model checking*. Néanmoins, en pratique, d'un côté, les outils basés sur le model checking se concentrent principalement sur l'augmentation de la précision, par exemple en diminuant le nombre de fausses alarmes, alors que d'un autre côté, les analyseurs statiques basés sur un treillis d'états abstraits se concentrent sur l'efficacité de l'analyse. Pour atteindre leur objectif, les premiers construisent à cette fin des arbres abstraits représentant les états atteignables et les raffinent quand nécessaire, alors que les seconds annotent chaque positions du programme avec un état du treillis et utilise des approximations conservatives pour converger rapidement. Dans cette thèse, nous tentons de regrouper dans un cadre général ces deux familles d'approches aussi bien que de nouvelles approches. Notre objectif est de mettre en évidence les similitudes et différences fondamentales de ces techniques, en mettant un fort accent sur la capacité d'implémenter aisément notre nouveau cadre général.

Premièrement, nous avons conçu et implémenté un cadre général, dénommé *analyse configurable de programmes* avec son algorithme d'analyse permettant la vérification automatique de logiciels. L'algorithme peut être configuré non seulement pour se comporter comme une analyse basée sur un arbre d'états abstraits atteignables (comme utilisé dans un model checker) ou une analyse basée sur un treillis (comme utilisé dans une analyse statique du flot de donnée) mais également pour définir des analyses intermédiaires qui n'avaient auparavant pas été considérées. Une instance d'analyse configurable de programmes se compose d'une ou plusieurs analyses de programme (par exemple une analyse basée sur des prédicats et une analyse de structures de données dynamiques. La manière dont les analyses s'exécutent et interagissent est contrôlée par plusieurs paramètres de notre algorithme générique de vérification. Pour notre évaluer notre approche, nous avons comparé expérimentalement des

combinaisons d'analyses symboliques sous diverses configurations. En variant la valeur des paramètres de notre algorithme, nous pouvons explorer un spectre d'analyse correspondant à différents compromis précision-coût, et nous avons réussi à mettre en évidence que certains choix de paramètres conduisent à des augmentations significatives de l'efficacité de l'analyse.

Deuxièmement, nous avons amélioré notre cadre général et son algorithme de vérification de manière à permettre l'ajustement dynamique de la précision de l'analyse en fonction des résultats accumulés. Dans la version originale de notre cadre général, les analyses peuvent être combinées de manière flexible mais uniquement tous les choix sont statiques alors que l'extension que nous proposons permet d'obtenir des combinaisons de plusieurs analyses où la précision de chacune des analyses peut être ajustée indépendamment. A titre d'illustration, nous arrêter de considérer la valeur exacte d'une variable (analyse explicite) au profit d'une analyse de prédicats relatifs à cette variable (analyse symbolique) au moment et à l'endroit où le nombre de valeurs différentes qui ont été rencontrées pour cette variable dépasse un certain seuil. Nous avons évalué expérimentalement notre mécanisme d'ajustement dynamique de la précision en considérant des combinaisons d'une analyse explicite avec une analyse symbolique. Nous avons analysé des fragments de code provenant d'un client et d'un serveur SSH ainsi que sur des exemples artificiels. Nous obtenons avec notre nouvelle approche des temps de vérification significativement inférieurs à une approche purement symbolique.

Troisièmement, nous ajoutons à notre cadre général le raffinement de l'abstraction, en plus de l'ajustement dynamique de la précision. Un raffinement ne peut qu'augmenter la précision de l'analyse alors que l'ajustement dynamique de la précision peut à la fois augmenter et diminuer la précision. De plus, lorsque qu'un raffinement a lieu, les états avec une précision trop basse sont écartés pour être remplacés par des états avec une précision plus élevée, alors qu'avec l'ajustement dynamique de la précision, aucun état n'est écarté lorsque la précision est changée. Nous utilisons notre cadre général pour présenter une nouvelle approche pour le raffinement d'une analyse de structures de données dynamiques (*shape analysis*). Le défi principal de ce type d'analyse est la découverte des informations dont l'analyse a besoin pour analyser la structure de donnée, ce qui inclut l'identification automatique du type de structure de donnée manipulée. Nous présentons une approche qui raffine par étape la précision de l'analyse (basée sur l'outil TVLA) et qui nous rapproche d'une solution complètement automatique à la vérification des implémentations de structures de données. Notre solution utilise deux techniques pour guide le raffinement. Première-

ment, durant l'exploration des états, une analyse explicite du tas collecte des instances de structures de données. Ces instances sont utilisées pour identifier le type de structure de données manipulées par le programme. Deuxièmement, lorsque d'une fausse alarme est découverte, nous choisissons l'abstraction la plus imprécise permettant d'éliminer la fausse alarme parmi toutes les abstractions que l'algorithme considère comme possible. Nous avons vérifié avec succès des programmes provenant d'une librairie de structures de données manipulant des listes doublement chaînées et des arbres binaires.

Toutes les techniques présentées dans cette thèse ont été implémentée en tant qu'extension à l'outil de vérification BLAST.

Mots-clés: *vérification logicielle; model checking; analyse statique; composition d'analyses statiques; raffinement d'abstractions; analyse de structures de donnée.*

Acknowledgments

First and foremost I would like to acknowledge the unconditional support of my parents and my family. They have constantly encouraged me to do what I liked and never questioned my choices. I would not embarrass them with a lengthy, emotional acknowledgement of all they gave to me, as outrageous display of emotions is not in my genes. It suffices to say that without their love and support, I would not be the person that I am and I cannot be thankful enough.

Two persons have had a crucial role in shaping me as a researcher: Tom and Dirk. The support of Tom, my thesis supervisor, has been a reassuring presence throughout the four years of my thesis. I met Tom when I took the first class he gave at EPFL in 2005. He made me discover the world of automatic verification, and convinced me that it was an area in which interesting research could be conducted. I can only applaud his skills as a teacher and as a researcher. Seldom can one hope to encounter a person as insightful and understanding as Tom. I started my collaboration with Dirk when he was a postdoctoral researcher in our lab at EPFL, and have not stopped collaborating with him ever since. Dirk helped me focusing my thoughts, by his attention to details, restraining my natural tendency of thinking quickly and carelessly. It has been a great pleasure to work with him even after he had left EPFL.

I would also like to acknowledge all the people that have been at some point member of our lab, including, in no particular order, Barbara, Jasmin, Laura, Verena, Andrey, Damien, Dejan, Dietmar, Laurent, Nir, Simon, Tatjana, Thomas, and Vasu. I would also like to thank Rupak who hosted me for three months in Los Angeles. A special note must be made about my officemate and friend Maria. I was lucky to meet Maria when she joined our lab and we have shared an office ever since. Somehow, we succeeded in spending four years together without any tension of any kind. We shared our moments of joy and

doubts. Chatting with her has always been a pleasure, and it is unarguable that we succeeded in creating the most pleasant work environment.

I cannot stress enough how important my friends have been throughout my thesis. I would particularly mention, in alphabetical order, Corinne, Frédéric, Grégory, Marc, Samuel, and Thierry, and all my friends inside and outside of EPFL.

Life at EPFL would not be quite the same without dedicated administrative staff. In our lab, Sylvie helped me through all administrative duties and Fabien provided an impeccable IT infrastructure, both in a particularly friendly manner. I would also like to thank the administration of the school of computer and communication sciences and of the doctoral program in computer, communication and information sciences. Collectively they provided excellent information and support throughout my life at EPFL. I would like to acknowledge in particular the excellent work of Sylviane and Cecilia. Both are dedicated to the well-being of the school and the student body.

I would like to thank for their time the member of my thesis jury, Emre, Tom, Dirk, Rupak and Viktor. The comments of the examiners on the draft of the thesis helped shaping its final form.

Finally, I would like to thank the sponsors that have generously funded my research: Microsoft Research through its European PhD Scholarship Program and the Swiss National Science Foundation.

Contents

Abstract **v**

Résumé **vii**

Acknowledgments **xi**

Contents **xiii**

List of Figures **xvii**

List of Tables **xix**

List of Algorithms **xxi**

Previous Publications **xxiii**

1 Introduction **1**

1.1 A Unified and Flexible Analysis Framework 2

1.2 Combination of Analyses 8

1.3 Dynamically Adjustable Precision 9

1.4 Combination of Analyses for Refinement 11

1.5 Structure of the Thesis 12

2 Preliminaries **15**

2.1 Programs 15

2.1.1 Control-Flow Automata and Types 15

2.1.2 Concrete Semantics 19

2.2 Verification Problem 21

2.3 Shape Analysis 23
2.3.1 Shape Classes 23
2.3.2 Two-valued Shape Graphs 24
2.3.3 Three-valued Shape Graphs, Abstraction and Embedding 26
2.3.4 Abstract Program Semantics 28
2.3.5 Shape Abstraction and Shape Regions 32
2.3.6 Tracking Definitions and Shape-Class Generators 33

3 Configurable Program Analysis 35

3.1 Motivation 35
3.2 Formalism and Algorithm 37
3.2.1 Preliminaries 37
3.2.2 Configurable Program Analysis (CPA) 39
3.2.3 Reachability Algorithm for CPA 43
3.2.4 CPA for Location Analysis 49
3.2.5 CPA for Predicate Analysis 50
3.2.6 CPA for Shape Analysis 51
3.2.7 Composite Program Analysis 52
3.3 Comparison with Data-flow Analysis and Abstract Interpretation 57
3.3.1 Encoding a Traditional Program Analysis as a CPA . . . 58
3.3.2 Encoding a CPA as a Join-Based Analysis 61
3.4 Application: Configuring Compositions of Analyses 62
3.4.1 Configuring Predicate Abstraction + Shape Analysis . . . 63
3.4.2 Configuring Predicate Abstraction + Pointer Analysis . . 71
3.5 Related Work 72
3.6 Conclusion 74

4 Dynamic Precision Adjustment 75

4.1 Motivation 75
4.2 Related Work 78
4.3 Program-Analysis Framework 79
4.3.1 CPA with Dynamic Precision Adjustment (CPA+) 79
4.3.2 Reachability Algorithm for CPA+ 82

4.3.3 Composition for CPA+ 86
4.4 Application: Combining Explicit and Symbolic Program Analyses 87
4.4.1 CPA+ for Location Analysis 88
4.4.2 CPA+ for Predicate Analysis 89
4.4.3 CPA+ for Shape Analysis 91
4.4.4 CPA+ for Explicit Value and Heap Analysis 91
4.4.5 Composition of Explicit, Predicate, and Location Analysis 96
4.4.6 Composition of Explicit, Shape, and Location Analysis . . 97
4.5 Experimental Evaluation 100
4.5.1 Explicit Value Analysis and Predicate Analysis 100
4.5.2 Explicit Heap Analysis and Shape Analysis 103
4.6 Conclusion 104

5 Shape Abstraction Refinement 107
5.1 Motivation 107
5.2 Related Work 111
5.3 Preliminaries 113
5.3.1 Path Formulas 113
5.3.2 Interpolation 121
5.4 Shape Analysis with Abstraction and Refinement 123
5.4.1 Interruptible CPA+ and Reachability Algorithm 124
5.4.2 Interruptible CPA+ for Path Analysis 126
5.4.3 Interruptible CPA+ for Explicit Heap Analysis 127
5.4.4 Composite Interruptible CPA+ for Path, Shape, and Explicit Heap Analysis 128
5.4.5 Model-Checking Algorithm (*ModelCheck*) 130
5.4.6 Algorithm for Abstraction from Explicit Heaps (*Abstract*) 131
5.4.7 Algorithm for Shape Refinement (*Refine*) 134
5.5 Implementation 137
5.5.1 Library of Shape-Class Generators 138
5.5.2 Manual Annotations of Data Types 141
5.6 Experimental Evaluation 142
5.6.1 Example Programs 142

5.6.2 Results 144
5.7 Conclusion 146

6 Conclusion **147**

Bibliography **151**

Curriculum Vitae **159**

List of Figures

1.1 A CPA allows to explore the entire precision-efficiency spectrum 6

2.1 Grammar of program operations. 16
2.2 Example C Program 17
2.3 Control-flow automaton of the program in Figure 2.2 18
2.4 Strongest postcondition of program operations 20
2.5 Example two-valued shape graph 25
2.6 Lattice of logical values in three-valued logic with respect to information order 26
2.7 Example three-valued shape graph 26
2.8 Example three-valued shape graph for a shape class with fewer instrumentation predicates 27
2.9 Result of applying *focus* on the shape graph of Figure 2.7 30
2.10 Result of applying *coerce* on the shape graphs of Figure 2.9 . . . 32

3.1 Example lattice for constant propagation with two variables . . 38
3.2 Example program and corresponding CFA 41
(a) C program 41
(b) CFA 41
3.3 Example C program 64
3.4 Example of shape graphs computed during the analysis of the program in Figure 3.3 65

4.1 Example program 77
4.2 Sample explicit-analysis state 91

4.3 Shape graph that is an abstraction of the sample explicit heap depicted in Figure 4.2 97

5.1 Example C program 109

5.2 Sample abstract states 110

(a) Sample explicit heap 110

(b) Sample shape graph 110

5.3 Definition of Con for each program operation 115

5.4 Path formula and its interpolant for an infeasible path of the program in Figure 2.3 118

5.5 Hierarchy of data structures 138

List of Tables

3.1 Configurations of predicate and shape analysis 67

3.2 Verification time for different configurations 68

(a) Time for predicate abstraction and shape analysis 68

(b) Time for predicate abstraction and pointer analysis 68

4.1 Performance evaluation of dynamic precision adjustment of predicate analysis and explicit value analysis 101

(a) Extreme examples 101

(b) SSH client/server software 101

4.2 Performance evaluation of dynamic precision adjustment of shape analysis and explicit heap analysis 103

5.1 Library of SCG used in the experiments 140

5.2 Verification time without and with shape refinement 143

List of Algorithms

3.1 $CPA(\mathbb{D}, P, e_0)$ 44

4.1 $CPA{+}(\mathbb{D}^{+}, P, e_0, \pi_0)$ 82

5.1 $ExtractInterpolants(t, \Gamma)$ 122

5.2 $PartialCPA{+}(\mathbb{D}^{+}, P, R_0, F_0)$ 125

5.3 $ModelCheck(P, l_{err}, M)$ 130

5.4 $Abstract(R, F, M, E)$ 132

5.5 $Refine(t, R, F, M, E)$ 135

Previous Publications

The research presented in Chapter 3 appeared as Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, “Configurable Software Verification: Concretizing the convergence of Model Checking and Program Analysis” in the *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, Lecture Notes in Computer Science 4590, pages 504–518. Springer, 2007.

The research presented in Chapter 4 appeared as Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, “Program Analysis with Dynamic Precision Adjustment” in the Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pages 29–38. IEEE, 2008.

The research presented in Chapter 5 appeared, using a different formalization, as Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, and Damien Zufferey, “Shape Refinement through Explicit Heap Analysis” in the Proceeding of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010), Lecture Notes in Computer Science 6013, pages 263–277. Springer, 2010.

Chapter 1

Introduction

Proving properties of software systems is a challenging problem that has interested mathematicians and computer scientists since the 1930s. Software systems were initially used in niche applications, at a time where computers had little memory, little processing power, and a prohibitive cost. Nowadays, software systems are applied in an ever-growing number of applications. Software is ever more pervasive, being used in a broad variety of embedded systems found in cars, planes, phones, and other personal gadgets, upto large information processing systems prevalent in the corporate world and on the Internet. As a consequence of our increased dependency on software for our everyday tasks and for business, software failures, generally referred to as *bugs*, have consequences that range from a mere loss of productivity to more dramatic consequences, from financial losses to endangerment of life. Over the last decade, the software industry has started to realize that testing, even if done with state-of-the-art technique, is not always sufficient to validate systems. The hardware industry, where quality control is crucial due to the costs involved in fixing bugs after chips have been shipped, pushed the development and the adoption of formal methods for automated verification. The same movement can be observed today in some part of the software industry, in particular for safety-critical code.

In this thesis, we attempt to build tools to provide an automatic answer to a particular question about programs: can the program enter a given set of *bad* states? The problem is know as the *reachability* problem. The set of bad states contains the states that violate a partial-correctness specification. Proving any safety property can be reduced to reachability. For instance, one can use reachability to check whether any assertion can be violated, or whether some temporal property is violated (e.g., the program has made an invalid sequence

of calls to a programming interface). We focus on automated techniques that do not require the verification engineer or the programmer to annotate the code with preconditions, postconditions, or loop invariants. The reachability problem cannot be solved by a naive, explicit exploration of all reachable states because software systems are inherently infinite (unlike hardware systems). For instance, a program may create lists on the heap of an unbounded length. In fact, the reachability problem is well-known to be undecidable for Turing-complete programming languages. As a consequence, no technique can hope to give a definite answer within a finite amount of time for every program (i.e., no technique is *complete*).

Static program analyses have been used to determine whether a given program is free of bad behaviors, without actually running the program. The result of a static analysis is generally conservative (i.e., the analysis is sound): a static analysis is used to categorically prove that a program is free of bad behaviors, but the analysis may fail, not terminate, or report false positives (i.e., the analysis may not be complete). In contrast, dynamic techniques (e.g., testing) rely on running the program and as a consequence report only actual bugs. Nevertheless, in general, they are not able to prove the absence of bugs because they can only cover a subset of all possible program executions.

There is a fundamental trade off between the precision and the computational complexity of a static analysis. The goal of our research is to build theoretical frameworks and tools to explore the precision/efficiency spectrum of program analyses. In the rest of this chapter, we present and put in context the main contributions of this thesis. Our central contribution is a unified framework for program analysis that can be configured to simulate many existing techniques. We then build on this framework to explore three different features that can improve the analysis. First, we provide a mechanism for flexible combination of analyses. Second, we extend our framework to allow the precision of the analysis to be changed while the analysis is running. Third, we show how combinations of analyses can be used to improve a refinement strategy.

1.1 A Unified and Flexible Analysis Framework

Historically, the first static analyses were used in compilers [Aho et al. 1986] to enable optimizations, mostly in the form of data-flow analysis. As a consequence, those analyses have a strong focus on efficiency and they target relatively simple properties. At the other end of the precision spectrum lies software

model-checking, a technique that attempts to prove sophisticated properties with as few false positives as possible at the expense of being computationally expensive. While some data-flow analyses can routinely analyze programs with millions of lines of code, software model-checking can only consider programs that are orders of magnitude smaller.

In the following, we give a brief overview of the three families of techniques for software verification that we mostly focus on: data-flow analysis, abstract interpretation, and software model-checking. The research on automatic software verification is not limited to those three techniques. For instance, type systems can be seen as a verification tool: they prevent certain bad behavior from occurring, but the type systems found in traditional programming languages provide very shallow analyses. More ambitious approaches include typestates [Strom and Yemini 1986; Field et al. 2003]. Other techniques generate verification conditions [Floyd 1967; Hoare 1969] and discharge them to theorem provers and decision procedures [Nelson and Oppen 1979; Leino and Nelson 1998; Flanagan et al. 2002; Barnett et al. 2005; Lam et al. 2005]. Traditionally, those techniques require the programmer or the verification engineer to provide preconditions, postconditions, and sometimes loop invariants. Bounded-model checking represents a subset of the paths in the program by a formula that a SAT solver analyzes [Clarke et al. 2001; Clarke et al. 2004]. The method is not sound in general because loops are unrolled only finitely many times. All those techniques have been successful in their own right, but their are out of the scope of our discussion.

Data-flow analysis Data-flow analysis has its root in traditional program analyses found in compilers [Kildall 1973; Aho et al. 1986] and is historically the first family of static analyses that has been thoroughly studied. As a consequence, data-flow analyses are generally focused on efficiently establishing simple properties. The analysis annotates program locations with facts that hold at the given location. For instance, an uninitialized variable data-flow analysis annotates program locations with sets of variables that might be uninitialized; a constant propagation analysis annotates program locations with a partial function from variables to constants, representing the fact that a variable has a constant value at that location; a shape analysis annotates program locations with shape graphs representing the shape of data structure instances on the heap. The facts are represented by elements of a semi-lattice [Davey and Priestley 1990] and the problem consists in finding the solution of a set of data-flow equa-

tions. Transfer functions model the effect of program operations on lattice elements (e.g., for an uninitialized variable analysis, the transfer function for an operations that assign a value to variable `x` maps a set U of variables to $U \setminus \{\mathtt{x}\}$). We briefly sketch in the following how a traditionnal iterative algorithm [Kildall 1973] solves the data-flow equations. The join operation of a semi-lattice yields the least upper-bound of a given set of elements, according to the pre-order of the lattice. The data-flow equations can be solved by a fix-point algorithm that starts with an initial state, iteratively applies transfer functions to elements according to program operations and, for locations with more than one predecessor (*join points*), merges information from all predecessors using the join operator of the lattice. Precision may be lost at join points because states are combined using the (possibly overapproximating) join operator, which potentially leads to path insensitivity. If the lattice does not have infinite ascending chains, the data-flow analysis is guaranteed to terminate using a number of lattice operations polynomial in the height of the lattice. Extensive research has been conducted in studying different iteration orders and computation strategies to improve the efficiency (but not the precision) of the analysis [Hecht and Ullman 1973; Kennedy 1975; Hecht 1977; Sharir 1980; Dhamdhere et al. 1992]. The crux to control the precision of the data-flow analysis lies in the lattice of data-flow facts (i.e., its *domain*). Analyses such as uninitialized variables or constant propagation result in efficient, polynomial-time analysis algorithms, whereas analyses such as shape analysis result in exponential analysis algorithms because of the complexity of the operations on lattice elements.

Abstract Interpretation Abstract interpretation provides a framework to systematically design analyses, based on the relation between a concrete domain and an abstract domain [Cousot and Cousot 1977]. When applied to program analysis [Cousot and Cousot 1979], the concrete domain is the set of program states (concrete states), and the concrete system captures precisely the semantics of the program. For example, if we assume a program with only integer variables, a concrete state is a mapping from variables to integers. The abstract domain contains abstract states representing (possibly infinite) sets of concrete program states. For example, an abstract domain for interval analysis contains for each variable an interval of values. The concrete and abstract domains are linked by an abstraction function (mapping sets of concrete states to an abstract state) and a concretization function (mapping an abstract state to the set of concrete states

it represents). Abstract interpretation works by *interpreting* the behavior on the abstract domain to conservatively deduce facts about the concrete program. Abstract interpreters work by computing a fix-point solution similarly to data-flow analyses. In fact data-flow analysis is a special case of abstract interpretation. To accelerate (or allow) convergence, widening operators can be used [Cousot and Cousot 1992; Bourdoncle 1993]. For instance, a widening operator for an interval analysis may replace some interval bounds by infinity. Similarly to data-flow analysis, the precision and efficiency of an abstract interpreter depend on the domain, but also on the widening operator. Abstract interpretation has been successfully applied to large code bases, in particular in the aeronautical industry, to prove numerical properties [Blanchet et al. 2003]

Software model-checking Software model-checking applies model checking [Clarke et al. 1999], an exhaustive state-space exploration technique, to software systems [Jhala and Majumdar 2009]. Because the state space of software systems is generally infinite, software model-checking needs to use symbolic representations of sets of states. Moreover, in order to make the analysis practical, it is generally necessary to resort to *abstraction*: the infinite-state concrete system is abstracted to a finite-state abstract system amenable to model checking. For instance, predicate abstraction [Graf and Saïdi 1997; Ball et al. 2001] can be used: given a set P of predicates over program variables, a set C of program states is abstracted by the strongest conjunction of predicates in P that holds for all states in C. For a given abstraction, the model-checking algorithm exhaustively explores the states of the finite abstract system. When an (abstract) path to the error location is found, it does not necessarily correspond to a path of the concrete program; it might be due to the abstraction being too coarse. As a result, the abstraction is refined such that the error path is not encountered in the refined abstract system (counterexample abstraction refinement [Clarke et al. 2003]). For example, for a predicate abstraction, the abstraction is refined by adding predicates to the set of tracked predicates. The overall process forms an abstract-check-refine loop [Saidi 2000; Ball and Rajamani 2001]. Software model checking has seen a tremendous development over the last decade, with many tool implementations available [Godefroid 1997; Holzmann 1997; Corbett et al. 2000; Havelund and Pressburger 2000; Ball and Rajamani 2002; Henzinger et al. 2002; Musuvathi et al. 2002; Andrews et al. 2004; Chaki et al. 2004; Clarke et al. 2005; Ivancic et al. 2005; Esparza et al. 2006]. Software model-checking

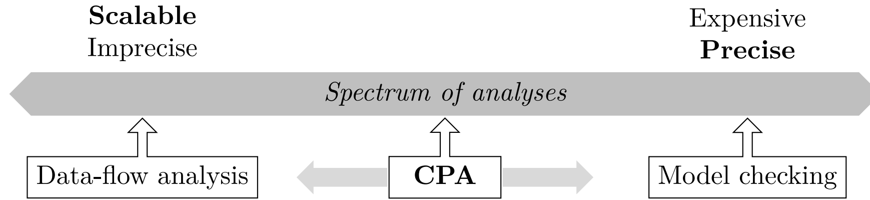


Figure 1.1: A CPA allows to explore the entire precision-efficiency spectrum

is very precise because it is path sensitive and the abstraction can be refined when necessary, but the analysis may not terminate because the abstraction might get refined infinitely often. The strength of software model checking is the verification of properties of control-intensive programs (by opposition to data-intesive programs) such as device drivers where predicate abstraction is the most suitable abstraction [Ball et al. 2006].

Abstractly, all three techniques attempt to build an overapproximation of the set of reachable states: in data-flow analyses and abstract interpreters, the set is represented by locations annotated by a lattice element or an abstract state; in model checkers, the set of reachable states can be represented by an abstract reachability tree. If the overapproximation represents no bad state, we can safely conclude that no bad state is reachable. All three techniques also rely on an abstract domain used to symbolically represent sets of program states. Nevertheless, different abstract domains are better suited to different analysis approaches. For instance, predicate abstraction is well-suited to software-model checking whereas the interval domain is not. The major point of divergence is the algorithm used to compute the overapproximation. For instance, data-flow analyses merge abstract states (with a possible precision loss) from different branches, while software model checkers never merge. The similarities of all approaches hint towards the creation of a unified framework that supersedes the different algorithms. In fact, it is well known that those approaches are theoretically equivalent in the sense that one can simulate the other [Steffen 1991; Schmidt 1998; Cousot and Cousot 1995]. Nevertheless, the corresponding research communities tend to work mostly in isolation. As a consequence, it is often hard to evaluate how the advances made by one of the communities apply to other communities, because they rely on different frameworks.

In this thesis, we present a unified, flexible framework for program analysis: *configurable program analysis*. The motivation of the framework goes beyond

the intellectual attractiveness of theoretical unification: we intended the framework to be practical, easily implementable, flexible, and compositional. Existing analyses correspond to particular instances of the framework, and they can be easily compared because they are expressed in a unified way. Not only can our framework express existing analyses, but it allows to define intermediate solutions that have not been considered in the past. As a consequence, we can picture our approach as going from a set of points to a continuous spectrum of analyses, as illustrated in Figure 1.1. A configurable program analysis (CPA) specifies separately the abstract domain and how the reachability analysis works via the definition of operators used in a generic reachability algorithm. We attempt to hard-code as few algorithmic choices as possible. Orthogonal aspects of the analysis are each represented by one operator that can be changed independently and a wide variety of configurations can be explored without changing the abstract domain. As a result, for a fixed abstract domain, we are able to configure the analysis to mimic existing algorithms as well as new algorithms by picking a particular set of operators. The operators of a CPA specifies (1) how abstract successor states are computed (transfer relation), (2) how information is merged among abstract states (merge operator), and (3) when an abstract state is already represented by the set of abstract states (termination check). In particular, the merge operator is useful to configure the algorithm to behave either like a software model checker, where no merging occurs, or like a data-flow analyzer, where all abstract states for a given location are merged together.

We adopted in our framework an operational view: operators control the behavior of our generic algorithm. In contrast, existing analysis frameworks tend to have a more declarative approach: the specification of the abstract domain and the corresponding abstract transfer relation fully characterize the analysis. For instance, a data-flow analysis is entirely defined by the lattice of facts and the corresponding transfer functions, but it is not possible to change the way states are combined without changing the domain. A consequence of our operational approach is that the implementation of the framework can follow precisely the theoretical concepts, and existing abstract domains can be more easily reused. Moreover, the implementation enables easy experimentation with different configurations. Automatic verification is often referred to as push-button verification; in the same spirit CPA can be pictured as adding sliders in addition to the one button, each controlling a particular aspect of the analysis.

1.2 Combination of Analyses

For the verification of realistic programs, it is generally not enough to use one analysis in isolation. Abstract domains are good to capture only a part of the concrete state precisely. As a consequence, if one needs to consider multiple aspects of the state space simultaneously, it is not good enough to run analyses separately: more precise results are obtained when analyses are combined. For example, a predicate abstraction captures well the possible values of integer variables, while a shape analysis [Jones and Muchnick 1982; Chase et al. 1990; Sagiv et al. 2002] captures well the content of the heap. The combination of the two analyses captures both the value of integer variables and the content of the heap. The combined analysis can be more precise than running both analyses separately: for instance, we may not merge states that have different predicate valuations, resulting in a more precise analysis result for the shape analysis.

The idea of combining analyses is not new. It has been studied both in the context of model checkers and abstract interpreters. In model checkers, the combination of predicate abstraction with shape analysis or a pointer analysis has been studied, both as a way to improve the precision and the efficiency [Beyer et al. 2006; Fischer et al. 2005]. In abstract interpreters, analyses are combined by building product domains. The weakest form of composition, known as the direct product, consists of a Cartesian product of the domain and all operations are performed independently on each component domains. It is well known that the direct product is oftentimes too imprecise [Cousot and Cousot 1979] and as a consequences other products have been defined such as the reduced product or the logical product [Cousot and Cousot 1979; Codish et al. 1993; Gulwani and Tiwari 2006].

We provide a mechanism to combine individual CPAs into a so-called *composite program analysis* in the CPA framework. Similarly to CPA, the composition mechanism uses operators to control the precision of the composition. In contrast, existing approaches modulate the precision of the analysis solely by using different product domains. A composite analysis is defined with respect to a finite set of component CPAs. The abstract domain of the composite analysis is the direct product of the domains of each component CPA. In addition to component CPAs, the composite analysis contains operators: a composite transfer relation, a composite merge operator, a composite termination check, and strengthening operators between domains. The composite operators are used to define the behavior of the composite analysis in the same way as operators of a CPA define the behavior of the analysis. While the composite domain is a di-

rect product, more precise analysis results are obtained by selecting appropriate operators. For example, the composite transfer relation can be an independent computation of component transfers (as in a normal direct product), or it can make use of strengthening operators to get a more precise result; and the composite merge operator can allow merging only when component CPA's elements agree.

We evaluated the CPA framework by considering and implementing several CPAs including one based on predicate abstraction, and one based on shape abstraction. We are able to compare experimentally different ways of combining analyses by evaluating different choices for the parameters of our generic algorithm rather than having to design a new implementation for every configuration. We formalized previously defined analyses [Beyer et al. 2006; Fischer et al. 2005] in the framework, and studied new configurations that were not studied previously. As a result we were able to explore different combinations of operators resulting in different precision and efficiency. We could identify in each case a different configuration that represented the best compromise between precision and efficiency. As a consequence, we can conclude that the flexibility that our framework enables is needed as no universally best configuration exists.

1.3 Dynamically Adjustable Precision

There exists cases where it is valuable to adjust the precision of the analysis while it is running, both for a single analysis and for a composite analysis.

For a single analysis, modulating the precision can be used to accelerate convergence. Consider for example an interval analysis. An abstract state is a set of constraints of the form $c \leq x$ or $x \geq c$ for variable x and integer constant c. Consider that the analysis is applied to a program with a loop in which variable i is incremented. An interval analysis of the program might not terminate: it might generate abstract states with constraints $i \leq b$ for ever increasing values of b. In such a case, termination could be achieved by replacing constraints of the form $i \leq b$ by $i \leq \infty$. The decision to set the bound to infinity is triggered by observing a sequence of increasing upper bounds for i. In abstract interpretation, widening operators have precisely that role and are crucial to the efficiency of the analysis [Cousot and Cousot 1977]: widening is applied after a certain number of steps to ensure convergence by locally loosing precision. Refinement in software model checkers can also be seen as an adjustment of the precision, but a refinement is only triggered when a false positive is hit. In

contrast, we are interested here in a more general precision adjustment approach where the precision can be adjusted at any time, based on previous results.

For a composite analysis where abstract domains differ in precision, we would like to switch between different analyses depending on the results obtained so far. For instance, a predicate abstraction analysis and an explicit value analysis both capture values of integer variables. A predicate abstraction can express arbitrary relations among variables such as $x + 3y \leq 10$, but predicates are taken from a finite set of predicates; and an explicit value analysis can capture precisely the value of variables but can only remember facts of the form $x = c$. We might want to start the analysis using only the explicit analysis. Then, if a variable has a number of different values in the reached set that exceeds a given threshold, we want to disable the (precise) explicit analysis for the variable, and enable the tracking of predicates symbolically representing the values. In other words, we want to decrease the precision of the explicit analysis (by tracking less variables) and increase the precision of the predicate analysis (by tracking more predicates). Such combinations are in the same spirit as attempts to combine static analysis with results obtained from testing [Yorsh et al. 2006; Gulavani et al. 2006]: testing is used to accelerate the construction of the proof.

CPAs and composite program analyses can define program analyses flexibly, but they are restricted to a pre-defined, fixed-precision analyses. To allow the kind of dynamic changes of the precision presented above, we extend the CPA framework. A configurable program analysis with dynamic precision adjustment (CPA+) is a CPA with an additional operator that adjusts the precision of the analysis. The precision adjustment function changes the precision of the analysis dynamically based on the set of reachable states computed so far. Similarly, a composite program analysis with dynamic precision adjustment is a composite program analysis built from CPA+'s with an additional composite precision adjustment function. The composite precision adjustment function can adjust the precision of the component analyses individually. All other operators of a CPA+ are parametric in the precision. The precision adjustment mechanism is not restricted to always increasing the precision (like refinement) or always decreasing the precision (like widening). It supports arbitrary change of precision, and in the case of composite analyses, it can decrease the precision of one analysis while increasing the precision of an other analysis.

We evaluated CPA+ by considering combinations of explicit and symbolic analyses: we considered the combination of an explicit value analysis and a predicate abstraction, and the combination of an explicit heap analysis and a shape analysis. In both cases, we start by using the explicit analysis until a

certain metric on the set of reached explicit elements exceeds a threshold (for an explicit value analysis, the metric is the number of different values; for an explicit heap analysis, the metric is the depth of the explicit heap). When the threshold is hit, the precision of the explicit (value or heap) analysis is decreased, the precision of the symbolic (predicate or shape) analysis is increased, and explicit values computed so far are abstracted to an appropriate symbolic state. Compared to a purely symbolic analysis, we observed a significant speed up in most cases. Compared to a purely explicit analysis, we could prove more programs safe because the symbolic analysis allows the proof to be completed.

1.4 Combination of Analyses for Refinement

In software model checkers counterexample-guided abstraction refinement [Clarke et al. 2003] is used to discover predicates on demand. As a result, the analysis only uses the predicates that are needed for a particular problem instance. Such an approach is required when the analysis would be prohibitively expensive if we were to always use the highest precision. Shape analysis is a analysis that is used to represent the content of the heap, for programs manipulating unbounded, recursive data structures. Because of its high accuracy, shape analysis can become very expensive and this advocates for the use of a refinement-based algorithm. The shape analysis defined by Sagiv et al. [Sagiv et al. 2002] uses three-valued logical structures (also referred to as shape graphs) to represent the content of the heap. The advantage of this approach compared to earlier approaches [Jones and Muchnick 1982; Chase et al. 1990] is that the analysis is parametric: the precision of the analysis is defined by a set of unary and binary predicates over nodes of a shape graph. The analysis relies on two types of predicates: core predicates derive from which pointers and fields the analysis intends to track, and instrumentation predicates are derived facts (e.g. reachability, cyclicity) that are crucial to control the precision of the analysis. Different kinds of data structures require different instrumentation predicates. The literature contains many instrumentation predicates that are known to be useful to prove properties of certain data structures [Sagiv et al. 2002]. As a consequence, it is possible to consider a lazy refinement algorithm where the analysis starts with a trivial precision, and where core and instrumentation predicates are added only when needed. Our previous work showed how core predicates can be discovered from spurious counterexample by pointers that play a role in the spuriousness proof [Beyer et al. 2006]. The counterexample analysis is based on interpolants [Craig 1957; McMillan 2003] of unsatisfiable path for-

mulas similarly to predicate-abstraction based model checkers [Henzinger et al. 2004]. Instrumentation predicates, on the other hand, are more delicate to discover. A few approaches attempt to automatically create good instrumentation predicates but they are limited by the fact that they require decision procedures that support transitive closure [Yorsh et al. 2007]. For instance, Loginov et al. attempted to use inductive learning to discover instrumentation predicates [Loginov et al. 2005].

Considering the successful combination of explicit and symbolic analyses when the precision is adjusted automatically, we show that a similar approach can be used to improve a refinement procedure for a shape analysis. We address the refinement of core predicates in the same as in the lazy shape-analysis algorithm (i.e., based on interpolants of unsatisfiable path formulas) and we contribute a new approach for the refinement of instrumentation predicates. Rather than generating arbitrary instrumentation predicates, we decided to rely on the rich set of well-studied instrumentation predicates presented in the literature: the algorithm chooses good instrumentation predicates from a rich, user-configurable library of instrumentation predicates.

An explicit heap analysis is combined with the shape analysis to guide the refinement. The explicit heaps produced by the explicit heap analysis represent precisely a fragment of the heap. We use explicit heaps to evaluate invariants of data structures in order to consider only relevant instrumentation predicates in future refinement steps. Moreover, we abstract explicit heaps to shape graphs at the end of the explicit analysis in order to speed up the shape analysis. When an infeasible error path that can only be ruled out by using more instrumentation predicates is encountered, we simulate the analysis on the path to identify the smallest set of predicates that is needed to rule out the infeasible error path.

We applied our refinement strategy to functions implementing low-level lists and trees manipulations, using a rich library of shape abstractions supporting lists and trees, based on instrumentation predicates found in the literature. The explicit analysis was able to provide useful hints to our refinement strategy, with a low overhead.

1.5 Structure of the Thesis

Chapter 2 covers notions that are used throughout the thesis: we define the programs that we consider in our exposition, we define precisely the reachability

problem, and we present a particular abstract domain (shape analysis) that represents symbolically the content of the heap.

Chapter 3 introduces the framework of configurable program analysis (CPA) and composite program analysis. We present an application of the framework where we evaluate experimentally various combinations of predicate abstraction with shape or pointer analysis.

Chapter 4 extends the CPA framework to support dynamic precision adjustment, both for single analyses and for composite analyses. We present an application of the framework to the combination of explicit and symbolic analyses.

Chapter 5 discusses a counterexample-based refinement strategy for a shape analysis, where the result of an explicit-heap analysis as well as the analysis of infeasible error paths are used to guide the refinement. To support refinement, we propose a modification of the CPA framework that allows a CPA to interrupt the analysis when a refinement is required.

Chapter 6 concludes the thesis with potential future lines of research.

Chapter 2

Preliminaries

In this chapter, we cover the notions that are used throughout the rest of this thesis. In particular, we define the set of programs we are considering and the verification problem we are solving. In addition, we present an existing abstraction of the content of the heap (shape analysis) that we use throughout the thesis when considering applications of our frameworks.

2.1 Programs

For presentation purposes, we consider flat programs (i.e., programs without function calls) in a simple imperative programming language that supports integer variables and heap-stored data structures. Program statements are expressed using a subset of C. All techniques presented in this thesis can be extended to support conservatively the entire C language, including interprocedural analysis using context-free reachability [Reps et al. 1995; Henzinger et al. 2004; Rinetzky et al. 2004]. Instead of using control-flow graphs to represent programs, we formalize programs using control-flow automata.

2.1.1 Control-Flow Automata and Types

A *control-flow automaton* (CFA) is a directed, labeled graph (L, E), where the set L of nodes represents the control locations of the program (program-counter values), and the set $E \subseteq L \times Ops \times L$ of edges represents the program *transfers*, i.e., an edge $(l, op, l') \in E$ represents the fact that the program can go from location l to location l' by executing operation op. Each edge is labeled with a program operation that can be either an assignment (basic block) or an assume

$$
\begin{array}{lcl}
\mathit{operation} & ::= & \mathit{var}\ \texttt{=}\ \mathit{expression} \\
 & | & \mathit{var}\ \texttt{=}\ \mathit{var} \\
 & | & \mathit{var}\ \texttt{=}\ \mathit{var}\texttt{->}\mathit{field} \\
 & | & \mathit{var}\texttt{->}\mathit{field}\ \texttt{=}\ \mathit{var} \\
 & | & \mathit{var}\ \texttt{=}\ \mathit{malloc}() \\
 & | & \mathit{assume}(\mathit{predicate}) \\
\mathit{expression} & ::= & \text{arithmetic expression over} \\
 & & \mathit{var}\ \text{and integer constants} \\
\mathit{predicate} & ::= & \mathit{var} \sim \mathit{expression} \\
 & | & \mathit{var}\ \texttt{==}\ \mathit{var} \\
 & | & \mathit{var}\ \texttt{!=}\ \mathit{var} \\
 & | & \mathit{predicate}\ \mathit{bop}\ \mathit{predicate} \\
 & | & \texttt{!}\,\mathit{predicate} \\
\sim & ::= & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{!=} \mid \texttt{==} \\
\mathit{bop} & ::= & \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

Figure 2.1: Grammar of program operations.

predicate (condition that must hold for control to proceed across the edge). The language *Ops* of program operations is defined by the grammar in Figure 2.1 (the starting meta-symbol is *operation*). The program operations are based on a set X of identifiers (denoted by *var* in the grammar), to identify program variables, and a set F of identifiers (denoted by *field* in the grammar), to identify fields. Variable identifiers and field identifiers can be either of type *integer* or of type *pointer* to a (possibly recursive) structure. Each *structure* is a set of field identifiers. Without loss of generality, we assume that all structures of a program are pairwise disjoint (i.e., each field identifier occurs in only one structure). The left-hand side of an assignment operation is called *lvalue*. The lvalues occurring in a program operation are restricted to *var* and *var->field*. The latter denotes the field *field* of the structure pointed to by pointer variable *var*. The expressions that can occur in program operations are side-effect free C expressions of arithmetic over variable identifiers and integer constants, without pointer dereferences. The right-hand side of an assignment to a variable can be an expression or a dereferenced pointer; the right-hand side of an assignment to a dereferenced pointer can only be a single variable. Moreover, pointer arithmetic is not allowed. Memory allocation for structures is modeled by the operation *var* = *malloc*(). Unlike the `malloc` function from the standard C library, the size of allocation is not specified (it is assumed to be one data structure element of the type pointed to by the identifier) and the allocation always succeeds.

The *type* of an identifier is given by function $T : (X \cup F) \to 2^F$ which maps an identifier i to its type with the following meaning: if we have $T(i) = \emptyset$

```
 1 typedef struct node {
 2   int data;
 3   struct node *next;
 4 } *List;
 5
 6 void foo() {
 7   List a, t, p;   int x;
 8   List a = (List) malloc(sizeof(struct node));
 9   if (a == NULL) exit(1);
10   p = a;
11   while (*) {
12     p->data = 1;
13     t = (List) malloc(sizeof(struct node));
14     if (t == NULL) exit(1);
15     p->next = t;
16     p = p->next;
17   }
18   p->data = 0;
19
20   p = a;
21   x = p->data;
22   while (x == 1) {
23     p = p->next;
24     x = p->data;
25   }
26   assert(x == 0);
27 }
```

Figure 2.2: Example C Program

then the type of i is integer, otherwise we have $T(i) = Y$ and the type of i is pointer to structure $Y \subseteq F$. An operation $op \in Ops$ is *well-typed* if the following conditions hold:

$op =$	i_1 `=` *expression*	implies	$T(i_1) = \emptyset \ \wedge \ T(i) = \emptyset$ for all identifiers i that occur in *expression*,
$op =$	i_1 `=` i_2	implies	$T(i_1) = T(i_2)$,
$op =$	i_1 `=` i_2->i_3	implies	$T(i_1) = T(i_3) \ \wedge \ i_3 \in T(i_2)$,
$op =$	i_1->i_2 `=` i_3	implies	$T(i_2) = T(i_3) \ \wedge \ i_2 \in T(i_1)$,
$op =$	i_1 `=` *malloc*()	implies	$T(i_1) \neq \emptyset$, and
$op =$	*assume*(*predicate*)	implies	*predicate* is well-typed.

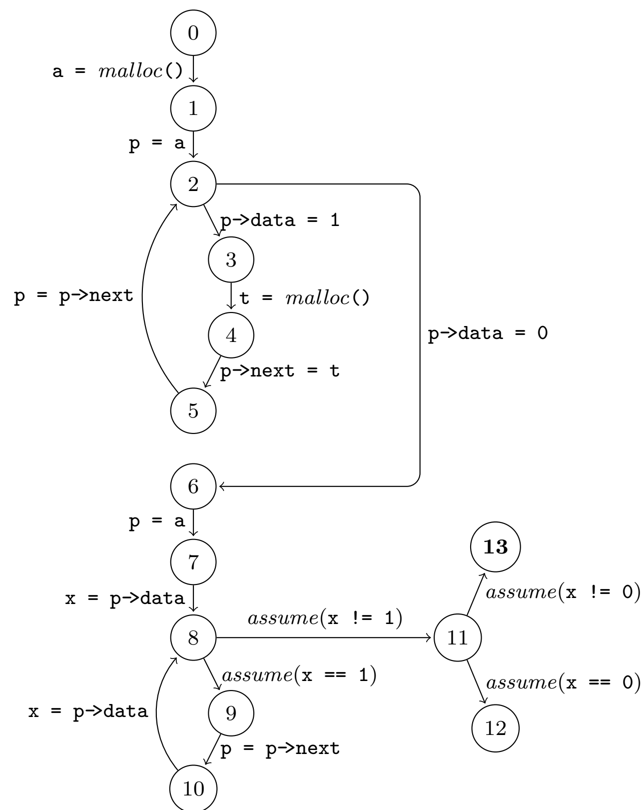


Figure 2.3: Control-flow automaton of the program in Figure 2.2

A predicate p is *well-typed* if the following conditions hold:

$p =$	$i_1 \sim$ *expression*	implies	$T(i_1) = \emptyset \ \wedge\ T(i) = \emptyset$
			for all identifiers i that occur in *expression*,
$p =$	i_1 `==` i_2	implies	$T(i_1) = T(i_2)$,
$p =$	i_1 `!=` i_2	implies	$T(i_1) = T(i_2)$,
$p =$	`!`p_1	implies	p_1 is well-typed, and
$p =$	p_1 *bop* p_2	implies	p_1 and p_2 are well-typed.

A *program* (G, T, l_0) consists of a CFA $G = (L, E)$, a type function T, and an initial control location $l_0 \in L$. A program is *well-typed* if all its operations are type-safe. For a well-typed program we denote the set of integer variables by $X_{int} = \{x \in X \mid T(x) = \emptyset\}$, and the set of pointer variables by $X_{ptr} = \{x \in X \mid T(x) \neq \emptyset\}$. In the reminder of the thesis, when we consider a program, we always assume implicitly that the program is well-typed. A *program path* t of length n is a sequence $(op_1 : l_1); \ldots; (op_n : l_n)$ of operations, such that $(l_{i-1}, op_i, l_i) \in E$ for all $1 \leq i \leq n$.

Example. We represent the example C program in Figure 2.2 by the well-typed program $P = (G, T, l_0)$, where G is the CFA given in Figure 2.3, l_0 is location 0, and the typing function T is defined as follows: $T(\texttt{a}) = T(\texttt{p}) = T(\texttt{t}) = T(\texttt{next}) = \{\texttt{data}, \texttt{next}\}$ and $T(\texttt{x}) = T(\texttt{data}) = \emptyset$. Note that that the CFA does not contain the tests for valid reference after allocation that the C program has because our simple programming language assumes that allocation never fails. The sequence (`a=`*malloc*`()` : 1); (`p=a` : 2) (`p->data=0` : 6); (`p=a` : 7); (`x=p->data` : 8); (*assume*(`x!=1`) : 11); (*assume*(`x!=0`) : 13); is a program path of P. □

2.1.2 Concrete Semantics

A *concrete state* (l, v, h) of a program (G, T, l_0) consists of the following three components. (1) The value $l \in L$ represents the current control location (position of the program counter). (2) The variable assignment $v : X \to \mathbb{Z}$ is a total function that maps every variable identifier to an integer number: integer variables are mapped to integer values, and pointer variables are mapped to structure addresses. (3) The heap assignment $h : \mathbb{Z} \rightharpoonup (F \rightharpoonup \mathbb{Z})$ is a partial function that maps every 'known' structure address to a field assignment, also called *structure cell* (memory content). A field assignment maps each field identifier of the structure to an integer number. The domain of the field assignment

Prog. operation op	Strongest postcondition $\mathsf{SP}(\varphi, op)$
s = e	$\exists \widehat{s} : \varphi_{[s \mapsto \widehat{s}]} \wedge s = e$
s_1 = s_2	$\exists \widehat{s_1} : \varphi_{[s_1 \mapsto \widehat{s_1}]} \wedge s_1 = s_2$
s_1 = s_2->f	$\exists \widehat{s_1} : \varphi_{[s_1 \mapsto \widehat{s_1}]} \wedge s_1 = \mathsf{sel}(h, s_2, f)$
s_1->f = s_2	$\exists \widehat{h} : \varphi_{[h \mapsto \widehat{h}]} \wedge h = \mathsf{upd}(\widehat{h}, s_1, f, s_2)$
s = $malloc()$	$\exists \widehat{h}, \exists \widehat{s} : \begin{pmatrix} \varphi_{[h \mapsto \widehat{h}, s \mapsto \widehat{s}]} \\ \wedge \big(\forall t : \mathsf{alloc}(h, t) \leftrightarrow (\mathsf{alloc}(\widehat{h}, t) \vee t = s)\big) \\ \wedge \big(\forall t : \mathsf{alloc}(\widehat{h}, t) \rightarrow s \neq t\big) \end{pmatrix}$
$assume(p)$	$\varphi \wedge p$

Figure 2.4: Strongest postcondition $\mathsf{SP}(\varphi, op)$ for a formula φ and each kind of program operation op.

is the structure. We denote the set of all concrete states by C. A set $r \subseteq C$ of concrete states is called *region*.

Example. Consider the program path of the previous example. At control location 8, the following concrete state could occur: $(8, v, h)$ with $v = \{a \mapsto 326873, p \mapsto 326873, t \mapsto 326873, x \mapsto 0\}$ and $h = \{326873 \mapsto \{next \mapsto 0, data \mapsto 0\}\}$. $\square$

The *concrete semantics* of a program path is defined in terms of the strongest-postcondition operator SP. For a set of concrete states represented by the formula φ and a program operation op, the formula $\mathsf{SP}(\varphi, op)$ represents the set of concrete successor states. We extend the operator SP from program operations to program paths in the natural way: $\mathsf{SP}(\varphi, t) = \mathsf{SP}(\varphi, op)$ if $t = (op : l)$, and $\mathsf{SP}(\varphi, t) = \mathsf{SP}(\mathsf{SP}(\varphi, op), t')$ if $t = (op : l); t'$. The definition of the strongest-postcondition operator for a formula φ and a program operation op is given in Figure 2.4. For a formula φ and two variable identifiers x and y, the formula $\varphi_{[x \mapsto y]}$ (renaming) is obtained from φ by replacing every free occurrence of x by y. We define the SP of formulas using a modified theory of arrays with equality, in order to represent field access and field updates (in the heap part of the concrete state). For a heap assignment h, a structure address a, a field identifier f, and a value c, the function sel returns the value stored by field identifier f of the structure at address a in h, i.e., $\mathsf{sel}(h, a, f) = h(a)(f)$; the function upd returns the updated heap assignment h' such that h' is the same as h except that the field identifier f of the structure at address a in h'

has value c, i.e., $\mathsf{upd}(h,a,f,c) = \big(h(a) \setminus \{(f,\cdot) \in h(a)\}\big) \ \cup \ \{(f,c)\}$[1]; finally, the predicate alloc holds if there is a mapping for a in h, i.e., $h(a)$ is defined. The theory includes two families of axioms to allow reasoning about our new constructs. The first family of axioms are congruence rules for upd, sel, and alloc. The second family of axioms expresses the relation between array updates (upd) and array reads (sel): if we read from an updated array, we get as a result either the updated value, or the value from the original array, based on whether or not the value at the structure address and field identifier that we are accessing have or have not been modified. Formally, we have the two axiom schemas $\mathsf{sel}(\mathsf{upd}(h,a,f,c),a,f) = c$ and $\mathsf{sel}(\mathsf{upd}(h,a,f,c),b,g) = sel(h,b,g)$ if $a \neq b$ or $f \neq g$. We refer to these axioms of the second family later in the text as read-over-write axioms.

A program path t is called *infeasible* if the formula $\mathsf{SP}(true,t)$ is unsatisfiable, and *feasible* if the formula $\mathsf{SP}(true,t)$ is satisfiable. A state c is *reachable* in P if $c \models \mathsf{SP}(true,t)$ for some program path t, i.e. the state c is a model for the formula $\mathsf{SP}(true,t)$. Similarly, a program location l is *reachable* if there exists some reachable program state whose location is l.

Example. The program path in the previous example is infeasible $\square$

A concrete transition relation is induced by the strongest post conditions. For a program state c, let φ_c be the formula is true only in c. Given a program $P = (G,T,l_0)$, the *concrete transition relation* $\rightarrow \subseteq C \times G \times C$ is defined as follows. For an edge $g = (l,op,l')$ of G, we have $(l,v,h)\xrightarrow{g}(l',v',h')$ if $(l',v',h') \models \mathsf{SP}(\varphi_{(l,v,h)},op)$. Given a set $S \subseteq C$ of concrete states, we denote by $Succ(P,S)$ the set of successors of S: $Succ(P,S) = \{c' \mid \text{there is } c \in S \text{ and an edge } g \text{ of } P \text{ s.t. } c\xrightarrow{g}c'\}$. Reachability can also be formulated in terms of the transition relation. A concrete state c_n is *reachable* from a region r in program P, denoted by $c_n \in Reach(P,r)$, if there exists a sequence of concrete states $\langle c_0,c_1,\ldots,c_n\rangle$ such that $c_0 \in r$ and for all $1 \leq i \leq n$, we have $c_{i-1}\xrightarrow{g}c_i$ for some CFA edge g of program P. Note that a state c is reachable (according to the definition in the previous paragraph) iff $c \in Reach(P,\{(l_0,\cdot,\cdot) \in C\})$.

2.2 Verification Problem

We focus on proving *safety* properties of programs. More specifically we are interested in proving or disproving that a given set of program states —the

[1]We denote by $\{(f,\cdot) \in h(a)\}$ the set $\{(f,x) \mid x \in \mathbb{Z} \text{ and } (f,x) \in h(a)\}$. We use similar notations in the rest of the thesis.

error states— is reachable. Formally, the *reachability problem* is defined as follows.

Problem 2.1. *Given a program P and a computable set B of states of P, is some state in B reachable?*

For the class of programs we consider, the reachability problem is well-known to be undecidable, i.e., there is exists no algorithm that can decide the problem, in a finite amount of time, for any arbitrary program. The undecidability of this problem is a fundamental result in the theory of computations that dates back to the foundational work of Alan Turing in the early 1930s. Despite its undecidability, the problem is recursively enumerable. Therefore, we aim at creating semi-algorithms for the problem that succeeds in answering the reachability problem for a rich class of programs. Nevertheless, there will always exist instances of the problem on which the algorithm does not terminate or stops with no conclusive answer.

In practice, we often consider a particular case of the reachability problem where we are interested in knowing whether a particular program location —the *error location*— is reachable. The problem is formalized as follows.

Problem 2.2. *Given a program P and a location l_{err} of P, is l_{err} reachable?*

The location-reachability problem is important because other safety properties of programs can be reduced to a location-reachability question on a modified program. As an example, consider an extension of our simple programming language that supports assertions, i.e., programs can contain operations of the form $assert(predicate)$. An assertion is *violated* if *predicate* evaluates to *false* for the pre-state of the operation. The assertion-violation-freedom problem can be formulated as follows.

Problem 2.3. *Given a program P with assertions, can any assertion of P be violated?*

Problem 2.3 can be reduced to a location reachability problem on a program without assertions. Given a program with assertions P_{assert}, we construct a program P by modifying P_{assert} as follows. First we insert a new location l_{err} in the CFA. Second, we replace every CFA edge $(l, assert(predicate), l')$ by two CFA edges $(l, assume(predicate), l')$ and $(l, assume(\texttt{!}predicate), l_{err})$. It can easily be seen that an assertion of P_{assert} is violated iff location l_{err} is reachable in P. More complex safety properties such as those expressible as temporal logic formula or automata can also be reduced to reachability by composing the program with a monitor automaton [Henzinger et al. 2002].

2.3 Shape Analysis

In this thesis, we have a particular focus on programs that manipulates unbounded, heap-stored data structures, such as lists or trees. Shape analysis provides a powerful symbolic representation of the heap. In this section, we give a brief introduction to a particular type of shape abstraction that is based on three-valued logical structures [Sagiv et al. 2002]. We reuse the terminology introduced for the lazy shape-analysis algorithm [Beyer et al. 2006] to talk about the shape abstraction (in particular, we reuse the notions of shape type, tracking definition, and shape-class generator)

2.3.1 Shape Classes

Shape abstraction *symbolically* represents instances of data structures by sets of *shape graphs*. We model the memory content by a set V of *heap nodes*. Each heap node represents one or more structure cells. Properties of the heap are encoded by predicates over nodes. The number of nodes that a predicate constrains is called the arity of the predicate, e.g., a predicate over one heap node is called *unary predicate* and a predicate over two heap nodes is called *binary predicate*. Nullary predicates constrain only variable identifiers, not structure cells. The predicates used in an instance of the analysis are structured in a shape class. A *shape class* $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of three sets of predicates over heap nodes:

1. a set P_{core} of core predicates,
2. a set P_{instr} of instrumentation predicates with $P_{core} \cap P_{instr} = \emptyset$, where each instrumentation predicate $p \in P_{instr}$ has an associated *defining formula* φ_p over core predicates, and
3. a set $P_{abs} \subseteq P_{core} \cup P_{instr}$ of abstraction predicates.

We denote the set of shape classes by $\mathcal{S}$. A shape class $\mathbb{S}$ *refines* a shape class $\mathbb{S}'$, written $\mathbb{S} \preccurlyeq \mathbb{S}'$, if (1) $P'_{core} \subseteq P_{core}$, (2) $P'_{instr} \subseteq P_{instr}$, and (3) $P'_{abs} \subseteq P_{abs}$. The partial order $\preccurlyeq$ induces a lattice of shape classes. We require the set P_{core} of core predicates to contain the (special) binary predicate *eq*. The predicate $eq(u, v)$ evaluates to true iff $u = v$.

Core predicates that are used to analyze data-structures include the following families of predicates. A *points-to predicate* $pt_x(v)$ is a unary predicate that has value 1 if pointer variable x points to a structure cell that is represented by v,

and value 0 otherwise. A *field predicate* $fd_\phi(v)$ is a unary predicate that has value 1 if field assertion ϕ holds for all structure cells that are represented by v, and value 0 otherwise. A field assertion is a predicate over the field identifiers of a structure. Therefore, field predicates represent the data content of a structure, rather than the shape of the structure. In addition, a binary predicate $f(u, v)$ is used to track pointers stored in fields: $f(u, v)$ has value 1 if field f of a structure cell represented by u points to a structure cell represented by v, and value 0 otherwise.

Example. Consider that we want to represent singly-linked lists by shape graphs. Each cell of the singly-linked list is composed of two fields: a pointer field *next* pointing to the next list element, and an integer field *data* storing some data associated with the list element. Moreover, we are interested in knowing the sign of the value stored in field *data*. For the analysis of such linked lists we can use a shape class $\mathbb{S}_{sll}$ composed of the following predicates. The set of core predicates of the shape class is composed of the special binary predicate *eq*; a binary predicate *next* such that $next(u, v)$ holds if the field *next* of node u points to node v; unary points-to predicates pt_a and pt_p; and unary field predicates $fd_{data=0}$ and $fd_{data>0}$, where *data* is the field for the data stored in a list element. The set of instrumentation predicates of the shape class is composed of unary reachability predicates $r_{a,next}$ and $r_{p,next}$, where $r_{x,next}(v)$ holds if node v is reachable from the node pointed to by pointer x following zero or more *next*-field. The defining formula of instrumentation predicate $r_{x,next}$ is $r_{x,next}(v) = \exists u : \ pt_x(u) \wedge next^*(u, v)$, where $next^*$ is the reflexive, transitive closure of *next*. $\square$

2.3.2 Two-valued Shape Graphs

Given a shape class $\mathbb{S}$, we define the notion of a two-valued shape graph, i.e., a shape graph whose nodes represent exactly one structure cell and whose predicates have valuations 0 or 1. A *two-valued shape graph*[2] $s^c = (V, val)$ shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of a set V of heap nodes and a valuation *val* in standard, two-valued logic of the predicates of $\mathbb{S}$: for a predicate $p \in P_{core} \cup P_{instr}$ of arity n, $val(p) : V^n \to \{0, 1\}$. The valuation of instrumentation predicates must satisfy the defining formula of the predicate: given an instrumentation predicate p of arity n with defining formula φ_p,

[2]In the terminology of Sagiv et al. two-valued shape graphs are called *concrete shape graphs*. We decided against using 'concrete' because in our setting a two-valued shape graph is an abstraction of sets of concrete states.

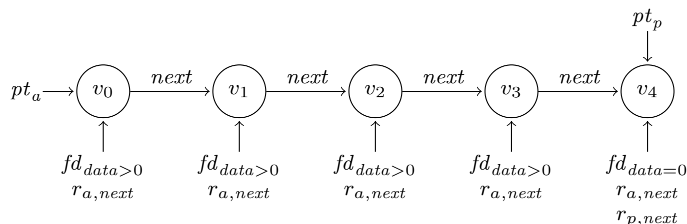


Figure 2.5: Example two-valued shape graph

$val(p)(v_1, \ldots, v_n) = \varphi_p(v_1, \ldots, v_k)$. The valuation of the special, core predicate eq is such that $val(eq)(v, v) = 1$ and $val(eq)(v_1, v_2) = 0$ if $v_1 \neq v_2$.

Example. Figure 2.5 shows the graphical representation of a two-valued shape graph for the shape class $\mathbb{S}_{sll}$. Valuations of binary predicates are represented by edges between nodes; valuations of unary predicates are represented by an edge from the predicate to a node. A solid edge indicates a value of 1, and the absence of edge indicates a value of 0. The shape graph represents lists of length 5 where the data field of all elements are strictly positive except for the last node whose value is null. Pointer a points to the first node, and pointer p points to the last node. □

A two-valued shape graph $g^c = (V, val)$ represents the same set of concrete states as the following state formula φ: for every node v in V, φ contains a variable x_v to represent the node; the (conjunctive) formula φ consists of the following conjuncts; for every node v in V, φ has the conjunct $\mathsf{alloc}(h, x_v)$; for every point-to predicate pt_p and node n in V, if $pt_p(n)$ evaluates to 0 then φ has the conjunct $x_v \neq p$, and if $pt_p(n)$ evaluates to 1 then φ has the conjunct $x_v = p$; for every field predicate fd_ϕ and node v in V, if $fd_\phi(v)$ evaluates to 0 then φ has the conjunct $\neg\phi[f \mapsto \mathsf{sel}(h, x_v, f)]$, and if $fd_\phi(v)$ evaluates to 1 then φ has the conjunct $\phi[f \mapsto \mathsf{sel}(h, x_v, f)]$, where f denotes a field identifier occurring in the field assertion ϕ; for every pair of nodes v and v' in V, if $eq(v, v')$ evaluates to 0 then φ has the conjunct $x_v \neq x_{v'}$, and if $eq(v, v')$ evaluates to 1 then φ has the conjunct $x_v = x_{v'}$; for every binary predicates f and for every pair of nodes v and v' in V, if $f(v, v')$ evaluates to 0 then φ has the conjunct $\mathsf{sel}(h, x_v, f) \neq x_{v'}$, and if $f(v, v')$ evaluates to 1 then φ has the conjunct $\mathsf{sel}(h, x_v, f) = x_{v'}$. We denote by $[\![g^c]\!]^c$ the set of concrete states represented by g^c. Note that the instrumentation predicates do not influence the set of concrete states represented by a two-valued shape graph because the valuation of instrumentation predicates is directly determined by the valuation of core predicates.

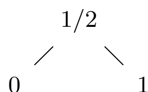


Figure 2.6: Lattice of logical values in three-valued logic with respect to information order

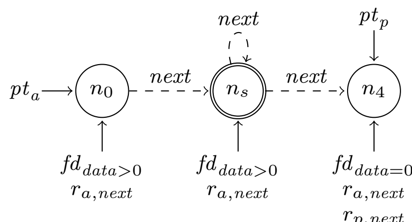


Figure 2.7: Example three-valued shape graph

2.3.3 Three-valued Shape Graphs, Abstraction and Embedding

Two-valued shape graphs can only constrain a fixed, finite number of nodes of the heap because every node represents exactly one structure cell. In this section, we introduce shape graphs that are expressed over Kleene's three-valued logic [Kleene 1987] rather than a two-valued logic, and can have summary nodes (representing an unbounded number of structure elements).

Three-valued logic. In addition to 1 (*true*) and 0 (*false*), *Kleene's three-valued logic* has a third truth value: 1/2. Truth value 1/2 means that the predicate has an 'unknown' value, i.e., the predicate may be either 0 or 1. Values 1 and 1/2 corresponds to *possible truth.* To represent the information content of truth values, we define a partial order $\sqsubseteq$ such that $l_1 \sqsubseteq l_2$ denotes that l_1 has less information than l_2. The information order $\sqsubseteq$ on truth values is defined as follows: for truth values l_1 and l_2, $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = 1/2$. Figure 2.6 depicts the semi-lattice of three-valued logic spanned by the information order.

Three-valued shape graphs. A *three-valued shape graph* $s = (V, val)$ for a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ consists of a set V of heap nodes and a valuation val in three-valued logic of the predicates of $\mathbb{S}$: for a predicate $p \in P_{core} \cup P_{instr}$ of arity n, $val(p) : V^n \to \{0, 1/2, 1\}$. Note that a two-valued shape graph is also a three-valued shape-graph. In the rest of the presentation we shall use the term *shape graph* for *three-valued shape graph.*

Example. Figure 2.7 shows the graphical representation of a three-valued shape graph for the shape class $\mathbb{S}_{sll}$. Summary nodes are indicated by double circles. Valuations of predicates are represented as for two-valued shape graphs, except

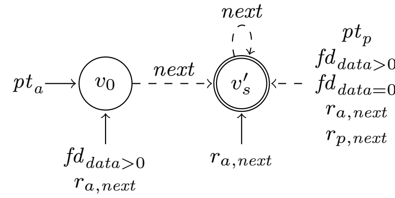


Figure 2.8: Example three-valued shape graph for a shape class with fewer instrumentation predicates

that an unknown value 1/2 is represented by a dotted edge. The shape graph represents lists of length 3 or more, where the data field of all elements is strictly positive except for the last node whose value is null. Pointer a points to the first node, and pointer p points to the last node. □

Abstraction. A two-valued shape graph can be abstracted to a three-valued shape graph with respect to a given shape class as follows [Sagiv et al. 2002]. Given a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ and a two-valued shape graph $g^c = (V^c, val^c)$, the $\mathbb{S}$-abstraction of g^c, denoted $\alpha_{\mathbb{S}}(g^c)$, is a three-valued shape graph $g = (V, val)$ obtained as follows. The set V of nodes is the set of equivalence classes of nodes V^c, when we consider that two nodes are equivalent if they agree on the valuations of all predicates in P_{abs}. For every predicate $p \in P_{core} \cup P_{instr}$, the valuation $val(p)$ is obtained by taking the join, according to the information order, of the valuation $val^c(p)$ for all nodes in the equivalence class. A node n in V that corresponds to a non-singleton equivalence class is called a *summary node*. The predicate eq is used to identify summary nodes: $val(eq)(n, n) = 1/2$ iff v is a summary node. The resulting three-valued shape graph has definite valuations for all predicates in the set P_{abs}. The valuation of non-abstraction predicates for a summary node v_s is 1/2 if the nodes from V^c represented by v_s have different valuations.

Example. The three-valued shape graph in Figure 2.7 is the abstraction of the two-valued shape graph in Figure 2.5, if the abstraction predicates of the shape class are pt_a, pt_p, $r_{a,next}$, and $r_{p,next}$. Summary node v_s represents nodes v_1, v_2 and v_3 of the two-valued shape graph. Figure 2.8 depicts a three-valued shape graph that is the abstraction of the two-valued shape graph in Figure 2.5, for a coarser shape class, where only pt_a and $r_{a,next}$ are abstraction predicates. Summary node v'_s represents nodes v_1, v_2, v_3 and v_4 of the two-valued shape graph. □

Embedding order of three-valued shape graphs. Given two three-valued shape graphs $g = (V, val)$ and $g' = (V', val')$ of a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$, and given a surjective function $f : V \to V'$, we say that f *embeds* S in S', denoted by $S \sqsubseteq^f S'$, if for every predicate p of arity n in $P_{core} \cup P_{instr}$ and every n-tuple of nodes $(v_1, \ldots, v_n)$, we have $val(p)(v_1, \ldots, v_n) \sqsubseteq val'(p)(f(v_1), \ldots, f(v_n))$. A shape graph g *can be embedded* in a shape graph g' if there exists a function f such that $g \sqsubseteq^f g'$. Relation $\sqsubseteq$ is a partial order modulo graph isomorphism. Note that a two-valued shape graph can be embedded in its $\mathbb{S}$-abstraction $\alpha_{\mathbb{S}}(g^c)$.

Example. The two-valued shape graph of Figure 2.7 can be embedded in the three-valued shape graph in Figure 2.7 by the surjective function $f = \{v_0 \mapsto v_0, v_1 \mapsto v_s, v_2 \mapsto v_s, v_3 \mapsto v_s, v_4 \mapsto v_4\}$. Similarly, the three-valued shape graph in Figure 2.7 can be embedded in the three-valued shape graph in Figure 2.8 by surjective function $f = \{v_0 \mapsto v_0, v_s \mapsto v'_s, v_4 \mapsto v'_s\}$ $\square$

Concretization. We use embedding in order to define the set of concrete states that a three-valued shape graph represents. Given a three-valued shape graph g, the set of concrete states represented by g, denoted by $[\![g]\!]$, is defined as the union of all sets of concrete states represented by some two-valued shape graph that can be embedded in g, i.e.,

$$[\![g]\!] = \bigcup \{ [\![g']\!]^c \mid g' \text{ is a two-valued shape graph and } g' \sqsubseteq g \}$$

The Embedding Theorem. The embedding theorem states that if a property is possibly true in a shape graph g, then it is also possibly true in a shape graph g' in which g can be embedded (i.e., $g \sqsubseteq g'$). A proof of the embedding theorem can be found in [Sagiv et al. 2002]. Dually, we have that if a property is definitely true (respectively false) in a shape graph g' then it is also definitely true (respectively false) in a shape graph g that embeds g' (i.e., $g \sqsubseteq g'$). As a consequence, we can soundly use three-valued shape graph as abstractions of two-valued shape graphs.

2.3.4 Abstract Program Semantics

For a given shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$, the $\mathbb{S}$-abstract semantics of a program operation is described as a set of predicate-update formula, one for each predicate in $P_{core} \cup P_{instr}$: for every program operation op and predicate p, the predicate-update formula φ_p^{op} defines the new valuation of the predicate after the execution of operation op_o. The predicate update formula that apply

to two-valued shape graphs can be used as well to update the predicate of three-valued shape graphs, interpreting the formula in the three-valued logic. Note that the valuation of instrumentation predicates might be derived using their defining formula after having applied the update formula, but this is less precise than using more precise update formula for the instrumentation predicates. While in principle its possible to derivate automatically the update formula, it requires the use of sophisticated theorem provers as the defining formula may contain transitive closure; in practice, we assume that update formula for every instrumentation predicate and operation are given. Given a shape graph $g = (V, val)$ and a control-flow edge (l, op, l'), we denote by $\mathsf{post}_{\mathbb{S}'}(g, (l, op, l'))$ the shape graph $g' = (V, val')$ of shape class $\mathbb{S}$, where predicate valuations val' are computed by applying on predicate valuations val the predicate-update formula corresponding to op. The abstract semantics is monotonic with respect to shape class refinement.

Improving the precision. To have more precise shape computations, the analysis needs a mechanism to *materialize* summary nodes, i.e., a mechanism to split a summary node into several nodes. In the framework of Sagiv et al., materialization is supported via two operations that are used to increase the precision of the abstract post computation: *focus* and *coerce*. In the following, we give a brief description of the two operations. Details about the implementation of the two operations can be found in [Sagiv et al. 2002]. Neither operation changes the set of states represented by a shape graph. The focus operation returns a set of shape graphs representing the same states as a shape graph but where a given formula has a definite value in each shape graph of the produced set (similar to a case split). The coerce, conversely, removes inconsistent shape graphs (representing no concrete states) and attempts to replace indefinite values (1/2) by definite values (0 or 1), wherever possible.

Given a shape graph $g = (V, val)$ and a formula $\varphi(n)$, the *focus* of g with respect to $\varphi(v)$ is the smallest set $G' = focus_\varphi(g)$ of shape graphs such that (1) $[\![G']\!] = g$, and (2) for every shape graph $g' = (V', val')$ in G' and for every node v' in V', $\varphi(v')$ evaluates to a definite value (0 or 1). The focus operation is extended to set of shape graphs in the natural way.

Example. Consider the shape class $\mathbb{S}_{sll}$ used in previous example. In order to increase the precision of the abstract post for operation `p = a->next`, we would like to materialize the node pointed to by `a->next`. We can use a focus operation with respect to the formula $\varphi(v) = \exists v'.pt_a(v') \wedge next(v, v')$. Figure 2.9 gives the set of shape graphs that results from focusing the shape graph given

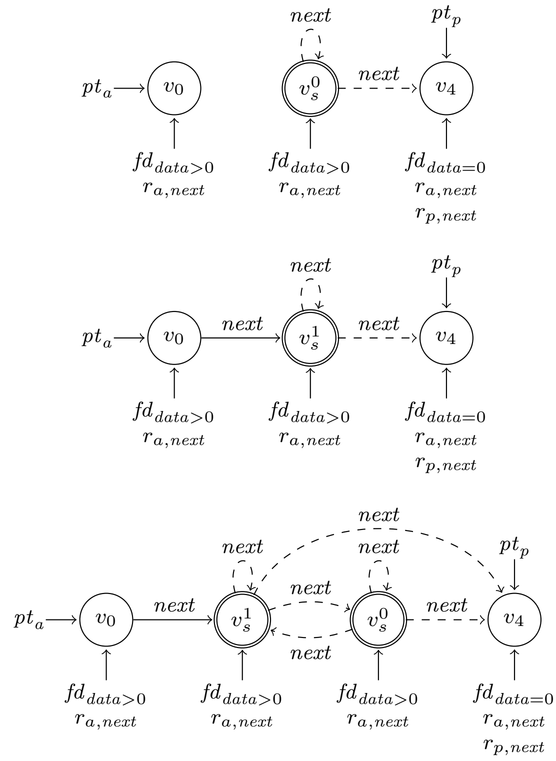


Figure 2.9: Result of applying *focus* for formula $\varphi(v) = \exists v'.pt_a(v') \wedge next(v, v')$ on the shape graph of Figure 2.7. In every shape graph, we have $\varphi(v_s^0) = 0$ and $\varphi(v_s^1) = 1$.

in Figure 2.7. In the shape graph in Figure 2.7, the formula φ evaluates to 1/2 on the summary node v_s. In the result of the focus, three cases are considered: (1) φ evaluates to 0 for all nodes represented by v_s, (2) φ evaluates to 1 for all nodes represented by v_s, and (3) φ evaluates to 1 for some nodes represented by v_s and evalutes to 0 for the rest of the nodes represented by v_s. Each of the three cases corresponds to one of the shape graph in the result of the operation. □

Given a set G of shape graphs, the *coerce* operation attempts to eliminate shape graphs that represents no concrete states and to sharpen predicate valuations without changing the set of concrete states that are represented. Coerce is applied with respect to a set of constraints. Constraints are derived from (1) the semantics of the programming language (e.g. $\forall v, v_1, v_2 : next(v, v_1) \wedge next(v, v_2) \Rightarrow v_1 = v_2$ or $\forall v : fd_{data=0}(v) \Rightarrow \neg fd_{data>0}(v)$), and (2) the definition of instrumentation predicates (e.g. $\forall v_1, v_2 : r_{a,next}(v_1) \wedge next(v_1, v_2) \Rightarrow r_{a,next}(v_2)$). The second category of constraints are automatically derived from the defining formula of instrumentation predicates. If a shape graph violates a constraint it represents no concrete state. If a predicate evaluates to 1/2 on some node and if the shape graph obtained by setting the valuation to 0 (resp. 1) violates a constraint, then the shape graph obtained by setting the valuation to 1 (resp. 0) represents the same set of concrete states. As for the focus operation, we have $[\![G]\!] = [\![coerce(G)]\!]$.

Example. Figure 2.10 represents the result of applying the coerce operation to the set of shape graphs in Figure 2.9. The first shape graphs violates the constraint $\forall v : r_{a,next}(v) \Rightarrow \exists v' : (pt_a(v) \wedge next^*(v', v))$, which is derived from the defining formula of the reachability instrumentation predicates (we have $r_{a,next}(n_s^0) = 1$, but $next^*(n_0, n_s^0) = 0$). In the second shape graph, node n_s^1 becomes a single node because of the constraint $\forall v, v_1, v_2 : next(v, v_1) \wedge next(v, v_2) \Rightarrow v_1 = v_2$ (we have $next(v_0, v_s^1) = 1$; therefore, $v_s^1 = v_s^1$, i.e., $eq(v_s^1, v_s^1) = 1$). □

Using focus and coerce, we can compute a more precise post by first applying focus (with respect to a focus formula that depends on the operation), then apply the predicate-update formulas, and finally, apply coerce. In the following, the abstract semantics $\mathsf{post}_{\mathbb{S}}$ refers to the improved semantics that yields for a shape graph the set of shape graphs resulting from the sequence of operation described above.

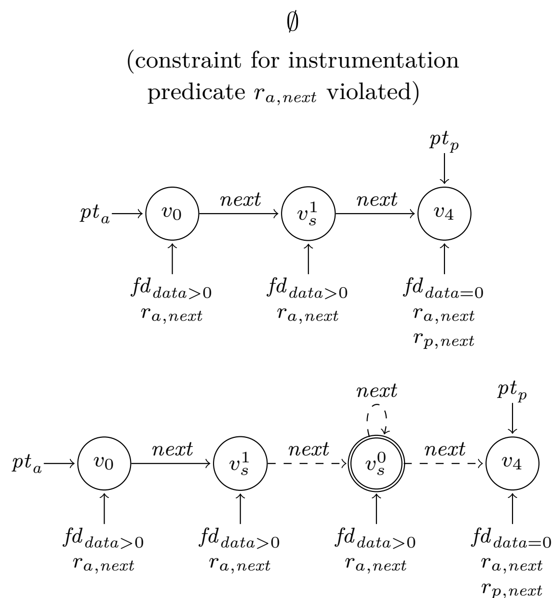


Figure 2.10: Result of applying *coerce* on the shape graphs of Figure 2.9

2.3.5 Shape Abstraction and Shape Regions

We have presented in the previous sections how shape graphs can be used to represent sets of program states. In this section we present the way in which we use shape graphs and classes as abstract states. To allow the simultaneous representation of different parts of the heap using different predicates, we abstract concrete states using sets of shape classes (rather than a single shape class).

The *shape abstraction* $\Psi \subseteq \mathcal{S}$ is a set of shape classes (different shape classes can be used to simultaneously track different data structures). The Ψ-abstraction, i.e., the result of applying a shape abstraction Ψ, is an abstract state, called shape region. A *shape region* $G = \{(\mathbb{S}_1, S_1), \ldots, (\mathbb{S}_n, S_n)\}$ consists of a set of pairs $(\mathbb{S}_i, S_i)$ where $\mathbb{S}_i$ is a shape class and S_i is a set of shape graphs for $\mathbb{S}_i$. A set of shape graphs represents the union of all states represented by some shape graph in the set, and a shape region represents the intersection of the states represented by each of the set of shape graphs it contains, i.e., $[\![\{(\mathbb{S}_1, S_1), \ldots, (\mathbb{S}_n, S_n)\}]\!] = \bigcap_{1 \leq i \leq n} \bigcup_{g \in G_i} [\![g]\!]$. Given a two-valued shape graph g^c and a shape abstraction $\Psi = \{\mathbb{S}_1, \ldots, \mathbb{S}_n\}$, we call the shape region $\{(\mathbb{S}_1, \{\alpha_{\mathbb{S}_1}(g^c)\}), \ldots, (\mathbb{S}_n, \{\alpha_{\mathbb{S}_n}(g^c)\})\}$ the Ψ-abstraction of g^c.

Let $\mathbb{S}$ and $\mathbb{S}'$ be two shape classes such that $\mathbb{S} \preccurlyeq \mathbb{S}'$. A shape graph g' for $\mathbb{S}'$ can be adjusted to a shape graph $g = E_{\mathbb{S}' \triangleright \mathbb{S}}(g')$ for $\mathbb{S}$ by a shape-graph adjustment function E, which leaves the set of nodes unchanged (i.e., $V = V'$),

and adjusts the predicates such that for each predicate p in $(P_{core} \cup P_{instr}) \setminus (P'_{core} \cup P'_{instr})$, the valuation of p is always 1/2, and all other predicates are unchanged. The set of concrete states represented by a shape graph is not changed by a shape-graph adjustment function, i.e., $[\![g]\!] = [\![E_{\mathbb{S} \triangleright \mathbb{S}'}(g)]\!]$. We extend the operator E to sets of shape graphs in the natural way. A shape region $\{(\mathbb{S}_1, S_1), \ldots, (\mathbb{S}_n, S_n)\}$ is *covered* by a shape region $\{(\mathbb{S}'_1, S'_1), \ldots, (\mathbb{S}'_n, S'_n)\}$, denoted by $\{(\mathbb{S}_1, S_1), \ldots, (\mathbb{S}_n, S_n)\} \sqsubseteq \{(\mathbb{S}'_1, S'_1), \ldots, (\mathbb{S}'_n, S'_n)\}$, if for every $1 \leq i \leq n$, (1) $\mathbb{S}_i \preccurlyeq \mathbb{S}'_i$, and (2) for each shape graph g in S_i, there exists a shape graph g' in S'_i such that $g \sqsubseteq E_{\mathbb{S}'_i \triangleright \mathbb{S}_i}(g')$. The relation $\sqsubseteq$ on shape regions is a partial order. We denote by $\mathcal{G}$ the set of all shape regions, and by $\mathcal{G}_\Psi$ the set of all shape regions of the form $\{(\mathbb{S}_1, \cdot), \ldots, (\mathbb{S}_n, \cdot)\}$ where $\Psi = \{\mathbb{S}_1, \ldots, \mathbb{S}_n\}$. For any shape abstraction Ψ, $(\mathcal{G}_\Psi, \sqsubseteq)$ is a lattice. The top element $\top_\Psi$ of the lattice is $\{(\mathbb{S}_1, \{g^\top_{\mathbb{S}_1}\}), \ldots, (\mathbb{S}_n, \{g^\top_{\mathbb{S}_n}\})\}$, where $g^\top_{\mathbb{S}} = (\{v\}, val)$ such that val maps every predicate of the shape class $\mathbb{S}$ to a constant valuation of 1/2. The bottom element $\bot_\Psi$ of the lattice is $\{(\mathbb{S}_1, \emptyset), \ldots, (\mathbb{S}_n, \emptyset))\}$. Abstract post operations are extended from shape graphs to shape regions in the natural way:

$$
\begin{aligned}
&\mathsf{post}_\Psi\left(\{(\mathbb{S}_1, S_1), \ldots, (\mathbb{S}_n, S_n)\}, g\right) \\
&= \left\{\left(\mathbb{S}_1, \bigcup_{s \in S_1} \mathsf{post}_{\mathbb{S}_1}(s, g)\right), \ldots, \left(\mathbb{S}_n, \bigcup_{s \in S_n} \mathsf{post}_{\mathbb{S}_n}(s, g)\right)\right\}
\end{aligned}
$$

where g denotes an edge of the CFA.

2.3.6 Tracking Definitions and Shape-Class Generators

Instead of directly considering shape classes, we separate two aspects of shape classes. First, a tracking definition provides information about which pointers and which field predicates need to be tracked on a syntactic level. Second, given a tracking definition, a shape-class generator determines which predicates are actually added to the shape class.

A *tracking definition* $D = (T, T_s, \Phi)$ consists of (1) a set T of *tracked pointers*, which is the set of variable identifiers that may be pointing to some node in a shape graph; (2) a set $T_s \subseteq T$ of *separating pointers*, which is the set of variable identifiers for which we want the corresponding predicates (e.g., points-to, reachability) to be abstraction predicates (i.e., precisely tracked, no value 1/2 allowed); and (3) a set Φ of field assertions. A tracking definition $D = (T, T_s, \Phi)$ *refines* a tracking definition $D' = (T', T'_s, \Phi')$, if $T' \subseteq T$, $T'_s \subseteq T_s$ and $\Phi' \subseteq \Phi$. We denote the set of all tracking definitions by $\mathcal{D}$. The coarsest tracking definition $(\emptyset, \emptyset, \emptyset)$ is denoted by D_0.

A *shape-class generator* (SCG) is a function $m : \mathcal{D} \rightarrow \mathcal{S}$ that takes as input a tracking definition and returns a shape class, which consists of core predicates, instrumentation predicates, and abstraction predicates. While useful SCGs contain points-to and field predicates for pointers and field assertions from the tracking definition, and the predicate *eq*, other predicates need to be added by appropriate SCGs. An SCG m *refines* an SCG m' (denoted by $m \sqsubseteq m'$) if $m(D) \preccurlyeq m'(D)$ for every tracking definition D. We require that the set of SCGs contains at least the coarsest element m_0, which is a constant function that generates for each tracking definition the shape class $(\emptyset, \emptyset, \emptyset)$. Furthermore, we require each SCG to be monotonic: given an SCG m and two tracking definitions D and D', if $D \preccurlyeq D'$, then $m(D) \preccurlyeq m(D')$.

A *shape type* $\mathbb{T} = (\sigma, m, D)$ consists of a structure type σ, an SCG m, and a tracking definition D. For example, consider the type $\sigma = \{\mathit{data}, \mathit{succ}\}$ (corresponding to the C type `struct node {int data; struct node *succ;}`) and the tracking definition $D = (\{l1, l2\}, \{l1\}, \{\mathit{data} = 0\})$. To form a shape type for a singly-linked list, we can choose an SCG that takes a tracking definition $D = (T, T_s, \Phi)$ and produces a shape class $\mathbb{S} = (P_{core}, P_{instr}, P_{abs})$ with the following components: the set P_{core} of core predicates contains the default binary predicate *eq* (for distinguishing summary nodes), a binary predicate *succ* for representing links between nodes in the list, a unary points-to predicate for each variable identifier in T, and a unary field predicate for each assertion in Φ. The set P_{instr} of instrumentation predicates contains for each variable identifier in T a reachability predicate. The set P_{abs} of abstraction predicates contains all core and instrumentation predicates about separating pointers from T_s. More precise shape types for singly-linked lists can be defined by providing an SCG that adds more instrumentation predicates (e.g., cyclicity).

A *shape-abstraction specification* is a set of shape types. The specification $\widehat{\Psi}$ defines a shape abstraction Ψ in the following way: a shape type $\mathbb{T} \in \widehat{\Psi}$ yields a shape class $\mathbb{S} \in \Psi$ with $\mathbb{S} = \mathbb{T}.m(\mathbb{T}.D)$. (We use the notation $X.y$ to denote the component y of a structure X.) Given a program P, the initial shape-abstraction specification $\widehat{\Psi}_0$ is defined as the set $\{(\sigma, m_0, D_0) \mid \sigma \text{ is a structure type of } P\}$; the initial shape region G_0 corresponds to $\top_\Psi$ where Ψ is the shape abstraction corresponding to $\widehat{\Psi}$. Region G_0 does not constrain the state space; it represents all program states.

Chapter 3

Configurable Program Analysis

3.1 Motivation

Automatic program verification requires a choice between precision and efficiency. The more precise a method, the fewer false alarms it will produce, but also the more expensive it is, and thus applicable to fewer programs. Historically, this trade-off was reflected in two major approaches to static verification: program analysis and model checking. While in principle, each of the two approaches can be (and has been) viewed as a subcase of the other [Schmidt 1998; Steffen 1991; Cousot and Cousot 1995], such theoretical relationships have had little impact on the practice of verification. Program analyzers, by and large, still target the efficient computation of few simple facts about large programs; model checkers, by contrast, focus still on the removal of false alarms through ever more refined analyses of relatively small programs. Emphasizing efficiency, static program analyzers are usually path-insensitive, because the most efficient abstract domains lose precision at the join points of program paths. Emphasizing precision, software model checkers, on the other hand, usually never join abstract domain elements (such as predicates), but explore an abstract reachability tree that keeps different program paths separate.

In order to experiment with the trade-offs, and in order to be able to set the dial between the two extreme points, we have developed and implemented a new framework that permits customized program analyses. Traditionally, customization has meant to choose a particular abstract interpreter (abstract domain and transfer functions, perhaps a widening operator) [Lev-Ami and Sa-

giv 2000; Dwyer and Clarke 1996; Martin 1998; Tjiangan and Hennessy 1992], or a combination of abstract interpreters [Gulwani and Tiwari 2006; Cousot and Cousot 1979; Codish et al. 1993; Lerner et al. 2002]. Here, we go a step further in that we also configure the execution engine of the chosen abstract interpreters. At one extreme (typical for program analyzers), the execution engine propagates abstract domain elements along the edges of the control-flow graph of a program until a fixpoint is reached [Cousot and Cousot 1977]. At the other extreme (typical for model checkers), the execution engine unrolls the control-flow graph into a reachability tree and decorates the tree nodes with abstract domain elements, until each node is 'covered' by some other node that has already been explored [Beyer et al. 2007]. In order to customize the execution of a program analysis, we define and implement a meta engine that needs to be configured by providing, in addition to one or more abstract interpreters, a *merge* operator and a *termination* check.

The merge operator indicates when two nodes of a reachability tree are merged, and when they are explored separately: in classical program analysis, two nodes are merged if they refer to the same control location of the program; in classical model checking, no nodes are merged. The termination check indicates when the exploration of a path in the reachability tree is stopped at a node: in classical program analysis, when the corresponding abstract state does not represent new (unexplored) concrete states (i.e., a fixpoint has been reached); in classical model checking, when the corresponding abstract state represents a subset of the concrete states represented by another node. Our motivation is practical, not purely theoretical: while it is theoretically possible to redefine the abstract interpreter to capture different merge operators and termination checks within a single execution engine, we wish to reuse abstract interpreters as building blocks, while still experimenting with different merge operators and termination checks. This is particularly useful when several abstract interpreters are combined. In this case, our meta engine can be configured by defining a *composite merge operator* from the component merge operators; a *composite termination check* from the component termination checks; but also a *composite transfer function* from the component transfer functions.

Combining the advantages of different execution engines for different abstract interpreters can yield dramatic results, as was shown by *predicated lattices* [Fischer et al. 2005]. That work combined predicate abstraction with a data-flow domain: the data-flow analysis becomes more precise by distinguishing different paths through predicates; at the same time, the efficiency of a lattice-based analysis is preserved for facts that are difficult to track by

predicates. However, the configuration of predicated lattices is just one possibility, combining abstract reachability trees for the predicate domain with a join-based analysis for the data-flow domain. Another example is *lazy shape analysis* [Beyer et al. 2006], where we combined predicate abstraction and shape analysis. Again, we 'hard-wired' one particular such combination: no merging of nodes; termination by checking coverage between individual nodes; Cartesian product of transfer functions. Our new, configurable implementation permits the systematic experimentation with many variations. We show that different configurations can lead to large, example-dependent differences in precision and performance. In particular, it is often useful to use non-Cartesian transfer functions, where information flows between multiple abstract interpreters, e.g., from the predicate state to the shape state (or lattice state), and vice versa. By choosing suitable abstract interpreters and configuring the meta engine, we can also compare the effectiveness and efficiency of symbolic versus explicit representations of values, and the use of different pointer alias analyses in software model checking.

In recent years we have observed a convergence of historically distinct program verification techniques. It is indeed difficult to say whether our configurable verifier is a model checker (as it is based on Blast) or a program analyzer (as it is configured by choosing a set of abstract interpreters and some parameters for executing and combining them). We believe that the distinction is no longer practically meaningful (it has not been theoretically meaningful for some time), and that this signals a new phase in automatic software-verification tools.

3.2 Formalism and Algorithm

3.2.1 Preliminaries

We consider programs, represented as control-flow automata (CFA), as defined in Section 2.1.

A *partial order* $\sqsubseteq \subseteq E \times E$ is a binary relation that is reflexive ($e \sqsubseteq e$ for all $e \in E$), antisymmetric (if $e \sqsubseteq e'$ and $e' \sqsubseteq e$ then $e = e'$), and transitive (if $e \sqsubseteq e'$ and $e' \sqsubseteq e''$ then $e \sqsubseteq e''$). The *least upper bound* for a subset $M \subseteq E$ of elements is the smallest element e such that $e' \sqsubseteq e$ for all $e' \in M$. The partial order $\sqsubseteq$ induces a *complete lattice* if any subset $M \subseteq E$ has a least upper bound $e \in E$ (cf. [Davey and Priestley 1990; Nielson et al. 1999] for more details). We denote a complete lattice that is induced by a set E and a partial oder $\sqsubseteq$ using the

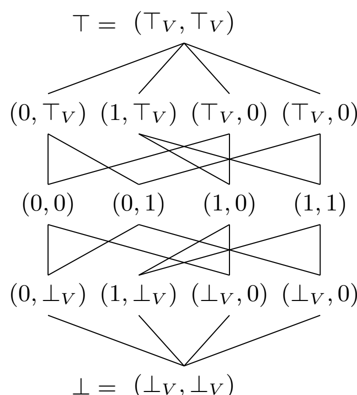


Figure 3.1: Example lattice for constant propagation with two variables

signature $(E, \top, \bot, \sqsubseteq, \sqcup)$, in order to assign symbols to special components: the join operator $\sqcup : E \times E \to E$ yields the least upper bound for two elements (we use the set notation $\sqcup\{e_1, e_2, \ldots\}$ to denote $e_1 \sqcup e_2 \sqcup \ldots$), the top element $\top$ is the least upper bound of the set E ($\top = \sqcup E$), and the bottom element $\bot$ is the greatest lower bound of the set E ($\bot = \sqcup \emptyset$).

Example. Let us consider the lattice $(V, \top, \bot, \sqsubseteq, \sqcup)$ that can be used for a constant-propagation analysis over two Boolean variables. The set V of lattice elements consists of variable assignments: $V = \{v : X \to \{\bot_V, 0, 1, \top_V\}\}$ where, for concreteness, assume that the set $X = \{x_1, x_2\}$ of variables consists of two variables. The partial order $\sqsubseteq$ is defined as $v \sqsubseteq v'$ if for all $x \in X$, $v(x) = v'(x)$ or $v(x) = \bot_V$ or $v'(x) = \top_V$. Figure 3.1 depicts this simple lattice as a graph. The nodes represent lattice elements, where a pair (c_1, c_2) denotes the variable assignment $\{x_1 \mapsto c_1, x_2 \mapsto c_2\}$. The edges represent the partial order (if read in the upwards direction), where reflexive and transitive edges are omitted. The top element $\top$ is the variable assignment with $\top(x) = \top_V$ for all $x \in X$. If used as abstract state in a program analysis, the lattice element $\top$ represents all concrete states. The bottom element $\bot$ is the variable assignment with $\bot(x) = \bot_V$ for all $x \in X$. Note that in a program analysis, not only does the lattice element $\bot$ represent the empty set of concrete states, but every variable assignment (abstract state) that has one variable assigned to $\bot_V$ cannot represent any concrete state. [1] □

[1] This leads to the notion of 'smashed bottom', where all variable assignments with at least one variable assigned to $\bot_V$ are subsumed by $\bot$. We do not stress this notion in our presentation, but our implementation supports it.

3.2.2 Configurable Program Analysis (CPA)

A *configurable program analysis* $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a termination check stop, which are explained in the following. These four components *configure* our algorithm (Algorithm 3.1 discussed in the next subsection) and influence the precision and cost of a program analysis.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, a complete lattice $\mathcal{E}$, and a concretization function $[\![\cdot]\!]$. The complete lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of a (possibly infinite) set E of elements, a top element $\top \in E$, a bottom element $\bot \in E$, a partial order $\sqsubseteq \,\subseteq E \times E$ (which defines the lattice structure), and a total function $\sqcup : E \times E \to E$ (the join operator). The function $\sqcup$ yields the least upper bound for two lattice elements, and the symbols $\top$ and $\bot$ denote the least upper bound of the set E and $\emptyset$, respectively. The lattice elements from E are the abstract states. Each abstract state represents a (possibly infinite) set of concrete states. The concretization function $[\![\cdot]\!] : E \to 2^C$ assigns to each abstract state its meaning, i.e., the set of concrete states that it represents. The concretization function is extended to sets of abstract states in the natural way: for $S \subseteq E$, $[\![S]\!] = \bigcup_{e \in S} [\![e]\!]$. The abstract domain determines the objective of the analysis. For soundness of the program analysis, the abstract domain must fulfill the following requirements:

   (a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$
   The abstract state $\top$ represents all concrete states, and the abstract state $\bot$ represents no concrete state.

   (b) $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$
   If the abstract state e' is greater than (or equal to) abstract state e, then e' must represent a greater (or the same) set of concrete states.

   (c) $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$
   The join operator is precise or over-approximates, i.e., the join of two abstract states e and e' must represent (at least) the concrete states represented by e and the concrete states represented by e'.

   Note that requirements (b) and (c) are equivalent because the join operator $\sqcup$ is defined as the least upper bound.

2. The *transfer relation* $\rightsquigarrow \,\subseteq E \times G \times E$ assigns to each abstract state e possible new abstract states e' that are abstract successors of e, and each

transfer is labeled with a control-flow edge g. We write $e \stackrel{g}{\rightsquigarrow} e'$ if $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ if there exists a g with $e \stackrel{g}{\rightsquigarrow} e'$. The transfer relation must fulfill the following requirement:

(d) $\forall e \in E, g \in G : \bigcup_{e \stackrel{g}{\rightsquigarrow} e'} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' \mid c \stackrel{g}{\rightarrow} c'\}$
The transfer relation over-approximates operations, i.e., every concrete successor c' of some concrete state c represented by e must be represented by some abstract successor e' of abstract state e.

3. The *merge operator* $\mathsf{merge} : E \times E \to E$ combines the information of two abstract states. The operator weakens the second parameter depending on the first parameter (the result of $\mathsf{merge}(e, e')$ can be anything between e' and $\top$). The merge operator must fulfill the following requirement:

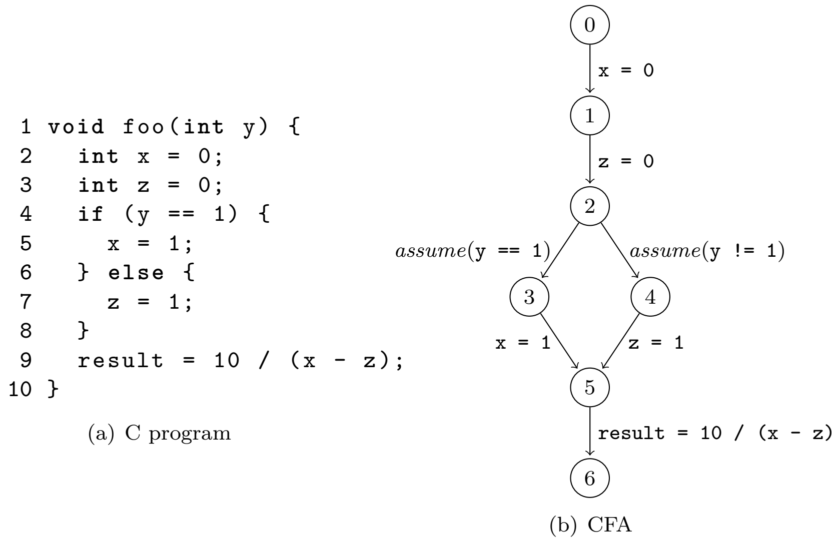
(e) $\forall e, e' \in E : e' \sqsubseteq \mathsf{merge}(e, e')$
The result of merge can only be more abstract than the second parameter. The operator merge is used to (conditionally) combine two abstract states e and e' in such a way that either the new abstract state subsumes both e and e', or subsumes only e' but uses information of e in order to widen e', or is the same as e'. In other words, the merge operator can implement a join operator, or a widening operator, or returns its second operand unchanged.

Note that the operator merge is not commutative, and is not the same as the join operator $\sqcup$ of the lattice, but merge can be based on $\sqcup$. Later we will use the following merge operators: $\mathsf{merge}^{sep}(e, e') = e'$ and $\mathsf{merge}^{join}(e, e') = e \sqcup e'$.

4. The *termination check* $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks if the abstract state that is given as first parameter is covered by the set of abstract states given as second parameter. The termination check can, for example, go through the elements of the set R that is given as second parameter and search for a single element that subsumes ($\sqsubseteq$) the first parameter, or —if D is a powerset domain[2]— can join the elements of R to check if R subsumes the first parameter. The termination check must fulfill the following requirement:

(f) $\forall e \in E, \forall R \subseteq E : \mathsf{stop}(e, R) = \mathit{true}$ implies $[\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$
If an abstract state e is considered as 'covered' by R, then every

[2]A *powerset domain* is an abstract domain such that $[\![e_1 \sqcup e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$.



(a) C program

(b) CFA

Figure 3.2: Example program and corresponding CFA

concrete state represented by e is represented by some abstract state from R.

Note that the termination check stop is not the same as the partial order $\sqsubseteq$ of the lattice, but stop can be based on $\sqsubseteq$. Later we will use the following termination checks (the second requires a powerset domain): $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ and $\mathsf{stop}^{join}(e, R) = (e \sqsubseteq \bigsqcup_{e' \in R} e')$.

The abstract domain on its own does not determine the precision of the analysis; each of the four configurable components (abstract domain, transfer relation, merge operator, and termination check) independently influences both precision and cost.

Requirements (a) to (f) imposed on a CPA are weak compared to other analysis frameworks (including data-flow analysis and abstract interpretation). For instance, we do not require that the transfer relation be the most precise transfer relation, or we do not require that the termination check be the most precise check. The additional flexibility is desired because we want to explore trade-offs between accuracy and computational cost.

Example. Figure 3.2 shows an example program on the left and the corresponding CFA on the right. The CFA has seven program locations ($L = \{0, 1, 2, 3, 4, 5, 6\}, l_0 = 0$) and three integer program variables ($X = X_{int} = \{x, y, z\}$). The initial region r_0 of this program is the set $\{(0, \cdot, \cdot) \in C\}$. The

concrete states at program location 3 that are reachable from the initial region, are the following:

$$\{(3, v, h) \in C \mid v(x) = 0 \wedge v(y) = 1 \wedge v(z) = 0\}$$

The set of concrete states at program location 5 that are reachable from the initial region are the following:

$$\{(5, v, h) \in C \mid (v(y) = 1 \wedge v(x) = 1 \wedge v(z) = 0) \vee (v(y) \neq 1 \wedge v(x) = 0 \wedge v(z) = 1)\}$$

Suppose we want to construct a CPA for constant propagation. The configurable program analysis $\mathbb{CO} = (D_{\mathbb{CO}}, \rightsquigarrow_{\mathbb{CO}}, \mathsf{merge}_{\mathbb{CO}}, \mathsf{stop}_{\mathbb{CO}})$ that tries to determine, for each program location, the value of each variable, consists of the following components (we use the set L of program locations, the set X_{int} of integer program variables, and the set $\mathbb{Z}$ of integer values):

1. The abstract domain $D_{\mathbb{CO}} = (C, \mathcal{E}, [\![\cdot]\!])$ consists of the following three components. The set C is the set of concrete states. The lattice $\mathcal{E}$ represents the abstract states, which store for each program location an abstract integer-variable assignment. Formally, the lattice:

$$\mathcal{E} = ((L \cup \{\top_L, \bot_L\}) \times (X_{int} \rightarrow \mathcal{Z}), (\top_L, v_\top), (\bot_L, v_\bot), \sqsubseteq, \sqcup)$$

with $\mathcal{Z} = \mathbb{Z} \cup \{\top_{\mathcal{Z}}, \bot_{\mathcal{Z}}\}$, is induced by the partial order $\sqsubseteq$ that is defined as $(l, v) \sqsubseteq (l', v')$ if $(l = l'$ or $l = \bot_L$ or $l' = \top_L)$ and for all x in X_{int} : $v(x) = v'(x)$ or $v(x) = \bot_{\mathcal{Z}}$ or $v'(x) = \top_{\mathcal{Z}}$. (The join operator $\sqcup$ yields the least upper bound, $v_\top$ is the abstract variable assignment with $v_\top(x) = \top_{\mathcal{Z}}$ for every $x \in X_{int}$, and $v_\bot$ is the abstract variable assignment with $v_\bot(x) = \bot_{\mathcal{Z}}$ for every $x \in X_{int}$.) A concrete state (l_c, v_c, h_c) *matches* an abstract program location l if $l_c = l \neq \bot_L$, or $l = \top_L$. Similarly, a concrete state (l_c, v_c, h_c) is *compatible* with an abstract variable assignment v if for all $x \in X_{int}$, $v_c(x) = v(x) \neq \bot_{\mathcal{Z}}$, or $v(x) = \top_{\mathcal{Z}}$. The concretization function $[\![\cdot]\!]$ assigns to an abstract state (l, v) all concrete states that match the program location l and are compatible with the abstract variable assignment v.

2. The transfer relation $\rightsquigarrow_{\mathbb{CO}}$ has the transfer $(l, v) \overset{g}{\rightsquigarrow} (l', v')$ if
(1) $g = (l, \mathit{assume}(p), l')$ and for all $x \in X_{int}$:

$$v'(x) = \begin{cases} \bot_{\mathcal{Z}} & \text{if } v(x) = \bot_{\mathcal{Z}} \text{ for some } x \in X_{int} \\ & \text{or the formula } \phi(p,v) \text{ is unsatisfiable} \\ c & \text{if the formula } \phi(p,v) \text{ is satisfiable and} \\ & c \text{ is the only satisfying assignment for variable } x \\ v(x) & \text{otherwise} \end{cases}$$

where, given a predicate p over variables in X, and an abstract variable assignment v,

$$\phi(p,v) := p \ \wedge \bigwedge_{x \in X_{int}, v(x) \neq \top_{\mathcal{Z}}, v(x) \neq \bot_{\mathcal{Z}}} x = v(x)$$

or

(2) $g = (l, \texttt{w} := e, l')$ and for all $x \in X_{int}$:

$$v'(x) = \begin{cases} eval(e,v) & \text{if } x = \texttt{w} \\ v(x) & \text{otherwise} \end{cases}$$

where, given an expression e over variables in X, and an abstract variable assignment v,

$$eval(e,v) = \begin{cases} \bot_{\mathcal{Z}} & \text{if } v(x) = \bot_{\mathcal{Z}} \text{ for some } x \in X_{int} \\ \top_{\mathcal{Z}} & \text{if } v(x) = \top_{\mathcal{Z}} \text{ for some } x \in X_{int} \text{ that occurs in } e \\ \top_{\mathcal{Z}} & \text{if } e \text{ is of the form } p\texttt{->}f \text{ or of the form } malloc() \\ z & \text{otherwise, where expression } e \text{ evaluates to } z \text{ when} \\ & \text{every variable } x \text{ is replaced by } v(x) \text{ in } e \end{cases}$$

or

(3) $l = l' = \top_L$ and $v' = v_\top$.

3. The merge operator is defined by

$$\mathsf{merge}_{\mathbb{CO}}((l,v),(l',v')) = \begin{cases} (l,v) \sqcup (l',v') & \text{if } l = l' \\ (l',v') & \text{otherwise} \end{cases}$$

(combine the two abstract variable assignments where the control flow joins).

4. The termination check is defined by $\mathsf{stop}_{\mathbb{CO}} = \mathsf{stop}^{sep}$.

The following lattice element is an example for an abstract state that is reachable for the program in Figure 3.2: $(2, \{x \mapsto 0, y \mapsto \top_{\mathcal{Z}}, z \mapsto 0\})$. □

3.2.3 Reachability Algorithm for CPA

The reachability algorithm *CPA* (Algorithm 3.1) computes, given a configurable program analysis, a program and an initial abstract state, a set of reachable abstract states, i.e., an overapproximation of the set of reachable concrete states. The configurable program analysis is given by the abstract domain D, the transfer relation $\rightsquigarrow$ for the input program, the merge operator merge, and the termi-

Algorithm 3.1 $CPA(\mathbb{D}, P, e_0)$

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, a program P, and
an initial abstract state $e_0 \in E$,
where E denotes the set of elements of the lattice of D
Output: a set of reachable abstract states
Variables: a set *reached* of elements of E and a set *frontier* of elements of E

```
1: frontier := {e0}
2: reached := {e0}
3: while frontier ≠ ∅ do
4:    choose e from frontier
5:    frontier := frontier \ {e}
6:    for each e' with e ~g~> e' for some CFA edge g of P do
7:       for each e'' ∈ reached do
8:          // combine with existing abstract state
9:          e_new := merge(e', e'')
10:         if e_new ≠ e'' then
11:            frontier := (frontier \ {e''}) ∪ {e_new}
12:            reached := (reached \ {e''}) ∪ {e_new}
13:      if stop(e', reached) = false then
14:         frontier := frontier ∪ {e'}
15:         reached := reached ∪ {e'}
16: return reached
```

nation check stop. The algorithm keeps updating two sets of abstract states, a set *reached* to store all abstract states that are found to be reachable, and a set *frontier* to store all abstract states that are not yet processed (work list). The state exploration starts with the initial abstract state e_0. In each iteration of the while loop, a current abstract state e is chosen in the frontier and removed from that list. The algorithm does not dictate the order in which the abstract states are chosen from the wait list: it is left to the implementation of the choose operation. The algorithm considers each successor e' of the current abstract state e, obtained from the transfer relation. Now, using the operator merge, the abstract successor state is combined with an existing abstract state from *reached*. If the operator merge has added information to the new abstract state e_{new}, such that the old abstract state e'' is strictly subsumed, then the old abstract state is replaced by the new abstract state in the sets *reached* and *frontier*.[3] If the resulting new abstract state e' is not covered by the set *reached* (according to the termination check stop) after the merge step, then it is added to the set *reached* and to the set *frontier*.[4] Note that the algorithm does not necessarily terminate. It is a consequence of the weak requirements imposed on the domain and

[3]Implementation remark: The operator merge can be implemented in a way that it operates directly on the reached set. If the set *reached* is stored in a sorted data structure, there is no need to iterate over the full set of reachable abstract states, but only over the abstract states that need to be combined.

[4]Implementation remark: The termination check can be done additionally before the merge process. This speeds up cases where the termination check is cheaper than the merge operator.

the operators of a CPA. For instance, the definition does not forbid infinite ascending chains in the lattice. While at first sight it might be considered as a weakness, we make use of the additional flexibility to make possible to express in our framework analyses that may not terminate. Such non-terminating analyses can take part in combined analyses such that the result, combined program analysis terminates. Nevertheless, the requirements are strong enough to prove that Algorithm *CPA* produces an overapproximation of the reachable program states when it terminates, as stated by the following theorem.

Theorem 3.1 (Soundness of *CPA*)**.** *Given configurable program analysis* $\mathbb{D}$*, a program* P*, and an initial abstract state* e_0*, Algorithm CPA computes a set of abstract states that over-approximates the set of reachable concrete states if it terminates:* $[\![\mathit{CPA}(\mathbb{D}, P, e_0)]\!] \supseteq \mathit{Reach}(P, [\![e_0]\!])$.

Proof. To prove the theorem, we first prove that the loop of the *CPA* algorithm has the following two loop invariants:

$$\forall c \in [\![e_0]\!] : \; c \in [\![\mathit{reached}]\!] \tag{3.1}$$

$$\forall c \in [\![\mathit{reached} \setminus \mathit{frontier}]\!] : \; \forall c' \text{ s.t. } c \!\to\! c' : c' \in [\![\mathit{reached}]\!] \tag{3.2}$$

Initially, both sets *reached* and *frontier* are the singleton set $\{e_0\}$. Consequently both invariants hold trivially.

We now prove that if the two invariants hold at the beginning of an iteration, they still hold at the end.

We first show that no operation of an iteration removes concrete states from the set $[\![\mathit{reached}]\!]$. Let R and W denote the values of variables *reached* and *frontier*, respectively, at the beginning of the iteration, and R' and W' their value at the end of the iteration. We show that the following proposition holds:

$$[\![R]\!] \subseteq [\![R']\!] \tag{3.3}$$

All operations except one either leave variable *reached* unchanged or add an element to it; we only need to consider the operation $\mathit{reached} := \big(\mathit{reached} \setminus \{e''\}\big) \cup \{e_{new}\}$. Let R^{pre} be the value of the variable *reached* before the operation, and R^{post} its value after the operation. The operation is only executed if $e_{new} \neq e''$; as a consequence, it is sufficient to show that $[\![e'']\!] \subseteq [\![e_{new}]\!]$. By construction of the algorithm, we have $e_{new} = \mathsf{merge}(e', e'')$. By requirement (e) on the merge operator, we have $e'' \sqsubseteq e_{new}$. By requirement (b) on the lattice order, it follows that $[\![e'']\!] \subseteq [\![e_{new}]\!]$.

The preservation of the first invariant immediately follows from (3.3).

To prove the preservation of the second invariant, we need to show that for all $c \in [\![R' \setminus W']\!]$, and all c' such that $c{\rightarrow}c'$, we have $c' \in [\![R']\!]$. Let e_{pop} be the element removed from the set *frontier* in the first operation of the loop. We observe that $R' \setminus W' = (R \setminus W) \cup \{e_{pop}\}$. From (3.3) it suffices to show that:

$$\{c' \mid \exists c \in [\![e_{pop}]\!] : c{\rightarrow}c'\} \subseteq [\![R']\!] \tag{3.4}$$

For all e' such that $e_{pop} \rightsquigarrow e'$, the algorithm adds e' to the set *reached* (but not to the set *frontier*) unless the termination check $\mathsf{stop}(e', \mathit{reached})$ succeeds. If e' is added to the set *reached*, trivially we have that $[\![e']\!] \subseteq [\![R']\!]$. Otherwise, by requirement (f) on the termination check stop, we also have $[\![e']\!] \subseteq [\![R']\!]$. (Note that the innermost for loop does not change the value of $\mathit{reached} \setminus \mathit{frontier}$.) Consequently, $\bigcup_{e_{pop} \rightsquigarrow e'} [\![e']\!] \subseteq [\![R']\!]$ holds. From requirement (d) on the transfer relation, we have $\{c' \mid \exists c \in [\![e_{pop}]\!] : c{\rightarrow}c'\} \subseteq \bigcup_{e_{pop} \rightsquigarrow e'} [\![e']\!]$. Consequently, the second invariant is also preserved.

Given loop invariants 3.1 and 3.2, if the algorithm terminates, we know that the output value $\widehat{R} = \mathit{CPA}(\mathbb{D}, P, e_0)$ satisfies the following two conditions:

$$\forall c \in [\![e_0]\!] : c \in [\![\widehat{R}]\!] \tag{3.5}$$

$$\forall c \in [\![\widehat{R}]\!] : \forall c' \text{ s.t. } c{\rightarrow}c' : c' \in [\![\widehat{R}]\!] \tag{3.6}$$

From (3.5) and (3.6) it follows that $[\![e_0]\!] \subseteq [\![\widehat{R}]\!]$, and that $[\![\widehat{R}]\!]$ is transitively closed under the concrete transition relation $\rightarrow$. Consequently, by the definition of $\mathit{Reach}(P, [\![e_0]\!])$, the theorem holds. $\square$

A consequence of Theorem 3.1 is that we can use Algorithm *CPA* to give definite negative answers to the location reachability problem, i.e., if *CPA* returns no abstract state representing states with the error location, then we can conclude that the error location is unreachable. The following corollary is a direct consequence of Theorem 3.1 and the definition of the reachability of a location.

Corollary 3.2. *Given configurable program analysis $\mathbb{D}$, program P with initial location l_0, and the error location l_{err}, let the error region $r_{err} = \{(l_{err}, \cdot, \cdot)\}$. Given abstract state e_0 such that $[\![e_0]\!]$ contains all concrete states with location l_0, if there are no $e \in \mathit{CPA}(\mathbb{D}, P, e_0)$ such that $[\![e]\!] \cap r_{err} \neq \emptyset$, then location l_{err} is unreachable.*

Note that we cannot conclude that a state is reachable if it is represented by some state returned by Algorithm *CPA*: the analysis might not be precise enough, resulting in a too coarse overapproximation.

Example. In the previous example we defined a CPA for constant propagation. Now we want to use such an analysis to determine that the program in Figure 3.2 can never encounter a 'division-by-zero' failure, i.e., $x \neq z$ at location 5. We can observe that such an error cannot occur in the program because variable x is either strictly greater or strictly smaller than variable z, depending on the value of variable y, but never equal.

We apply Algorithm *CPA* to the configurable program analysis $\mathbb{CO}$ and the program in Figure 3.2. We start the algorithm with the initial abstract state $(0, \{x \mapsto \top, y \mapsto \top, z \mapsto \top\})$. After a few iterations, we have added to the set *reached* the abstract state $(5, \{x \mapsto 1, y \mapsto 1, z \mapsto 0\})$ (when taking the edge from program location 3 to 5). Later, the algorithm examines the edge from program location 4 to 5, and the variable e' in the algorithm is assigned the abstract state $(5, \{x \mapsto 0, y \mapsto \top, z \mapsto 1\})$. One of the states e'' (that is already in the set of reached abstract states) will be the aforementioned abstract state $e'' = (5, \{x \mapsto 1, y \mapsto 1, z \mapsto 0\})$. Since the program locations are identical in both abstract states, the algorithm replaces e'' in the set *reached* with the abstract state $(5, \{x \mapsto \top, y \mapsto \top, z \mapsto \top\})$, since the merge operator joins the abstract states.

Algorithm *CPA* terminates after a number of iterations that is polynomial in the size of the program, and returns the following set of abstract states:

$$\begin{array}{ll} (1, \{x \mapsto \top, y \mapsto \top, z{\mapsto}\top\}), & (2, \{x \mapsto 0, y \mapsto \top, z \mapsto 0\}), \\ (3, \{x \mapsto 0, y \mapsto 1, z \mapsto 0\}), & (4, \{x \mapsto 0, y \mapsto \top, z \mapsto 0\}), \\ (5, \{x \mapsto \top, y \mapsto \top, z \mapsto \top\}), & (6, \{x \mapsto \top, y \mapsto \top, z \mapsto \top\}) \end{array}$$

Therefore, the analysis conclude that the program is potentially unsafe because program variables x and z can have the same value at location 5 and thus, a division-by-zero can occur. However, in the concrete program this is not possible, and the false alarm is due to the over-approximation of the merge operator (which in this case simulates a classical data-flow analysis): when control flow meets at location 5, the two variable assignments are combined, and the information from the two different branches is lost.

If the reachability algorithm had been hardwired to always behave as the analysis described above, we would have had to change the domain to increase the precision of the analysis. In contrast, our configurable program analysis

allows us to parameterize the algorithm, using the same abstract domain but a different merge operator: suppose we replace $\mathsf{merge}_{\mathbb{CO}}$ by merge^{sep}. Now, whenever the locations are equal, the assignments are not combined, instead, the two different abstract states are kept separately. Using this configuration of the program analysis, we obtain a more accurate result: the set *reached* contains both abstract states $(5, \{x \mapsto 1, y \mapsto 1, z \mapsto 0\})$ and $(5, \{x \mapsto 0, y \mapsto \top, z \mapsto 1\})$, from which we can conclude that variables x and z can never have the same value. We get the correct answer because the variable assignments from the different branches are not interchangeable. Therefore, no 'division-by-zero' failure can occur in this program.

While the latter analysis is more precise, it comes with a high cost: if there are n 'if' branches in the control flow, then there are potentially 2^n abstract states, and the algorithm may not terminate in the presence of loops, because the abstract states for the same program location are never combined. Thus, the choice of the merge operator depends on the intended precision and cost of the analysis. $\square$

The reachability algorithm for CPA can be configured to behave similarly to data-flow analysis or model checking.

Data-Flow Analysis. Data-flow analysis is the problem of assigning to each program location a lattice element that over-approximates the set of possible concrete states at that program location. The least solution (smallest over-approximation) can be found by computing the least fixpoint, by iteratively applying the transfer relation to abstract states and joining the resulting abstract state with the abstract state that was assigned to the program location in a previous iteration. The decision whether the fixpoint is reached is usually based on a working list of data-flow facts that were newly added. In our configurable program analysis, the data-flow setting can be realized by choosing the merge operator merge^{join} and the termination check stop^{sep}. Note that a configurable program analysis can model improvements (in precision or efficiency) for an existing data-flow analysis without redesigning the abstract domain of the existing data-flow analysis. For example, a new data-flow analysis that uses the powerset domain 2^D, instead of D itself, can be represented by a configurable program analysis reusing the domain D and its operators, but using an appropriate the merge operator merge^{sep} instead of merge^{join}. Other domains based on a subset of the powerset domain 2^D can also be considered by choosing new merge operators different from merge^{sep} and merge^{join}. A more detailed

comparison of CPA with the traditional frameworks of data-flow analysis and abstract interpretation can be found in Section 3.3.

Model Checking. A typical model-checking algorithm explores the abstract reachable state space of the program by unfolding the CFA, which results in an *abstract reachability tree*. For a given abstract state, the abstract successor state is computed and added as successor node to the tree. Branches in the CFA have their corresponding branches in the abstract reachability tree, but since two paths never meet, a join operator is never applied. This tree data structure supports the required path analysis in counterexample-guided abstraction refinement (CEGAR), as well as reporting a counterexample if a bug is found. The decision whether the fixpoint is reached is usually implemented by a coverage check, i.e., the algorithm checks each time a new node is added to the tree if the abstract state of that node is already subsumed by some other node. BLAST's model-checking algorithm can be instantiated as a configurable program analysis by choosing the merge operator merge^{sep} and the termination check stop^{sep}.

Combinations of Model Checking and Program Analysis. Due to the fact that the model-checking algorithm never uses a join operator, the analysis is automatically path-sensitive. In contrast, path-sensitivity in data-flow analysis requires the use of a more precise data-flow lattice that distinguishes abstract states on different paths. On the other hand, due to the join operations, the data-flow analysis can reach the fixpoint much faster in many cases. In contrast, in our approach the path-sensitivity is determined by the merge operator. Different abstract interpreters exhibit significant differences in precision and cost, depending on the choice for the transfer relation, the merge operator and the termination check. Therefore, we need a mechanism to combine the best choices of operators for different abstract interpreters when composing the resulting program analyses.

In the following three subsections, we present three configurable program analyses that we have used in our experiments.

3.2.4 CPA for Location Analysis

A configurable program analysis $\mathbb{L} = (D_{\mathbb{L}}, \rightsquigarrow_{\mathbb{L}}, \mathsf{merge}_{\mathbb{L}}, \mathsf{stop}_{\mathbb{L}})$ that tracks the reachability of program locations consists of the following components:

1. The domain $D_{\mathbb{L}}$ is based on the flat lattice for the set L of program locations: $D_{\mathbb{L}} = (C, \mathcal{L}, [\![\cdot]\!])$, with $\mathcal{L} = (L \cup \{\top, \bot\}, \top, \bot, \sqsubseteq, \sqcup)$, $\bot \sqsubseteq l \sqsubseteq \top$ and

$l \neq l' \Rightarrow l \not\sqsubseteq l'$ for all elements $l, l' \in L$ (this implies $\bot \sqcup l = l, \top \sqcup l = \top, l \sqcup l' = \top$ for all elements $l, l' \in L, l \neq l'$), and $[\![\top]\!] = C$, $[\![\bot]\!] = \emptyset$, and for all l in L, $[\![l]\!] = \{(l, \cdot, \cdot) \in C\}$.

The element $\top$ represents the fact that the program location is not known, and the element $\bot$ represents non-reachability.

2. The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ if $g = (l, op, l')$, and the transfer $\top \overset{g}{\rightsquigarrow}_{\mathbb{L}} \top$ for all $g \in G$.

   The transfer determines the syntactical successor in the CFA without considering the semantics of the operation *op*.

3. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{L}} = \mathsf{merge}^{sep}$.

4. The termination check considers abstract states individually:
   $\mathsf{stop}_{\mathbb{L}} = \mathsf{stop}^{sep}$.

This (very limited) abstract domain can be used to perform a syntactical reachability analysis, for example to eliminate control-flow that can never be executed. We shall see in Section 3.2.7 how the CPA for location analysis can be used to track the program location when combined with other CPAs, in order to separate the concern of location tracking from the other analyses.

3.2.5 CPA for Predicate Analysis

A program analysis based on Cartesian predicate abstraction was defined by Ball et al. [2001]. We show in this section how to build a CPA that expresses their analysis, given a fixed, finite set P of predicates, with *false* $\in P$, that are tracked by the analysis. The configurable program analysis $\mathbb{P} = (D_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}})$ for predicate analysis, consists of the following components: (for a set of predicates $r \subseteq P$, we write φ_r to denote the conjunction of all predicates in r, in particular $\varphi_\emptyset = true$):

1. The domain $D_{\mathbb{P}} = (C, \mathcal{P}, [\![\cdot]\!])$ is based on predicates that represent regions, with $\mathcal{P} = (2^P, \emptyset, P, \sqsubseteq, \sqcup)$, where the partial order $\sqsubseteq$ is defined as $r \sqsubseteq r'$ if $r \supseteq r'$ (note that if $r \sqsubseteq r'$ then φ_r implies $\varphi_{r'}$). The least upper bound $r \sqcup r'$ is given by $r \cap r'$ (note that $\varphi_{r \sqcup r'}$ is implied by $\varphi_r \vee \varphi_{r'}$). The element $\top = \emptyset$ leaves the abstract state unconstrained (*true*), i.e., every concrete state is represented. The element $\bot = P$ represents the empty set of concrete states (because *false* $\in P$). The concretization function $[\![\cdot]\!]$ is defined by $[\![r]\!] = \{c \in C \mid c \models \varphi_r\}$.

2. The transfer relation $\rightsquigarrow_{\mathbb{P}}$ has the transfer $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$, if $\mathsf{post}(\varphi_r, g)$ is satisfiable and r' is the largest set of predicates such that φ_r implies $\mathsf{pre}(p, g)$ for each $p \in r'$, where $\mathsf{post}(\varphi, g)$ and $\mathsf{pre}(\varphi, g)$ denote the strongest postcondition and the weakest precondition, respectively, for a formula φ and a control-flow edge g. The two operators post and pre are defined such that $[\![\mathsf{post}(\varphi, g)]\!] = \{c' \in C \mid \exists c \in C : c \overset{g}{\rightarrow} c' \wedge c \models \varphi\}$ and $[\![\mathsf{pre}(\varphi, g)]\!] = \{c \in C \mid \exists c' \in C : c \overset{g}{\rightarrow} c' \wedge c' \models \varphi\}$. The Cartesian abstraction of the successor state is obtained by considering every predicate in P separately, which is usually implemented by $|P|$ calls to a theorem prover.

3. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{P}} = \mathsf{merge}^{sep}$.

4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{P}} = \mathsf{stop}^{sep}$.

3.2.6 CPA for Shape Analysis

In Section 2.3 we presented shape analysis, a static analysis that uses finite structures (shape graphs) to represent instances of heap-stored data structures. In this section, we show how we can build a configurable program analysis $\mathbb{S}$ for shape analysis that uses shape regions of a fixed shape abstraction Ψ as abstract states. The configurable program analysis $\mathbb{S} = (D_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, \mathsf{merge}_{\mathbb{S}}, \mathsf{stop}_{\mathbb{S}})$ for shape analysis, consists of the following components:

1. The domain $D_{\mathbb{S}} = (C, \mathcal{D}, [\![\cdot]\!])$ is based on shape regions, where the abstract domain is the lattice $\mathcal{D}$ of shape regions for the shape abstraction Ψ using the pre-order on shape regions defined in Section 2: $\mathcal{D} = (\mathcal{G}_{\Psi}, \bot_{\Psi}, \top_{\Psi}, \sqsubseteq, \sqcup)$. The concretization function is defined in Section 2.

2. The transfer relation $\rightsquigarrow_{\mathbb{S}}$ has the transfer $G \overset{g}{\rightsquigarrow}_{\mathbb{S}} G'$ if $G' = \mathsf{post}_{\Psi}(G, g)$ (abstract post successor as described in Section 2.3.4 and).

3. The merge operator combines elements when control flow meets: $\mathsf{merge}_{\mathbb{S}} = \mathsf{merge}^{join}$.

4. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{S}} = \mathsf{stop}^{sep}$.

3.2.7 Composite Program Analysis

A configurable program analysis can be composed of several configurable program analyses. A *composite program analysis* $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$[5] consists of two configurable program analyses $\mathbb{D}_1$ and $\mathbb{D}_2$ sharing the same set of concrete states and E_1 and E_2 being their respective sets of abstract states, a composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2)$, a composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \to (E_1 \times E_2)$, and a composite termination check $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \to \mathbb{B}$. The three composites $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ are expressions over the components of $\mathbb{D}_1$ and $\mathbb{D}_2$ ($\rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i$), as well as the operators $\downarrow$ and $\preceq$ (defined below). The composite operators can manipulate lattice elements only through those components, never directly (e.g., if $\mathbb{D}_1$ is already a result of a composition, then we cannot access the tuple elements of abstract states from E_1, nor redefine merge_1). The only way of using additional information is through the operators $\downarrow$ and $\preceq$. The *strengthening* operator $\downarrow : E_1 \times E_2 \to E_1$ computes a stronger element from the lattice set E_1 by using the information of a lattice element from E_2; it must fulfill the requirement $\downarrow(e, e') \sqsubseteq e$. The strengthening operator can be used to define a composite transfer relation $\rightsquigarrow_\times$ that is stronger than a pure product relation. For example, if we combine predicate abstraction and shape analysis, the strengthening operator $\downarrow_{\mathbb{S},\mathbb{P}}$ can 'sharpen' the field predicates of the shape graphs by considering the predicate region. Furthermore, we allow the definitions of composite operators to use the *compare* relation $\preceq \subseteq E_1 \times E_2$, to compare elements of different lattices. For example, the compare relation can be used to accelerate the convergence of the algorithm, for example in the transfer relation: suppose we have just computed an element (e_1, e_2) and we have $e_1 \preceq e_2$, then we can replace (e_1, e_2) by $(e_1, \top_2)$ without changing the set of concrete states that are represented.

To guarantee that a configurable program analysis can be built from the composite program analysis, we impose the following requirements on the three composite operators:

(d) $\forall e_1 \in E_1, e_2 \in E_2, g \in G$:
$\bigcup_{(e_1,e_2) \overset{g}{\rightsquigarrow}_\times (e_1',e_2')} ([\![e_1']\!] \cap [\![e_2']\!]) \supseteq \bigcup_{c \in [\![e_1]\!] \cap [\![e_2]\!]} \{c' \mid c \overset{g}{\to} c'\}$
(the composite transfer relation over-approximates operations)

(e) $\forall e_1, e_1' \in E_1, e_2, e_2' \in E_2$:
$(\widehat{e_1}, \widehat{e_2}) = \mathsf{merge}_\times((e_1, e_2), (e_1', e_2'))$ implies $e_1' \sqsubseteq_1 \widehat{e_1} \;\wedge\; e_2' \sqsubseteq_2 \widehat{e_2}$

[5]We extend this notation to any finite number of $\mathbb{D}_i$.

(each component of the result of the composite merge operator is more abstract than the respective component of the second parameter)

(f) $\forall e_1 \in E_1, e_2 \in E_2, \forall R \subseteq E_1 \times E_2$:
$\mathsf{stop}_\times((e_1, e_2), R) = \mathit{true}$ implies $[\![e_1]\!] \cap [\![e_2]\!] \subseteq \bigcup_{(e'_1, e'_2) \in R} ([\![e'_1]\!] \cap [\![e'_2]\!])$
(if a pair of abstract states is considered as 'covered' by R, then the concrete states represented by the pair are also represented by some pair of abstract states from R)

For a given composite program analysis $\mathcal{C} = (\mathbb{D}_1, \mathbb{D}_2, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, we can construct the configurable program analysis $\mathbb{D}_\times = (D_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where the product domain $D_\times$ is defined as the direct product of D_1 and D_2: $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$. The product lattice is $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$ with $(e_1, e_2) \sqsubseteq_\times (e'_1, e'_2)$ if $e_1 \sqsubseteq_1 e'_1$ and $e_2 \sqsubseteq_2 e'_2$ (and for the join operation the following holds $(e_1, e_2) \sqcup_\times (e'_1, e'_2) = (e_1 \sqcup_1 e'_1, e_2 \sqcup_2 e'_2)$). The product concretization function $[\![\cdot]\!]_\times$ is such that $[\![(d_1, d_2)]\!]_\times = [\![d_1]\!]_1 \cap [\![d_2]\!]_2$.

Theorem 3.3 (Composition of CPAs)**.** *Given a composite program analysis* $\mathcal{C}$*, the result of the composition* $\mathbb{D}_\times$ *is a configurable program analysis.*

Proof. The requirements (a), (b), and (c) for abstract domains of CPAs follow from the construction of $D_\times$ as product domain (abstract domains are closed under product). The operators of the composition $D_\times$ fulfill the requirements (d), (e), and (f) of CPAs, by applying the definitions of product lattice and product concretization function. □

Note that the product lattice is a direct product of the component lattices. The literature agrees that this direct product itself is often not sharp enough [Cousot and Cousot 1979; Codish et al. 1993]. Even improvements over the direct product (e.g., the reduced product [Cousot and Cousot 1979] or the logical product [Gulwani and Tiwari 2006]) do not solve the problem completely. However, in a configurable program analysis, we can specify the desired degree of 'sharpness' in the composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$. For a given product domain, the definitions of the three composite operators determine the precision of the resulting configurable program analysis. In particular analyses equivalent to more precise domain products can be defined by choosing appropriate operators. In previous approaches, a redefinition of basic operations was necessary, but using configurable program analysis, we can reuse the existing abstract interpreters.

Example. We now revisit the problem of constant propagation that we used in previous examples (cf. Section 3.2.2, 3.2.3), and define a configurable program analysis for this problem in a more convenient and flexible way: as a composite program analysis. The composite program analysis $\mathcal{C}$ consists of the configurable program analysis $\mathbb{L}$ for locations and a new configurable program analysis $\mathbb{CO}'$ that is similar to $\mathbb{CO}$, except that locations are omitted from abstract elements because they can be handled by the location CPA.
The CPA $\mathbb{CO}' = (D_{\mathbb{CO}'}, \rightsquigarrow_{\mathbb{CO}'}, \mathsf{merge}_{\mathbb{CO}'}, \mathsf{stop}_{\mathbb{CO}'})$ consists of the following components:

1. The abstract domain $D_{\mathbb{CO}'} = (C, \mathcal{E}, [\![\cdot]\!])$ consists of the following three components. The set C is the set of concrete states. The complete lattice $\mathcal{E} = (X_{int} \to \mathcal{Z}, v_\top, v_\bot, \sqsubseteq, \sqcup)$ represents the abstract states, which are abstract integer variable assignments, with $\mathcal{Z} = \mathbb{Z} \cup \{\top, \bot\}$; the partial order $\sqsubseteq$ is defined as $v \sqsubseteq v'$ if $\forall x \in X_{int} : (v(x) = v'(x)$ or $v(x) = \bot$ or $v'(x) = \top)$. The concretization function $[\![\cdot]\!]$ assigns to an abstract state v all concrete states that are compatible with the abstract variable assignment v.

2. The transfer relation $\rightsquigarrow_{\mathbb{CO}'}$ has the transfer $v \overset{g}{\rightsquigarrow} v'$ if
(1) $g = (l, \mathtt{assume}(p), l')$ and for all $x \in X_{int}$:
$$v'(x) = \begin{cases} \bot & \text{if } v(x) = \bot_{\mathcal{Z}} \text{ for some } x \in X_{int} \\ & \text{or the formula } \phi(p, v) \text{ is unsatisfiable} \\ c & \text{if the formula } \phi(p, v) \text{ is satisfiable and} \\ & c \text{ is the only satisfying assignment for variable } x \\ v(x) & \text{otherwise} \end{cases}$$
or
(2) $g = (l, \mathtt{w} := e, l')$ and for all $x \in X_{int}$:
$$v'(x) = \begin{cases} eval(e, v) & \text{if } x = \mathtt{w} \\ v(x) & \text{otherwise} \end{cases}$$
where functions ϕ and *eval* are defined as in the example of Section 3.2.2.

3. The merge operator is defined by $\mathsf{merge}_{\mathbb{CO}'} = \mathsf{merge}^{join}$.

4. The termination check is defined by $\mathsf{stop}_{\mathbb{CO}'} = \mathsf{stop}^{sep}$.

We can now build a composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{CO}', \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ with the following composite operators:

1. The composite transfer relation $\rightsquigarrow_\times$ has the transfer $(e_1, e_2) \overset{g}{\rightsquigarrow}_\times (e_1', e_2')$ if $e_1 \overset{g}{\rightsquigarrow}_{\mathbb{L}} e_1'$ and $e_2 \overset{g}{\rightsquigarrow}_{\mathbb{CO}'} e_2'$.

2. The composite merge operator $\mathsf{merge}_\times$ is defined by:

$$\mathsf{merge}_\times((e_1, e_2), (e_1', e_2')) = \begin{cases} (e_1, \mathsf{merge}_{\mathbb{CO}'}(e_2, e_2')) & \text{if } e_1 = e_1' \\ (e_1', e_2') & \text{otherwise} \end{cases}$$

3. The composite termination check is defined by $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

Note that the CPA resulting from this composite program analysis is equivalent to the CPA $\mathbb{CO}$, which we had previously defined. □

In the following subsections, we present composite program analyses that have the same behavior as analyses found in the literature. We focus on the Blast tool [Beyer et al. 2007] as an example of a predicate-abstraction-based software model checker, and its extensions to support additional abstractions [Fischer et al. 2005; Beyer et al. 2006]. We use the same composition of analyses in our experiments.

Blast's Domain

The program analysis that is implemented in the tool Blast [Beyer et al. 2007] can be expressed as the composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{P}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where the components are the configurable program analysis $\mathbb{L}$ for program locations and the configurable program analysis $\mathbb{P}$ for predicate analysis. We construct the composite transfer relation $\rightsquigarrow_\times$ such that we have the transfer $(l, r) \overset{g}{\rightsquigarrow}_\times (l', r')$ if $l \overset{g}{\rightsquigarrow}_\mathbb{L} l'$ and $r \overset{g}{\rightsquigarrow}_\mathbb{P} r'$. We choose the composite merge operator $\mathsf{merge}_\times = \mathsf{merge}^{sep}$ and the composite termination check $\mathsf{stop}_\times = \mathsf{stop}^{sep}$.

Blast's Domain with Shape Analysis

The combination of predicate abstraction and shape analysis used in the lazy shape-analysis algorithm [Beyer et al. 2006] can now be expressed as the composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{P}, \mathbb{S}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ with the three components location analysis $\mathbb{L}$, predicate abstraction $\mathbb{P}$, and shape analysis $\mathbb{S}$. In our previous work [Beyer et al. 2006] we used a configuration that corresponds to the composite transfer relation $\rightsquigarrow_\times$ that has the transfer $(l, r, s) \overset{g}{\rightsquigarrow}_\times (l', r', s')$ if $l \overset{g}{\rightsquigarrow}_\mathbb{L} l'$ and $r \overset{g}{\rightsquigarrow}_\mathbb{P} r'$ and $s \overset{g}{\rightsquigarrow}_\mathbb{S} s'$, the composite merge operator $\mathsf{merge}_\times = \mathsf{merge}^{sep}$, and the composite termination check $\mathsf{stop}_\times = \mathsf{stop}^{sep}$. Our new tool allows us to define the three composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ in many different ways, and we report the results of our experiments in Section 3.4.

Blast's Domain with Pointer Analysis

Fischer et al. presented a particular combination (called *predicated lattices*) of predicate abstraction and a data-flow analysis for pointers [Fischer et al. 2005]. Their analysis can be expressed as the composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{P}, \mathbb{A}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$, where $\mathbb{A}$ is a configurable program analysis for pointers, which tracks pointer aliases, memory allocations, and value information. The transfer relation $\rightsquigarrow_\times$ has the transfer $(l, r, d) \overset{g}{\rightsquigarrow}_\times (l', r', d')$ if $l \overset{g}{\rightsquigarrow}_\mathbb{L} l'$ and $r \overset{g}{\rightsquigarrow}_\mathbb{P} r'$ and $d \overset{g}{\rightsquigarrow}_\mathbb{A} d'$. We can mimic the algorithm of Fischer et al. by choosing the composite termination check $\mathsf{stop}_\times = \mathsf{stop}^{sep}$ and the composite merge operator that joins the third elements if the first two agree:

$$\mathsf{merge}_\times((l, r, d), (l', r', d')) = \begin{cases} (l', r', \mathsf{merge}_\mathbb{A}(d, d')) & \text{if } l = l' \text{ and } r = r' \\ (l', r', d') & \text{otherwise} \end{cases}$$

with $\mathsf{merge}_\mathbb{A}(d, d') = d \sqcup_\mathbb{A} d'$.

Note that we can more easily compare related approaches once they have been formalized in our general framework. In this particular instance, we observe that the fundamental difference in the reachability algorithm used for predicated lattices and for the approach used in lazy shape-analysis is how states are merged: the two composite analyses use different composite merge operators.

Remark: Location Domain. Traditional data-flow analyses do not consider the location domain as a separate abstract domain; they assume that the program locations are always explicitly analyzed. In contrast, we leave this completely up to the analysis designer. We find it interesting to consider the program counter as just another program variable, and define a location domain that makes the program counter explicit when composed with other domains. As a result, the other abstract domains are released from defining the program location handling, and only the parameters for the composite program analysis need to be set. This keeps different concerns separate. Usually, only the program counter is modeled explicitly, and all other variables are represented symbolically (e.g., by predicates or shapes). We have the freedom to treat *any* program variable explicitly (c.f. Section 4.4), not only the program counter; this may be useful for loop indices. Conversely, we can treat the program counter symbolically, and let other variables 'span' the abstract reachability tree.

3.3 Comparison with Data-flow Analysis and Abstract Interpretation

In practice, both data-flow analysis and abstract interpretation are based on similar fixpoint algorithms (the possible improvements that exist in abstract interpreters such as widening operators are discussed later), and we refer in this section to both approaches as *traditional program analysis*. Data-flow analysis and abstract interpretation are often expressed as the solution of a set of equations. We discuss in this section the classical monotone framework for forward, join data-flow analysis. Iterative algorithms are used to compute the least fixpoint of those equations (know as Maximum Fixed Point solution (MFP) in the literature). For a program $P = (L, G, l_0, T)$ and a lattice $\mathcal{E} = (E, \sqsubseteq)$ of an abstract domain D, the data-flow equations, which constraint a function $\mathit{Analysis} : L \to E$ representing abstract states associated with program locations, are defined as follows:

$$
\begin{array}{rcl}
\mathit{Analysis}(l_0) & \sqsupseteq & d_0 \\
\forall l \in L: \ \mathit{Analysis}(l) & = & \bigsqcup\{\mathit{post}(\mathit{Analysis}(l'), g) \mid g = (l', \mathit{op}, l) \in G\}
\end{array}
$$

where *post* denotes the (monotonic) transfer function such that $\mathit{post}(d, g)$ represents the successors of state d when edge g is taken. The transfer function fulfills the following requirement: $[\![\mathit{post}(d, g)]\!] \supseteq \bigcup_{c \in [\![d]\!]}\{c' \mid c \xrightarrow{g} c'\}$. A solution A of the data-flow equations can be interpreted as the set of program states $\bigcup_{l \in L}\{c \in C \mid c = (l, \cdot, \cdot) \text{ and } c \in [\![A(l)]\!]\}$.

Traditional program analyses bear many similarities with our new CPA framework. For instance, they both use abstract states to represent sets of concrete states, and both perform a fix-point computation to build an overapproximation of the reachable states. The major difference lies in the 'shape' of the overapproximation that the analysis computes, i.e., what are the possible set of abstract states that the analysis produces. In the case of CPA, the framework does not impose any a-priori restriction on the set of abstract states. Only the values of the operators (in particular the merge operator) influence the shape of the result. For instance, when the merge operator is merge^{join}, then the result of the analysis will be a unique abstract state. In contrast, traditional program analyses attempt to annotate every program location with exactly one abstract state. In principle, the limitation imposed by traditional program analyses can be circumvented by considering more precise abstract do-

mains, in particular power set domains. In this section, we explore in details the link between CPA and traditional program analysis.

3.3.1 Encoding a Traditional Program Analysis as a CPA

We build a CPA that behaves as a traditional program analysis for a given abstract domain D, i.e., the CPA computes the same set of abstract reachable states as the traditional program analysis. We need to capture with the concepts provided by the CPA framework the fact that the analysis maintains one state per program location. A prerequisite is that abstract states of the CPA need to contain both a location and an element from the abstract domain. It translates most naturally as a composite program analysis $\mathcal{C} = (\mathbb{L}, \mathbb{D}, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ that combines the location CPA $\mathbb{L}$ presented in Section 3.2.4 with a CPA $\mathbb{D} = (D, \rightsquigarrow_\mathbb{D}, \mathsf{merge}_\mathbb{D}, \mathsf{stop}_\mathbb{D})$ based on the abstract domain D of the traditional program analysis. We denote the CPA corresponding to the composite program analysis $\mathcal{C}$ by $\mathbb{D}_\times$. To ensure that at most one state exists for a given program location, we need to select appropriate operators for CPA $\mathbb{D}$ and appropriate composite operators. The composite transfer relation is Cartesian:

$$(l, d) \overset{g}{\rightsquigarrow} (l', d') \text{ if } l \overset{g}{\rightsquigarrow}_\mathbb{L} l' \text{ and } d \overset{g}{\rightsquigarrow}_\mathbb{D} d'$$

The transfer relation of CPA $\mathbb{D}$ is based on transfer functions of the analysis:

$$d \overset{g}{\rightsquigarrow}_\mathbb{D} d' \text{ if } d' = post(d, g)$$

Note that $\rightsquigarrow_\mathbb{D}$ fulfills requirement (d) by definition of the transfer function $post$. The composite merge operator joins states when they agree on their location:

$$\mathsf{merge}_\times((l, d), (l', d')) = \begin{cases} (l', \mathsf{merge}_\mathbb{D}(d, d')) & \text{if } l = l' \\ (l', d') & \text{otherwise} \end{cases}$$

The merge operator of CPA $\mathbb{D}$ merges states using the join operator of the lattice:

$$\mathsf{merge}_\mathbb{D}(d, d') = d \sqcup_\mathbb{D} d' \;\; \big(= \mathsf{merge}^{join}(d, d')\big)$$

The termination checks compare (using the pre-order) the state with the state in the reached set for the same location:

$$\begin{aligned} \mathsf{stop}_\times((l, d), R) &= \mathsf{stop}_\mathbb{D}(d, \{d' \mid (l, d') \in R\}) \\ \mathsf{stop}_\mathbb{D}(d, R) &= \; \exists d' \in R : d \sqsubseteq_\mathbb{D} d' \; = \mathsf{stop}^{sep}(d, R) \end{aligned}$$

This particular configuration of merge operators and termination checks ensures that there is at most one abstract state for a given location. Moreover we can build a solution of the data-flow equations from the set of reachable states computed by the analysis.

Theorem 3.4. *Given a program* $P = (G, E, l_0, T)$ *and an initial abstract state* d_0 *of* $\mathbb{D}$*, let* $R = CPA(\mathbb{D}_\times, P, (l_0, d_0))$*. The set of reached states* R *satisfies the following:*

(1) $\forall l \in L : |\{(l, \cdot) \in R\}| \leq 1$

(2) $R[l_0] \sqsupseteq d_0$

(3) $\forall l \in L : R[l] = \bigsqcup\{post(R[l'], g) \mid g = (l', op, l) \in G\}$

where $R[l] = d$ *if* $(l, d) \in R$ *and otherwise* $R[l] = \bot$

Proof. We prove (1) by showing that the main loop of Algorithm *CPA* has the following invariant:

$$|\{(l, \cdot) \in reached\}| \leq 1$$

Initially the invariant trivially holds because $reached = \{(l_0, d_0)\}$. Assume the invariant holds at the beginning of an iteration. Let (l, d) be the element that was removed from *frontier*. For every (l', d') such that $(l, d) \rightsquigarrow_\times (l', d')$, we study how lines 7-15 change the set *reached*. The innermost for-loop (lines 7-12) modifies *reached* only if $\mathsf{merge}_\times$ modifies its second argument. By definition of the composite merge, this happens only when the location matches. Because the invariant holds at this point, there is either zero or one element with location l' in *reached*. If there is no element with location l' in *reached*, then the termination check (line 13) returns *false*, (l', d') is added to reached, and the invariant is preserved. If there is one element (l', d'') in *reached*, then the merge operator returns the element $(l', d' \sqcup d'')$. If $d'' = d' \sqcup d''$, then *reached* is not modified in the innermost loop, else $(l', d'' \sqcup d')$ replaces (l', d'') in *reached*. (In both cases the invariant is preserved.) The termination check returns *true* because $d' \sqsubseteq d' \sqcup d''$ and therefore the invariant is preserved.

We can observe from the proof of (1) that lines 4-15 of the algorithm *CPA* when called for CPA $\mathbb{D}_\times$ can be equivalently rewritten as:

```
4: choose (l, d) from frontier  // reached[l] = d
5: frontier := frontier \ {(l, d)}
6: for each (l', d') with d' = post(d, g) for CFA edge g = (l, op, l') of P do
7:     if there is (l', d'') ∈ reached then  // reached[l'] = d''
8:         d_new := d' ⊔ d''
```

9: **if** $d_{new} \neq d''$ **then**
10: $\quad$ $frontier := \big(frontier \setminus \{(l', e'')\}\big) \cup \{(l', d_{new})\}$
11: $\quad$ $reached := \big(reached \setminus \{(l', e'')\}\big) \cup \{(l', d_{new})\}$
12: // $reached[l'] = d' \sqcup d''$
13: **else** // $reached[l'] = \bot$
14: $frontier := frontier \cup \{(l', d')\}$
15: $reached := reached \cup \{(l', d')\}$ // $reached[l'] = d' = d' \sqcup \bot$

The code is annotated with the valuation of $reached[\cdot] : L \to E$ defined as follows:

$$reached[l] = \begin{cases} d & \text{if } (l, d) \in reached \\ \bot & \text{if } \neg\exists d : (l, d) \in reached \end{cases}$$

At the beginning of Algorithm *CPA*, we have $reached[l_0] = d_0$ and $reached[l] = \bot$ for $l \neq l_0$. For an element (l, d) in *frontier* and for every edge $g = (l, op, l')$, the algorithm updates $reached[l']$ with its old value joined with $\mathsf{post}(d, g)$. Finally, $(l', reached[l'])$ is added to *frontier* if it was changed. This algorithm corresponds exactly to the traditional fix-point algorithm for data-flow analysis [Kildall 1973], and therefore it is a solution to the data-flow equations (2) and (3). For the details of a proof the reader can refer, for example, to Nielson et al. [1999]. □

Note on widening. While the algorithm used in abstract interpreters is based on a similar idea, it provides one additional feature to accelerate the convergence towards a fixpoint: widening. A widening operator takes a set of abstract states (representing previously computed abstract states for a given location) and returns a new abstract state, larger (according to the pre-order) than all elements in the set. Widening cannot be in general simulated by a CPA based on the same domain. The reason is that the merge operator can only consider one state at a time. To simulate the widening operator, we need access to global information about the set of reached states, and the merge operator only provide access to one state at a time. In the next chapter we present an extension of the CPA framework where a new operator of the analysis (precision adjustment function) takes as input the current set of reached states and can change an abstract state accordingly. Consequently, our extended framework is able to simulate widening (as discussed later in Section 4.3.1).

3.3.2 Encoding a CPA as a Join-Based Analysis

It is not possible to encode an arbitrary CPA as a traditional program analysis (data-flow analysis or abstract interpretation) because the requirements on the operators of a CPA are too weak to guarantee that there is always a corresponding traditional program analysis. In the last subsection, we have seen that a traditional program analysis is translated in a CPA that uses the merge operator merge^{join}. In this subsection, we show how an arbitrary CPA can be converted to a CPA whose merge operator always produce a result larger than both its arguments (a key property of a traditional program analysis). As a consequence, the reached set will contain exactly one abstract state because the termination check will always succeed. This setting corresponds to the behavior of a traditional program analysis.

More specifically, given a CPA $\mathbb{D} = (D, \rightsquigarrow_{\mathbb{D}}, \mathsf{merge}_{\mathbb{D}}, \mathsf{stop}_{\mathbb{D}})$, we build a CPA $\mathbb{D}_j = (D_j, \rightsquigarrow_j, \mathsf{merge}_j, \mathsf{stop}_j)$ such that the merge operator merge_j satisfies the (stronger) requirement:

$$\forall e_1, e_2 : \mathsf{merge}_j(e_1, e_2) \sqsupseteq e_1 \sqcup e_2$$

We mentioned in Section 3.2.3 on page 48 that an analysis based on merge^{sep} (the most precise merge operator) corresponds to an analysis based on merge^{join} on the powerset domain. The domain of $\mathbb{D}_j$ needs to be able to simulate arbitrary CPAs including those based on merge^{sep}; therefore, the domain of $\mathbb{D}_j$ needs to be the powerset of the domain of $\mathbb{D}$: let E be the set of abstract states on which the domain D of CPA $\mathbb{D}$ is based; the domain D_j of CPA $\mathbb{D}_j$ is based on 2^E. One abstract state of the power set domain represents a set of abstract states. The idea for the construction of the operators of D_j is to use one abstract state of D_j to represent the set of reachable states produced by $\mathbb{D}$. The transfer relation $\rightsquigarrow_j$ has the transfer $S \overset{g}{\rightsquigarrow}_j S'$ if $S' = \bigcup_{e \in S} \{e' \mid e \overset{g}{\rightsquigarrow}_{\mathbb{D}} e'\}$. The merge operator merge_j depends on the merge operator and the termination check of the CPA. Let us first define the auxiliary function $m : 2^E \times 2^E \rightarrow 2^E$ as follows: $m(S, S') = \bigcup_{e \in S, e' \in S'} \mathsf{merge}_{\mathbb{D}}(e, e')$. The join function used by the abstract interpreter is defined as follows: $\mathsf{merge}_j(S, S') = m(S, S') \cup \{e \in S \mid \neg\mathsf{stop}_{\mathbb{D}}(e, m(S, S'))\}$. The termination check compare states individually: $\mathsf{stop}_j = \mathsf{stop}^{sep}$.

Discussion. We have established that in principle CPAs can be expressed in a framework that always join states similarly to traditional program analyses. The CPA framework does not attempt to encompass analyses that could not be represented in other frameworks, including abstract interpretation; rather, we

claim that a CPA decomposes concepts, so that (1) understanding the behavior and various parameters of the algorithm is easier, and (2) the framework is closer to the actual implementation. Indeed, a CPA distinguishes the kind of data that the analysis tracks (expressed as the domain) from the way abstract states are computed (transfer function), combined (merge operator) and compared (termination check). Thanks to a more operational approach, CPAs are closer to implementations, and they avoid the use of more complex domains that would be necessary to simulate a CPA. In particular, for a given domain, we can modulate the precision of a CPA between a traditional program analysis on the domain and a traditional program analysis on the powerset of the domain only by changing the merge operator.

Moreover, CPAs are flexible with respect to the precision of their operators. As a result, because the transfer relation and the termination check are explicit operator of the CPA, one can explore the use of weaker (but more efficient to implement) transfer relations or termination checks. This flexibility is not readily expressible in the abstract interpretation framework, unless the abstract domain is adapted accordingly.

One objective of the CPA framework is to allow for flexible composition of analyses. Composite program analyses are easily constructed in our framework, particularly for complex combinations. As an example, let us consider the composite analysis given in the last section that combines Blast domain with a pointer analysis. Because of the particular composite merge operator, the set of computed states is such that there is only one pointer-analysis abstract state for a given pair of location and set of predicates. As a result, an equivalent abstract interpreters would use as domain partial functions from pairs of location and set of predicates to pointer-analysis abstract states.

Overall we claim that CPA is a particular way of looking at computation of reachable states via fixpoint computation, which is close to implementation and allows rich and flexible compositions of analyses.

3.4 Application: Configuring Compositions of Analyses

We evaluated our new approach on several combinations of program analyses, under several different configurations. In this section we present the different combinations we have considered and present the result of our experimental evaluation.

Implementation. We implemented the framework of configurable program analysis as an extension of the model checker Blast, in order to be able to reuse many components that are necessary for an analysis tool but out of our focus in this work. Blast supports recursive function calls, as well as pointers and recursive data structures on the heap. For representing the shape-analysis domain we use parts of the three-valued logic analyzer (Tvla) implementation [Lev-Ami and Sagiv 2000]. For pointer-alias analysis, we use the implementation that comes with Cil [Necula et al. 2002]. We use the configuration of Fischer et al. [2005] to compare with predicated lattices.

3.4.1 Configuring Predicate Abstraction + Shape Analysis

For our first set of experiments, we consider the combination of predicate abstraction and shape analysis. To demonstrate the impact of various configurations on performance and precision, we ran our algorithm on the set of example C programs used to evaluate the lazy shape-analysis algorithm [Beyer et al. 2006], extended by some programs to explore scalability. In all cases, refinement is not used: the analyzer is given as input appropriate predicate and shape abstractions. These examples can be divided into three categories: (1) examples that require only unary and binary (shape) predicates to be proved safe (`list_`i, `simple`, and `simple_backw`), (2) examples that require in addition nullary predicates (`alternating` and `list_flag`), and (3) an example that requires that information from the nullary predicates is used to compute the new value of unary and binary predicates (`list_flag2`). A summary of the operators used in each configuration (A-F) is given in Table 3.1. The verification times are given in Table 3.2(a) for the six different configurations. When Blast fails to prove the program safe for a given configuration, a false alarm (fa) is reported.

A: Predicated Lattice (merge-pred-join, stop-sep)

In our first configuration we use the traditional model-checking approach (no join) for the predicate abstraction, and the predicated-join approach for the shape analysis. We use the symbols and notions that were introduced in the last section, for the location analysis, predicate abstraction, and shape analysis. Configuration A corresponds to the following composite operators:

1. $(l, r, G) \overset{g}{\rightsquigarrow}_{\times} (l', r', G')$ if $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$ and $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} r'$ and $G \overset{g}{\rightsquigarrow}_{\mathbb{S}} G'$

```
 1 typedef struct node {
 2   int h;
 3   struct node *n;
 4 } *List;
 5
 6 void foo(int flag) {
 7   List a = (List) malloc(sizeof(struct node));
 8   if (a == NULL) return;
 9   List p = a;
10   while (random()) {
11     if (flag)  p->h = 1;
12     else       p->h = 2;
13     p->n = (List) malloc(sizeof(struct node));
14     if (p->n == NULL) return;
15     p = p->n;
16   }
17   p->h = 3;
18 }
```

Figure 3.3: Example C program

2. $\mathsf{merge}_\times((l, r, G), (l', r', G')) = \begin{cases} (l', r', \mathsf{merge}_\mathbb{S}(G, G')) & \text{if } l = l' \text{ and } r = r' \\ (l', r', G') & \text{otherwise} \end{cases}$

3. $\mathsf{stop}_\times((l, r, G), R) = \mathsf{stop}^{sep}((l, r, G), R)$

The transfer relation is Cartesian, i.e., the successors of the different components are computed independently (cf. [Beyer et al. 2006]). The merge operator joins the shape graphs of abstract regions that agree on both the program location and the predicate region. The predicate regions are never joined. Termination is checked using the coverage against single abstract states (stop^{sep}). This configuration corresponds to a *predicated lattice* [Fischer et al. 2005].

Example. To illustrate the difference between the various configurations, we use the C program in Figure 3.3. This program constructs a list that contains the data values 1 or 2, depending on the value of the variable `flag`, and ends with a single 3. We illustrate the example using the following abstractions. In the predicate abstraction, we keep track of the nullary predicate *flag*. In the shape analysis, we represent the list pointed to by program variable `a` by shape graphs of a shape class that contains points to predicates pt_a and pt_p, field predicates $fd_{h=1}$, $fd_{h=2}$ and $fd_{h=3}$, and a binary predicate n. (These abstractions can be automatically discovered by the refinement procedure described in Chapter 5, but we do not focus on refinement in this chapter.) Figure 3.4 shows some shape graphs that are encountered during the analysis.

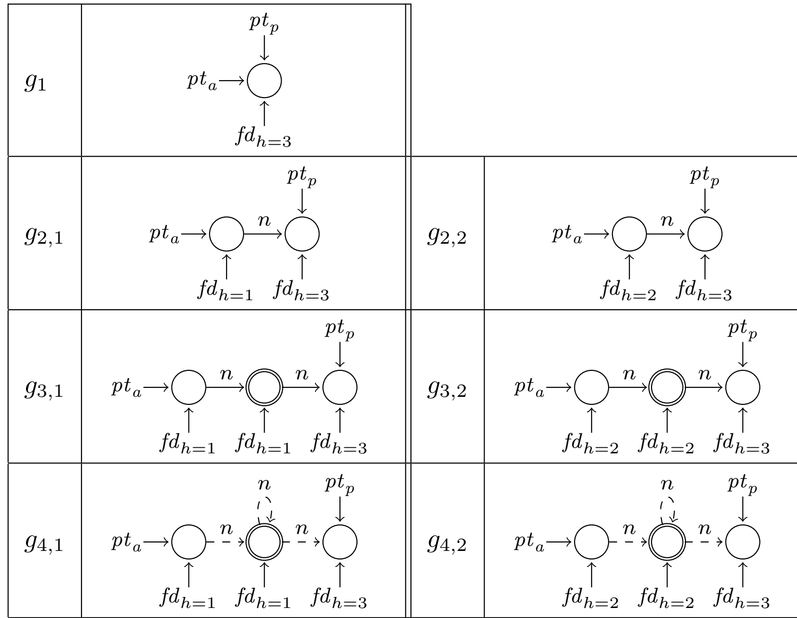


Figure 3.4: Example of shape graphs computed during the analysis of the program in Figure 3.3

To understand how this composite program analysis works on this example, we consider abstract states for which the program location component has the value 18 (program exit point). Because of the merge operator, abstract states that agree on both program location and predicates are joined. Consequently, shape graphs corresponding to lists with different lengths are collected in a single abstract state. When the program analysis has computed all reachable abstract states, we therefore find at most one abstract state per program location and predicate valuation, e.g., $(18, \mathit{flag}, \{g_1, g_{2,1}, g_{3,1}, g_{4,1}\})$ and $(18, \neg\mathit{flag}, \{g_1, g_{2,2}, g_{3,2}, g_{4,2}\})$.

Experimental Results. Precision: Shape analysis is based on a powerset domain, and therefore the join has no negative effect on the precision of the analysis. *Performance:* The idea behind the join in data-flow analysis is to keep the number of abstract states small for efficiency and progress reasons, and in a typical data-flow analysis the join operations are efficient. However, since an abstract state contains a set of shape graphs in our analysis, the effect is the opposite: the join operations add extra work, because larger sets of shape graphs need to be manipulated. In addition, when the algorithm computes successors of a joined set, the work that may have been done already for some subset is repeated. This results in unnecessarily many, highly expensive operations.

B: As Precise as Model Checking (merge-sep, stop-sep)

Now we want to avoid that the merge operator causes join overhead in the analysis when computing abstract successor states. This is easy to achieve in our composite program analysis: we replace the composite merge operator $\mathsf{merge}_\times$ by the merge operator merge^{sep}. The new composite program analysis joins neither predicate regions nor shape regions, and corresponds to the reachability algorithm used in *lazy shape analysis* [Beyer et al. 2006].

Example. Since this composite program analysis is not joining elements, there is no reached abstract state with a set of shape graphs of size larger than 1 (unlike in the previous configuration A). Instead, we maintain distinct abstract states. In particular, at the exit program location, the set of reached abstract states contains the following abstract states:

$$\begin{array}{llll} (18, \mathit{flag}, \{g_1\}), & (18, \mathit{flag}, \{g_{2,1}\}), & (18, \mathit{flag}, \{g_{3,1}\}), & (18, \mathit{flag}, \{g_{4,1}\}), \\ (18, \neg\mathit{flag}, \{g_1\}), & (18, \neg\mathit{flag}, \{g_{2,2}\}), & (18, \neg\mathit{flag}, \{g_{3,2}\}), & (18, \neg\mathit{flag}, \{g_{4,2}\}) \end{array}$$

This set of abstract states represents exactly the same set of concrete states as the result of the previous analysis (configuration A). Note that the shape graph $g_{3,i}$ is subsumed by shape graph $g_{4,i}$ but is not eliminated by the algorithm because the abstract state with $g_{3,i}$ was added to the set *reached* before $g_{4,i}$.

Experimental Results. All examples in our experiments have smaller run times using this configuration, and the precision in the experiments does not change, compared to configuration A. *Precision:* Shape analysis is based on a powerset domain, and therefore, joins are precise. The precision of the predicated lattice is the same as the precision of this variant without joins. *Performance:* Although the number of explored abstract states is slightly higher, this configuration improves the performance of the analysis. The size of lattice elements (i.e., the average number of shape graphs in an abstract state) is considerably smaller than in the predicated-lattice configuration (A). Therefore, we achieve a better performance, because operations (in particular the successor computations) on small sets of shape graphs are much more efficient than on large sets.

C: More Precision Using an Improved Transfer Relation (merge-sep, stop-sep, transfer-new)

From the first to the second configuration, we could improve the performance of the analysis. Now, we show how the precision of the analysis can be improved. We replace the Cartesian transfer relation [Beyer et al. 2006] by a new, more

	Predicate CPA		Shape CPA		Composite CPA		
	merge	stop	merge	stop	$\rightsquigarrow_\times$	merge$_\times$	stop$_\times$
A	sep	sep	join	sep	Cartesian	pred-join	sep
B	sep	sep	join	sep	Cartesian	sep	sep
C	sep	sep	join	sep	strengthened	sep	sep
D	sep	sep	join	join	Cartesian	pred-join	pred-join
E	sep	sep	join	join	Cartesian	join-shape	pred-join
F	join	sep	join	join	Cartesian	join	join

Table 3.1: Configurations of predicate and shape analysis

precise transfer relation that does not compute successors completely independently for the different component analyses:

$$(l, r, G) \overset{g}{\rightsquigarrow}_\times (l', r', G') \text{ if } l \overset{g}{\rightsquigarrow}_\mathbb{L} l' \text{ and } r \overset{g}{\rightsquigarrow}_\mathbb{P} r' \text{ and } G \overset{g}{\rightsquigarrow}_\mathbb{S} G'' \text{ and } G' = \downarrow_{\mathbb{S},\mathbb{P}}(G'', r')$$

The strengthening operator improves the precision of the transfer relation by using the predicate region to sharpen the shape information.

Example. For the example program, the strengthening operator has no effect, because the nullary predicate *flag* has no relation with any predicates used in the shape graph. The strengthening operator would prove useful if, for example, the shape graphs had in addition a unary field predicate $h = x$ (indicating that the field `h` of a node has the same value as the program variable `x`), and the predicate abstraction had the nullary predicate $x = 3$. Consider the operation at line 17 (`p->h = 3`). The successor of the shape graph before applying the strengthening operator can only update the unary field predicate $h = x$ to value 1/2, while the unary field predicate $h = 3$ can be set to value 1 for the node pointed to by p. Supposing $x = 3$ holds in the predicate region of the abstract successor, the strengthening operator updates the field predicate $h = x$ to value 1 as well.

Experimental Results. This configuration results in an improvement in precision over published results for a 'hard-wired' configuration [Beyer et al. 2006], at a reasonable cost. *Precision:* Because of the strengthening operator, the abstract successors are more precise than using the Cartesian transfer relation. Therefore, the whole analysis is more precise. *Performance:* The cost of the strengthening operator is small compared to the cost of the shape-successor computation. Therefore, the performance is not severely impacted when compared to a Cartesian transfer relation.

Table 3.2: Verification time for different configurations

(a) Time for predicate abstraction and shape analysis (false alarm: FA)

Program	A pred-join stop-sep	B merge-sep stop-sep	C merge-sep stop-sep transfer-new	D merge-sep stop-join	E merge-join-shape stop-join	F merge-join stop-join join preds
simple	0.53 s	0.32 s	0.40 s	0.34 s	0.51 s	0.50 s
simple_backw	0.43 s	0.28 s	0.26 s	0.31 s	0.44 s	0.45 s
list_1	0.42 s	0.37 s	0.41 s	0.32 s	0.41 s	0.41 s
list_2	5.24 s	0.85 s	1.25 s	0.86 s	5.34 s	5.36 s
list_3	138.97 s	1.79 s	2.62 s	2.10 s	132.08 s	132.07 s
list_4	$>$ 600 s	9.67 s	15.44 s	11.87 s	$>$ 600 s	$>$ 600 s
alternating	0.86 s	0.61 s	0.96 s	0.60 s	FA	FA
list_flag	0.69 s	0.49 s	0.79 s	0.46 s	FA	FA
list_flag2	FA	FA	0.81 s	FA	FA	FA

(b) Time for predicate abstraction and pointer analysis

Program	CFA nodes	LOC	A: original merge-join	B: more precision merge-sep
s3_clnt	272	2 547	0.68 s	0.83 s
s3_srvr	322	2 542	0.56 s	0.59 s
cdaudio	968	18 225	33.50 s	$>$ 600 s
diskperf	549	14 286	248.33 s	$>$ 600 s

D: As Precise as Model Checking with Improved Termination Check (merge-sep, stop-join)

Now we try to achieve another improvement over configuration B: we replace the termination check with one that checks the abstract state against the join of the reached abstract states that agree on program locations and predicates. To achieve this modification, we would need to change the component CPA for shape analysis to use the termination check stop^{join}. Specifically, we replace the CPA $\mathbb{S}$ in the composition by the CPA $\mathbb{S}'$ defined as follows:

$\mathbb{S}' = (D_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, \mathsf{merge}_{\mathbb{S}}, \mathsf{stop}^{join})$

The composite check operator is then modified as follows:

$\mathsf{stop}_{\times}((l, r, G), R) \;=\; \mathsf{stop}_{\mathbb{S}'}(G, \{G' \mid (l, r, G') \in R\})$

The previous termination check was going through the set of already reached abstract states, checking against every abstract state for coverage. Alternatively, abstract states that agree on the predicate abstraction can be summarized by one single abstract state that is used for the termination check. This is sound because the shape-analysis domain is a powerset domain.

Example. To illustrate the use of the new termination check in the example, consider a set of reached abstract states that contains at some intermediate step the following abstract states: $(18, \mathit{flag}, \{g_1\})$, $(18, \mathit{flag}, \{g_{2,1}\})$, $(18, \neg\mathit{flag}, \{g_1\})$, and $(18, \neg\mathit{flag}, \{g_{2,2}\})$. If we want to apply the termination check to the abstract state $(18, \mathit{flag}, \{g_1, g_{2,1}\})$ and the given set of reached abstract states, we check whether the set $\{g_1, g_{2,1}\}$ of shape graphs is a subset of the join of all shape graphs already found for this program location and valuation of predicates (that is, the set $\{g_1, g_{2,1}\}$). The check would not be positive at this point using termination check stop^{sep}.

Experimental Results. The overall performance impact is slightly negative. *Precision:* This configuration does not change the precision for our examples. *Performance:* We expected improved performance by (1) avoiding many single coverage checks because of the summary abstract state, and (2) fewer successor computations, because we may recognize earlier that the fixpoint is reached. However, the performance impact in our examples is negligible, because a very small portion of the time is spent on termination checks, and the gain is more than negated by the overhead due to the joins.

---

[8] A: predicated join; B: no join (model checking); C: no join and more precise transfer relation; D: no join, termination check with join; E: normal join of shapes (data-flow analysis); F: join for predicate abstraction. All experiments were run on a 3 GHz Intel Xeon processor.

E: Join at Meet-Points as in Data-Flow Analysis for Shapes (merge-join-shape, stop-join)

To compare with a classical data-flow analysis for shape analysis ran independently of a model checker using predicate abstraction, we adapt the previous configuration such that the abstract elements of the shape analysis are joined where the control flow meets, independently of the predicate region. We use the following merge operator, which joins with *all* previously computed shape graphs for the program location of the abstract state:

$$\mathsf{merge}_\times((l,r,G),(l',r',G')) = \begin{cases} (l',r',\mathsf{merge}_\mathbb{S}(G,G')) & \text{if } l = l' \\ (l',r',G') & \text{otherwise} \end{cases}$$

Example. The composite program analysis encounters, for example, the abstract state $(18, \mathit{flag}, \{g_1, g_{2,1}, g_{2,2}, g_{3,1}, g_{3,2}, g_{4,1}, g_{4,2}\})$, which contains shape graphs for lists that contain either 1s or 2s despite the fact that *flag* has the value *true*. Therefore, we note a loss of precision compared to the previous configurations, because the less precise merge operator looses the correlation between the value of the nullary predicate *flag* and the shape graphs.

Experimental Results. The analysis is not able to prove several of the examples that were successfully verified with previous configurations. *Precision:* The shape-analysis component has lost path-sensitivity: the resulting shape graphs are similar to what a classical fixpoint algorithm for data-flow analysis would yield. Therefore, the analysis is less precise. *Performance:* The run time is similar to configuration A.

F: Predicate Abstraction with Join (merge-join for preds)

We now evaluate a composite program analysis that is similar to a classical data-flow analysis without path sensitivity, i.e., both predicates and shapes are joined for the abstract states that agree on the program location. To achieve this goal, we replace component CPA $\mathbb{P}$ for predicate analysis by a CPA $\mathbb{P}'$ that is the same as $\mathbb{P}$ but with the merge operator merge^{join}:

$$\mathbb{P}' = (D_\mathbb{P}, \rightsquigarrow_\mathbb{P}, \mathsf{merge}^{join}, \mathsf{stop}_\mathbb{P})$$

The composite merge operator is modified as follows:

$$\mathsf{merge}_\times((l,r,G),(l',r',G')) = \begin{cases} (l',\mathsf{merge}_{\mathbb{P}'}(r,r'),\mathsf{merge}_\mathbb{S}(G,G')) & \text{if } l = l' \\ (l',r',G') & \text{otherwise} \end{cases}$$

This composite program analysis corresponds exactly to a data-flow analysis on the direct product of the two lattices: the set of reached abstract states contains only one abstract state per program location, because the merge operator joins abstract states of the same program location.

Example. At program location 18, at the end of the analysis, we have only one abstract state: $(18, \mathit{true}, \{g_1, g_{2,1}, g_{2,2}, g_{3,1}, g_{3,2}, g_{4,1}, g_{4,2}\})$.

Experimental Results. This configuration can prove the same example programs as configuration E, and the run times are also similar to configuration E. *Precision:* This composite program analysis is the least precise in our set of configurations, because the merge operator joins both the predicates and the shape graphs independently, for a given program location. While join is suitable for many data-flow analyses, Cartesian predicate abstraction becomes very imprecise when predicate regions are joined, because then it is not possible to express disjunctions of predicates by the means of separate abstract states for the same program location. *Performance:* Compared to configuration E, the number of abstract states is smaller (only one per program location), but the shape graphs have the same size. Therefore, this configuration is less precise, although not more efficient.

Summary

For our set of examples, the experiments have shown that configuration C is the best choice, and we provided justifications for the results. However, we cannot conclude that configuration C is the preferred configuration for *any* combination of abstract interpreters, and we provide evidence for this in the next subsection.

3.4.2 Configuring Predicate Abstraction + Pointer Analysis

In the experimental setting of this subsection we show that for a certain kind of abstract interpreter the join is not only better, but that algorithms without join show prohibitive performance, or do not terminate. We consider the combination of Blast's predicate domain and a pointer-analysis domain, as described at the end of Section 3.2. In Table 3.2(b) we report the performance results for two different algorithms: configuration A for a "predicated lattice," as described by Fischer et al. [2005], and configuration B for an algorithm without join, using the merge operator merge^{sep}. The experiments give evidence that the number of abstract states explodes, and blows up the computational overhead, but the gained precision is not even necessary for proving our example programs correct.

3.5 Related Work

Unifying Model Checking and Program Analysis. Program analysis [Aho et al. 1986] has its roots in compiler optimization and therefore is required to be efficient, often at the expense of precision; model checking [Clarke et al. 1999] has its roots in program verification and thus is required to be precise, usually at the expense of scalability. Accordingly, each of the two approaches has different strengths and weaknesses. It has long been known that, in theory, program analysis can be reduced to model checking, and vice versa [Steffen 1991; Schmidt 1998; Cousot and Cousot 1995]. The contribution presented in this chapter is a *unifying formal implementation framework*, which supports different practical combinations of both approaches, enabling experimentation that can ultimately lead to more powerful software-verification tools.

Algorithms for Model Checking. Most current software model checkers are based on predicate abstraction [Graf and Saïdi 1997]. The exhaustive state-space exploration of the model checking procedure has been successfully implemented using two different types of algorithms. Iterative model checkers, such as Slam [Ball and Rajamani 2002] and Blast [Beyer et al. 2007], perform abstract post operations along the control-flow graph in order to compute abstract successor states and check the reachability of the error location. Bounded model checkers, such as CBMC [Clarke et al. 2004] and SATAbs [Clarke et al. 2005], first transform the program into a Boolean formula by unwinding the control-flow graph, and then check the resulting formula for satisfiability. Nevertheless, the latter is not sound in general because it considers a finite number of unrolling of loops.

Algorithms for Program Analysis. Our execution algorithm (Algorithm 3.1) is given as an abstract sketch of the basic steps that a program-analysis engine has to perform. It does not dictate the details of the iteration order, i.e., the implementation of the 'choose' operation in line 4 of Algorithm 3.1. The extreme depth-first and breadth-first orders are in general not optimal, and finding an optimal ordering of the (chaotic) iterations in a fixpoint computation is a well-known problem in computing science. For example, Bourdoncle analyzed different iteration orders for data-flow analysis which are based on the topological ordering of the control-flow nodes [Bourdoncle 1993]. We have decoupled the choice of iteration algorithm from the abstract domain, and the combination of domains. Moreover, static analyzers such as Astrée [Blanchet et al. 2002] use delayed joins, or path partitioning [Mauborgne and Rival 2005], to improve the

precision and efficiency of the analysis. We can model these techniques within our framework by changing only the merge operator.

Combinations for Model Checking. Fischer et al. have combined the abstract domain of predicate abstraction with a lattice-based data-flow domain to track pointer and value information [Fischer et al. 2005]. The data-flow information becomes more precise by distinguishing different paths through predicates. This configuration represents just one possibility, namely, combining abstract reachability trees for the predicate domain with a join-based analysis for the data-flow domain. Our contribution is to provide a formalism that makes it possible to experiment with flexible combinations of abstract domains, where the algorithm can be parameterized with a merge operation and a termination check. Another example is lazy shape analysis [Beyer et al. 2006], which 'hard-wires' one particular combination of predicate abstraction and shape analysis.

Combinations for Program Analysis. There are many successful approaches for the customization of data-flow domains [Lev-Ami and Sagiv 2000; Dwyer and Clarke 1996; Martin 1998; Tjiangan and Hennessy 1992], and for the combination of abstract interpreters [Gulwani and Tiwari 2006; Cousot and Cousot 1979; Codish et al. 1993; Lerner et al. 2002]. We go one step further, by parameterizing the abstract interpreters in such a way that the analysis algorithm (and therefore the precision) can be controlled so that the domain is analyzed using join-based data-flow analysis, or using tree-based model checking, or using anything in between. The direct product of domains is usually not 'sharp' enough [Cousot and Cousot 1979; Codish et al. 1993], and improvements [Cousot and Cousot 1979; Gulwani and Tiwari 2006] are often hard-coded for particular domains. We prefer to let the designer of the abstract interpreter specify the desired degree of 'sharpness' using the composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$. Our goal is to encode techniques such as delayed join [Blanchet et al. 2002] or path partitioning [Mauborgne and Rival 2005] using our parameters, instead of hard-coding them with a fixed analysis. A recent article [Cousot et al. 2008] on the ASTRÉE project emphasizes the importance of flexible communication between abstract domains, and that standard abstract interpretation [Cousot and Cousot 1977] is not expressive enough to implement practical combinations of program analyses. Cousot et al. also observe that complete lattices and Galois connections are not necessary for program analysis, but that the concretization function matters [Cousot et al. 2008]. Because the framework of abstract interpretation is sometimes too restrictive for practically meaningful analyses, we formalize program analyses using the less restrictive framework of configurable program analysis (CPA).

3.6 Conclusion

We have modified BLAST from a tree-based software model checker to a tool that can be configured using different lattice-based abstract interpreters, composite transfer functions, merge operators, and termination checks, transcending the traditional boundaries between traditional program analyses and software model checking. Specifically configured extensions of BLAST with lattice-based analysis had been implemented before, e.g., in predicated lattices [Fischer et al. 2005] and in lazy shape analysis [Beyer et al. 2006]. As a side-effect, we can now express the algorithmic settings presented in these papers in a simple and systematic way, and moreover, we have found different configurations that perform even better.

We have achieved a flexible combination of model-checking and program-analysis algorithms, and abstract domains that can be used at different precisions. In the next chapter, we present an extension of the framework that adjusts the precision *during the analysis* in order to verify different parts of the program with different precisions.

Chapter 4

Dynamic Precision Adjustment

4.1 Motivation

The success of program analysis depends on finding a delicate trade-off between the precision of an analysis and its cost. If the analysis is not precise enough, the result is an overwhelming number of false alarms; if the analysis is too expensive, it will not scale to large programs. Traditional research in program analysis has focused on efficient, scalable analyses that are specified by the user before being executed by a tool [Cousot and Cousot 1977; Sagiv et al. 2002]. Research in model checking, on the other hand, has focused on expressive, expensive analyses (such as predicate abstraction) whose precision can be increased automatically, during execution of the analysis, as much as necessary [Clarke et al. 2003; Ball and Rajamani 2002]. The traditional approach has the drawbacks that when the result of an analysis is inconclusive, the user has to start a new analysis, e.g., with greater precision; and that a given analysis and precision are applied globally to the whole program. The framework of configurable program analysis (CPA) presented in the previous chapter does not address this particular problem because the domain and the parameters are fixed before the analysis start and cannot be changed during the reachability analysis. The model-checking approach touts to overcome these drawbacks [Henzinger et al. 2002]. However, not only is it expensive to automatically increase the precision of an analysis (e.g., predicate discovery [Chaki et al. 2003; Henzinger et al. 2004]), but in order to support an automatic precision-refinement procedure, the analysis itself (predicate abstraction) is so expressive as to be prohibitively expensive. As a result, there remains a gap of several orders of magnitude between the size of

programs that can be model checked and the size of programs that yield to traditional static analyses [Blanchet et al. 2003].

In this chapter, we present and evaluate a new way of increasing the precision of an analysis during execution. What we do is best viewed as running several different analyses simultaneously and using their results on-line to adjust their respective precisions. For example, we may run an explicit analysis, which tracks the values of a set of program variables, in parallel with a predicate analysis, which tracks the values of a set of predicates. We may start by tracking all variables explicitly, and no predicates. As soon as we encounter, during the analysis, a specified threshold number of different values of a variable, we may switch from tracking explicitly the value of the variable to tracking a predicate (or set of predicates) involving the variable. In other words, we dynamically decrease the precision of one analysis (tracking fewer variables explicitly) and at the same time increase the precision of another analysis (tracking more predicates). This scheme, which can be applied to any combination of program analyses, has several advantages. Compared with the purely predicate-based precision refinement performed by many current software model checkers [Ball and Rajamani 2002; Beyer et al. 2007], predicates are used only when and where a simpler analysis fails. Compared with the explicit and symbolic analyses performed by execution-based model checkers [Godefroid 1997; Sen et al. 2005], predicates (or other forms of widening) can be introduced when and where they are needed to complete a proof. Compared with traditional program analyzers [Blanchet et al. 2003; Sagiv et al. 2002], the precision of an analysis can be refined on-line, during the analysis, when and where necessary.

Motivating example. The program shown in Figure 4.1 is inspired by the code in an SSH server software. We verify that the location at label `ERROR` is not reachable. For the analysis we use the already mentioned combination of an explicit analysis and a predicate analysis, where new predicates are introduced whenever the number of encountered values of a variable exceeds a given threshold. Initially, the explicit analysis tracks the values of the variables `st`, `ok`, `cmd`, and `p`. For each of the first three variables, no more than five different values are encountered. However, the loop counter `p` assumes a number of different values that cannot be bounded by a constant. The precision adjustment that takes place when the number of values for variable `p` hits, say, 10, prevents the explicit analysis from exploring infinitely many concrete values. Then, the precision adjustment injects the predicate $p < n$ into the predicate analysis and turns off the explicit analysis for variable `p`. By tracking the value of the predicate $p < n$ in the subsequent iteration, the loop analysis terminates and

```
 1 int main() {
 2   int st = 0, ok = 0, p;
 3   int *a = getarray();
 4   int n = length(a);
 5   int cmd = readcmd();
 6   while (1) {
 7     switch (st) {
 8     case 0:
 9       if (cmd == 177) { st = 1; }
10       else { st = 2; }
11       break;
12     case 1:
13       if (cmd == 177) { st = 3; cmd = readcmd(); }
14       else { st = 0; }
15       break;
16     case 2:
17       if (cmd != 177) {
18         cmd = 79; st = 4; p = cmd;
19         while (p > 0) { --p; *(a+p) = 0; }
20         if (p > 0) goto ERROR;
21       }
22       else { goto ERROR; }
23       break;
24     case 3:
25       if (cmd == 78) { st = 4; ok = 1; }
26       else { st = 0; cmd = readcmd(); }
27       break;
28     case 4:
29       if (ok) { if (cmd != 78) goto ERROR; }
30       else { if (cmd != 79) goto ERROR; }
31       goto CONTINUE;
32     }
33   }
34  CONTINUE:
35   p = cmd;
36   while (p < n) { ++p; *(a+p) = 0;}
37   if (p < n) goto ERROR;
38   return 0;
39  ERROR:
40   return 1;
41 }
```

Figure 4.1: Example program

the falsehood of the predicate prevents the symbolic execution from entering the error state. During the analysis, we simultaneously performed a *refinement* of the predicate abstract domain, because we added a new predicate, and an *abstraction* of the explicit abstract domain, because we removed a variable. We call this a dynamic (on-line) precision adjustment of the analysis.

We can also apply our approach to run an explicit heap analysis in parallel with the shape analysis presented in Section 2.3 [Sagiv et al. 2002]. We start tracking heap-stored data structures with an explicit, concrete representation of the heap content. When a certain number of concrete heap cells is reached for a given data structure, we switch from tracking heap content explicitly to an appropriate symbolic representation based on shape graphs.

We present our scheme as an extension of the formalism of configurable program analysis (CPA) presented in the previous chapter. While CPA already allows a flexible composition of several analyses, and the on-line transfer of information from one analysis to another, this transfer was limited to local information about the current abstract state. Rather than computing on abstract states only, we compute on pairs (e, π) consisting of an abstract state e and a precision π. For example, e may be the value of a floating-point variable and π the number of digits in the mantissa of the floating-point representation; together they represent a set of possible real numbers. In our algorithm, the precision of an analysis can be changed depending on all abstract state–precision pairs computed so far, i.e., depending on global information. For example, a new predicate may be introduced depending on the fact that a certain set of explicit values of a variable have been encountered (collected) in the analysis so far. Or, one analysis may be switched on or off depending on the accumulated results produced by another analysis so far.

4.2 Related Work

First, it is theoretically possible to model dynamic precision adjustments within a configurable program analysis, but to do so, one would have to encode the set of previously encountered abstract states as part of each abstract state. This would result in a contrived abstract domain and would not allow the reuse of existing analyses.

Second, there have been several previous proposals for combining explicit and symbolic analyses [Yorsh et al. 2006; Gulavani et al. 2006]. While their emphasis is on executing program parts in actual runtime environments in order

to provide inexpensive information to a symbolic search for proof, we view both explicit and symbolic execution as program analyses whose precisions can be adjusted and traded off dynamically. This gives us greater flexibility. While the previous approaches apply explicit program execution to all variables, we choose dynamically and automatically which variables to track explicitly.

Third, in predicate-based abstraction refinement algorithms, new predicates are introduced based on information obtained from counterexamples [Clarke et al. 2003; Ball and Rajamani 2002]. By contrast, we introduce new predicates based on information obtained from the abstract states encountered so far during the analysis. Our work is more closely related to the automatic discovery of invariants from state samples [Ernst et al. 2007], such as linear relationships between variables, and to widening operators used in abstract interpretation [Cousot and Cousot 1977]. However, unlike widening, we propose a framework that does not relax the precision of individual abstract states but adjusts the precision of entire analyses going forward. In particular, the precision of any component analysis may be increased as well as decreased, depending on the accumulated results so far.

4.3 Program-Analysis Framework

We introduce the concept of configurable program analysis with dynamic precision adjustment (CPA+). A CPA+ makes it possible to dynamically adjust the precision of an analysis while exploring the abstract state space of the program. A composite CPA+ can control the precision of its component analyses during the verification process: it can make a component analysis more abstract, and more efficient (in the extreme: switch it off completely), and it can make a component analysis more precise, and more expensive (in the extreme: track the values for each variable with the best precision possible).

4.3.1 Configurable Program Analysis with Dynamic Precision Adjustment (CPA+)

For presentation and formalization purposes, we use the framework of configurable program analysis (Chapter 3). This provides the advantage of making explicit the new, orthogonal capability that dynamic precision adjustment gives the analysis. We add two components to the CPA: in order to pair abstract-domain elements with precisions, we introduce a set of precisions, and in order to adjust the precision of such pairs, we introduce a precision adjustment oper-

ator. Thus the new algorithm for dynamic precision adjustment will be able to maintain and adjust for each abstract state its precision.

A *configurable program analysis with dynamic precision adjustment* (CPA+) $\mathbb{D}^+ = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, a merge operator merge, a termination check stop, and a precision adjustment function prec, which are explained in the following.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ is defined by the set C of concrete states, the semi-lattice $\mathcal{E}$ of abstract states, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ consists of the (possibly infinite) set E of abstract domain elements, the top element $\top \in E$, the bottom element $\bot \in E$, the partial order $\sqsubseteq \subseteq E \times E$, and the function $\sqcup : E \times E \to E$ (the join operator). The function $\sqcup$ yields the least upper bound for two lattice elements, and the symbols $\top$ and $\bot$ denote the least upper bound of the set E and $\emptyset$, respectively. The concretization function $[\![\cdot]\!] : E \to 2^C$ assigns to each abstract state e its meaning, i.e., the set of concrete states that it represents. For soundness of the program analysis, the abstract domain must fulfill the following requirements:

   (a) $[\![\top]\!] = C$ and $[\![\bot]\!] = \emptyset$

   (b) $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$

   (c) $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$
   (the join operator is precise or overapproximate)

   Note that requirements (b) and (c) are equivalent because the join operator $\sqcup$ is defined as the least upper bound.

2. The *set* Π *of precisions* determines the possible precisions of the abstract domain. The program analysis uses elements from Π to keep track of different precisions for different abstract states. A pair (e, π) is called *abstract state* e *with precision* π. The operators on the abstract domain are parametric in the precision. Note that the definition of the concretization function does not depend on the precision, i.e., an abstract state with precision represents the same set of concrete states, irrespective of the precision.

3. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns to each abstract state e possible new abstract states e' with precision π which are abstract successors of e, and each transfer is labeled with a control-flow edge g. We

write $e \overset{g}{\rightsquigarrow} (e', \pi)$ if $(e, g, e', \pi) \in \rightsquigarrow$, and $e \rightsquigarrow (e', \pi)$ if there exists a g with $e \overset{g}{\rightsquigarrow} (e', \pi)$.

The transfer relation must fulfill the following requirement:

(d) $\forall e \in E, g \in G, \pi \in \Pi : \bigcup_{e \overset{g}{\rightsquigarrow} (e', \pi)} [\![e']\!] \;\supseteq\; \bigcup_{c \in [\![e]\!]} \{c' \mid c \overset{g}{\rightarrow} c'\}$
(the transfer relation overapproximates operations for every fixed precision)

4. The *merge operator* $\mathsf{merge} : E \times E \times \Pi \rightarrow E$ weakens the second parameter using the information of the first parameter, and returns a new abstract state of the precision that is given as third parameter.

   The merge operator must fulfill the following requirement:

   (e) $\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq \mathsf{merge}(e, e', \pi)$
   (the result of merge can only be more abstract than the second parameter)

5. The *termination check* $\mathsf{stop} : E \times 2^E \times \Pi \rightarrow \mathbb{B}$ checks if the abstract state given as first parameter with the precision given as third parameter, is covered by the set of abstract states given as second parameter. The termination check can, for example, go through the elements of the set R that is given as second parameter and search for a single element that subsumes ($\sqsubseteq$) the first parameter, or —if D is a powerset domain[1]— can join the elements of R to check if R subsumes the first parameter.

   The termination check must fulfill the following requirement:

   (f) $\forall e \in E,\; R \subseteq E,\; \pi \in \Pi :$
   $\mathsf{stop}(e, R, \pi) = \mathit{true} \;\;\Rightarrow\;\; [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$
   (if an abstract state e is considered to be 'covered' by R, then every concrete state represented by e is represented by some abstract state from R)

6. The *precision adjustment function* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \rightarrow E \times \Pi$ computes a new abstract state and a new precision, for a given abstract state with precision and a set of abstract states with precision. The precision adjustment function may perform widening of the abstract state, in addition to a change of precision.

   The precision adjustment function must fulfill the following requirement:

[1] A *powerset domain* is an abstract domain s.t. $[\![e \sqcup e']\!] = [\![e]\!] \cup [\![e']\!]$.

Algorithm 4.1 $CPA{+}(\mathbb{D}^{+}, P, e_0, \pi_0)$

Input: a CPA with dynamic precision adjustment $\mathbb{D}^{+} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$, a program P, an initial abstract state $e_0 \in E$ with precision $\pi_0 \in \Pi$, where E denotes the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Variables: a set *reached* of elements of $E \times \Pi$, a set *frontier* of elements of $E \times \Pi$

```
frontier := {(e_0, π_0)};
reached := {(e_0, π_0)};
while frontier ≠ ∅ do
  pop (e, π) from frontier;
  // Adjust the precision.
  (ê, π̂) = prec(e, π, reached);
  for each e' with ê ~g~> (e', π̂) for some CFA edge g of P do
    for each (e'', π'') ∈ reached do
      // Combine with existing abstract state.
      e_new := merge(e', e'', π̂);
      if e_new ≠ e'' then
        frontier := (frontier \ {(e'', π'')}) ∪ {(e_new, π̂)};
        reached := (reached \ {(e'', π'')}) ∪ {(e_new, π̂)};
    // Add new abstract state?
    if stop(e', {e | (e, ·) ∈ reached}, π̂) = false then
      frontier := frontier ∪ {(e', π̂)};
      reached := reached ∪ {(e', π̂)}
return {e | (e, ·) ∈ reached}
```

(g) $\forall e, \widehat{e} \in E, p, \widehat{p} \in \Pi, R \subseteq E \times \Pi :$
$(\widehat{e}, \widehat{p}) = \mathsf{prec}(e, p, R) \quad \Rightarrow \quad [\![e]\!] \subseteq [\![\widehat{e}]\!]$

Classical widening and strengthening operators can only decrease and increase precision, respectively. The new operator prec can be used for both increasing and decreasing the precision of abstract states. A CPA (without dynamic precision adjustment) is also capable of a limited form of widening (via the operator merge). But in contrast with the precision adjustment function, the merge operator has only access to one abstract state at a time. Neither the transfer relation nor the merge operator can use global knowledge about the overall analysis progress, i.e., cannot access the set of reachable states. In order to adjust the *global* precision of a program analysis, we need the precision adjustment operator of a CPA+. This operator can be used to realize a global, dynamic change of precision, by taking into account all visited abstract states.

4.3.2 Reachability Algorithm for CPA+

The abstract domain of a program analysis defines a way to describe abstract program states. An analysis algorithm needs to know in addition a precision

for the abstract states (for example, which variable to ignore). Usually this information is hard-wired in either the abstract-domain elements or the algorithm. Algorithm *CPA+* keeps for every abstract state a precision, i.e., we use a pair (e, π) with $e \in E$ and $\pi \in \Pi$ to describe an abstract state and the precision at which it was computed. This means that the precision of the analysis depends on the *context*, i.e., abstract states on which the algorithm is working. The precision used in the algorithm can change dynamically from abstract state to abstract state, by adjusting the precision using the function prec. Each of the three other configurable components (transfer relation, merge operator, and termination check) is parameterized with the precision of the resulting abstract state.

Algorithm *CPA+* (Algorithm 4.1) computes, for a given configurable program analysis with dynamic precision adjustment, a given program, and an initial abstract state with precision, a set of reachable abstract states, i.e., an overapproximation of the set of reachable concrete states. The configurable program analysis with dynamic precision adjustment is given by the abstract domain D, the precisions Π, the transfer relation $\rightsquigarrow$ of the input program, the merge operator merge, the termination check stop, and the precision adjustment function prec. The algorithm keeps updating two sets of abstract states with precision: a set *reached* to store all abstract states with precision that are found to be reachable, and a set *frontier* to store all abstract states with precision that are not yet processed. The state exploration starts from the initial abstract state e_0 with precision π_0. For a current abstract state e with precision π, the algorithm first adjusts the (local) precision of the algorithm using the precision adjustment function prec, based on the set of reached abstract states with precision. Next the algorithm considers each successor e' with the new precision $\widehat{\pi}$, according to the transfer relation. Now, using the given operator merge, the abstract successor state is combined with each existing abstract state in *reached*. If the operator merge has added information to the new abstract state, such that the old abstract state is strictly subsumed, then the new abstract state with precision replaces the old abstract state with precision in the sets *reached* and *frontier* (or is added if the old abstract state with precision is not in the set). If after the merge step the resulting new abstract state with precision is not covered by the set *reached*, then it is added to the set *reached* and to the set *frontier*.

Theorem 4.1 (Soundness of *CPA+*)**.** *Given a configurable program analysis with dynamic precision adjustment $\mathbb{D}^+$, a program P, and an initial abstract*

state e_0 *with precision* π_0*, Algorithm CPA+ computes a set of abstract states that overapproximates the set of reachable concrete states:*

$$[\![CPA+(\mathbb{D}^{+}, P, e_0, \pi_0)]\!] \supseteq Reach(P, [\![e_0]\!])$$

Proof. The proof resembles the proof of soundness of *CPA* (Theorem 3.1). Given a set of states with precisions $R \subseteq 2^{E \times \Pi}$, let $\sigma(R) = \{e \mid \exists \pi : (e, \pi) \in R\}$. We prove that the out-most loop of Algorithm *CPA+* satisfies the following two loop invariants:

$$\forall c \in [\![e_0]\!] : \; c \in [\![\sigma(\mathit{reached})]\!] \tag{4.1}$$

$$\forall c \in [\![\sigma(\mathit{reached} \setminus \mathit{frontier})]\!] : \; \forall c' \text{ s.t. } c \to c' : c' \in [\![\sigma(\mathit{reached})]\!] \tag{4.2}$$

Initially, both sets *reached* and *frontier* are the singleton set $\{(e_0, \pi_0)\}$; therefore initially $\sigma(\mathit{reached}) = \sigma(\mathit{frontier}) = \{e_0\}$. Both invariants hold trivially.

We now prove that if the two invariants hold at the beginning of an iteration, then they hold at the end of the iteration.

We first show that no operation in an iteration removes concrete states from the set $[\![\sigma(\mathit{reached})]\!]$. Let R and W denote the values of variables *reached* and *frontier*, respectively, at the beginning of the iteration, and R' and W' their values at the end of the iteration. We show that the following proposition holds:

$$[\![\sigma(R)]\!] \subseteq [\![\sigma(R')]\!] \tag{4.3}$$

All operations except one leave the variable *reached* unchanged or add an element to it; we only need to consider the operation $\mathit{reached} := \big(\mathit{reached} \setminus \{e''\}\big) \cup \{e_{new}\}$. Let R^{pre} be the value of the variable *reached* before the operation, and R^{post} its value after the operation. The operation is executed only if $e_{new} \neq e''$; as a consequence, we only need to show that $[\![e'']\!] \subseteq [\![e_{new}]\!]$. By construction of the algorithm, we have $e_{new} = \mathsf{merge}(e', e'', \widehat{\pi})$. By requirement (e) on the merge operator, we have $e'' \sqsubseteq e_{new}$, and by requirement (b) on the lattice order, we conclude that $[\![e'']\!] \subseteq [\![e_{new}]\!]$.

The preservation of the first invariant immediately follows from (4.3).

To prove the preservation of the second invariant, we need to show that for all $c \in [\![\sigma(R' \setminus W')]\!]$, and all c' such that $c \to c'$, we have $c' \in [\![\sigma(R')]\!]$. Let (e_{pop}, π_{pop}) be the element with precision removed from the set *frontier* in the

first operation of the loop. We observe that $R' \setminus W' = (R \setminus W) \cup \{(e_{pop}, \pi_{pop})\}$. From Proposition 4.3 it suffices to show that:

$$\{c' \mid \exists c \in [\![e_{pop}]\!] : c \rightarrow c'\} \subseteq [\![\sigma(R')]\!] \tag{4.4}$$

The first operation of the loop is the precision adjustment function: let $(\widehat{e}, \widehat{\pi}) = \mathsf{prec}(e_{pop}, R, \pi_{pop})$. By requirement (g) on the precision adjustment function, we have that $[\![e_{pop}]\!] \subseteq [\![\widehat{e}]\!]$; therefore we can prove (4.4) by proving that $\{c' \mid \exists c \in [\![\widehat{e}]\!] : c \rightarrow c'\} \subseteq [\![\sigma(R')]\!]$. For all e' such that $\widehat{e} \rightsquigarrow (e', \widehat{\pi})$, the algorithm adds e' to the set *reached* (but not *frontier*) unless the termination check $\mathsf{stop}(e', \mathit{reached}, \widehat{\pi})$ succeeds. If e' is added to the set *reached*, trivially we have that $[\![e']\!] \subseteq [\![R']\!]$. Otherwise, by requirement (f) on the termination check stop, we also have $[\![e']\!] \subseteq [\![R']\!]$. Consequently, $\bigcup_{\widehat{e} \rightsquigarrow e'} [\![e']\!] \subseteq [\![R']\!]$ holds. From requirement (d) on the transfer relation, we have $\{c' \mid \exists c \in [\![\widehat{e}]\!] : c \rightarrow c'\} \subseteq \bigcup_{\widehat{e} \rightsquigarrow (e', \widehat{\pi})} [\![e']\!]$. Consequently, the second invariant is also preserved.

Given loop invariants (4.1) and (4.2), if the algorithm terminates, we know that the output value $\widehat{R} = \text{CPA+}(\mathbb{D}^+, e_0, \pi_0)$ satisfies the following two conditions:

$$\forall c \in [\![e_0]\!] : \; c \in [\![\sigma(\widehat{R})]\!] \tag{4.5}$$

$$\forall c \in [\![\sigma(\widehat{R})]\!] : \forall c' \text{ s.t. } c \rightarrow c' : c' \in [\![\sigma(\widehat{R})]\!] \tag{4.6}$$

From (4.5) and (4.6) it immediately follows that $[\![e_0]\!] \subseteq [\![\sigma(\widehat{R})]\!]$, and that $[\![\sigma(\widehat{R})]\!]$ is transitively closed under the concrete transition relation $\rightarrow$. Consequently, by the definition of $\mathit{Reach}(P, [\![e_0]\!])$, the theorem holds. $\square$

We observe that the CPA+ framework is an extension of the CPA framework. More specifically, for every CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ we can build a CPA+ $\mathbb{D}^+ = (D, \Pi, \rightsquigarrow^+, \mathsf{merge}^+, \mathsf{stop}^+, \mathsf{prec})$ for the same domain as follows. The set of precision Π contains a singleton precision π_0. The transfer relation $\rightsquigarrow^+$ has the transfer $e \overset{g}{\rightsquigarrow}{}^+ (e', \pi_0)$ if the transfer relation $\rightsquigarrow$ has the transfer $e \overset{g}{\rightsquigarrow} e'$. The merge operator merge^+ and the termination check stop^+ are based on their CPA counterpart: $\mathsf{merge}^+(e, e', \pi_0) = \mathsf{merge}(e, e')$, and $\mathsf{stop}^+(e, \pi_0, R) = \mathsf{stop}(e, R)$. The precision adjustment function never changes the precision: $\mathsf{prec}(e, \pi_0, R) = (e, \pi_0)$. Algorithm *CPA* applied to $\mathbb{D}$ and Algorithm *CPA+* applied to $\mathbb{D}^+$ returns the same result when run on the same program and the same initial abstract element.

Theorem 4.2 (CPA+ is an extension of CPA)**.** *The CPA+ $\mathbb{D}^+$ corresponding to a valid CPA $\mathbb{D}$ is a valid CPA+, and for every program P and initial abstract state e_0, we have CPA$(\mathbb{D}, P, e_0)$ = CPA+$(\mathbb{D}^+, P, e_0, \pi_0)$, where π_0 is the only precision of $\mathbb{D}^+$.*

Proof. Requirements (a) to (f) hold because of the corresponding requirement for the CPA $\mathbb{D}$. Requirement (g) trivially holds because the precision adjustment function never changes the element. By replacing the operators of $\mathbb{D}^+$ by their definition in Algorithm *CPA+*, we immediately observe that we get an algorithm equivalent to *CPA*. □

4.3.3 Composition for CPA+

A configurable program analysis with dynamic precision adjustment can be composed of several configurable program analyses with dynamic precision adjustment. A *composite CPA+* $\mathcal{C} = (\mathbb{D}^+{}_1, \mathbb{D}^+{}_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$[2] consists of two configurable program analyses with dynamic precision adjustment $\mathbb{D}^+{}_1$ and $\mathbb{D}^+{}_2$ sharing the same set of concrete states, E_1 and E_2 being their respective sets of abstract states, a composite set of precisions $\Pi_\times$, a composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2) \times \Pi_\times$, a composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \times \Pi_\times \rightarrow (E_1 \times E_2)$, a composite termination check $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2} \times \Pi_\times \rightarrow \mathbb{B}$, and a composite precision adjustment function $\mathsf{prec}_\times : (E_1 \times E_2) \times \Pi_\times \times 2^{(E_1 \times E_2) \times \Pi_\times} \rightarrow (E_1 \times E_2) \times \Pi_\times$. The three composites $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ are expressions over the components of $\mathbb{D}^+{}_1$, $\mathbb{D}^+{}_2$, and $\Pi_\times$ ($\rightsquigarrow_i, \mathsf{merge}_i, \mathsf{stop}_i, \mathsf{prec}_i, [\![\cdot]\!]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i$), as well as the operators $\downarrow$ and $\preceq$. The *strengthening* operator $\downarrow$ and the *compare* relation $\preceq$ can be used to increase the precision of the composite operators in a way similar to their use in CPA (Section 3.2.7).

To guarantee that a configurable program analysis with dynamic precision adjustment can be built from the composite program analysis with dynamic precision, we impose the requirements for CPA+ operators (d, e, f, g) from Section 4.3.1 on the four composite operators.

For a given composite program analysis with dynamic precision adjustment $\mathcal{C} = (\mathbb{D}^+{}_1, \mathbb{D}^+{}_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, we can construct the CPA+ $\mathbb{D}^+{}_\times = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, where the product domain $D_\times$ is defined as the direct product of D_1 and D_2: $D_\times = D_1 \times D_2 = (C, \mathcal{E}_\times, [\![\cdot]\!]_\times)$. The product lattice is $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), (\bot_1, \bot_2), \sqsubseteq_\times, \sqcup_\times)$

[2]We extend this notation to any finite number of $\mathbb{D}^+{}_i$.

with $(e_1, e_2) \sqsubseteq_\times (e'_1, e'_2)$ if $e_1 \sqsubseteq_1 e'_1$ and $e_2 \sqsubseteq_2 e'_2$ (and for the join operation, we have $(e_1, e_2) \sqcup_\times (e'_1, e'_2) = (e_1 \sqcup_1 e'_1,\ e_2 \sqcup_2 e'_2)$). The product concretization function $[\![\cdot]\!]_\times$ is such that $[\![(d_1, d_2)]\!]_\times = [\![d_1]\!]_1 \cap [\![d_2]\!]_2$.

Similarly to composite analyses in the CPA framework, we can specify the desired degree of 'sharpness' in the composite operators $\rightsquigarrow_\times$, $\mathsf{merge}_\times$, and $\mathsf{stop}_\times$ rather than by defining new product domains. In addition to that, the CPA+ allows the analysis algorithm to specify a local precision for each abstract state and to adjust this precision dynamically during the analysis. As a consequence, the precision adjustment function can be used to increase the precision of one component analysis and decrease the precision of another component analysis, because the (composite) precision operator can access information of both. In addition to the limited form of widening provided by the composite merge operator, in the case of composition, the CPA framework provides a limited form of strengthening, in the composite transfer relation, via the operator $\downarrow$. Nevertheless, this mechanism is only local, and only the composite precision adjustment function can change globally and dynamically the composite precision (and as a consequence the precision of the component analyses as well).

Theorem 4.3 (Composition of CPA+)**.** *Given a composite program analysis with dynamic precision adjustment $\mathcal{C}$, the result $\mathbb{D}^{+}{}_\times$ of the composition is a configurable program analysis with dynamic precision adjustment.*

Proof. The requirements (a), (b), and (c) for abstract domains of CPA+ follow from the construction of $D_\times$ as product domain (abstract domains are closed under product). The operators of the composite CPA+ fulfill the requirements (d), (e), (f), and (g) of CPA+ by definition. $\square$

4.4 Application: Combining Explicit and Symbolic Program Analyses

The motivation of this work is the need for a formalism that allows us to compose a program analysis from known parts and that makes it possible to dynamically adjust the precision of the algorithm, locally, and across different abstract domains. In particular, we are interested in cases where we first attempt to use an explicit analysis (tracking precisely valuations of variable and content of the heap), and later switch to a symbolic analysis (enabling a powerful representation of infinite set of states), leveraging the results obtained by the explicit analysis so far. More specifically, we are aiming at formalizing the following

two analyses. The first analysis uses an explicit value analysis and a predicate analysis. Our goal is to track variables explicitly in the analysis, and at the point where it becomes too expensive to track all possible values, we abstract the values that we have seen so far by predicates and add these predicates to a predicate analysis. The second analysis uses an explicit heap analysis and a shape analysis. Our goal is to track the content of the heap explicitly, and at the point where the explicit heaps represent too large data structures, we abstract the explicit heap to a shape graph and proceed using only the shape analysis.

We first introduce component CPA+'s and then the two composite CPA+'s. The first three component CPA+'s (location, predicate, and shape analysis) are based on CPAs presented in the last chapter. The other component CPA+'s introduce explicit analyses.

4.4.1 CPA+ for Location Analysis

We extend the CPA $\mathbb{L}$, which tracks the syntactical reachability of program locations, to a single-precision CPA+. The *CPA+ for location analysis* $\mathbb{L}^+ = (D_{\mathbb{L}^+}, \Pi_{\mathbb{L}^+}, \rightsquigarrow_{\mathbb{L}^+}, \mathsf{merge}_{\mathbb{L}^+}, \mathsf{stop}_{\mathbb{L}^+}, \mathsf{prec}_{\mathbb{L}^+})$, consists of the following components:

1. The domain $D_{\mathbb{L}^+}$ is based on the flat lattice for the set L of program locations:
   $D_{\mathbb{L}^+} = (C, \mathcal{E}, [\![\cdot]\!])$, with $\mathcal{E} = (L \cup \{\top, \bot\}, \top, \bot, \sqsubseteq, \sqcup)$,
   $\bot \sqsubseteq l \sqsubseteq \top$ and $l \neq l' \Rightarrow l \not\sqsubseteq l'$ for all elements $l, l' \in L$
   (this implies $\bot \sqcup l = l, \top \sqcup l = \top, l \sqcup l' = \top$ for all elements $l, l' \in L, l \neq l'$),
   and $[\![\top]\!] = C$, $[\![\bot]\!] = \emptyset$, and for all $l \in L$: $[\![l]\!] = \{(l, \cdot, \cdot) \in C\}$.

2. There is only one precision, which is the set of all locations: $\Pi_{\mathbb{L}^+} = \{L\}$.

3. The transfer relation $\rightsquigarrow_{\mathbb{L}^+}$ has the transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}^+} (l', \pi)$ if $g = (l, \cdot, l')$, and the transfer $\top \overset{g}{\rightsquigarrow}_{\mathbb{L}^+} (\top, \pi)$ for all $g \in G$ (the syntactical successor in the CFA without considering the semantics of the operation *op*).

4. The merge operator does not combine elements when control flow meets:
   $\mathsf{merge}_{\mathbb{L}^+}(e, e', \pi) = e'$.

5. The termination check considers abstract states individually:
   $\mathsf{stop}_{\mathbb{L}^+}(e, R, \pi) = (e \in R)$.

6. The precision is never adjusted: $\mathsf{prec}_{\mathbb{L}^+}(e, \pi, R) = (e, \pi)$.

4.4.2 CPA+ for Predicate Analysis

We extend the CPA $\mathbb{P}$ for predicate analysis to a CPA+ with the precision representing the set of tracked predicates. The result is a program analysis for Cartesian predicate abstraction that tracks the validity of predicates in the precision. The *CPA+ for predicate analysis* $\mathbb{P}^+ = (D_{\mathbb{P}^+}, \Pi_{\mathbb{P}^+}, \rightsquigarrow_{\mathbb{P}^+}, \mathsf{merge}_{\mathbb{P}^+}, \mathsf{stop}_{\mathbb{P}^+}, \mathsf{prec}_{\mathbb{P}^+})$ consists of the following components:

1. The domain $D_{\mathbb{P}^+} = (C, \mathcal{E}, [\![\cdot]\!])$ is based on predicates that represent regions (of concrete states). The semi-lattice $\mathcal{E} = (2^{\mathcal{P}}, \emptyset, \mathcal{P}, \supseteq, \cap)$ models the abstract states as finite subsets $e \subseteq \mathcal{P}$ of predicates, where $\mathcal{P}$ is the (infinite) set of quantifier-free predicates over variables from X (using linear-arithmetic expressions and equality with uninterpreted function symbols). The concretization function $[\![\cdot]\!] : 2^{\mathcal{P}} \to 2^C$ assigns to each abstract state r its meaning, i.e., the set of concrete states that it represents: $[\![r]\!] = \{c \in C \mid c \models \varphi_r\}$, where for a set of predicates $r \subseteq \mathcal{P}$, φ_r denotes the conjunction of all predicates in r (in particular $\varphi_\emptyset = \mathit{true}$).

2. The set of precisions $\Pi_{\mathbb{P}^+} = 2^{\mathcal{P}}$ models a precision for an abstract state as a set of predicates. For a predicate p, if p is in precision π, then p is tracked by the analysis when precision π is used, and if p is in an abstract state e, then p is true in all concrete states represented by the abstract state e.

3. The transfer relation $\rightsquigarrow_{\mathbb{P}^+}$ has the transfer $e \overset{g}{\rightsquigarrow}_{\mathbb{P}^+} (e', \pi)$, if $\mathsf{post}(\varphi_r, g)$ is satisfiable and r' is the largest set of predicates from precision π such that $\varphi_r \Rightarrow \mathsf{pre}(p, g)$ for each $p \in r'$, where $\mathsf{post}(\varphi, g)$ and $\mathsf{pre}(\varphi, g)$ denote the strongest postcondition and the weakest precondition, respectively, for a formula φ and a control-flow edge g (as defined in the context of the CPA $\mathbb{P}$ in Section 3.2.5).

4. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{P}^+}(e, e', \pi) = e'$.

5. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{P}^+}(e, R, \pi) = (\exists e' \in R : e \supseteq e')$.

6. The precision adjustment function does not change the abstract state with precision: $\mathsf{prec}_{\mathbb{P}^+}(e, \pi, R) = (e, \pi)$.

If a predicate is not in the precision set and not in the abstract state, then this predicate is not tracked. If a predicate is in the precision set but not in

the abstract state, then there exists a concrete state represented by the abstract state for which the predicate is not true. If a predicate is in the abstract state, then the predicate is true for all concrete states represented by the abstract state.

Note: Local precision

In comparison to the predicate analysis performed by software model checkers such as Blast, the above formalization has two limitations. First, we assume a fixed universe of possible predicates — isolated automatic refinement of the predicate abstraction (e.g., using counterexample-guided abstraction refinement) is not discussed in this context, as our goal is to point out possibilities to refine the abstract states by using information from other component analyses. Second, we use the same global precision at every program location. In contrast, software model checkers often use local precisions: different predicates are tracked at different program locations. Our framework provides an elegant way to enable local precisions by composing a predicate analysis with a location analysis: we can define a composite analysis that uses a map from locations to sets or predicates as composite precision and gives to the component predicate analysis the precision corresponding to the current location. Consider the composite program analysis with dynamic precision adjustment $\mathcal{C}_{loc} = (\mathbb{L}^{+}, \mathbb{P}^{+}, \Pi_{\times}, \rightsquigarrow_{\times}, \mathsf{merge}_{\times}, \mathsf{stop}_{\times}, \mathsf{prec}_{\times})$ defined as follows:

1. A composite precision $\pi_{\times} \in \Pi_{\times}$ is a function from elements of the location CPA+ to precisions of the predicate analysis: $\pi_{\times} : (L \cup \{\top, \bot\}) \to \Pi_{\mathbb{P}^{+}}$.
2. The composite transfer relation has the transfer $(l, r) \overset{g}{\rightsquigarrow}_{\times} (l', r', \pi_{\times})$ if $l \overset{g}{\rightsquigarrow}_{\times} (l', L)$ and $r \overset{g}{\rightsquigarrow}_{\times} (r', \pi_{\times}(l'))$.
3. The composite merge operator never merges: $\mathsf{merge}_{\times}((l, P), (l', P'), \pi_{\times}) = (l', P', v')$.
4. The composite termination check consider composite states individually: $\mathsf{stop}_{\times}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq_{\times} e')$.
5. The composite precision adjustment function does not change the abstract state with precision: $\mathsf{prec}_{\times}((l, r), \pi_{\times}) = (l, r)$.

Note that the solution outlined above (unlike hard-coded local precisions) can be adapted to select the precision for a component analysis based on the value of any other component analysis.

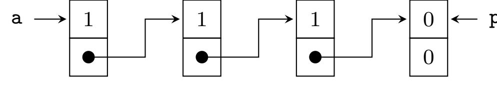


Figure 4.2: Sample explicit-analysis state

4.4.3 CPA+ for Shape Analysis

We extend the CPA $\mathbb{S}$ for shape analysis to a CPA+ with the precision being the shape abstraction specification on which the analysis is based. The result is a program analysis for shape analysis that tracks shape regions of the shape classes corresponding to the precision. The *CPA+ for shape analysis* $\mathbb{S}^{+} = (D_{\mathbb{S}^{+}}, \Pi_{\mathbb{S}^{+}}, \rightsquigarrow_{\mathbb{S}^{+}}, \mathsf{merge}_{\mathbb{S}^{+}}, \mathsf{stop}_{\mathbb{S}^{+}}, \mathsf{prec}_{\mathbb{S}^{+}})$ consists of the following components:

1. The domain $D_{\mathbb{S}^{+}} = (C, \mathcal{D}, [\![\cdot]\!])$ is based on shape regions, where the abstract domain is the lattice $\mathcal{D}$ of shape regions (for any shape abstraction) based on the order on shape regions and the concretization function presented in Section 2.
2. The set of precisions $\Pi_{\mathbb{S}^{+}} = \{\widehat{\Psi} \mid \widehat{\Psi}$ is a shape abstraction specification$\}$ models a precision for an abstract state as a shape abstraction specification. The precision specifies which predicates are used in the shape graphs.
3. The transfer relation $\rightsquigarrow_{\mathbb{S}}$ has the transfer $G \overset{g}{\rightsquigarrow}_{\mathbb{S}} (G', \widehat{\Psi})$ if $G' = \mathsf{post}_{\Psi}(G, g)$, (abstract post successor as described in Section 2.3.4) where Ψ is the shape abstraction corresponding to $\widehat{\Psi}$.
4. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{S}^{+}}(G, G', \widehat{\Psi}) = G'$.
5. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{S}^{+}}(G, R, \widehat{\Psi}) = (\exists G' \in R : G \sqsubseteq G')$.
6. The precision adjustment function does not change the abstract state with precision: $\mathsf{prec}_{\mathbb{S}^{+}}(G, \widehat{\Psi}, R) = (G, \widehat{\Psi})$.

4.4.4 CPA+ for Explicit Value and Heap Analysis

We present an analysis that tracks explicitly both the value of variables and the content of the heap. We then show how the analysis can be specialized to be a pure value analysis (explicit value analysis, similar to constant propagation) or a pure heap analysis (explicit heap analysis). The *CPA+ for explicit analysis* $\mathbb{E}^{+} = (D_{\mathbb{E}^{+}}, \Pi_{\mathbb{E}^{+}}, \rightsquigarrow_{\mathbb{E}^{+}}, \mathsf{merge}_{\mathbb{E}^{+}}, \mathsf{stop}_{\mathbb{E}^{+}}, \mathsf{prec}_{\mathbb{E}^{+}})$, consists of the following components:

1. The abstract domain $D_{\mathbb{E}^*} = (C, \mathcal{E}, [\![\cdot]\!])$ is based on the semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ defined as follows. The set E of abstract elements contains elements of the form $H = (v, h)$, consisting of the two following elements: (1) the variable assignment $v : X \to \mathbb{Z}_\top$ is a total function that maps each variable identifier (integer or pointer variable) to an integer (representing an integer value or a structure address) or the special value $\top$ (representing the value 'unknown'); and (2) the heap assignment $h : \mathbb{Z} \rightharpoonup (F \rightharpoonup \mathbb{Z}_\top)$ is a partial function that maps valid structure addresses to field assignments, also called *structure cell* (memory content). A field assignment is a total function that maps each field identifier of the structure to an integer (representing an integer value or a structure address), or the special value $\top$. In addition, the set E contains the element $\bot$ to denote an abstract states representing no concrete state. The top element $\top = (v_\top, h_\top) \in E$, with $v_\top(x) = \top_\mathcal{Z}$ for all $x \in X$ and $h_\top = \emptyset$, has has no explicit value for any variable and represents every heap. The partial order $\sqsubseteq \subseteq E \times E$ is defined as $\bot \sqsubseteq H$ for all $H \in E$, and $(v, h) \sqsubseteq (v', h')$ if (1) for every $x \in X$, we have $v(x) = v'(x)$ or $v'(x) = \top$, and (2) for every $a \in \mathit{dom}(h')$, $f \in \mathit{dom}(h'(a))$, we have $h(a)(f) = h'(a)(f)$, or $h'(a)(f) = \top$. The join $\sqcup : E \times E \to E$ yields the least upper bound.

   An explicit heap represents all states with matching value and heap contents, for those variables and heap cells that have valuations different from $\top$, modulo a renaming of the addresses. We define the set of addresses that occur in an element $H = (v, h)$ of E, denoted by *addresses*(H), as follows:

$$\begin{array}{rcl} \mathit{addresses}(H) & = & \mathit{dom}(h) \\ & \cup & \{v(x) \neq \top \mid x \in \mathit{dom}(v) \text{ with } T(x) \neq \emptyset\} \\ & \cup & \{h(a)(f) \neq \top \mid a \in \mathit{dom}(h) \text{ with } T(f) \neq \emptyset\} \end{array}$$

   Given an element $H = (v, h)$ of E and an isomorphism α from *addresses*(H) to a set $A' \subseteq \mathbb{Z}$, we define $H|_\alpha = (v', h')$ as follows:

$$\text{For every } x \in \mathit{dom}(v): \quad v'(x) = \begin{cases} v(x) & \text{if } T(x) = \emptyset \\ \alpha(v(x)) & \text{otherwise} \end{cases}$$

$$\text{For every } a \in \mathit{dom}(h): \quad h'(\alpha(a))(f) = \begin{cases} h(a)(f) & \text{if } T(f) = \emptyset \\ \alpha(h(a)(f)) & \text{otherwise} \end{cases}$$

   Two elements H_1 and H_2 of E are *heap-isomorphic*, denoted $H_1 \approx H_2$ if there exists an isomorphism α from *addresses*(H_1) to *addresses*(H_2) such

that $H_1|_\alpha = H_2$, i.e., H_1 and H_2 are identical modulo a renaming of the addresses. The concretization function $[\![\cdot]\!] : E \to 2^C$ is defined as follows:

$$[\![H]\!] = \bigcup_{(v',h') \approx H} \left\{ (l, v_c, h_c) \in C \;\middle|\; \begin{array}{l} \forall x \in X : \\ \quad v'(x) \neq \top \Rightarrow v_c(x) = v'(x) \\ \wedge\ \forall a \in dom(h'), f \in dom(h'(a)) : \\ \quad v'(a)(f) \neq \top \Rightarrow h_c(a)(f) = v'(a)(f) \end{array} \right\}$$

Example. Figure 4.2 depicts an abstract state $H = (v, h)$ of the explicit analysis, where $v = \{a \mapsto 1, p \mapsto 4\}$ and

$$h = \{ \begin{array}{l} 1 \mapsto \{data \mapsto 1, next \mapsto 2\},\ 2 \mapsto \{data \mapsto 1, next \mapsto 3\}, \\ 3 \mapsto \{data \mapsto 1, next \mapsto 4\},\ 4 \mapsto \{data \mapsto 0, next \mapsto 0\} \end{array} \}$$

Note that the four addresses in the heap could be replaced by any four distinct values, without changing the set of concrete states represented by the abstract state. □

2. A precision $\pi = (\nu, \kappa)$ from the set of precisions $\Pi_{\mathbb{E}^+}$ specifies both the maximum number of values that the analysis tracks explicitly for a given variable, and the maximum depth of the explicit heap from a given pointer variable. The first component $\nu : X \to (\mathbb{N} \cup \{\infty\})$ is a function that specifies for each variable a threshold value on the maximal number of different, explicitly stored values, where $\nu(x) = 0$ means the value of variable x is not tracked explicitly, and $\nu(x) = \infty$ means the value of variable x is tracked explicitly no matter how many different values exist. The second component $\kappa : X_{ptr} \to (\mathbb{N} \cup \{\infty\})$ is a function that specifies for each pointer variable x a threshold value on the maximal *depth* that is tracked explicitly for the data structure pointed to by x, where $\kappa(x) = 0$ means that the heap elements reachable from x are not tracked explicitly, and $\kappa(x) = \infty$ means that the analysis does not impose a limit on the number of heap elements it tracks explicitly for the heap elements reachable from x.

3. The transfer relation $\rightsquigarrow_{\mathbb{E}^+}$ has the transfer $(v, h) \overset{g}{\rightsquigarrow} (H', (\nu, \kappa))$ if one of the following holds:

   (1) $g = (\cdot, assume(p), \cdot)$ and either (a) $p_{/v} \Rightarrow false$ and $H' = \bot$, or (b) $p_{/v} \not\Rightarrow false$ and $H' = (v', h)$ and for all $x \in X$:

$$v'(x) = \begin{cases} \top & \text{if } \nu(x) = 0 \text{ or } p_{/v} \Rightarrow (x = \top) \\ c & \text{if } \nu(x) \neq 0 \text{ and } p_{/v} \Rightarrow (x = c) \\ v(x) & \text{otherwise} \end{cases}$$

where $p_{/v}$ denotes the predicate p with all occurrences of a variable $y \in X$ replaced by $v(y)$. (Note that the heap part is not changed because our programming language forbids pointer dereferences inside assume statements.)

(2) $g = (\cdot, \texttt{w}_1 := \texttt{w}_2\texttt{->f}, \cdot)$, $H' = (v', h)$ and for all $x \in X$:

$$v'(x) = \begin{cases} h(v(x))(\texttt{f}) & \text{if } x = \texttt{w}_1 \text{ and } v(x) \in dom(h) \\ \top & \text{if } x = \texttt{w}_1 \text{ and } v(x) \notin dom(h) \\ v(x) & \text{if } x \neq \texttt{w}_1 \end{cases}$$

(3) $g = (\cdot, \texttt{w} := malloc(), \cdot)$, $H' = (v', h')$ and for all $x \in X$:

$$v'(x) = \begin{cases} \top & \text{if } x = \texttt{w} \text{ and } \kappa(x) = 0 \\ a_{new} & \text{if } x = \texttt{w} \text{ and } \kappa(x) > 0 \text{ and } a_{new} \notin dom(h) \\ v(x) & \text{otherwise} \end{cases}$$

and $h' = \begin{cases} \{(a_{new}, \{(f, \top) \mid f \in F\})\} \cup h & \text{if } \kappa(\texttt{w}) > 0 \\ h & \text{otherwise} \end{cases}$

(4) $g = (\cdot, \texttt{w} := exp, \cdot)$, $H' = (v', h)$ and for all $x \in X$:

$$v'(x) = \begin{cases} \top & \text{if } \nu(x) = 0 \\ exp_{/v} & \text{if } \nu(x) \neq 0 \text{ and } x = \texttt{w} \\ v(x) & \text{otherwise} \end{cases}$$

where $exp_{/v}$ denotes the evaluation of an expression exp over X for an abstract value assignment v and explicit heap h:

$$exp_{/v} = \begin{cases} \top & \text{if } x \in X \text{ occurs in } exp \text{ with } v(x) = \top \\ c & \text{otherwise, where expression } exp \text{ evaluates to } c\text{, after} \\ & \text{each occurrence of variable } x \in X \text{ is replaced by } v(x) \end{cases}$$

(5) $g = (\cdot, \texttt{w}_1\texttt{->f} := \texttt{w}_2, \cdot)$, $H' = (v, h')$ and for all $a \in dom(h)$:

$$h'(a) = \begin{cases} \{(g, x) \in h(a) \mid g \neq \texttt{f}\} \cup \{(\texttt{f}, x')\} & \text{if } a = v(\texttt{w}_1) \neq \top \\ h(a) & \text{if } a \neq v(\texttt{w}_1) \neq \top \\ \{(g, x) \in h(a) \mid g \neq \texttt{f}\} & \text{if } v(\texttt{w}_1) = \top \end{cases}$$

Note that there are no transfer from $\bot$.

4. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{E}^+}(e, e', \pi) = e'$.

5. The termination check considers abstract states individually: $\mathsf{stop}_{\mathbb{E}^+}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq e')$.

6. The precision adjustment function sets values or structure fields to $\top$ when the thresholds on the number of different values or on the depth of the heap

are exceeded. Given an element $H = (v, h)$ and an address a, the *depth* of H from a, denoted by $depth(H, a)$, is defined as the maximum number of node on an acyclic path starting from a in the directed graph whose nodes are addresses and where an edge from a_1 to a_2 exists if $h(a_1)(f) = a_2$ for some field f, starting from a. We denote by $ReachAddr(H, a)$ the set of addresses reachable from a in H, i.e. the set of nodes reachable from a in the graph given in the previous definition. The precision adjustment function computes a new abstract state with precision as follows: $\mathsf{prec}_{\mathbb{E}^+}((v, h), (\nu, \kappa), R) = ((v', h'), (\nu', \kappa'))$ if the following holds:

(1) for all $x \in X_{int}$: $\nu'(x) = \nu(x)$ and if $|R(x)| \geq \nu(x)$ then $v'(x) = \top$, otherwise $v'(x) = v(x)$, where $R(x)$ denotes the set $\{v(x) \mid ((v, \cdot), \cdot) \in R\}$ of explicit values held in R for variable x, and

(2) Let $Y = \{y \in X_{ptr} \mid v(y) \neq \top$ and $depth(H, v(x)) \geq \kappa(x)\}$ and $A_Y = \bigcup_{y \in Y} ReachAddr(H, v(y))$. We have that (a) for all x in X_{ptr}, if $v(x) \in A_Y$, then $\kappa'(x) = 0 \ \wedge \ v'(x) = \top$, otherwise $\kappa'(y) = \kappa(y) \ \wedge \ v'(x) = v(x)$, and (b) $h' = \{(a, \cdot) \in h \mid a \notin A_Y\}$.

In the following, we present two specialized, explicit-analysis CPA+, whose precisions are limited to a subset of the precision of $\mathbb{E}$. The first analysis tracks the value of a subset of the integer variables with a fixed threshold on the number of different values, and the second analysis tracks explicitly the data structures pointed to by a subset of the pointer variables with a fixed threshold on the depth of the data structures.

For k in $\mathbb{N} \cup \{\infty\}$, the CPA+ for *explicit value analysis* $\mathbb{C}^+ = (D_{\mathbb{C}^+}, \Pi_{\mathbb{C}^+}, \rightsquigarrow_{\mathbb{C}^+}, \mathsf{merge}_{\mathbb{C}^+}, \mathsf{stop}_{\mathbb{C}^+}, \mathsf{prec}_{\mathbb{C}^+})$ *with threshold* k is defined as follows. The set of precisions $\Pi_{\mathbb{C}^+} = 2^{X_{int}}$ models a precision for an abstract state as a set of integer variables that needs to be tracked. The operators $\rightsquigarrow_{\mathbb{C}^+}$, $\mathsf{merge}_{\mathbb{C}^+}$, $\mathsf{stop}_{\mathbb{C}^+}$, $\mathsf{prec}_{\mathbb{C}^+}$ behave as the corresponding operator in $\mathbb{E}^+$ if precisions are transformed as follows. A precision V of $\mathbb{C}^+$ corresponds to the precision $\pi = (\nu, \kappa)$ defined as follows: for every variable x in X, $\nu(x) = k$ if $x \in V$, and $\nu(x) = 0$ otherwise; for every variable x in X_{ptr}, $\kappa(x) = 0$. Because the precision of the heap part of the analysis κ is a constant function to 0, no data structure on the heap is ever tracked and the heap-component of an abstract element always remains empty. As a consequence, we can restrict the domain $D_{\mathbb{C}^+}$ to abstract elements with empty explicit heaps, and instead of writing $(v, \emptyset)$ in the following, we shall denote an abstract element of the explicit value analysis only by its variable assignment v.

For k in $\mathbb{N} \cup \{\infty\}$, the CPA+ for *explicit heap analysis* $\mathbb{H}^+ = (D_{\mathbb{E}^+}, \Pi_{\mathbb{H}^+}, \rightsquigarrow_{\mathbb{H}^+}, \mathsf{merge}_{\mathbb{H}^+}, \mathsf{stop}_{\mathbb{H}^+}, \mathsf{prec}_{\mathbb{H}^+})$ *with threshold* k is defined as follows. The set of precisions $\Pi_{\mathbb{H}^+} = 2^{X_{ptr}}$ models a precision for an abstract state as a set of pointer variables that needs to be tracked. The operators $\rightsquigarrow_{\mathbb{H}^+}$, $\mathsf{merge}_{\mathbb{H}^+}$, $\mathsf{stop}_{\mathbb{H}^+}$, $\mathsf{prec}_{\mathbb{H}^+}$ behave as the corresponding operator in $\mathbb{E}^+$ if precisions are transformed as follows. A precision Θ of $\mathbb{H}^+$ corresponds to the precision $\pi = (\nu, \kappa)$ defined as follows: for every variable x in X, $\nu(x) = \infty$ if x is in Θ, and $\nu(x) = 0$ otherwise; for every variable x in X_{ptr}, $\kappa(x) = k$ if x is in Θ, and $\kappa(x) = 0$ otherwise. We call an abstract state of the explicit heap analysis an *explicit heap* to emphasize the fact that it focuses on capturing the content of the heap.

4.4.5 Composition of Explicit Value, Predicate, and Location Analysis

We construct a composite CPA+ that dynamically changes the precision across different analyses, based on the analyses defined in the previous subsections. Specifically, we define a composite CPA+ that on-the-fly abstracts the explicit value CPA+ (by switching it off for certain variables) and refines the predicate CPA+ (by adding predicates), using information from the reachable set of abstract states with precision. The predicates added to the predicate CPA+ characterize the sample set of values provided by the explicit CPA+.

The composite program analysis with dynamic precision $\mathcal{C}_1 = (\mathbb{L}^+, \mathbb{P}^+, \mathbb{C}^+, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, is based on the CPA+ for location analysis $\mathbb{L}^+$, the CPA+ for predicate analysis $\mathbb{P}^+$, the CPA+ for explicit value analysis $\mathbb{C}^+$, and the following components:

1. The composite precision is defined by $\Pi_\times = \Pi_{\mathbb{L}^+} \times \Pi_{\mathbb{P}^+} \times \Pi_{\mathbb{C}^+}$. The initial precision of the composite analysis is $(L, \emptyset, X_{int})$, i.e. initially, we track no predicates and all variables are tracked explicitly.

2. The composite transfer relation $\rightsquigarrow_\times$ has the transfer $(l, P, v) \overset{g}{\rightsquigarrow}_\times ((l', P', v'), (\pi_l, \pi_P, \pi_v))$ if $l \overset{g}{\rightsquigarrow}_{\mathbb{L}^+} (l', \pi_l)$ and $P \overset{g}{\rightsquigarrow}_{\mathbb{P}^+} (P', \pi_P)$ and $v \overset{g}{\rightsquigarrow}_{\mathbb{C}^+} (v', \pi_v)$.

3. The composite merge operator never merges: $\mathsf{merge}_\times((l, P, v), (l', P', v'), \pi) = (l', P', v')$.

4. The composite termination check consider composite states individually: $\mathsf{stop}_\times(e, R, \pi) = (\exists e' \in R : e \sqsubseteq_\times e')$.

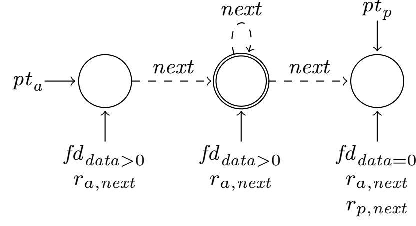


Figure 4.3: Shape graph that is an abstraction of the sample explicit heap depicted in Figure 4.2

5. The composite precision adjustment function is uses relies on the precision adjustment function of $\mathbb{C}^+$. Remember that the composite precision adjustment function of $\mathbb{C}^+$ stops the explicit analysis for a variable x if the number of values for x exceeds the specified threshold. If this situation occurs, then the precision of the predicate analysis is increased by adding all predicates computed by an abstraction function $\mathit{InferPred} : (X \times 2^{X \to \mathcal{Z}}) \to 2^{\mathcal{P}}$. Formally, the composite precision adjustment function is defined by:
$$\mathsf{prec}_\times((l,P,v),(\pi_l,\pi_P,\pi_v),R) = ((l',P',v'),(\pi'_l,\pi'_P,\pi'_v))$$
$$\text{if } \begin{cases} (l',\pi'_l) = (l,\pi_l) \\ (v',\pi'_v) = \mathsf{prec}_{\mathbb{C}^+}(v,\pi_v,\{(v'',\pi''_v) \mid ((l,p,v''),(\cdot,\cdot,\pi''_v)) \in R\}) \\ \pi_P^{new} = \bigcup_{x \in X: v(x) \neq \top \wedge v'(x) = \top} \mathit{InferPred}(x,\{v'' \mid ((l,P,v''),\cdot) \in R\}) \\ \pi'_P = \pi_P \cup \pi_P^{new} \\ P' = P \cup \{p \in \pi_P^{new} \mid v \models p\} \end{cases}$$

Our implementation of *InferPred* applies a simple heuristic for discovering relationships between variables and values, based on predicates that syntactically occur in the program and user-defined predicates. More interesting implementations, which mine predicates that best generalize the sample set of explicit value assignments, can rely on the techniques used in dynamic approaches. For instance, the tool DAIKON [Ernst et al. 2007], automatically detects partial invariants from program executions, and could also be used to implement *InferPred*.

4.4.6 Composition of Explicit Heap, Shape, and Location Analysis

The combination of explicit value and predicate analysis focuses on the analysis of the values of scalar variables. In this section in contrast we focus on the content of the heap. We analyze data structures on the heap using a composition of the CPA+ for explicit heap analysis and the CPA+ for shape analysis. The shape analysis is defined with respect to a maximum shape abstraction

specification $\widehat{\Psi}_{max}$. The composite precision adjustment function tries to find a trade-off between the large space requirements of the explicit heap analysis and the expensive computations of the shape analysis. The explicit analysis is used until the size of a structure in the explicit heaps exceeds a threshold. When the threshold is exceeded, the explicit analysis is deemed too expensive and the explicit heap is converted into a shape graph. Thereafter, the analysis proceeds with tracking more precise shape regions and not storing explicit information anymore for that particular structure.

The composite program analysis with dynamic precision $\mathcal{C}_2 = (\mathbb{L}^+, \mathbb{S}^+, \mathbb{H}^+, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, is based on the CPA+ for location analysis $\mathbb{L}^+$, the CPA+ for shape analysis $\mathbb{S}^+$, the CPA+ for explicit heap analysis $\mathbb{H}^+$, and the following components:

1. The composite precision is defined by $\Pi_\times = \Pi_{\mathbb{L}^+} \times \Pi_{\mathbb{S}^+} \times \Pi_{\mathbb{H}^+}$. The initial precision is $(L, \widehat{\Psi}_\emptyset, X_{ptr})$, where $\widehat{\Psi}_\emptyset$ is the shape abstraction obtained from $\widehat{\Psi}^{max}$ by replacing the tracked pointers by an empty set: $\widehat{\Psi}_\emptyset = \{(\sigma, m, (\emptyset, \emptyset, \Phi)) \mid (\sigma, m, (\cdot, \cdot, \Phi)) \in \widehat{\Psi}^{max}\}$.

2. The transfer relation $\rightsquigarrow_\times$ has the transfer $(l, G, H) \overset{g}{\rightsquigarrow}_\times ((l', G', H'), (\pi_l, \pi_G, \pi_H))$ if $l \overset{g}{\rightsquigarrow}_{\mathbb{L}^+} (l', \pi_l)$ and $G \overset{g}{\rightsquigarrow}_{\mathbb{S}^+} (G', \pi_G)$ and $H \overset{g}{\rightsquigarrow}_{\mathbb{H}^+} (H', \pi_H)$.

3. The composite merge operator never merges: $\mathsf{merge}_\times((l, G, H), (l', G', H'), \pi) = (l', G', H')$.

4. The composite termination check consider composite states individually: $\mathsf{stop}_\times(e, R, \pi) = (\exists e' \in R : e \sqsubseteq_\times e')$.

5. The composite precision adjustment function relies on the precision adjustment function of $\mathbb{H}^+$. Remember that the composite precision adjustment function of $\mathbb{H}^+$ stops the explicit analysis for variables pointing to a data structure whose depth exceeds the specified threshold. If this situation occurs, then the precision of the shape analysis is increased by adding to the tracking definitions of the current precision the pointers that were removed from the explicit precision. Moreover, the function *HeapToShape* generalizes from a set of explicit data structures to shapes, i.e., the function *HeapToShape* is called to abstract heaps to shape regions, for those data structures whose pointers have been removed from the precision of the explicit heap analysis. Function *HeapToShape* takes as parameter an explicit heap, a set of pointers, and a shape abstraction specification, and returns a set of shape regions of the shape abstraction specified by the

third argument, which represents the data structures in the explicit heap pointed to by some pointer in the set given as second argument. Formally, the composite precision adjustment function is defined by:

$$\mathsf{prec}_\times((l,G,H),(\pi_l,\pi_G,\pi_H),R) = ((l',G',H'),(\pi'_l,\pi'_G,\pi'_H))$$
$$\text{if } \begin{cases} (l',\pi'_l) = (l,\pi_l) \\ (H',\pi'_H) = \mathsf{prec}_{\mathbb{H}^+}(H,\pi_H,\{(H'',\pi''_H) \mid ((l,G,H''),(\cdot,\cdot,\pi''_H)) \in R\}) \\ \pi'_G = \big\{(\sigma,m,(T',T'_s,\Phi)) \mid \\ \qquad (\sigma,m,(T,T_s,\Phi)) \in \pi_G \wedge (\sigma,m,(T^{max},T^{max}_s,\Phi)) \in \widehat{\Psi}^{max} \\ \qquad \wedge\, T' = T \cup \{x \in \pi_H \setminus \pi'_H \mid \sigma \text{ is the type of } x\} \\ \qquad \wedge\, T'_s = T^{max}_s \cap T'\big\} \\ G' = G \sqcup \mathit{HeapToShape}(H,(\pi_H \setminus \pi'_H),\pi'_G) \end{cases}$$

Example. The explicit heap H represented in Figure 4.2 has depth 4 from the addresses stored in pointer a. If it were encountered during an analysis with threshold 3, the precision adjustment function of the explicit heap analysis would remove a and p (because the heap cell pointed by p is reachable from a) from its precision and would return the explicit heap $\top$. Suppose the maximal shape abstraction specification $\widehat{\Psi}^{max}$ of the analysis contains a tracking definition that tracks pointers a and p and field assertions $data = 0$ and $data > 0$, and a shape-class generator for singly-linked list that generates binary predicate *next*, unary reachability predicates $r_{x,next}$ for every pointer x in the tracking definition, and the usual points-to and field predicates. The composite precision adjustment function would set the precision of the shape analysis to $\widehat{\Psi}^{max}$ and would abstract explicit heap H to the shape region returned by $\mathit{HeapToShape}(H,\{a,p\},\widehat{\Psi}^{max})$, which contains the shape graph depicted in Figure 4.3. □

Limitations

Note that the shape analysis component does not support isolated refinement (e.g., counterexample-guided abstraction refinement). We suppose that we know a-priori the shape abstraction that we need to use, and we only use the explicit analysis to (1) enable shape analysis when needed and (2) to accelerate shape analysis by allowing the analysis to build on the result of the explicit analysis. The next chapter introduces a refinement technique for shape analysis using the result of an explicit heap analysis, which also incorporates the features of the CPA+ defined in the present section.

4.5 Experimental Evaluation

We implemented the configurable program analysis with dynamic precision adjustment as an extension of the BLAST toolkit, building on our implementation of configurable program analysis. Given a C program and command-line parameters that select the $\mathbb{D}^{+}$ to use for analysis, the tool computes abstract reachable states, and checks that no assertions are violated, in a fully automatic way. All experiments presented in this section were run on a GNU/Linux machine with a Intel Core 2 Duo 6700 processor and 4 GB RAM.

We cannot directly compare with the experiments presented in the previous chapter (Section 3.4) to evaluate the CPA framework, because CPA can perform only fixed instantiations of the precision parameters of CPA+. Nevertheless, we provide a comparison of CPA+ with extreme instances (pure predicate or shape analysis, and pure explicit analysis) of CPA.

4.5.1 Explicit Value Analysis and Predicate Analysis

For the first part of our evaluation, we used the composite CPA+ $\mathcal{C}_1$ that results from combining the CPA+ $\mathbb{L}^{+}$, $\mathbb{P}^{+}$, and $\mathbb{C}^{+}$. Table 4.1 reports verification times results for two example sets (a) and (b). We experimented with several configurations for the threshold that triggers the injection of a predicate into the predicate analysis and switches off the explicit value analysis for a variable. The value k given in the table correspond to the threshold of the explicit value analysis, i.e., the number of different values after which the explicit tracking is stopped. The function *InferPred* of the composite precision adjustment function returns predicates which are currently the result of simple heuristics. Possible alternative implementations include considering syntactic predicates from the program or predicates returned by BLAST's lazy counterexample-guided refinement.

The first set of example programs consists of some constructed, relatively small examples. The programs `ex1` and `ex2` exhibit different scenarios for which the combination of the two analyses is better than any of the component analyses on its own. Program `ex1` is the example from Figure 4.1. The results show that it is best to track variables with a small number of possible values explicitly and the loop index symbolically. The purely explicit analysis ($k = \infty$) fails for `ex1` due to the unbounded number of iterations in one of the loops. Program `ex2` is like `ex1` but the exit condition of the second loop is changed so that the number of iterations is bounded by a known constant. Therefore, the purely

(a) Extreme examples

Program	Symbolic	Explicit + Symbolic		Explicit
	$k = 0$	$k = 1$	$k = 5$	$k = \infty$
ex1	0.46 s	0.17 s	0.21 s	—
ex2	0.43 s	0.16 s	0.21 s	1.00 s
ex3_1	0.16 s	0.13 s	0.11 s	0.08 s
ex3_2	0.40 s	0.24 s	0.14 s	0.09 s
ex3_4	1.41 s	0.58 s	0.20 s	0.12 s
ex3_8	6.53 s	2.10 s	0.31 s	0.18 s
loop1	25.20 s	26.01 s	22.78 s	0.16 s
loop2	279.84 s	277.07 s	258.79 s	0.44 s
square	—	—	—	0.08 s

(b) SSH client/server software

Program	Symbolic	Explicit + Symbolic	
	$k = 0$	$k = 1$	$k = 2$
s3_clnt.1	27.61 s	2.87 s	7.73 s
s3_clnt.2	22.14 s	2.47 s	3.58 s
s3_clnt.3	14.75 s	2.54 s	3.55 s
s3_clnt.4	10.61 s	2.52 s	3.56 s
s3_srvr.1	7.95 s	1.50 s	2.25 s
s3_srvr.2	5.00 s	1.39 s	2.13 s
s3_srvr.3	3.61 s	1.44 s	2.11 s
s3_srvr.4	4.32 s	1.41 s	2.14 s
s3_srvr.6	67.93 s	1.49 s	2.17 s
s3_srvr.7	35.59 s	1.95 s	2.56 s
s3_srvr.8	4.88 s	1.44 s	2.17 s
s3_srvr.9	33.20 s	1.94 s	2.67 s
s3_srvr.10	4.58 s	1.47 s	2.16 s
s3_srvr.11	36.64 s	1.95 s	2.60 s
s3_srvr.12	64.78 s	1.50 s	2.21 s
s3_srvr.13	125.91 s	2.04 s	2.64 s
s3_srvr.14	68.27 s	1.52 s	2.19 s
s3_srvr.15	5.01 s	1.68 s	2.12 s
s3_srvr.16	69.12 s	1.51 s	2.28 s

Table 4.1: Performance evaluation of dynamic precision adjustment on two sets of examples for the combination of predicate analysis and explicit value analysis

explicit analysis succeeds, but the symbolic analysis is still faster due to the high number of different values of the loop counter. Program `ex3_1` results from removing the loops in `ex1`. It shows that the purely explicit analysis is always best. The three variations `ex3_2`, `ex3_4` and `ex3_8` are extended versions of `ex3_1`, with a larger number of program locations, which is twice, four times and eight times the size of `ex3_1`, respectively. This experimental setting illustrates the exponential run time for the symbolic analysis versus the linear run time for the explicit value analysis.

To verify programs `loop1` and `loop2`, it is necessary to unroll a loop 100 and 200 times, respectively. A predicate analysis is prohibitively expensive because it requires a predicate for every value of the variable: $i = 0, i = 1, ..., i = 100$. Program `square` contains a function that computes the square using additions exclusively (strength reduction). Predicate analysis of this program fails for two reasons. First, even when unrolling the loop, the symbolic analysis could not find the predicates necessary to prove safety. Second, the main program uses an expression in an assert statement that is beyond the decision procedure used in Blast. The explicit value analysis is able to prove the program safe efficiently.

The second set of programs (`s3_clnt.`i, `s3_srvr.`i) represents the subroutine for the connection handshake protocol (state machine) of the SSH client and server, for which we verify several protocol-specified safety properties (one line in the table for one property). In all experiments, the combination analysis significantly outperforms the pure predicate analysis ($k = 0$), sometimes by orders of magnitude. The reason is that most predicates in the pure predicate analysis are (mis-) used to track just one single explicit value, for which the explicit value analysis is superior. The configuration for $k = 2$ always needs more time for the verification task, because it tries to track a second value for variables that require symbolic analysis; for these programs it is faster to switch to symbolic tracking after the analysis has found out that one value is not sufficient. A purely explicit analysis results in false alarms for all examples, because there are variables that have to be analyzed symbolically. (The programs contain branches whose outcome depends on inputs or uninitialized variables.)

Discussion

Overall, the combination speeds up significantly the analysis, as fewer predicates are needed. On realistic examples, a small threshold provides a good compromise in term of performance: it prevents the explicit value analysis from running for a needlessly long time when a variable can have too many different

Program	Shapes	Explicit + Shapes		Explicit
		$k = 3$	$k = 5$	$k = \infty$
list_1	0.24 s	0.17 s	0.19 s	—
list_2	0.84 s	0.85 s	1.08 s	—
list_3	2.99 s	3.30 s	5.06 s	—
list_4	8.80 s	11.40 s	24.39 s	—
list_bnd4_1	0.35 s	0.28 s	0.07 s	0.06 s
list_bnd4_2	1.02 s	1.22 s	0.22 s	0.21 s
list_bnd4_3	2.90 s	4.40 s	1.23 s	1.15 s
list_bnd4_4	7.45 s	12.78 s	6.14 s	5.90 s
list_flags_1	0.44 s	0.36 s	0.36 s	—
list_flags_2	1.36 s	1.08 s	1.19 s	—
list_flags_3	4.95 s	3.70 s	4.15 s	—
list_flags_4	19.66 s	13.89 s	16.33 s	—
list_flags_5	79.10 s	59.09 s	74.50 s	—
list_bnd2_f1	0.42 s	0.06 s	0.05 s	0.07 s
list_bnd4_f1	0.69 s	0.51 s	0.11 s	0.08 s
list_bnd4_f2	2.38 s	1.46 s	0.21 s	0.19 s
list_bnd4_f3	8.19 s	4.86 s	0.57 s	0.52 s
list_bnd4_f4	31.65 s	18.40 s	1.94 s	2.14 s
list_bnd4_f5	130.94 s	74.72 s	9.64 s	9.48 s

Table 4.2: Performance comparison on examples for the combination of shape analysis and explicit heap analysis

values, and is enough to provide a great speedup by avoiding the addition of predicates about those variables that have a constant value or only a very few different values.

4.5.2 Explicit Heap Analysis and Shape Analysis

For the second part of our evaluation, we used the composite CPA+ $\mathcal{C}_2$ that results from combining the CPA+ $\mathbb{L}^{+}$, $\mathbb{S}^{+}$, and $\mathbb{H}^{+}$. Table 4.2 reports the verification times for a set of small programs that manipulate lists. We experimented with several configurations for the threshold value k of the explicit heap analysis, which is the number of nodes in the list after which we call the function *HeapToShape* and switch off the explicit heap analysis for the heap. The extreme threshold $k = 0$ switches the explicit analysis off for the whole analysis, and $h = \infty$ keeps the explicit heap analysis always on.

The first four lines of the table let us analyze a negative effect of explicit heap analysis: the lists generated by program `list_i` represent an unbounded sequence of i different data elements in an ascending order. The shape graph of the shape analysis contains $O(i)$ nodes, whereas the size of the explicit structure not only has a size proportional to the length of the list, but explores many

different versions of the list for a given length (combinatorial explosion). The next four examples are variations of the first four, but create lists of bounded length (`list_bnd4_i`). The results show an effect similar to the unbounded case, i.e., the explicit heap analysis suffers from combinatorial explosion for larger values of i, and the summarization in the shape graphs overcompensates the overhead for the expensive shape operations. But the explicit heap analysis can be more efficient when the number of different lists that need to be examined is relatively small.

The third set of examples `list_flags_i` again produces lists of unbounded length. The performance numbers indicate that the explicit heap analysis can be helpful —if the threshold is reasonably small— for constructing the abstract shape graphs by conversion from explicit heap structures (speedup of 24 % for threshold $k = 3$). The last set of examples produces lists of bounded length, but requires larger and many shape graphs to prove the property. This shows that the overhead of expensive shape-analysis operations can be effectively avoided by explicit heap tracking when the control flow of the program limits the number of possible structures for the explicit heap analysis and leads to many different shape graphs in the shape analysis.

Discussion

So far, our preliminary experiments have not identified an absolute advantage of one of the configurations, but that different configurations lead to significantly different performance, and that further work is necessary to leverage the potential that the combination offers.

4.6 Conclusion

We have provided an algorithm and tool for experimenting with the combination of several program analyses. Our method allows the on-line transfer of information between the different analyses, and this information can be used to increase or decrease the precision of any analysis on-the-fly. Using our tool, we showed that it can be beneficial to combine predicate abstraction with an explicit analysis because many predicates are used to track only specific values of variables. Our method allows us to dynamically change the partition between the variables that are analyzed symbolically using predicates and those that are analyzed explicitly. We also studied the effects of combining a symbolic, graph-based shape analysis with an explicit heap analysis. The new formalism

and tool allow us to quickly set up such experiments and study the effects of different parameter settings.

In the next chapter, we present an extension of the same ideas (combinations and change of precision) in the context of a refinement-based analysis. In particular, we show how to use explicit heaps to provide hints for the refinement of a shape abstraction.

Chapter 5

Shape Abstraction Refinement

5.1 Motivation

Proving the safety of programs that use dynamically-allocated data structures on the heap is a major challenge due to the difficulty of finding appropriate abstractions. For cases where the correctness property intimately depends on the shape of the data structure, researchers have over the last decade designed abstractions that are collectively known as shape analysis. The approach that we presented in Section 2.3 and that we have used in the experiments presented in the previous two chapters is an example of shape analysis, where heaps are represented by three-valued logical structures [Sagiv et al. 2002]. The precision of the analysis is specified by a set of predicates over nodes (unary and binary) representing core facts (e.g., points-to and field predicates) and derived facts (e.g., reachability). The latter category of predicates —the so-called instrumentation predicates— are crucial to control how precise the analysis is. First, they can keep track of relevant properties; second, they allow for more precise successor computations; and third, when used as abstraction predicates, they can control node summarization.

In our previous work, we presented an automatic abstraction-refinement loop for shape analysis (as well as predicate abstraction) [Beyer et al. 2006]. If a chosen abstraction is too coarse to prove the desired correctness property, a spurious counterexample path is identified, i.e., a path of the abstract program which witnesses a violation of the property but has no concrete counterpart. We analyzed such counterexample paths in order to determine a set of additional pointers and field predicates which, when tracked by the abstraction, remove the spurious counterexample. These core predicates are then added to the analysis,

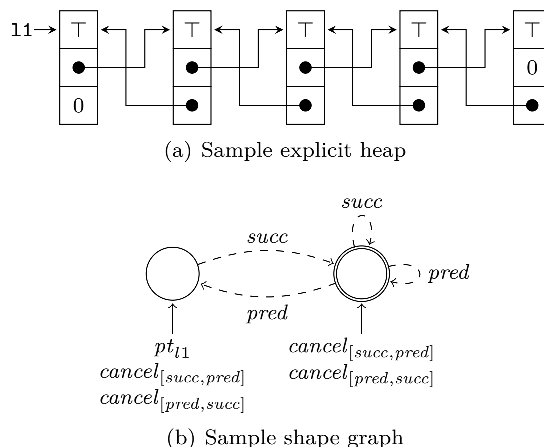
and a new attempt is made at proving the property. A main shortcoming of that work is that the refinement loop never automatically discovers the shape class (e.g., doubly-linked list, binary tree) that is suitable for proving the desired property, and it never adds new instrumentation predicates to the analysis. Consequently, programs can only be verified if all necessary shape classes and instrumentation predicates are "guessed" by the verification engineer when an abstraction is seeded. In the absence of such a correct guess, the method will iteratively track more and more core predicates, until either timing out or giving up because no more relevant predicates can be found.

In this chapter, we focus on the stepwise refinement of a shape analysis by automatically increasing the precision of the shape classes via instrumentation predicates. Similarly to dynamic precision adjustment, refinement increases the precision of the analysis across refinement iterations; unlike dynamic precision adjustment, the analysis removes states with coarser precision from the reached set when a refinement occurs. The refinement strategy that we propose is guided by the analysis of spurious counterexamples (i.e., infeasible paths to an error location). Suppose that counterexample analysis (e.g., following Beyer et al. [2006]) indicates that we need to track the heap structure to which a pointer p points, in order to verify the program. We can encounter two situations: (1) we do not yet track p and we do not know to which kind of data structure p points; or (2) we already track the shape of the heap structure to which p points but the tracked shape class is too coarse and may lack some necessary instrumentation predicates. We address situation (1) by running an explicit heap analysis in order to identify the shape of the data structure from samples, and situation (2) by selecting the coarsest refinement from a lattice of plausible shape classes. Our implementation provides such plausible shape classes by default for standard data structures like lists and trees, but also supports a flexible way to extend the existing shape classes.

Motivating example. We illustrate our method on a simple program that manipulates doubly-linked lists (Figure 5.1). First, two (acyclic) doubly-linked lists of arbitrary length are generated (`alloc_list`); then the two lists are concatenated; finally, the program checks if the result is a valid doubly-linked list (`assert_dll`). Our algorithm automatically verifies that no assertion in this program is violated. The algorithm starts with a trivial abstraction, where no predicates are tracked, and the reachability analysis using this abstraction finds an abstract error path. The algorithm checks whether this abstract error path corresponds to a concrete error path of the program by building a path formula (i.e., a formula which is satisfiable iff the path is a concrete error path). The

```
 1 typedef struct node {
 2   int data;
 3   struct node *succ, *prev;
 4 } *List;
 5
 6 List alloc_list() {
 7   List r = (List) malloc(sizeof(struct node));
 8   List p = r;
 9   if (r == 0) exit(1);
10   while (*) {
11     List t = (List) malloc(sizeof(struct node));
12     if (t == 0) exit(1);
13     p->succ = t;
14     t->pred = p;
15     p = p->succ;
16   }
17   return r;
18 }
19
20 void assert_dll(List p) {
21   while ((p != 0) && (p->succ != 0)) {
22     assert(p->succ->pred == p);
23     p = p->succ;
24   }
25 }
26
27 void main() {
28   List l1 = alloc_list();
29   List l2 = alloc_list();
30
31   List p = l1;
32   while (p->succ != 0) {
33     p = p->succ;
34   }
35   p->succ = l2;
36   l2->pred = p;
37
38   assert_dll(l1);
39 }
```

Figure 5.1: Example C program



(a) Sample explicit heap

(b) Sample shape graph

Figure 5.2: Sample abstract states

path formula of the first abstract error path is unsatisfiable; therefore, this is an infeasible error path (also called spurious counterexample), and the abstraction is refined using an interpolation-guided refinement process. The following atoms occur in interpolants for the first path formula: pointer equalities among `l1`, `l2`, and `p`; `l1->succ` = `p`; and `l2->pred` = `p`. Since the interpolants mention pointers of a recursive data structure, we need to observe them via a shape analysis tracking `l1`, `l2`, and `p` (and their aliases).

But it is not enough to know which pointers to analyze; we also need to know their data structures, in order to determine the shape abstraction, because different data structures require different instrumentation predicates. Since it is the first time we encounter this data structure, our algorithm uses an explicit heap analysis to collect explicit heap samples that would occur during program execution. We graphically illustrate an explicit heap that is collected by the explicit heap analysis in Figure 5.2(a). A node (rectangle with three boxes) represents one structure element; the first box represents the integer value for the field `data`; the second and third box represent the pointer values of the fields `succ` and `prev`, respectively. An arrow represents a pointer valuation. A symbol $\top$ in a box represents an unknown value. When a threshold is hit (e.g., once we have collected explicit heaps with at least 5 nodes each), we stop the explicit heap analysis, and extract the shape class from the explicit heap samples by checking which data structure invariants they satisfy. In the example heap, all nodes satisfy the invariant for acyclic singly-linked lists for each field individually, and the invariant for doubly-linked lists (for every node n, the predecessor of the successor of n is n itself), but not the invariant for binary trees (acyclic graph formed by the two field pointers). Knowing that the data structure is

not a tree, and because both fields `pred` and `succ` occur in interpolants, we restrict the search for a shape abstraction to those suitable for doubly-linked lists. We refine the shape abstraction by choosing the coarsest shape class for doubly-linked lists, i.e., in addition to points-to predicates, we track two binary predicates for the fields `pred` and `succ`, and no instrumentation predicates.

The refined abstraction is still not fine enough to prove the program safe, because we find a new abstract error path. Its path formula is unsatisfiable, but the interpolant-based analysis of the abstract error path does not yield any new predicates. Therefore, we have to search for a finer shape class that contains instrumentation predicates as well. From the previous analysis we know that we have a doubly-linked list. We use a binary search to find, in the given lattice, the coarsest abstraction specification that eliminates the abstract error path. In our example, the tool discovers the necessity to track the unary instrumentation predicates $\mathit{cancel}[\texttt{succ}, \texttt{pred}]$ and $\mathit{cancel}[\texttt{pred}, \texttt{succ}]$ in addition to previously tracked predicates. For a node v, the predicate $\mathit{cancel}[f_1, f_2](v)$ holds if the following condition is fulfilled: if the field f_1 of an element represented by v points to an element represented by some node v', then the field f_2 of the element represented by v' points back to the element represented by v. After this last refinement step, the abstract reachability analysis proves that no assertion is violated. Figure 5.2(b) shows a shape graph that is reachable at the entry point of function `assert_dll`. A node represents a single structure element, and a summary node (drawn as a double circle) represents one or more structure elements. Unary predicate valuations are represented by arrows (or the absence of arrows) from predicates to nodes; binary predicate valuations are represented by arrows between nodes, labeled with the predicate. We can observe that the instrumentation predicates $\mathit{cancel}[\texttt{succ}, \texttt{pred}]$ and $\mathit{cancel}[\texttt{pred}, \texttt{succ}]$ have a valuation of 1 for all nodes in the data structure. Due to the information carried by those instrumentation predicates, we are able to prove the program safe.

5.2 Related Work

Counterexample-guided abstraction refinement (CEGAR) [Clarke et al. 2003] has been used in the context of several different abstractions.

First and foremost, it has been successfully applied in several predicate-abstraction based verifiers [Godefroid 1997; Holzmann 1997; Corbett et al. 2000; Havelund and Pressburger 2000; Ball and Rajamani 2002; Musuvathi et al.

2002; Andrews et al. 2004; Chaki et al. 2004; Clarke et al. 2005; Ivancic et al. 2005; Esparza et al. 2006; Beyer et al. 2007] and some of them also rely on interpolation to infer predicates. Nevertheless, none of the existing software model checkers have an heap analysis to the extent that we are providing, with the strength of shape analysis and the efficiency of abstraction refinement. There have also been proposals to encode shape analysis within predicate-abstraction frameworks [Dams and Namjoshi 2003; Balaban et al. 2005]. So far they apply only to restricted settings, such as singly-linked lists, or they need help from the user for computing abstractions.

Gulavani and Rajamani [2006] proposed CEGAR-based widening operators in the general context of abstract interpretation and applied it to shape analysis.

In the specific context of shape analysis, refinement of a TVLA-based shape analysis has also been studied. Loginov et al. [2005] proposed a technique to learn new instrumentation predicates from imprecise verification results. Our previous work [Beyer et al. 2006] studied how to combine nullary predicate abstraction and shape analysis, and how to refine shape analysis by discovering new core predicates. Nevertheless, we did not address in our previous work the refinement of SCGs.

Our predefined library of SCGs contains well-known instrumentation predicates for the analysis of programs manipulating lists and trees [Sagiv et al. 2002]. More sophisticated predicates have been proposed [Reineke 2005] and can be easily added to the library of SCGs. By relying on a given, fixed library of SCGs, we are still not discovering automatically more general instrumentation predicates. However, there are inherent limitations on what first-order theorem provers can deduce about three-valued abstractions that use transitive-closure predicates such as reachability [Yorsh et al. 2007]. Nevertheless, recent advances in quantified interpolants may lift those limitations [McMillan 2008].

Other approaches to shape analysis exist but refinement is not as central as in our context. Separation logic has been used to design abstract domains that can be used in abstract interpreters [O'Hearn et al. 2001; Reynolds 2002; Distefano et al. 2006], but the question of refinement is less central. The analysis is generally tailored to support only particular data structures and as a result the analysis scales to large programs [Yang et al. 2008]. Recent work considered the extension of the approach to broader families of related data structures [Berdine et al. 2007; Guo et al. 2007]. There exists other attempts to use interpolation for the verification of data structures but they attempt to infer quantified invariants and are more geared towards the verification of program using a data-

structure library [Kapur et al. 2006]. Symbolic shape analysis based on Boolean heaps [Podelski and Wies 2005; Wies 2009] bears many similarities with three-valued logic-based approaches. Couterexample-guided abstraction refinement of Boolean heap abstractions has been considered [Podelski and Wies 2010]. Not only is the domain refined, but the transfer relation is also refined. We have not considered the refinement of the transfer relation but this form of refinement can also be integrated into our framework.

Our current work is also in the tradition of combining symbolic and explicit analyses for program verification. In particular, combinations of symbolic abstraction methods with concrete program execution (testing) to build safety proofs have received much attention recently. Such techniques have been applied in the context of predicate abstraction-based model checkers to accelerate the state construction and guide the refinement [Gulavani et al. 2006; Beckman et al. 2008; Kröning et al. 2004; Yorsh et al. 2006], and in the context of constraint-based invariant generation [Gupta et al. 2009]. We explored in the previous chapter the use of precision adjustment to switch between explicit and symbolic steps during a reachability analysis. We build on the same basic idea in the context of refinement. To the best of our knowledge, no existing technique uses explicit heaps to guide the refinement of a shape abstraction.

5.3 Preliminaries

We describe in the following the techniques that we use in order to analyze counterexamples. First we present how we can construct a formula from a program path such that the formula is satisfiable if and only if the program path is feasible. Second we present a technique to extract useful facts from unsatisfiable path formulas based on interpolation. The same techniques are used in the lazy shape-analysis algorithm [Beyer et al. 2006].

5.3.1 Path Formulas

To check if a given program path is infeasible, we could in theory compute the strongest postcondition. However, due to the large number of quantifiers and the use of our modified theory of arrays, this operation can be expensive to compute. To make the check of infeasibility more efficient, we construct a path formula. A *path formula* is the conjunction of several constraints, one per operation on the program path, such that the program path is infeasible iff the path formula is unsatisfiable. We use an approach similar to static single-

assignment form (SSA) [Cytron et al. 1989], which introduces a new constant for each assignment operation, in order to avoid quantification. If a program path is found to be infeasible, we can learn the reason for its infeasibility —and how to eliminate it— from Craig interpolants for the path formula (as detailed in the next subsection). Every path formula is a formula in the quantifier-free theory of rational linear arithmetic and equality with uninterpreted function symbols. Interpolation is decidable for this theory; satisfiability and interpolants can be computed by state-of-the art decision procedures [Beyer et al. 2008; Cimatti et al. 2008].

The technique for building path formulas used in the previous version of Blast [Henzinger et al. 2004] cannot be reused directly, because it does not handle recursive data structures. However, since the number of structure cells possibly involved in a program path is bounded, we can still produce a finite formula. The address of each structure cell that is accessed on a program path must have been previously assigned to a pointer variable (because we consider a restricted set of possible lvalues). To refer to integer values and addresses in the path formula, we use SSA-like lvalue constants. For a program path of length n, an *lvalue constant* is either $\langle var, l\rangle$ (an identifier constant), or $\langle\langle var, l\rangle\text{->}field, l'\rangle$ with position labels $l, l' \in [0..n]$ and $l' \geq l$. An *lvalue* is either *var*, or $\langle var, l\rangle\text{->}field$. The labels l and l' identify the positions in the program path where the lvalues may have been modified. An lvalue map θ is a function from lvalues to labels. The *lvalue-renaming function* $\mathsf{sub}(\theta, v)$ is defined by $\mathsf{sub}(\theta, s) = \langle s, \theta(s)\rangle$ and $\mathsf{sub}(\theta, s\text{->}f) = \langle \mathsf{sub}(\theta, s)\text{->}f, \theta(\mathsf{sub}(\theta, s)\text{->}f)\rangle$, where s is a variable identifier and f is a field identifier. We extend the function sub to expressions and predicates in the natural way.

To simplify the path formula using alias information, the function may maps a label and an lvalue constant to the set of identifier constants that may have the same value, i.e., $\langle s, l_s\rangle \in \mathsf{may}(l, c)$ if, after the l-th operation of the program path, the value of c may be equal to the value of s after the l_s-th operation of the program path. The function may is not essential for the path formula: it is used only to reduce the size of the path formula by taking into account information that two pointers are guaranteed not to be equal.

The *path formula* for a program path t is the conjunction of all formulas in the final constraint map, i.e., the path formula for t of length l is $\bigwedge_{0\leq i\leq l} \Gamma_i$ with $(\cdot, \Gamma) = \mathsf{Con}(t)$.

The constraint-construction function Con takes as input a triple $((\theta, \Gamma), l, op_l)$, which consists of a pair (θ, Γ) of an lvalue map θ and a constraint map Γ

Operation op_l	New map θ' and Alloc'	Constraint $\Gamma'(l)$
s = e	$\theta'(s) = l$	$\mathsf{sub}(\theta', s) = \mathsf{sub}(\theta, e)$
s_1 = s_2	$\theta'(s_1) = l$ $\forall f \in T(s_1) : \theta'(\langle s_1, l\rangle\text{->}f) = l$	$\mathsf{eqvar}((s_1, \theta'), (s_2, \theta))$
s_1 = s_2->f	$\theta'(s_1) = l$ $\forall f \in T(s_1) : \theta'(\langle s_1, l\rangle\text{->}f) = l$	$\mathsf{sub}(\theta', s_1) = \mathsf{sub}(\theta, s_2\text{->}f)$ $\wedge \bigwedge_{c \in \mathsf{may}(l\text{-}1, \mathsf{sub}(\theta, s_2\text{->}f))} \begin{pmatrix} \mathsf{sub}(\theta, s_2\text{->}f) = c \\ \Rightarrow \mathsf{eqvar}((s_1, \theta'), (c, \theta)) \end{pmatrix}$
s_1->f = s_2	$\theta'(\langle s_1, \theta(s_1)\rangle\text{->}f) = l$ $\forall c \in \mathsf{may}(l\text{-}1, \langle s_1, \theta(s_1)\rangle) :$ $\theta'(c\text{->}f) = l$	$\mathsf{sub}(\theta', s_1\text{->}f) = \mathsf{sub}(\theta, s_2)$ $\wedge \bigwedge_{c \in \mathsf{may}(l\text{-}1, \mathsf{sub}(\theta, s_1))} \begin{pmatrix} \mathsf{ite}.(c = \mathsf{sub}(\theta, s_1)) \\ .\mathsf{sub}(\theta', c\text{->}f) = \mathsf{sub}(\theta, s_2) \\ .\mathsf{sub}(\theta', c\text{->}f) = \mathsf{sub}(\theta, c\text{->}f) \end{pmatrix}$
s = $malloc()$	$\theta'(s) = l$ $\forall f \in T(s) : \theta'(\langle s, l\rangle\text{->}f) = l$ $\mathsf{Alloc}' = \mathsf{Alloc} \cup \{\langle s, l\rangle\}$	$\bigwedge_{a \in \mathsf{Alloc}} \langle s, l\rangle \neq a$
$assume(p)$		$\mathsf{clos}^*(\theta, true, p)$

Figure 5.3: Definition of Con for each program operation: $(\theta', \Gamma') = \mathsf{Con}((\theta, \Gamma), l, op_l)$.

from position labels to first-order logic formulas over lvalue constants, a position label l, and the operation op_l at position l, and produces as output a pair (θ', Γ') of a new lvalue map and a new constraint map. For a given program path t, we define $\mathsf{Con}(t)$ as (θ_n, Γ_n) that is obtained by the recursion $(\theta_l, \Gamma_l) = \mathsf{Con}((\theta_{l\text{-}1}, \Gamma_{l\text{-}1}), l, op_l)$, where l is the position label of op_l in the program path. The map θ_0 is the constant function $\lambda x : 0$, and Γ_0 is the constant function $\lambda x : \emptyset$. The map θ_l differs from $\theta_{l\text{-}1}$ only in the lvalue that may be modified by op_l, which is mapped to l by θ_l. The map Γ_l results from extending $\Gamma_{l\text{-}1}$ by mapping l to the constraint derived from op_l. We derive constraints from operations similarly to the previous approach [Henzinger et al. 2004]. A modification to the previous approach is necessary for assignments to pointers: we cannot 'unroll' a recursive data structure and refer to all reachable structure cells, because this would yield an infinite formula. Additionally, we need to add aliasing constraints when several lvalue constants may point to the same structure cell. The formal definition of the function Con is given in Figure 5.3, where the operations in the first column are defined by the grammar in Figure 2.1.

The function Con for operations of the form s = e assigns in θ' the new position label l to variable identifier s, since the new value of s must be represented by a new identifier constant. The constraint for position l represents the equality of the new identifier constant and the renamed expression. No aliasing needs to be considered since s is of type integer ($T(s) = \emptyset$).

For operations of the form s_1 = s_2, Con assigns the new position label l not only to the variable identifier s_1, but also to all field identifiers of s_1 (if s_1 is of type pointer). The constraint is given by function eqvar, which produces a constraint that represents the equality of two variable identifiers and all their field identifiers (if there are any):

$$\mathsf{eqvar}((s_1, \theta_1), (s_2, \theta_2)) \;=\; \left(\begin{array}{l} \mathsf{sub}(\theta_1, s_1) = \mathsf{sub}(\theta_2, s_2) \quad \wedge \\ \displaystyle\bigwedge_{f \in T(s_1)} \mathsf{sub}(\theta_1, s_1\text{->}f) = \mathsf{sub}(\theta_2, s_2\text{->}f) \end{array} \right)$$

For operations of the form s_1 = s_2->f, the constraint has not only to represent the equality that results from the assignment, but also all possible equalities with pointer variables and their field identifiers, for pointers that are aliased with s_2->f.

For operations of the form s_1->f = s_2, the new lvalue map assigns a new position label to the field identifier s_1->f and to all field identifiers f that can be accessed through other pointers that may be aliased with s_1. The constraint

represents the equality that results from the assignment, and in addition it assigns values to the new lvalue constants for each field identifier f: if the pointer is actually aliased with s_1, then the lvalue constant for its field identifier f is equal to the identifier constant for s_2, and if not, then the lvalue constant does not change.

For allocation operations, besides assigning new position labels to the variable identifier and its field identifiers, we keep track of the new structure cell in the allocation set Alloc, and the constraint represents that the new structure address is different from all previously allocated structure addresses.

For assume operations, the lvalue map does not change, but the assume predicate is renamed, and extended by field-identifier equalities. The function $\mathsf{clos}^*(\theta, b, p)$ produces, given an assume predicate p, the predicate that results from replacing all equalities $s_1 = s_2$ occurring positively (or negatively, depending on the value of the Boolean value b) by $\mathsf{eqvar}((s_1, \theta), (s_2, \theta))$:

$$\mathsf{clos}^*(\theta, b, p) = \begin{cases} \mathsf{clos}^*(\theta, b, p_1)\ \mathit{bop}\ \mathsf{clos}^*(\theta, b, p_2) & \text{if } p \equiv (p_1\ \mathit{bop}\ p_2) \\ \neg\ \mathsf{clos}^*(\theta, \neg b, p_1) & \text{if } p \equiv (\neg p_1) \\ \mathsf{eqvar}((s_1, \theta), (s_2, \theta)) & \text{if } p \equiv (s_1\texttt{==}s_2) \text{ and } b \equiv \mathit{true} \\ \neg\ \mathsf{eqvar}((s_1, \theta), (s_2, \theta)) & \text{if } p \equiv (s_1\texttt{!=}s_2) \text{ and } b \equiv \mathit{false} \\ \mathsf{sub}(\theta, p) & \text{otherwise} \end{cases}$$

Example. Recall the program whose CFA is given in Figure 2.3 and the infeasible program path used in the example in Section 2.1. Figure 5.4 contains the constraint of the path formula corresponding to the infeasible program path. Note that the conjunction of the constraints is unsatisfiable. □

The following lemma justifies the use of path formulas as replacement for the concrete semantics in our model-checking algorithm, for precisely determining whether a path is infeasible.

Lemma 5.1. *Let P be a program and let t be a program path of P. The program path t is infeasible iff the path formula for t is unsatisfiable.*

Proof. We need to show that the path formula for t and the strongest postcondition of *true* and t are equisatisfiable. Consider a given program path t. We denote the strongest postcondition of *true* and t by $\varphi_{\mathsf{SP}} = \mathsf{SP}(\mathit{true}, t)$. We denote the path formula for t by $\varphi_{\mathsf{Con}} = \bigwedge_i \Gamma_i$, where $(\cdot, \Gamma) = \mathsf{Con}(t)$. In the following, we prove that φ_{SP} and φ_{Con} are equisatisfiable.

	Operation	Constraint of the path formula
	Interpolant: *true*	
1	`a=`$malloc$`()`	$true$
	Interpolant: *true*	
2	`p=a`	$\langle p,2\rangle = \langle a,1\rangle \wedge \langle\langle p,2\rangle\text{->}data,2\rangle = \langle\langle a,1\rangle\text{->}data,1\rangle$ $\wedge\langle\langle p,2\rangle\text{->}next,2\rangle = \langle\langle a,1\rangle\text{->}next,1\rangle$
	Interpolant: $\langle p,2\rangle = \langle a,1\rangle$	
3	`p->data=0`	$\langle\langle p,2\rangle\text{->}data,3\rangle = 0$ $\wedge \begin{pmatrix} \text{ite}.(\langle a,1\rangle = \langle p,2\rangle) \\ .\langle\langle a,1\rangle\text{->}data,3\rangle = \langle\langle p,2\rangle\text{->}data,3\rangle \\ .\langle\langle a,1\rangle\text{->}data,3\rangle = \langle\langle a,1\rangle\text{->}data,1\rangle \end{pmatrix}$
	Interpolant: $\langle\langle a,1\rangle\text{->}data,3\rangle = 0$	
4	`p=a`	$\langle p,4\rangle = \langle a,1\rangle \wedge \langle\langle p,4\rangle\text{->}data,4\rangle = \langle\langle a,1\rangle\text{->}data,3\rangle$ $\wedge\langle\langle p,4\rangle\text{->}next,4\rangle = \langle\langle a,1\rangle\text{->}next,3\rangle$
	Interpolant: $\langle\langle p,4\rangle\text{->}data,4\rangle = 0$	
5	`x=p->data`	$\langle x,5\rangle = \langle\langle p,4\rangle\text{->}data,4\rangle$
	Interpolant: $\langle x,5\rangle = 0$	
6	$assume$(`x!=1`)	$\langle x,5\rangle \neq 1$
	Interpolant: $\langle x,5\rangle = 0$	
7	$assume$(`x!=0`)	$\langle x,5\rangle \neq 0$
	Interpolant: *false*	

Figure 5.4: Constraints of the (unsatisfiable) path formula for an (infeasible) path of the program in Figure 2.3 and interpolants at every cut point

To prove equisatisfiability, we build a reversible mapping from terms occurring in the strongest postcondition (including sel applications, but not upd applications) to terms in the path formula. We denote this partial mapping by μ: for a term e in our modified theory of arrays of the form s or $\mathsf{sel}(h, s, f)$, $\mu(e)$ is a term over lvalue constants. Moreover, we will show that the path formula contains conjuncts that represent all necessary instantiations of the two axioms of our modified theory of arrays used in SP. Note that terms containing array updates (i.e., applications of the upd function) do not occur directly in the translated formula; instead we instantiate all relevant axioms that depends on them (in particular read-over-write axioms).

The proof is by structural induction over the program path.

The base case is trivial to prove (both formulas are *true*). The mapping μ is initially the following function:

$$\mu(e) = \begin{cases} \langle s, 0\rangle & \text{if } e = s \text{ with } s \in X \\ \langle \langle s, 0\rangle\text{->}f, 0\rangle & \text{if } e = \mathsf{sel}(h, s, f) \text{ with } s \in X \text{ and } f \in T(s) \end{cases}$$

Note that it is equivalent to renaming lvalues using the initial mapping θ_0 used in the path formula construction.

We prove the induction case by assuming that the lemma holds for a program path t, and by showing that the lemma holds for the program path $t' = t\ ;\ (op : \cdot)$. Let l' be the length of program path t'. We have that $\mathsf{SP}(true, t') = \mathsf{SP}(\mathsf{SP}(true, t), op)$. Let $(\theta, \Gamma) = \mathsf{Con}(t)$ and $(\theta', \Gamma') = \mathsf{Con}(t')$. We have that for all $0 \leq i < l'$, $\Gamma'_i = \Gamma_i$. Therefore we can rewrite the induction hypothesis as $\mathsf{SP}(true, t)$ and $\bigwedge_{0 \leq i < l'} \Gamma'_i$ are equisatisfiable. Moreover, we assume that μ is the mapping computed in the induction step for program path t. In the following, we define a new mapping μ' for the formulas about t'. We consider six cases, each one corresponding to one kind of program operation, and for each case (1) we define the new mapping μ' by extending the previous mapping μ by associating to newly modified terms lvalue constants with label l', and performing variable-identifier renaming where appropriate, and (2) we show that we have instantiated all relevant axioms for any possible suffix of the program path.

Case $op = s$ = e*:* The new mapping is defined as follows:

$$\mu'(e) = \begin{cases} \langle s, l'\rangle & \text{if } e = s \\ \mu(\widehat{e}) & \text{otherwise, where } \widehat{e} = e[\widehat{s}/s] \end{cases}$$

Because the program is well-typed, it follows that s is a variable of type integer (i.e., $T(s) = \emptyset$), and therefore s cannot occur as a structure address in a sel

expression, i.e. there are no expressions of the form in $\mathsf{sel}(\cdot, s, \cdot)$ occurring in the strongest postcondition. Therefore, no new axioms need to be instantiated.

Case $op = s_1$ `=` s_2*:* The new mapping is defined as follows:

$$\mu'(e) = \begin{cases} \langle s_1, l' \rangle & \text{if } e = s_1 \\ \langle \langle s_1, l' \rangle \texttt{->} f, l' \rangle & \text{if } e = \mathsf{sel}(h, s_1, f) \text{ with } f \in T(s_1) \\ \mu(\widehat{e}) & \text{otherwise, where } \widehat{e} = e[\widehat{s_1}/s_1] \end{cases}$$

In the path formula, we instantiate the congruence axioms to add all terms of the form $\mathsf{sel}(h, s_1, f) = \mathsf{sel}(h, s_2, f)$ with $f \in T(s_1)$ through the use of the eqvar function. (Note that we have that $T(s_1) = T(s_2)$ because the program is well-typed.) Those are the only additional axiom instances necessary. New read-over-write axiom instances are not needed since they can be inferred from previously instantiated read-over-write axioms (by induction hypothesis, we know all relevant read-over-write axioms triggered by upd applications have been instantiated for variable identifiers occurring in the prefix t), the newly introduced congruence axioms, and transitivity of equality.

Case $op = s_1$ `=` s_2`->`f*:* The new mapping is defined as in the previous case. By comparison with the previous case, we need to instantiate new congruence axioms for terms with nested sel function applications. Lvalue constants are intended to represent values without such a nesting. To accommodate this restriction, the path formula instantiates the congruence axioms for any identifier constants that is equal to the new value of pointer s_1. The construction is guaranteed to be sound under the hypothesis that may has the aforementioned soundness requirement.

Case $op = s_1$`->`f `=` s_2*:* The new mapping is defined as follows:

$$\mu'(e) = \begin{cases} \langle \langle s_1, l' \rangle \texttt{->} f, l' \rangle & \text{if } e = \mathsf{sel}(h, s_1, f) \\ \langle \langle s_2, l_2 \rangle \texttt{->} f, l' \rangle & \text{if } e = \mathsf{sel}(h, s_2, f) \text{ and } \mu(s_2) = \langle s_2, l_2 \rangle \\ & \text{and } \langle s_2, l_2 \rangle \in \mathsf{may}(l' - 1, \mu(s_1)) \\ \mu(e) & \text{otherwise} \end{cases}$$

The postcondition formulas never refer to heap assignment other than the most recent ones. Consequently μ' does not contain any mapping for expressions referring to the old value of the heap assignment $\widehat{h}$ in contrast to the other cases; all mappings are updated to refer to the new heap assignment h. In the path formula, read-over-write axioms are instantiated for all possible identifier constants for which it is relevant. Note that this directly relies on the soundness of the may operator since axioms are not instantiated for those value that are

not possibly equal to s_1 (according to may). Moreover, for terms in μ where field identifiers different from f occur, the mapping is not changed, which has the same result has instantiating read-over-axioms for the case where field identifiers are different.

Case $op = s$ = $malloc()$: In the case of allocations, we need to refer to previously allocated addresses. Allocated addresses are represented by the set Alloc in the path formula, and by the predicate alloc in the strongest postcondition. We can see that the set Alloc contains the μ-translation of all pointers t for which $\mathsf{alloc}(h, t)$ holds; therefore, the set of inequalities we introduce in the strongest postcondition is the same as its counterpart in the path formula. The mapping is changed in the following way:

$$\mu'(e) = \begin{cases} \langle s, l' \rangle & \text{if } e = s \\ \mu(\widehat{e}) & \text{otherwise, where } \widehat{e} = e[\widehat{s}/s] \end{cases}$$

No new axiom needs instantiation since only inequalities with a fresh identifier are added to the formula.

Case $op = assume(p)$: In this case, the mapping is left unchanged (i.e., $\mu' = \mu$). Similarly to the first case, we instantiate congruence axioms for those equalities that occur positively in the predicate (it is the role of the clos^* function as it can be seen from its definition). □

Note that the proof could only be done because of the restriction on the programming language, which enforces that each relevant structure cell (i.e. those that are ever dereferenced along the program path) was pointed in the past by some pointer variable. Our implementation supports C code by transforming the code in the simplified programming language, introducing auxiliary variables.

5.3.2 Interpolation

Interpolation is a technique to get small reasons as to why a formula is unsatisfiable. The notion of interpolant was introduced by Craig [1957] and has gained considerable attention in the verification community over the last decade [McMillan 2005a], as a way to extract information from counterexamples for refinement of predicate abstraction over different theories [Henzinger et al. 2004; Esparza et al. 2006; Kapur et al. 2006], as well as for other domains including shape analysis [Beyer et al. 2006], and as a way to approximate transition relations [Jhala and McMillan 2005].

Algorithm 5.1 *ExtractInterpolants*(t, Γ)

Input: an infeasible program path $t = (op_1 : l_1); \ldots; (op_n : l_n)$, a constraint map Γ
Output: a map Π from the locations of t to sets of atomic predicates
$\Pi(l) := \emptyset$ for each control location l
$\psi := \mathit{true}$
for $i := 1$ to $n - 1$ **do**
  $\varphi^- := \psi \wedge \Gamma(i)$
  $\varphi^+ := \bigwedge_{i+1 \leq j \leq n} \Gamma(j)$
  $\psi := \text{ITP}(\varphi^-, \varphi^+)$
  $\Pi(l_i) := \mathsf{Atoms}(\mathsf{Clean}(\psi))$
return Π

For two formulas φ^- and φ^+, such that $\varphi^- \wedge \varphi^+$ is unsatisfiable, a *Craig interpolant* ψ satisfies the three following conditions: (1) formula φ^- implies formula ψ, (2) the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and (3) ψ contains only symbols that are common to φ^- and φ^+. For the theory used in our path formula (propositional formula with equality), interpolants can be computed by readily available tools [McMillan 2005b; Rybalchenko and Sofronie-Stokkermans 2007; Beyer et al. 2008; Cimatti et al. 2008].

The interpolants contain all information that we need in order to eliminate an infeasible program path in further iterations of the analysis. In our context, we use interpolants as a way to extract from unsatisfiable path formulas both pointers and field assertions that a shape analysis should track in order to eliminate an infeasible program path. To extract predicates (and later pointers and field assertions from the predicates) from a path formula, we use Algorithm *ExtractInterpolants*.

Algorithm *ExtractInterpolants* (5.1) takes as input an infeasible program path t and a constraint map Γ, and returns a function Π that assigns to each CFA location of t a set of (nullary) predicates. The constraint map that is given as input is the result of the constraint-construction function Con. The returned function Π contains for every control location of t all predicates that are necessary to eliminate an infeasible program path t from the overapproximation. These predicates can be used to refine a predicate precision in a way that makes the abstract program path also infeasible, or, in our case, to refine tracking definitions.

The algorithm splits the (infeasible) path formula at every control location and computes an inductive interpolant. Function $\text{ITP}(\varphi^-, \varphi^+)$ returns a formula ψ that is a Craig interpolant of φ^- and φ^+. Our algorithm produces interpolants that are *inductive*, i.e. the interpolant at location l_{i+1} is implied by the interpolant at location l_i and the constraint for the operation from l_i

to l_{i+1}. Inductive interpolants are guaranteed to eliminate the infeasible program path. In the actual implementation, we use a function ITP that takes as input a sequence of n formulas and produces a sequence of $n-1$ inductive interpolants [McMillan 2005b]. This way the procedure can be more efficient by deriving all interpolants from the same proof of unsatisfiability. For a given formula, the function Clean replaces each lvalue constant by its lvalue, i.e., removes the labels. The function Atoms returns the set of atomic predicates (no Boolean operators) of a formula. [1]

Example. Consider the infeasible program path and its path formula shown in Figure 5.4. The interpolants at each cut points are indicated at the figure. For example, the interpolant after the fourth operation of the path (at control location 7) is $\langle\langle a, 1\rangle\text{->}data, 3\rangle = 0$, and the interpolant after the fifth (at control location 8) is $\langle\langle p, 4\rangle\text{->}data, 4\rangle = 0$. The extracted predicates (after applying Clean and Atoms) are $a\text{->}data = 0$ for control location 7, and $p\text{->}data = 0$ for control location 8. □

5.4 Shape Analysis with Abstraction and Refinement

We introduce a new verification algorithm that is based on abstraction *and* refinement. Shape types can be refined in two different ways: either we refine the shape type's tracking definition, or we refine the shape type's SCG. In both cases, the resulting shape class is guaranteed to be finer, because SCGs are monotonic. Previous work has shown how tracking definitions can be refined, by extracting information from infeasible error paths using interpolation [Beyer et al. 2006]. Our approach is based on this algorithm, and proposes a novel technique to refine SCGs, by combining information from two sources. The first source of information consists of explicit heaps, which are used to restrict the refinement to SCGs that are designed to support the kind of data structure (e.g., doubly-linked list, binary tree) that the program manipulates. When we discover pointers to data structures for the first time, we run an explicit heap analysis of the program until we encounter explicit heaps with a depth that exceeds a given threshold. The explicit heaps that have been computed are queried for data structure invariants, and are then abstracted to shape graphs. The second source of information consists of infeasible error paths. We simulate shape analysis with different SCGs along the path to determine the coarsest SCG

[1] In principle, it would be possible to use the original interpolants for the predicate precision, but BLAST is based on a Cartesian abstraction, which usually requires to split the formulas.

that is able to eliminate the infeasible path. A library of SCGs that supports standard data structures like lists and trees is available in Blast.

We express the reachability analysis that is performed between refinement iteration as a composite program analysis with dynamic precision adjustment. In order to allow the analysis to terminate before a fixed point is reached in case an error is reached, we adapt the precision adjustment function and the reachability algorithm so as to interrupt the analysis when a refinement might be needed. In this section, first, we introduce the concept of an interruptible CPA+, a CPA+ with the new precision adjustment function; second, we present the modified reachability algorithm; third, we present the component CPA+ and the composite CPA+ that we use; fourth, we describe our overall counterexample-guided refinement loop and the details of our refinement strategy.

5.4.1 Interruptible CPA+ and Reachability Algorithm

We adapt the CPA+ framework to allow the analysis to terminate before a fixed-point is reached in case a refinement of the precision is needed. The resulting analysis is called an interruptible CPA+. The only difference between a CPA+ and an interruptible CPA+ is the precision adjustment function prec: in addition to returning an abstract state with precision, the function may also return the special value *interrupt* meaning that the reachability analysis should be interrupted. Analogously, we can define composite interruptible CPA+ by modifying the composite precision adjustment function.

An *interruptible CPA+* $\mathbb{D}^{+} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ has the same components and requirements as a CPA+ except for the precision adjustment function prec:

> Let E be the set of abstract elements of domain D. The *precision adjustment function* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \rightarrow (E \times \Pi) \cup \{\mathit{interrupt}\}$ either returns a new abstract state and a new precision or returns the special value *interrupt*, for a given abstract state with precision. The precision adjustment function has to fulfill the following requirement:
>
> (g) $\forall e, \widehat{e} \in E, p, \widehat{p} \in \Pi, R \subseteq E \times \Pi :$
> $(\widehat{e}, \widehat{p}) = \mathsf{prec}(e, p, R) \quad \Rightarrow \quad [\![e]\!] \subseteq [\![\widehat{e}]\!]$
>
> Note that requirement (g) leaves the freedom to the analysis to return special value *interrupt* at any time.

Algorithm 5.2 *PartialCPA+*$(\mathbb{D}^+, P, R_0, F_0)$

Input: a CPA+ with early termination $\mathbb{D}^+ = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
a program P, a set of abstract states with precision $R_0 \subseteq E \times \Pi$,
a subset F_0 of R_0 representing its frontier,
where E denotes the set of elements of the semi-lattice of D
Output: a set of reachable abstract states with precision and its frontier
Variables: a set *reached* of elements of $E \times \Pi$,
a set *frontier* of elements of $E \times \Pi$

```
frontier := F_0;
reached := R_0;
while frontier ≠ ∅ do
  pop (e, π) from frontier;
  // Adjust the precision or interrupt the analysis.
  r = prec(e, π, reached);
  if  r = interrupt  then
    frontier := frontier ∪ {(e, π)};
    return  (reached, frontier)
  (ê, π̂) = r;
  for each e' with ê ~g~> (e', π̂) for some CFA edge g of P do
    for each (e'', π'') ∈ reached do
      // Combine with existing abstract state.
      e_new := merge(e', e'', π̂);
      if e_new ≠ e'' then
        frontier := (frontier \ {(e'', π'')}) ∪ {(e_new, π̂)};
        reached := (reached \ {(e'', π'')}) ∪ {(e_new, π̂)};
    // Add new abstract state?
    if stop(e', {e | (e, ·) ∈ reached}, π̂) = false then
      frontier := frontier ∪ {(e', π̂)};
      reached := reached ∪ {(e', π̂)}
return  (reached, ∅)
```

A CPA+ is a special case of an interruptible CPA+: any valid CPA+ is also a valid interruptible CPA+.

We provide a composition mechanism for interruptible CPA+ similar to the composition mechanism for CPA and CPA+. A *composite interruptible CPA+* $\mathcal{C} = (\mathbb{D}^+_1, \mathbb{D}^+_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$[2] has the same components as a composite CPA+ except for $\mathbb{D}^+_1$ and $\mathbb{D}^+_2$ that are interruptible CPA+'s, and for the composite precision adjustment function: $\mathsf{prec} : E_\times \times \Pi_\times \times 2^{E_\times \times \Pi_\times} \to (E_\times \times \Pi_\times) \cup \{\mathit{interrupt}\}$, where $E_\times$ is the Cartesian product of the sets of abstract states of $\mathbb{D}^+_1$ and $\mathbb{D}^+_2$. A composite interruptible CPA+ $\mathcal{C} = (\mathbb{D}^+_1, \mathbb{D}^+_2, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$ corresponds to the interruptible CPA+ $\mathbb{D}^+ = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$ where $D_\times$ is the domain based on the direct product of the domains of $\mathbb{D}^+_1$ and $\mathbb{D}^+_2$. It can be proved in the same way as for composite CPA+, that the resulting interruptible CPA+ fulfills the requirements of an interruptible CPA+.

[2] We extend this notation to any finite number of $\mathbb{D}^+_i$.

We adapt the reachability algorithm so that the analysis is interrupted when the precision adjustment function returns *interrupt*. To allow the analysis to resume from intermediate results, the algorithm is seen as a transformation of a reached set and its frontier (both are sets of states with precision). The reached set represents the reachable states computed so far, and its frontier is a subset of the reached set that contains those abstract states whose successors have not yet been added to the reached set. Algorithm *PartialCPA+* (5.2) takes as input an interruptible CPA+, a program, an initial set of abstract states and its frontier. The algorithm attempts to iteratively compute reachable states by computing successors of state in the frontier until either the frontier becomes empty (fixpoint), or the precision adjustment function returned value *interrupt*. The reached set and its frontier are iteratively updated as they were updated in the reachability algorithm for a CPA+. The analysis returns a larger set of reached states with precision and its frontier. We can distinguish between the two outcomes of the algorithm by considering the returned frontier: in case the analysis was interrupted, the frontier is nonempty.

Theorem 5.2. *Given an interruptible CPA+ $\mathbb{D}^+$, a program P, an initial set of abstract states with precision R_0 and its frontier $F_0 \subseteq R_0$,*
if PartialCPA+$(\mathbb{D}^+, P, R_0, F_0)$ terminates and returns (R', F'), we have:
(1) $[\![\sigma(R_0)]\!] \subseteq [\![\sigma(R')]\!]$ and
(2) if $Succ(P, [\![\sigma(R_0 \setminus F_0)]\!]) \subseteq [\![\sigma(R_0)]\!]$, then $Succ(P, [\![\sigma(R' \setminus F')]\!]) \subseteq [\![\sigma(R')]\!]$, where $\sigma(R)$ denotes $\{e \mid \exists \pi : (e, \pi) \in R\}$.

Proof. We do not detail the proof as it follows exactly the same structure as the proof of loop invariants of Algorithm *CPA+* (Theorem 4.1). The loop bodies have different behavior only when the precision adjustment function returns value *interrupt*. In that case, property (1) holds because no operation removes concrete states from the set *reached* (as proved before); and property (2) holds because the element popped from *frontier* is re-added to *frontier* when prec returns *interrupt*. □

5.4.2 Interruptible CPA+ for Path Analysis

We extend the CPA+ $\mathbb{L}^+$, which tracks the syntactical reachability of program locations, to an interruptible CPA+ $\mathbb{L}_i^+$ that tracks the program path that lead to the current location. The precision of the analysis is a set of locations, at which the analysis must be interrupted. The *interruptible CPA+ for path*

analysis $\mathbb{L}_i^+ = (D_{\mathbb{L}_i^+}, \Pi_{\mathbb{L}_i^+}, \rightsquigarrow_{\mathbb{L}_i^+}, \mathsf{merge}_{\mathbb{L}_i^+}, \mathsf{stop}_{\mathbb{L}_i^+}, \mathsf{prec}_{\mathbb{L}_i^+})$, consists of the following components:

1. The domain $D_{\mathbb{L}_i^+}$ is based on the flat lattice for the set L^* of sequences of program locations: $D_{\mathbb{L}_i^+} = (C, \mathcal{E}, [\![\cdot]\!])$, with $\mathcal{E} = (L^* \cup \{\top, \bot\}, \top, \bot, \sqsubseteq, \sqcup)$, $\bot \sqsubseteq p \sqsubseteq \top$ and $l \neq p' \Rightarrow p \not\sqsubseteq p'$ for all elements $p, p' \in L^*$ (this implies $\bot \sqcup p = p, \top \sqcup p = \top, p \sqcup p' = \top$ for all paths $p, p' \in L^*, p \neq p'$). For a nonempty word p over locations L, we denote by $last(p)$ the last location of the word. An abstract element (i.e., a path) represents the set of concrete states whose location is the last location in the path: $[\![\top]\!] = C$, $[\![\bot]\!] = \emptyset$, and for all $p \in L^*$: $[\![p]\!] = \{(l, \cdot, \cdot) \in C \mid last(p) = l\}$.
2. The precision of the analysis is a set of location: $\Pi_{\mathbb{L}_i^+} = 2^L$.
3. The transfer relation $\rightsquigarrow_{\mathbb{L}_i^+}$ has the transfer $p \overset{g}{\rightsquigarrow}_{\mathbb{L}_i^+} (p', \pi)$ if $g = (l, \cdot, l')$ and $last(p) = l$ and $p' = pl'$, and the transfer $\top \overset{g}{\rightsquigarrow}_{\mathbb{L}_i^+} (\top, \pi)$ for every CFA edge g and $\pi \in \Pi_{\mathbb{L}_i^+}$.
4. The merge operator does not combine elements when control flow meets: $\mathsf{merge}_{\mathbb{L}_i^+}(p, p', \pi) = p'$.
5. The termination check considers only the last location of the path: $\mathsf{stop}_{\mathbb{L}_i^+}(p, R, \pi) = (\exists p' \in R : last(p) = last(p'))$.
   Note that the termination is sound because of the definition of the concretization function.
6. The precision adjustment function returns *interrupt* when the path ends at a location in the precision; and otherwise, the precision is not adjusted:
   $\mathsf{prec}_{\mathbb{L}^+}(p, \pi, R) = \begin{cases} interrupt & \text{if } last(p) \in \pi \\ (p, \pi) & \text{otherwise} \end{cases}$

Typically, the precision of the analysis contains the error location. The analysis terminates when the error location is reached, and at that point the abstract state contains the program path that led to the error location. We can use the abstract element to determine whether the error path is feasible. Note that by modifying the precision adjustment function, the analysis can be interrupted when the path satisfies an arbitrary property, e.g., the path violates a certain protocol.

5.4.3 Interruptible CPA+ for Explicit Heap Analysis

We extend the CPA+ $\mathbb{H}^+$, which tracks explicitly the content of the heap, to an interruptible CPA+ $\mathbb{H}_i^+$. The precision of the analysis is a set of pointers whose

data structures are tracked explicitly, and the analysis is interrupted when the depth of a data structure in the explicit heap exceeds the threshold k. (In contrast, $\mathbb{H}^+$ discards parts of the explicit heap when a tracked data structures becomes too deep.) The *interruptible CPA+ for explicit heap analysis* $\mathbb{H}^+_i = (D_{\mathbb{H}^+}, \Pi_{\mathbb{H}^+}, \rightsquigarrow_{\mathbb{H}^+}, \mathsf{merge}_{\mathbb{H}^+}, \mathsf{stop}_{\mathbb{H}^+}, \mathsf{prec}_{\mathbb{H}^+_i})$, has the same components as $\mathbb{H}^+$ except for the precision adjustment function:

6. The precision adjustment function returns *interrupt* when the threshold is hit, and otherwise the precision is never adjusted:
$$\mathsf{prec}_{\mathbb{H}^+_i}(H, \Theta, R) = \begin{cases} \mathit{interrupt} & \text{if } H = (v, h) \wedge \exists p \in \Theta : v(p) \neq \top \wedge \mathit{depth}(H, v(p)) \geq k \\ (H, \pi) & \text{otherwise} \end{cases}$$

5.4.4 Composite Interruptible CPA+ for Path, Shape, and Explicit Heap Analysis

We present a composite interruptible CPA+ that combines the interruptible CPA+ $\mathbb{L}^+_i$ for path analysis, the CPA+ $\mathbb{S}^+$ for shape analysis (Section 4.4.3), and the CPA+ $\mathbb{H}^+_i$ for explicit heap analysis. The composite analysis is interrupted if one of the following condition is satisfied: (1) a composite abstract state representing some concrete state whose location is an error location is encountered, or (2) the depth of an explicit heap has exceeded a threshold. The two cases correspond to situations where we want to change the precision and update the reached states before proceeding further with the analysis: in case (1) we stop in order to check the feasibility of the error path (potentially leading to a refinement); in case (2) we stop in order to select meaningful shape class generators based on the explicit heap. In addition, the composite analysis supports local precisions for the shape analysis and the explicit heap analysis (similarly to the example given for predicate abstraction in Section 4.4.2). The composite interruptible CPA+ $\mathcal{C}_{sr} = (\mathbb{L}^+_i, \mathbb{S}^+, \mathbb{H}^+_i, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$, is based on the interruptible CPA+ $\mathbb{L}^+_i$, the CPA+ $\mathbb{S}^+$, the interruptible CPA+ $\mathbb{H}^+_i$, and the following components:

1. The composite precision is defined by:
$$\Pi_\times = \Pi_{\mathbb{L}^+_i} \times \{\widehat{\Psi} : L \to \Pi_{\mathbb{S}^+}\} \times \{\Theta : L \to \Pi_{\mathbb{H}^+_i}\}$$
The initial precision is $(\{l_{err}\}, \widehat{\Psi}_0, \Theta_0)$, where l_{err} denote the error location, and for every l in L, $\widehat{\Psi}_0(l)$ is a trivial shape abstraction specification

containing for every structure type an empty tracking definition and the smallest shape class generator m_0, and $\Theta_0(l) = \emptyset$.

2. The transfer relation $\rightsquigarrow_\times$ has the transfer $(p, G, H) \overset{g}{\rightsquigarrow}_\times ((p', G', H'), (\pi_p, \pi_G, \pi_H))$ for $g = (l, op, l')$ if $p \overset{g}{\rightsquigarrow}_{\mathbb{L}_i^+} (p', \pi_p)$ and $G \overset{g}{\rightsquigarrow}_{\mathbb{S}^+} (G', \pi_G(l'))$ and $H \overset{g}{\rightsquigarrow}_{\mathbb{H}_i^+} (H', \pi_H(l'))$.

3. The composite merge operator never merges:

$$\mathsf{merge}_\times((p, G, H), (p', G', H'), \pi_\times) = (p', G', H')$$

4. The composite termination check considers states separately:

$$\begin{aligned} &\mathsf{stop}_\times((p, G, H), R, (\pi_p, \pi_G, \pi_H)) \\ &= \exists (p', G', H') \in R : \begin{pmatrix} & \mathsf{stop}_{\mathbb{L}_i^+}(p, \{p'\}, \pi_p) \\ \wedge & \mathsf{stop}_{\mathbb{S}^+}(G, \{G'\}, \pi_G) \\ \wedge & \mathsf{stop}_{\mathbb{H}_i^+}(H, \{H'\}, \pi_H) \end{pmatrix} \end{aligned}$$

5. The composite precision adjustment function interrupts the analysis if the precision adjustment function of the path analysis or of the explicit heap analysis returns *interrupt* (the precision adjustment function of the shape analysis never returns *interrupt*):

$$\begin{aligned} &\mathsf{prec}_\times((p, G, H), (\pi_p, \pi_G, \pi_H), R) \\ &= \begin{cases} \mathit{interrupt} & \text{if } r_p = \mathit{interrupt} \text{ or } r_H = \mathit{interrupt} \\ ((p', G', H'), (\pi'_p, \pi'_G, \pi'_H)) & \text{if } r_p = (p', \pi'_p) \text{ and } r_G = (G', \pi'_G) \\ & \text{and } r_H = (H', \pi'_H) \end{cases} \\ &\text{where } r_p = \mathsf{prec}_{\mathbb{L}_i^+}(p, \pi_p, \{(p'', \pi''_p) \mid ((p, \cdot, \cdot), (\pi''_p, \cdot, \cdot)) \in R\}) \\ &\text{and } r_G = \mathsf{prec}_{\mathbb{S}^+}(G, \pi_G, \{(G'', \pi''_G) \mid ((\cdot, G'', \cdot), (\cdot, \pi''_G, \cdot)) \in R\}) \\ &\text{and } r_H = \mathsf{prec}_{\mathbb{H}_i^+}(H, \pi_H, \{(H'', \pi''_H) \mid ((\cdot, \cdot, H''), (\cdot, \cdot, \pi''_H)) \in R\}) \end{aligned}$$

The interruptible CPA+ corresponding to the composite interruptible CPA+ $\mathcal{C}_{sr}$ is referred in the following as $\mathbb{D}_{sr}^+$. Note that the path analysis allows to store the same information in the set of reached states as the information that is found in an abstract reachability tree (ART), a data structure often used in software model checkers [Beyer et al. 2007]: the path-analysis state encodes the path from the root of the ART to the current node.

Algorithm 5.3 $ModelCheck(P, l_{err}, M)$

Input: a program P, an error location l_{err} of P,
a lattice M of SCGs with finite height
Output: either a set of abstract states that overapproximates the set of reachable states to witness safety,
or an error path to witness the existence of a feasible error path
Variables: an explicit-heap precision Θ, a shape-abstraction specification $\widehat{\Psi}$,
an ART A, a mapping E from types to sets of enabled SCGs

```
for each pointer type σ in P do
  E(σ) := M
for each location l of P do
  Ψ̂(l) := Ψ̂_∅;   Θ(l) := ∅;
reached := {((l_0, G_0, H_0), ({l_err}, Ψ̂, Θ))};
frontier := {((l_0, G_0, H_0), ({l_err}, Ψ̂, Θ))};
while true do
  (reached, frontier) := PartialCPA+(D⁺_sr, P, reached, frontier);
  if frontier = ∅ then
    print "Yes. The program is safe. Certificate:" σ(reached); stop;
  else if there is (p, G, H) in frontier s.t. last(p) = l_err then
    let t be the program path for the sequence of locations p
    (·, Γ) := Con(t);
    if ⋀_{1≤i≤|p|} Γ(p) ⊢ false; then // t is infeasible due to a too coarse precision
      (reached, frontier, E) := Refine(t, reached, frontier, M, E);
    else // t is feasible; the error is really reachable
      print "No. The program is unsafe. Counterexample path:" t; stop;
  else // threshold exceeded, switch off explicit tracking
    (reached, frontier, E) := Abstract(reached, frontier, M, E);
```

5.4.5 Model-Checking Algorithm (*ModelCheck*)

Our analysis algorithm is based on the composite interruptible CPA+ described in the previous section. Algorithm *PartialCPA+* is called iteratively. While the partial reachability analysis returns a nonempty frontier (i.e., the set of reached states is not a complete overapproximation of the reachable states), we either report an error path if a feasible error path has been found, and otherwise, we adjust the precision and restart the analysis with a different initial precision.

Algorithm *ModelCheck* (Algorithm 5.3) takes as input a program P, an error location l_{err} of P, and a lattice M of SCGs. The algorithm tries to prove (or disprove) that l_{err} is not reachable in any concrete program execution. It maintains a set of reachable states with precision and its frontier. In addition, it maintains a mapping from program types to sets of enabled SCGs (subsets of M). Only enabled SCGs are considered during refinement. In a first step, the algorithm initializes the set of reachable states with precision and its frontier: initially, both contains an abstract state representing all states whose location is the initial location of P; and the initial composite precision consists of the

singleton $\{l_{err}\}$, a trivial shape precision and an empty set of tracked pointers at every location. All SCGs are initially enabled for any type. Then a check-refine loop is executed until either the program is declared safe or a feasible path to the error location is found.

In every iteration, we first call procedure *PartialCPA+* (Algorithm 5.2) with the interruptible CPA+ $\mathbb{D}^{+}_{sr}$ (defined in Section 5.4.4) to extend, for the given program P, the given set of states with precision R towards a set of states that is closed under abstract successors (i.e., the frontier is empty) starting the exploration from its frontier F. Procedure *PartialCPA+* stops if one of the following conditions is fulfilled:

(a) The reachability analysis encounters an abstract state whose path ends at the error location. The error abstract state is in the returned frontier.

(b) The reachability analysis reaches a fixpoint, i.e., the frontier becomes empty, and no state in the reached set represents error states.

(c) The reachability analysis determines that the last computed abstract state has an explicit heap suitable for abstraction. The frontier is nonempty but the reached set contains no error state.

Algorithm *ModelCheck* distinguishes the different outcomes of *PartialCPA+* based on the frontier. (1) If the frontier is empty, then the overall algorithm can stop and report that the program is safe. (2) If the frontier is nonempty but contains no error state, then the threshold for the explicit heap analysis was reached; in other words, the explicit heap analysis has collected enough information to guide the refinement of the shape-abstraction specification. Procedure *Abstract* is called to analyze explicit heaps in order to restrict enabled SCGs, to refine SCGs in the shape-abstraction specification, and to replace explicit heaps in the reached set by shape graphs. (3) If the frontier is nonempty and contains a state whose path ends at the error location, then either (3a) the path is feasible, and the overall algorithm can stop and report an error, or (3b) the path is infeasible, and procedure *Refine* will try to find a more suitable precision because the path was encountered due to a too coarse precision. Procedure *Refine* may fail due to the absence of a suitable, fine-enough SCG in the lattice of SCGs. Note that Algorithm *ModelCheck* may not terminate, in case it produces finer and finer precisions to rule out longer and longer infeasible error paths.

5.4.6 Algorithm for Abstraction from Explicit Heaps (*Abstract*)

Algorithm 5.4 *Abstract*(R, F, M, E)

Input: a set of abstract states with precision R and its frontier F,
a set M of SCGs, and a type-to-SCGs mapping E
Output: a set of abstract states with precision and its frontier and
a type-to-SCGs mapping E

```
let π' = ({l_err}, Ψ̂, Θ) be the precision of states in F
refined := ∅
for each abstract state ((p, G, H), ·) in F with H = (v, h) do
  let l := last(p)
  for each pointer p ∈ Θ(l) s.t. depth(H, v(p)) > k do
    let σ be T(p);
    refined := refined ∪ {σ}
    choose (σ, m, D) ∈ Ψ̂(l)
    // evaluate invariants on explicit heap, and update precisions
    E(σ) := E(σ) ∩ SCGsFromExplicit(H, p)
    let m' be the coarsest SCG in E(σ)
    replace (σ, m, D) by (σ, m', D) in Ψ̂(l)
    // switch to shape analysis mode
    remove all x from Θ(l) s.t. T(x) = T(p)
// remove explicit heaps, and update shape graphs and precision
π' = ({l_err}, Ψ̂, Θ)
Y := {x ∈ X_ptr | T(x) ∈ refined}
for each abstract state ((l, G, H), π) in F do
  let G' := G ⊔ HeapToShape(H, Y, Ψ̂(l))
  let H' := ForgetPtrs(H, Y)
  replace ((l, G, H), π) by ((l, G', H'), π') in R and F
return (R, F, E)
```

When the explicit heap analysis has generated sufficiently large explicit heaps, Algorithm *Abstract* (Algorithm 5.4) is called to extract information from explicit heaps in order to choose a suitable SCG and to abstract explicit heaps to shape graphs. The algorithm takes as input a set of abstract states with precision R and its frontier F, a lattice of SCGs, and a mapping E from types to sets of enabled SCGs. Upon termination, the algorithm returns the updated set of states with precision and its frontier, and the updated mapping.

The algorithm first determines all pointers that point into a data structure whose depth exceeds the threshold k. Function *SCGsFromExplicit* analyzes an explicit heap and returns all relevant SCGs: every SCG is annotated with a set of invariants that must be fulfilled by explicit heaps for the SCG to be relevant (e.g., all SCGs generating instrumentation predicates for trees are annotated with the tree-ness invariant). For each SCG m, function *SCGsFromExplicit* evaluates the invariants of m on explicit heap H, and if all those invariants are fulfilled, the function enables m for its structure type. Then the precision is updated: pointer p and all other pointers of the same type are removed from the explicit-heap precision, and we refine the SCG of the chosen shape type to

be the coarsest enabled SCG for the structure type. After the refinement of the SCG, we erase the portion of the explicit heap whose pointers are not tracked anymore (function *ForgetPtrs*), and augment the corresponding shape region by the result of abstracting the erased portion of the explicit heap to shape graphs (function *HeapToShape*). Function *ForgetPtrs* takes as input an explicit heap H and a set of pointers P and returns an explicit heap where the information about any pointers in P and data structures pointed to by P are forgotten. Recall that for an explicit heap and a pointer variable p, $\mathit{ReachAddr}(H, p)$ denotes the set of addresses reachable from p. Formally, we have:

$\mathit{ForgetPtrs}((v, h), Y) = (v', h')$ if
(1) for every x in X_{int}, $v'(x) = v(x)$, and
(2) for every x in X_{ptr}, if $x \in Y$ then $v'(x) = \top$ else $v'(x) = v(x)$, and
(3) $h' = \{(a, F) \in h \mid a \notin \bigcup_{y \in Y} \mathit{ReachAddr}(H, v(y))\}$

The result of *HeapToShape* is a shape region with a single shape graph for each shape class that results from applying the newly refined SCG to the current tracking definitions, for those pointers erased from the explicit heap.

Example. Suppose that the analysis has computed the explicit heap graph that we depicted in Figure 5.2(a) on page 110 and that the threshold is 3. Figure 5.2(a) graphically depicts an explicit heap $H = (v, h)$ with $v = \{\mathit{l1} \mapsto 1\}$ and

$$\begin{aligned} h = \{ \quad & 1 \mapsto \{\mathit{data} \mapsto \top, \mathit{succ} \mapsto 2, \mathit{prev} \mapsto 0\}, \\ & 2 \mapsto \{\mathit{data} \mapsto \top, \mathit{succ} \mapsto 3, \mathit{prev} \mapsto 1\}, \\ & 3 \mapsto \{\mathit{data} \mapsto \top, \mathit{succ} \mapsto 4, \mathit{prev} \mapsto 2\}, \\ & 4 \mapsto \{\mathit{data} \mapsto \top, \mathit{succ} \mapsto 5, \mathit{prev} \mapsto 3\}, \\ & 5 \mapsto \{\mathit{data} \mapsto \top, \mathit{succ} \mapsto 0, \mathit{prev} \mapsto 4\} \quad \} \end{aligned}$$

The depth of the heap starting from the address stored in $\mathit{l1}$ is $\mathit{depth}(H, 1) = 5$, which exceeds the threshold. The set of SCGs for doubly-linked lists is included in $\mathit{SCGsFromExplicit}(H, \{\mathit{l1}\})$ because H fulfills the invariants of a valid doubly-linked list. An example of an SCG for doubly-linked lists is an SCG m_1 that generates binary predicates for fields *succ* and *pred* and the instrumentation predicates $\mathit{cancel}_{[succ,pred]}$ and $\mathit{cancel}_{[pred,succ]}$. The shape graph represented in Figure 5.2(b) is a possible abstraction of the explicit heap represented in Figure 5.2(a), i.e., for a shape abstraction specification $\widehat{\Psi}$ tracking pointer $\mathit{l1}$ and using an SCG m_1, $\mathit{HeapToShape}(H, \{\mathit{l1}\}, \widehat{\Psi})$ returns a shape region that contains the shape graph depicted in Figure 5.2(b). The result of $\mathit{ForgetPtrs}(H, \{\mathit{l1}\})$ is an empty heap $H' = (\{\mathit{l1} \mapsto \top\}, \emptyset)$. $\square$

In the next iteration of reachability, the extension of the reached set will continue from the frontier which contains abstract states with the newly computed shape graphs. Note that converting an explicit heap to a shape graph is significantly less expensive than obtaining the shape graph via abstract post computations, and is similar to the precision adjustment function of the composite CPA+ that combines an explicit-heap analysis with a shape analysis (Section 4.4.6).

Theorem 5.3. *Given a program P, a set of abstract states with precision R and its frontier $F \subseteq R$, a lattice M of SCGs and a type-to-SCGs mapping E, $\mathit{Abstract}(R, F, M, E)$ terminates and returns (R', F', E') such that:*
(1) $[\![\sigma(R)]\!] \subseteq [\![\sigma(R')]\!]$ and
(2) if $\mathit{Succ}(P, [\![\sigma(R \setminus F]\!])) \subseteq [\![\sigma(R)]\!]$, then $\mathit{Succ}(P, [\![\sigma(R' \setminus F')]\!]) \subseteq [\![\sigma(R')]\!]$.

Proof. The only operation that modifies abstract states is the abstraction from explicit heaps to shape graphs. The theorem follows immediately from the definitions of *ForgetPtrs* and *HeapToShape*: we have $[\![H]\!] \subseteq [\![\mathit{ForgetPtrs}(H, Y)]\!]$ and $[\![H]\!] \subseteq [\![\mathit{HeapToShape}(H, Y, \widehat{\Psi}(l))]\!]$. □

5.4.7 Algorithm for Shape Refinement (*Refine*)

When an infeasible error path is found, it is due to the shape precision being too coarse. Algorithm *Refine* tries to produce a finer precision and to use it to refine the reached set such that an error state for the same path does not occur anymore when the reachability analysis is called on the refined reached set. Algorithm *Refine* (Algorithm 5.5) takes as input a set R of reached states with precision and its frontier F, a lattice of SCGs, and a mapping from types to sets of enabled SCGs. The algorithm assumes that there is one state (p, G, H) in F whose path p ends at the error location and is infeasible. Upon termination, a refined set of states with (refined) precision and its frontier, and a (possibly updated) mapping from types to sets of enabled SCGs are returned.

The first step of the algorithm analyzes the infeasible error path. We compute the (inductive) interpolants of the (unsatisfiable) path formula corresponding to the path from the root to node n, for every location on the path (*ExtractInterpolants*). We use the interpolants to check whether we can find new pointers or field assertions to track by analyzing atoms occurring in interpolants. If we find a pointer that we have to track, we add it to the set of tracked separating pointers, and add all its aliases to the set of tracked pointers. If it is the first time we encounter a pointer, we need to know which kind of data structure

Algorithm 5.5 *Refine*(t, R, F, M, E)

Input: a program path t, a set of abstract states with precision R and its frontier F, a set M of SCGs, and a type-to-SCGs mapping E
Output: a set of abstract states with precision and its frontier and a type-to-SCGs mapping E
Variables: an interpolant map Π

```
let ({l_err}, Ψ̂, Θ) be the precision of the states in F;
Π := ExtractInterpolants(t, Γ);
for i := 1 to |t| do
  choose (σ, m, D) from Ψ̂(l_i), with D = (T, T_s, P)
  // Step 1: Refine the tracking definitions
  for each atom φ ∈ Π(l_i) do
    if some pointer p occurs in φ, and type(p) matches σ then
      add p and all elements of alias(p) to D.T
      add p to D.T_s
      if pointer p is dereferenced in φ then
        add to D.P the field assertion corresponding to φ
  // Step 2: Start explicit heap analysis or refine the SCG
  for each pointer p in D.T do
    if p ∉ Θ(l_i) and m = m_0 then
      // p was not analyzed before, switch to explicit heap analysis mode
      add p to Θ(l_i)
    if p ∉ Θ(l_i) and m ≠ m_0 then
      // in shape analysis mode: binary-search refinement
      m' := FineTune(t, m, E(σ))
      if m = m' then // the binary search cannot refine; extend the search
        add to E(σ) every m'' ∈ M s.t. m ⋢ m''
        m' := FineTune(t, m, E(σ))
      replace (σ, m, D) by (σ, m', D) in Ψ̂(l_i)
    if Θ(l_i) or Ψ̂(l_i) was changed then
      remove from R and F all abstract states (p, ·, ·) such that l_i occurs in p;
      add to F all abstract states (p, ·, ·) ∈ R s.t. last(p) is a predecessor of l_i;
if Ψ̂ and Θ did not change then
  print "Refinement failed on path:" t; stop;
let R' = {((p, G, H), ({l_err}, Ψ̂, Θ)) | ((p, G, H), ·) ∈ R and last(p) ≠ l_err};
let F' = {((p, G, H), ({l_err}, Ψ̂, Θ)) | ((p, G, H), ·) ∈ F and last(p) ≠ l_err};
return (R', F', E)
```

it is pointing to in order to enable only a subset of the SCGs in M. To discover this information, we cannot rely exclusively on syntactical type information. For example, the types for doubly-linked lists and binary trees (without parent pointers) have the same syntactical structure. We enable an explicit heap analysis of the data structure by adding the pointer to the precision of the explicit heap analysis, and the SCG is the trivial SCG m_0. If we considered the pointer before, then the explicit analysis was switched on, and we refined the SCG to a non-trivial SCG. In this case, the explicit heap analysis need not be run again because it will not provide new information. Instead, we decide to fine-tune the SCG by using a binary-search-like exploration of the lattice of enabled SCGs. If the fine-tuning fails to yield a finer SCG, it may still be the case that there exists a fine-enough SCG in the lattice of all SCGs that is prevented from being found because the explicit heap analysis over-restricted the set of enabled SCGs. In this case, we extend the set of enabled SCGs to include all SCGs from M that are not coarser than the current SCG.

Procedure *FineTune* takes as input an infeasible program path t, the current SCG m and a lattice M of SCGs. The procedure searches for the coarsest SCG m' such that m' *rules out* path t, i.e., the abstract strongest postcondition of the program path represents no states when SCG m is replaced by m' in the shape-abstraction specification. Note that we only compute shape regions along the given path t at this point, not along any other program path. To make the search more efficient, we try to prune in each iteration approximately half of the candidate SCGs. Because of the monotonicity of SCGs, if a given SCG cannot rule out t, then no coarser SCG can. The algorithm maintains a set C of candidates. The set C is initialized with all SCGs in M that are finer than m. We repeat the following steps until no more SCGs can be removed from C. We select a subset S of SCGs as small as possible such that the set of SCGs coarser than some SCG in S contains as many elements as the set of SCGs finer than some SCG in S. If no SCG in S rules out t, we remove from C all SCGs coarser or equal to a SCG in S; otherwise, we keep in C only those SCGs that are coarser or equal to some SCG in S that rules out t. When the loop terminates, if $C = \emptyset$, then the fine-tuning failed and we return m; otherwise, we choose one SCG m' in C that generates the fewest predicates when applied to the current tracking definition, and return m'.

Theorem 5.4. *Given a program P, a program path t, a set of abstract states with precision R and its frontier $F \subseteq R$, a lattice M of SCGs and a type-to-SCGs mapping E, if Refine(t, R, F, M, E) terminates normally and returns (R', F', E') then:*

(1) $[\![\sigma(R)]\!] \cap \{(l_{err}, \cdot, \cdot) \in C\} = \emptyset$ *and*
(2) if $Succ(P, [\![\sigma(R \setminus F]\!])) \subseteq [\![\sigma(R)]\!]$*, then* $Succ(P, [\![\sigma(R' \setminus F')]\!]) \subseteq [\![\sigma(R')]\!]$.

Proof. Every time a refinement occurs at location l, we remove all states whose path contains l and add to the frontier the abstract states whose location is a predecessor of l. As a consequence, property (1) holds because the error states are removed, and property (2) holds because we add the predecessor states to the frontier. □

Theorem 5.5. *Given a program* P*, a location* l_{err}*, and a lattice* M *of SCGs:*

(1) if $ModelCheck(P, l_{err}, M)$ *terminates and report a safety certificate* R*, then* $Reach(P, \{(l_0, \cdot, \cdot) \in C\}) \subseteq [\![R]\!]$ *and* $[\![R]\!] \cap \{(l_{err}, \cdot, \cdot) \in C\} = \emptyset$*; and*

(2) if $ModelCheck(P, l_{err}, M)$ *terminates and report an error path* t*, then* t *is a feasible path containing* l_{err}.

Proof. The theorem follows directly from Theorems 5.1, 5.2, 5.3 and 5.4. □

Note that we do not guarantee that the analysis terminate. There are programs that result in an infinite sequence of refinements, in case the interpolation-based counterexample analysis discovers new field predicates for longer program paths. Predicate-abstraction based model-checkers share the same caveat. Moreover, the overall algorithm may fail if the library M of SCGs do not contain suitable SCGs. It is a fundamental shortcoming of our approach. In practice, if the library is rich enough, the problem does not occur for a large class of programs.

5.5 Implementation

The algorithm presented in this paper is implemented as an extension to BLAST. The implementation builds on our extension to BLAST to support the CPA and CPA+ frameworks. TVLA is integrated into BLAST as a particular implementation of a shape-analysis module. In addition to the shape analysis discussed in this paper, our implementation supports nullary predicate abstraction and its refinement based on interpolants, similarly to lazy shape analysis [Beyer et al. 2006]. In the following we detail two important aspects of our implementation: first we discuss how the library of shape class generators is implemented and the default library that we provide with the tool; second we discuss alternative methods to select appropriate SCGs based on information given by the user (programmer or verification engineer).

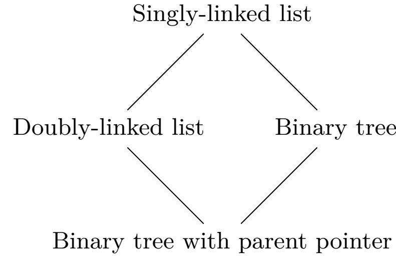


Figure 5.5: Hierarchy of data structures. Finer data structures are drawn lower.

5.5.1 Library of Shape-Class Generators

Our algorithm relies on a pre-defined library of SCGs. In order to decouple the specification of the library of SCGs from the implementation of the overall verification algorithm, we make use of a domain-specific language (DSL) for specifying SCGs. Code is generated from the DSL, compiled as a library, and linked against the verification tool. As a result, the task of supporting new data structures or to enhance existing SCGs is easy and decoupled from the analysis and refinement engine.

The DSL is structured in entries. Each entry defines SCGs to analyze one type of data structure (e.g., our implementation contains one DSL entry for singly-linked lists, one for doubly-linked lists, one for binary trees, and one for binary trees with parent pointers). Every entry in the DSL consists of three parts:

1. the maximal set of core and instrumentation predicates that can be used during the analysis (parametric in the tracking definition),

2. the definition of transfer functions to describe how predicate valuations are modified by abstract post computations corresponding to program operations, and

3. a set of invariants on explicit heaps such that if the invariants hold on sample explicit heaps then the SCGs corresponding to the entry can be enabled (used in the implementation of *SCGsFromExplicit*).

A refinement relation among DSL entries is specified separately. The refinement relation allows to express that a data structure can also be seen as a simpler data structure. For instance, a doubly-linked list can also be seen as a singly-linked list (if we ignore one of the pointer fields).

The set of predicates of a DSL entry (part 1) defines the most refined SCG $m_\top$, i.e., it contains all predicates that could be tracked for the data structure. However, not every predicate may be necessary to verify a program. Therefore, one entry spans a lattice of SCGs rather than a single SCG. A shape predicate p_1 depends on a shape predicate p_2 if p_1 is an instrumentation predicate and p_2 occurs in the defining formula of p_1. A DSL entry spans the lattice of all SCGs m' such that (a) $m' \sqsubseteq m_\top$ and (b) for every tracking definition D, the shape class $\mathbb{S} = m'(D)$ is such that if p_1 is in $\mathbb{S}$ and predicate p_1 depends on predicate p_2, then p_2 is in $\mathbb{S}$. The task of selecting an SCG within the lattice is the responsibility of the *FineTune* function used in the *Refine* algorithm.

Table 5.1 lists the four DSL entries that we have implemented. Each row of the table represents one DSL entry. For each entry, we give the name of the data structure for which the entry is relevant; how many recursive fields are tracked using binary core predicates; the instrumentation predicates defined by the entry; and the invariants that explicit heaps should satisfy. The refinement relation among entries is shown in Figure 5.5.

The instrumentation predicates that our implementation support are well-known instrumentation predicates found in the literature [Sagiv et al. 2002]. We detail thereafter the meaning of the instrumentation predicates. In the following, p denotes any pointer variable; f, f_1, f_2 any recursive field, and v, v_1, v_2 any shape graph node.

– Unary reachability *reachability*$[p, f](v)$ holds if elements represented by node v are reachable from the element pointed to by p, by following zero, one or more f-fields.

– Binary reachability *reachability2*$[f](v_1, v_2)$ holds if elements represented by node v_2 are all reachable from any element represented by node v_1 by following zero, one or more f-fields.

– Unary cyclicity *cyclicity*$[f](v)$ holds if the elements represented by v are reachable from themselves by following one or more f-fields.

– Unary cancellation *cancel*$[f_1, f_2](v)$ holds if, when field f_1 of node v points to node v', then field f_2 of v' points back to v.

– Binary down points-to *down*$[f_1, f_2](v_1, v_2)$ holds if field f_1 or field f_2 of v_1 points to v_2.

Data structure	Rec. fields	Instrumentation predicates / Invariants
Singly-linked list (sll)	1 (`n`)	$\mathit{reachability}[p,\texttt{n}](v)$ for each pointer p $\mathit{cyclicity}[\texttt{n}](v)$
		—
Doubly-linked list (dll)	2 (`s`, `p`)	$\mathit{reachability2}[f](v_1,v_2)$ for each rec. field f $\mathit{reachability}[p,f](v)$ for each pointer p, rec. field f $\mathit{cyclicity}[f](v)$ for each rec. field f $\mathit{cancel}[\texttt{s},\texttt{p}]$, $\mathit{cancel}[\texttt{p},\texttt{s}]$
		$\forall p_1,p_2:\ (p_1\texttt{->s==}p_2 \rightarrow p_2\texttt{->p==}p_1) \wedge (p_1\texttt{->p==}p_2 \rightarrow p_2\texttt{->s==}p_1)$
Binary tree without parent pointer (tree)	2 (`l`, `r`)	$\mathit{down}[\texttt{l},\texttt{r}](v_1,v_2)$, $\mathit{down}^*[\texttt{l},\texttt{r}](v_1,v_2)$ $\mathit{downReachability}[p,\texttt{l},\texttt{r}](v)$ for each pointer p $\mathit{downCyclicity}[\texttt{l},\texttt{r}](v)$
		$\forall p:\mathit{reach}(p\texttt{->l}) \cap \mathit{reach}(p\texttt{->r}) = \emptyset$
Binary tree with parent pointer (tree+p)	3 (`l`, `r`, `p`)	$\mathit{down}[\texttt{l},\texttt{r}](v_1,v_2)$, $\mathit{down}^*[\texttt{l},\texttt{r}](v_1,v_2)$ $\mathit{downReachability}[p,\texttt{l},\texttt{r}](v)$ for each pointer p $\mathit{downCyclicity}[\texttt{l},\texttt{r}](v)$ $\mathit{cancel}[\texttt{l},\texttt{p}]$, $\mathit{cancel}[\texttt{r},\texttt{p}]$
		$\forall p:\ \mathit{reach}(p\texttt{->l}) \cap \mathit{reach}(p\texttt{->r}) = \emptyset$ $\forall p_1,p_2:\ (p_1\texttt{->l==}p_2 \rightarrow p_2\texttt{->p==}p_1) \wedge (p_1\texttt{->r==}p_2 \rightarrow p_2\texttt{->p==}p_1)$

Table 5.1: Library of SCG used in the experiments. For each data structure (DSL entry), we give the set of instrumentation predicates that can be enabled, and invariants that must hold in explicit heaps

- Unary downward reachability *downReachability*$[p, f_1, f_2](v)$ holds if elements represented by node v are reachable from the element pointed to by p, by following zero, one or more f_1 or f_2-fields.
- Unary downward cyclicity *downCyclicity*$[f_1, f_2](v)$ holds if the elements represented by v are reachable from themselves by following one or more f_1 or f_2-fields.

For every data structure, we (maximally) track the following instrumentation predicates:

- for singly-linked lists: reachability (unary) and cyclicity (unary);
- for doubly-linked lists: reachability (unary and binary) separately for each pointers, cyclicity (unary), and cancellation (unary);
- for trees (with and without parent pointers): down points-to and its transitive closure ($down^*$, binary), downward reachability (unary), and downward cyclicity (binary), and in addition, for trees with a parent pointer, cancellation for left and parent, and right and parent (unary).

5.5.2 Manual Annotations of Data Types

When writing a program, programmers generally know the kind of data structure a pointer of a given type points to. For instance, if the programmer creates a structure type `struct tree_cell`, its intention is likely to model a tree (rather than a doubly-linked list). As a consequence, given a pointer type, the programmer would know which entries of the SCG library are relevant. Therefore, if the verification tool knew this information, it would not have to run the explicit analysis to guess which SCG to use. We designed a mechanism for the programmer (or the verification engineer) to communicate to the tool which SCGs to use via *code annotations*.

For a given structure (`struct`) type of the C program, the programmer specifies which entry from the library of SCGs to use by creating an alias of the type with a special name that the tools recognizes. More specifically, if a program type T is aliased (using `typedef`) to a name of the form `__SA__`*`entry`*`_`..., then only SCGs in the library corresponding to *`entry`* are enabled. For example, suppose the program contains the following type:

```
struct tree_node {
  tree_node *l;
```

```
    tree_node *r;
    void *data;
};
```

The programmer adds the following type alias to specify that `struct tree_node` represents a tree:

```
typedef struct tree_node __SA__tree_l_r;
```

Note that annotated code are valid C programs and the behavior of the program is not affected by the annotation. In richer programming languages, instead of using type aliases, a more elegant solution would be the use of an annotation mechanism provided by the programming language (as for example annotations in Java 1.5).

For many programs an even simpler heuristics can be used successfully to identify the kind of data structures, based on the following simple observation: a programmer is likely to give meaningful names to the types he declares. For example, a type representing a tree is likely to contain 'tree' in its name. We have implemented such a simple heuristics in our tool and it can be optionally enabled by the user.

5.6 Experimental Evaluation

Based on the implementation described in the previous section, we evaluated experimentally our approach on example programs manipulating different kind of data structures.

5.6.1 Example Programs

We evaluate our technique on code taken from the open-source C library for data-structures GDSL 1.4 [3]. We consider non-trivial low-level functions operating on doubly-linked lists and trees. Each function is inserted in client code, non-deterministically simulating valid uses of the function. The client code inputs arbitrary valid data structures to the function, and on return, checks that a given property is preserved. Both the instance generation and the property check are expressed as regular C code. In the case of property check, we cannot rely on assertions because we are limited to what can be expressed

[3] Available at `http://home.gna.org/gdsl/`

Program	Maximal SCG			With refinement				
	SCG	# instr. pred. families	Time	SCG	# instr. pred. families / max	#refinements	Annotation	Explicit
`cancel_list_link`	dll	3	10.04 s	dll	1/3	1 td, 1 scg	12.65 s	13.76 s
`cancel_list_insert_after`	dll	3	23.62 s	dll	1/3	1 td, 1 scg	24.41 s	26.82 s
`cancel_list_insert_before`	dll	3	30.90 s	dll	3/3	2 td, 2 scg	69.01 s	77.22 s
`cancel_list_remove`	dll	3	4.42 s	dll	2/3	1 td, 1 scg	28.49 s	29.05 s
`acyclic_list_link`	dll	3	11.57 s	sll	2/2	1 td, 1 scg	6.32 s	6.49 s
`acyclic_list_insert_after`	dll	3	24.21 s	sll	2/2	1 td, 1 scg	23.57 s	26.06 s
`acyclic_list_insert_before`	dll	3	34.53 s	dll	3/3	2 td, 2 scg	80.81 s	88.21 s
`acyclic_list_remove`	dll	3	4.23 s	sll	2/2	1 td, 2 scg	96.77 s	99.75 s
`bintree_rotate_left`	tree+p	5	> 9000 s	tree	2/4	3 td, 2 scg	414.28 s	521.31 s
`bintree_rotate_right`	tree+p	5	>9000 s	tree	2/4	3 td, 1 scg	419.24 s	437.30 s
`bintree_rotate_left_right`	tree+p	5	>9000 s	tree	2/4	2 td, 2 scg	7023.41 s	7401.74 s
`treep_rotate_left`	tree+p	5	>9000 s	tree+p	2/5	4 td, 2 scg	180.58 s	66.63 s
`treep_rotate_right`	tree+p	5	>9000 s	tree+p	2/5	4 td, 2 scg	402.70 s	384.19 s
`treep_rotate_left_right`	tree+p	5	>9000 s	tree+p	2/5	4 td, 2 scg	1175.14 s	1189.42 s

Table 5.2: Verification time of Blast on functions from the GDSL library, using (a) maximal SCG, or shape refinement with (b) program annotations or (c) explicit heap analysis to determine the SCG

with a C expression rather than considering more general assertion languages. The benchmarks `cancel_*` and `acyclic_*` operate on doubly-linked lists, and check, respectively, for the preservation of the structure of a doubly-linked list (i.e., the backward pointer of the node pointed to by a given node's forward pointer points back to the given node, and vice versa), and for acyclicity following forward pointers. The benchmarks `bintree_*` and `treep_*` operate on binary trees, and check, respectively, for the preservation of acyclicity following left and right pointers, and for the validity of parent pointers with respect to left and right pointers.

5.6.2 Results

All examples could be proved safe by Blast after a few refinement steps. Table 5.2 reports the execution time of Blast on a GNU/Linux machine with an Intel Core Duo 2 6700 processor and 4 GB of memory. The first part of the table reports the results with the most refined (maximal) SCG used for all pointers in the program, and therefore no refinement is needed. The first column reports the kind of data structure and the number of instrumentation predicate families used by the SCG. (We call *instrumentation predicate family* a collection of predicates capturing a particular property, e.g., all the unary instrumentation predicates for reachability form a predicate family; the unary instrumentation predicate for cyclicity forms a predicate family; etc.) The second column reports the verification time. The second part of the table reports the results when refinement is used. The first column of this part of the table reports the SCG and number of enabled instrumentation predicates families (compared to maximum). The second column reports the number of each kind of refinements: the first kind (td) corresponds to the refinement of a tracking definition (i.e., a new pointer or a new field predicate is discovered), and the second kind (scg) corresponds to the refinement of SCGs (i.e., new instrumentation predicates are introduced). The information in the first and second columns is identical for both configurations with refinement. To evaluate the impact of the explicit heap analysis on performance, we replace in one experimental setting the procedure *Abstract* by a procedure that enables the suitable set of SCGs based on our knowledge of the data structures, encoded as annotations for Blast in the code. Therefore, the third column reports verification times for the experiments when using annotations to determine the type of data structures (explicit heap analysis disabled), and the fourth column, when using the explicit heap analysis to infer the type of data structures. We run the explicit heap analysis until five

different samples of data structures containing (at least) four structure nodes are collected. In all examples, both tracking definitions and SCGs are refined. In most examples, the finest SCG is not needed (only a subset of available predicates is used). Note that for three out of four `acyclic_*` benchmarks, a shape class for singly-linked lists (considering only the forward pointer) is sufficient to prove safety.

The explicit heap analysis correctly identifies the data-structure in every example. The run time for explicit-heap based refinement is comparable to annotation-guided refinement. The variations between the two result from two sources: (1) the overhead of performing the explicit heap analysis, and (2) the abstraction from explicit heaps to shape graphs and the subsequent extension of the set of reachable states. On all examples, the explicit heap analysis accounts for a negligible fraction of the execution time. Most of the runtime is consumed by (symbolic) shape operations in Tvla. On the one hand, some shape-graph computations are saved. But on the other hand, depending on how large the set of reachable states is when *Abstract* is executed, many explicit heaps may abstract to the same shape graph, subsequently causing an overhead. Infeasible error paths may also have different lengths resulting in different interpolation and refinement timings. On small examples, the refinement contributes most of the total execution time (up to nearly 50%): most of the time is spent in the path simulations of *FineTune*. On larger examples, most of the time is spent in the final iteration of the reachability analysis, in particular, while computing abstract shape successors using Tvla. Overall, we conclude that the explicit heap analysis provides reliable information for the refinement, for a reasonable overhead.

Our refinement strategy outperforms the direct use of the most refined SCG on large examples (involving trees), because the refinement allows for the use of significantly less instrumentation predicates, compared to the most refined SCGs. On smaller examples, though, the run time can be larger if refinement is used, due to the high portion of time spent on refinement and the high number of instrumentation predicates we need, compared to the most refined case. The final reachability analysis sometimes takes significantly less time if the most refined SCG is used; one particular case is the two `list_remove` examples. The reason is that the SCG discovered by our refinement strategy (which only tracks the forward pointer) happens to generate more different shape graphs than the most refined SCG (which tracks both pointers), although the former generates less predicates than the latter.

5.7 Conclusion

We developed a novel approach to the refinement of a particular kind of abstraction: shape abstraction based on logical structures. To express our new algorithm, we have extended the framework presented in the previous chapter (CPA+) to support the interruption of the analysis when a refinement is desired (because large-enough explicit heaps have been computed) or when an error path is encountered (because an abstract error has been found). The main insight is the use of an explicit analysis in order to make good guesses as to which abstraction are suitable and to restrict the search to relevant abstractions. Our experiments show that the mechanism we suggest is successful in identifying data structures, and do not incur a significant overhead. We believe that for other abstract domains, it make sense to take advantage of information provided by a cheap, explicit analysis to guide the refinement.

Chapter 6

Conclusion

Software verification is an inherently difficult problem, on which a lively research community is working. When the goal is as difficult as automatic software verification, it is imperative to bring to bear insights and optimizations no matter if they originated in model checking, program analysis, or automated theorem proving. The similarities between those approaches (and other approaches including testing) and the need to make them work together have been stressed several times by the community [Steffen 1991; Schmidt 1998; Cousot and Cousot 1995; Yorsh et al. 2006; McMillan 2009]. It is high time we abolished the artificial frontiers that have kept research communities segregated in the past if we want to push the boundaries of programs, for which ever more complex properties can be proven.

In this thesis we presented a unified framework for software verification. Our approach intended to be pragmatic and practical. Configurable program analysis captures the essence of overapproximation-based methods to software verification. The key insight is that existing approaches can be viewed as particular instantiations (or configurations) of a generic verification algorithm. We showed how the CPA framework enables to express classical approaches and new approaches uniformly, in a way that make side-by-side comparison easier. As such, we believe that it could be an excellent educational tool, where the understanding of the fundamental similarities and differences between algorithms is made clear. It should be noted that it is not a replacement to existing frameworks such as abstract interpretation. Our framework does not go in the same theoretical depth as abstract interpretation as a general study of the relation between abstract and concrete domains. Rather it focuses on the feature of

the reachability algorithm and can build upon previous and future research on improving abstract interpreters.

From an implementation perspective we strongly believe that the design choices made while designing the CPA framework form a sound foundation for a versatile implementation of program analyses. For the purpose of this thesis we extended an existing model checker because it enabled us to quickly prototype a working system based on existing building blocks from the Blast model checker and the Tvla analyzer. It helped us validate our approach and its applications. The group of Dirk Beyer has implemented over the last few years an implementation of the CPA framework called CPAchecker [Beyer and Keremoglu 2009]. Their implementation matches the theoretical concepts that we presented and therefore provides an ideal experimentation field, where a broad variety of abstract domains could be parameterized and combined in novel ways.

We see a great potential in the idea of encapsulating orthogonal aspects of an analysis in parameters of a generic algorithm. Parameterization enables easy experimentation of different configurations. As a result, we are able to evaluate experimentally the quality of certain choices and we can identify 'sweet spots' for the value of parameters for particular abstract domains or combinations of abstract domains. In this thesis, we have defined four parameters that control the analysis (transfer, merge, termination, precision adjustment). We believe that the same idea could be applied to control other aspects of the analysis. An obvious example is the exploration order, i.e., how to select the abstract state to remove from the frontier Extreme cases would be DFS and BFS orders, and intermediate solutions include random search and chaotic search. A study similar to the study of Bourdoncle [1993] on the influence of the exploration order could be conducted by considering the different values of the new operator. An other example is the work of Beyer et al. on variable-size block encoding [Beyer et al. 2009; Wendler 2010]. They studied the influence of the points at which abstraction is performed for a Boolean predicate abstraction. One extreme, small-block encoding, corresponds to the case where abstraction is performed at every location; the other extreme, large-block encoding, corresponds to the case where abstraction occurs only at loop heads. The latter was shown to be most efficient for predicate abstraction.

Static analyses strive to prove safety, but when they fail, in general, little useful information is provided to the user. Moreover, although the analysis might have succeeded in proving some part of the program safe, the user has no easy way of knowing which parts were successfully verified and which parts caused

the failure. An extension to our analysis algorithm could enable the analysis to output an assumption under which the program is safe. Implementing such an approach on top of our framework could provide a configurable way to control how complete the safety proof is. It is yet another example of an orthogonal parameter of an analysis that could be made configurable.

One theme that is pervasive to this thesis is the idea of combining analyses. Combination enhances modularity because it allows to use different abstract domains to adequately capture different aspects of concrete program states. One example is the location domain that frees the other domains from tracking location information. Moreover, we have seen in three different contexts that combining analysis generates better algorithms. First, combination on its own provides more precise results than running analyses independently. Second, combination in conjunction with dynamic precision adjustment allows analyses of different precisions to collaborate so as to take advantage of the strengths of each analysis. Third, combination in conjunction with refinement enables the result of a cheap analysis to contribute information for the refinement of an expensive analysis. For every of these directions, there is potential for future research. Combination of analysis has received a lot of focus in the past; yet, few approaches and tools are able to use a multitude of different analyses simultaneously. The uniformity of the CPA framework would enable to experiment with the combination of an arbitrary large number of analyses, and an interesting research question is how to determine which analysis to use for which program variables, either based on the program structure (a priori heuristics) or based on the partial verification results (as in CPA+). The least explored area is the use of combinations for refinement, and we are convinced that refinement strategies for domains other than three-valued-logic-based shape analysis could be enhanced using information extracted from an explicit analysis or from concrete program executions.

Our focus has been on reachable state space overapproximation. Underapproximation techniques are equally interesting, in particular in the context of bug finding. Some attempts have been made to integrate testing with software verification [Yorsh et al. 2006; McMillan 2009] and approaches such as directed testing are definitely at the border between symbolic approaches and testing [Godefroid et al. 2005; Sen et al. 2005; Godefroid and Klarlund 2005]. A framework similar to CPA in spirit but with different requirements can be developed to represent underapproximation approaches, or even combinations of underapproximation and overapproximation methods. Such a framework would open vast lines of research.

As a final note, if software verification is to succeed in solving its grand challenge [Hoare 2003], we need researchers to be able to exchange and reuse each other's ideas not only at a theoretical level but also at a practical level. The lack of standard frameworks, input and output languages, and broad benchmark sets results in a research field where it is hard to evaluate the real impact of new solutions. We hope that our work is one step in the right direction, by providing a unified and flexible framework, in which new solutions can be easily compared head to head.

Bibliography

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., and Xie, Y. 2004. Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 3114. Springer, 484–487.

Balaban, I., Pnueli, A., and Zuck, L. D. 2005. Shape analysis by predicate abstraction. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI).* Lecture Notes in Computer Science 3385. Springer, 164–180.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. 2006. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference.* ACM, 73–85.

Ball, T., Podelski, A., and Rajamani, S. K. 2001. Boolean and cartesian abstractions for model checking C programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Lecture Notes in Computer Science 2031. Springer, 268–283.

Ball, T. and Rajamani, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of 8th International SPIN Workshop on Model Checking Software.* Lecture Notes in Computer Science 2057. Springer, 103–122.

Ball, T. and Rajamani, S. K. 2002. The Slam project: Debugging system software via static analysis. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 1–3.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO).* Lecture Notes in Computer Science, vol. 4111. Springer, 364–387.

Beckman, N., Nori, A. V., Rajamani, S. K., and Simmons, R. J. 2008. Proofs from tests. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, 3–14.

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. 2007. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science, vol. 4590. Springer, 178–192.

Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., and Sebastiani, R. 2009. Software model checking via large-block encoding. In *Proceedings of*

the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD). IEEE, 25–32.

Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. 2007. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer 9,* 5-6, 505–525.

Beyer, D., Henzinger, T. A., and Théoduloz, G. 2006. Lazy shape analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 4144. Springer, 532–546.

Beyer, D. and Keremoglu, M. E. 2009. CPAchecker: A tool for configurable software verification. Tech. Rep. SFU-CS-2009-02, Simon Fraser University. January.

Beyer, D., Zufferey, D., and Majumdar, R. 2008. CSIsat: Interpolation for LA+EUF. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 5123. Springer, 304–308.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation.* Lecture Notes in Computer Science 2566. Springer, 85–108.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI).* ACM, 196–207.

Bourdoncle, F. 1993. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications.* Lecture Notes in Computer Science 735. Springer, 128–141.

Chaki, S., Clarke, E. M., Groce, A., Jha, S., and Veith, H. 2004. Modular verification of software components in C. *IEEE Trans. Softw. Eng. 30,* 6, 388–402.

Chaki, S., Clarke, E. M., Groce, A., and Strichman, O. 2003. Predicate abstraction with minimum predicates. In *Proceedings of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME).* Lecture Notes in Computer Science 2860. Springer, 19–34.

Chase, D. R., Wegman, M. N., and Zadeck, F. K. 1990. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI).* ACM, 296–310.

Cimatti, A., Griggio, A., and Sebastiani, R. 2008. Efficient interpolant generation in satisfiability modulo theories. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Lecture Notes in Computer Science, vol. 4963. Springer, 397–412.

Clarke, E. M., Biere, A., Raimi, R., and Zhu, Y. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design 19,* 1, 7–34.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50,* 5, 752–794.

Clarke, E. M., Grumberg, O., and Peled, D. A. 1999. *Model Checking.* MIT.

Clarke, E. M., Kroening, D., and Lerda, F. 2004. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science 2988. Springer, 168–176.

Clarke, E. M., Kroening, D., Sharygina, N., and Yorav, K. 2005. SatAbs: SAT-based predicate abstraction for ANSI-C. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science 3440. Springer, 570–574.

Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M. G., and Hermenegildo, M. 1993. Improving abstract interpretations by combining domains. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM, 194–205.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. ACM, 439–448.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Conference Record of 4th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 269–282.

Cousot, P. and Cousot, R. 1992. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*. Lecture Notes in Computer Science 631. Springer, 269–295.

Cousot, P. and Cousot, R. 1995. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *Proceedings of the 7th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science 939. Springer, 293–308.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 2008. Combination of abstractions in the Astrée static analyzer. In *Proceedings of the 11th Asian Computing Science Conference on Secure Software and Related Issues (ASIAN'06)*. Lecture Notes in Computer Science 4435. Springer, 272–300.

Craig, W. 1957. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log. 22,* 3, 250–268.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1989. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 25–35.

Dams, D. and Namjoshi, K. S. 2003. Shape analysis through predicate abstraction and model checking. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Lecture Notes in Computer Science 2575. Springer, 310–324.

Davey, B. A. and Priestley, H. A. 1990. *Introduction to Lattices and Order*. Cambridge University Press.

Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. 1992. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 212–223.

Distefano, D., O'Hearn, P. W., and Yang, H. 2006. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 3920. Springer, 287–302.

Dwyer, M. B. and Clarke, L. A. 1996. A flexible architecture for building dataflow analyzers. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*. IEEE, 554–564.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69,* 1-3, 35–45.

Esparza, J., Kiefer, S., and Schwoon, S. 2006. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science 3920. Springer, 489–503.

Field, J., Goyal, D., Ramalingam, G., and Yahav, E. 2003. Typestate verification: Abstraction techniques and complexity results. In *Proceedings of the 10th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 2694. Springer, 439–462.

Fischer, J., Jhala, R., and Majumdar, R. 2005. Joining data flow with predicates. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 227–236.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 234–245.

Floyd, R. W. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*. AMS, 19–32.

Godefroid, P. 1997. Model checking for programming languages using VeriSoft. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 174–186.

Godefroid, P. and Klarlund, N. 2005. Software model checking: Searching for computations in the abstract or the concrete. In *Proceedings of the 5th International Conference on Integrated Formal Methods*. Lecture Notes in Computer Science, vol. 3771. Springer, 20–32.

Godefroid, P., Klarlund, N., and Sen, K. 2005. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 213–223.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with Pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science 1254. Springer, 72–83.

Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., and Rajamani, S. K. 2006. Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 117–127.

Gulavani, B. S. and Rajamani, S. K. 2006. Counterexample-driven refinement for abstract interpretation. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Lecture Notes in Computer Science 3920. Springer, 474–488.

Gulwani, S. and Tiwari, A. 2006. Combining abstract interpreters. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 376–386.

Guo, B., Vachharajani, N., and August, D. I. 2007. Shape analysis with inductive recursion synthesis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 256–265.

Gupta, A., Majumdar, R., and Rybalchenko, A. 2009. From tests to proofs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Lecture Notes in Computer Science 5505. Springer, 262–276.

Havelund, K. and Pressburger, T. 2000. Model checking Java programs using Java PathFinder. *Int. J. Softw. Tools Technol. Transfer 2,* 4, 366–381.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs.* Elsevier North-Holland.

Hecht, M. S. and Ullman, J. D. 1973. Analysis of a simple algorithm for global data flow problems. In *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL).* ACM, 207–217.

Henzinger, T. A., Jhala, R., Majumdar, R., and McMillan, K. L. 2004. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 232–244.

Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. 2002. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 2404. Springer, 526–538.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 58–70.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12,* 10, 576–580.

Hoare, C. A. R. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM 50,* 1, 63–69.

Holzmann, G. J. 1997. The Spin model checker. *IEEE Trans. Softw. Eng. 23,* 5, 279–295.

Ivancic, F., Yang, Z., Ganai, M. K., Gupta, A., Shlyakhter, I., and Ashar, P. 2005. F-Soft: Software verification platform. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 3576. Springer, 301–306.

Jhala, R. and Majumdar, R. 2009. Software model checking. *ACM Computing Surveys (CSUR) 41,* 4, 1–54.

Jhala, R. and McMillan, K. L. 2005. Interpolant-based transition relation approximation. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science, vol. 3576. Springer, 39–51.

Jones, N. D. and Muchnick, S. S. 1982. A flexible approach to interprocedural data-flow analysis and programs with recursive data structures. In *Conference*

Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL). ACM, 66–74.

Kapur, D., Majumdar, R., and Zarba, C. G. 2006. Interpolation for data structures. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005).* ACM, 105–116.

Kennedy, K. W. 1975. Node listings applied to data flow analysis. In *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages (POPL).* ACM, 10–21.

Kildall, G. A. 1973. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages (POPL).* ACM, 194–206.

Kleene, S. C. 1987. *Introduction to Metamathematics.* North-Holland, Amsterdam, Hollad.

Kröning, D., Groce, A., and Clarke, E. M. 2004. Counterexample-guided abstraction refinement via program execution. In *Proceedings of the 6th International Conference on Formal Engineering Methods, Formal Methods and Software Engineering (ICFEM).* Lecture Notes in Computer Science 3308. Springer, 224–238.

Lam, P., Kuncak, V., and Rinard, M. C. 2005. Hob: A tool for verifying data structure consistency. In *Proceedings of the 14th International Conference on Compiler Construction (CC).* Lecture Notes in Computer Science, vol. 3443. Springer, 237–241.

Leino, K. R. M. and Nelson, G. 1998. An extended static checker for Modula-3. In *Proceedings of the 7th International Conference on Compiler Construction (CC).* Lecture Notes in Computer Science 1383. Springer, 302–305.

Lerner, S., Grove, D., and Chambers, C. 2002. Composing data-flow analyses and transformations. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 270–282.

Lev-Ami, T. and Sagiv, M. 2000. Tvla: A system for implementing static analyses. In *Proceedings of the 7th International Symposium on Static Analysis (SAS).* Lecture Notes in Computer Science 2280. Springer, 280–301.

Loginov, A., Reps, T. W., and Sagiv, M. 2005. Abstraction refinement via inductive learning. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 3576. Springer, 519–533.

Martin, F. 1998. Pag: An efficient program analyzer generator. *Int. J. Softw. Tools Technol. Transfer 2,* 1, 46–67.

Mauborgne, L. and Rival, X. 2005. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Symposium on Programming on Programming Languages and Systems (ESOP).* Lecture Notes in Computer Science 3444. Springer, 5–20.

McMillan, K. L. 2003. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science 2725. Springer, 1–13.

McMillan, K. L. 2005a. Applications of craig interpolants in model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Lecture Notes in Computer Science, vol. 3440. Springer, 1–12.

McMillan, K. L. 2005b. An interpolating theorem prover. *Theor. Comput. Sci. 345,* 1, 101–121.

McMillan, K. L. 2008. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Lecture Notes in Computer Science, vol. 4963. Springer, 413–427.

McMillan, K. L. 2009. What's in common between test, model checking, and decision procedures? In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS).* Lecture Notes in Computer Science, vol. 5825. Springer, 35–36.

Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. 2002. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI).* USENIX.

Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. 2002. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC).* Lecture Notes in Computer Science 2304. Springer, 213–228.

Nelson, G. and Oppen, D. C. 1979. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. 1,* 2, 245–257.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis.* Springer.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL).* Lecture Notes in Computer Science, vol. 2142. Springer, 1–19.

Podelski, A. and Wies, T. 2005. Boolean heaps. In *Proceedings of the 12th International Symposium on Static Analysis (SAS).* Lecture Notes in Computer Science, vol. 3672. Springer, 268–283.

Podelski, A. and Wies, T. 2010. Counterexample-guided focus. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 249–260.

Reineke, J. 2005. Shape analysis of sets. M.S. thesis, Saarland University, Germany.

Reps, T. W., Horwitz, S., and Sagiv, M. 1995. Precise interprocedural data-flow analysis via graph reachability. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 49–61.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS).* IEEE Computer Society, 55–74.

Rinetzky, N., Sagiv, M., and Yahav, E. 2004. Interprocedural functional shape analysis using local heaps. Tech. Rep. TAU-CS-26/04, Tel-Aviv University.

Rybalchenko, A. and Sofronie-Stokkermans, V. 2007. Constraint solving for interpolation. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI).* Lecture Notes in Computer Science 4349. Springer, 346–362.

Sagiv, M., Reps, T. W., and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24,* 3, 217–298.

Saidi, H. 2000. Model-checking-guided abstraction and analysis. In *Proceedings of the 7th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science 1824. Springer, 377–396.

Schmidt, D. A. 1998. Data-flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 38–48.

Sen, K., Marinov, D., and Agha, G. 2005. Cute: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 263–272.

Sharir, M. 1980. A new approach to flow analysis in optimizing compilers. *Computer Languages 5*, 141–153.

Steffen, B. 1991. Data-flow analysis as model checking. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS)*. Springer, 346–365.

Strom, R. E. and Yemini, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12,* 1, 157–171.

Tjiangan, S. W. K. and Hennessy, J. 1992. Sharlit: A tool for building optimizers. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 82–93.

Wendler, P. 2010. Software verification based on adjustable large-block encoding. M.S. thesis, University of Passau, Germany.

Wies, T. 2009. Symbolic shape anaylsis. Ph.D. thesis, University of Freiburg, Germany.

Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. W. 2008. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 5123. Springer, 385–398.

Yorsh, G., Ball, T., and Sagiv, M. 2006. Testing, abstraction, theorem proving: Better together! In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 145–156.

Yorsh, G., Reps, T. W., Sagiv, M., and Wilhelm, R. 2007. Logical characterizations of heap abstractions. *ACM Trans. Comput. Log. 8,* 1, 5.

Curriculum Vitae

Grégory Théoduloz

Education

2006 – 2010	Ph.D. in Computer Science Ecole Polytechnique Fédérale de Lausanne (EPFL)
2001 – 2006	M.Sc. in Computer Science Ecole Polytechnique Fédérale de Lausanne (EPFL)

Experience

Spring 2010	Teacher of “Theoretical Computer Science” Ecole Polytechnique Fédérale de Lausanne (EPFL)
2006 – 2009	Teaching Assistant for “Theoretical Computer Science” Ecole Polytechnique Fédérale de Lausanne (EPFL)
Summer 2006	Visiting student with Prof. Rupak Majumdar University of California, Los Angeles
2005 – 2006	Teaching and Research Assistant Models and Theory of Computation, EPFL
2003 – 2005	Teaching Assistant for undergraduate and graduate classes School of Computer and Communication Sciences, EPFL

Publications

1. Grégory Théoduloz, “Integrating Shape Analysis in the Model Checker BLAST”, *Master thesis*, Ecole Polytechnique Fédérale de Lausanne, 2006.

2. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, "Lazy Shape Analysis", *Proceedings of the 18th International Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, Springer, 2006.
3. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis", *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, Springer, 2007.
4. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, "Program Analysis with Dynamic Precision Adjustment", *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 29–38. IEEE, 2008.
5. Dirk Beyer, Thomas A. Henzinger, Grégory Théoduloz, and Damien Zufferey, "Shape Refinement through Explicit Heap Analysis", *Proceeding of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, Lecture Notes in Computer Science 6013, pages 263–277. Springer, 2010.

Academic Awards

1. Cousin Award (best grade average first and second undergraduate years), 2003
2. Dommer Award (best grade average for engineering sections), 2006
3. Elca Award (best M.Sc. in Computer Science grade average), 2006
4. Unicible Award (Master thesis in Computer Science of an outstanding quality), 2006
5. Microsoft Research PhD Scholarship (3 years), 2007

Coordinates

Models and Theory of Computation
School of Computer and Communication Sciences
Ecole Polytechnique Féderale de Lausanne (EPFL)
CH-1015 Lausanne

Email: gregory.theoduloz@a3.epfl.ch
Web: http://me.gtz.ch

Personal Details

Date of birth:	7 February 1983
Citizenship:	Swiss
Marital status:	Single