HAL
open science

Static analysis of numerical properties in the presence of pointers

Zhoulai Fu

To cite this version:

Zhoulai Fu. Static analysis of numerical properties in the presence of pointers. Other [cs.OH]. Université de Rennes; Université européenne de Bretagne (2007-2016), 2013. English. NNT : 2013REN1S060 . tel-00918593

HAL Id: tel-00918593

https://theses.hal.science/tel-00918593v1

Submitted on 13 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Zhoulai FU

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires
Composante universitaire : Université de Rennes 1

Static Analysis of Numerical Properties in the Presence of Pointers

Thèse soutenue à Rennes le 22 Juillet, 2013

devant le jury composé de :

Mario SUDHOLT
Professeur, École de Mines de Nante / rapporteur

Laurent MAUBORGNE
Chargé de recherche, IMDEA Software / rapporteur

Antoine MINE
Chargé de recherche, ENS Ulm / examinateur

Sophie Pinchinat
Professeur, ISTIC, Université de Rennes 1 / examinateur

David PICHARDIE
Chargé de recherche, INRIA / co-directeur de thèse

Thomas JENSEN
Directeur de recherche, INRIA / directeur de thèse

Acknowledgment

The present thesis has been done in the frame of my PhD contract with Université de Rennes 1, in cooperation with INRIA, Rennes, France. I appreciate the grant AMX from the French Ministry of Research. I also received financial support from AX – l'association des anciens élèves et diplômés de l'école polytechnique. Sincere acknowledgments are due to my fellow alumni Vincent Mignotte, Yves Stierlé and Nicolas Zarpas. The discussions with them at 5 rue Descartes, Paris, turned out to be a crucial point in my professional life that provided me encouragement to continue with my research career.

Thanks to my supervisors Thomas Jensen and David Pichardie for proofreading my thesis. I would also like to thank them for their generously supporting me to attend two summer schools. There are certainly things beyond financials — my PhD was never smooth, but looking back, I have grown up, learned to be positive, and learned to be independent.

I am profoundly grateful to Laurent Mauborgne. Near the end of my PhD Laurent invited me to his lab. He listened to me and gave me full confidence. It was a pleasure to work with someone with such humor, humanity, and fierce intelligence. Thank you, Laurent.

Sincere thanks are due to Scott Livingston, with whom I was lucky enough to have as an officemate and friend at IMDEA Software, Madrid. I learned from Scott a sense of altruism, dignity and respect toward life, even including animals, like bats, which had been large concepts for my humble mind.

I am also indebted to many people for their perspective and technique guide. They don't necessarily know me personally. Thanks to Patrick Cousot for allowing me to take his abstract interpretation course as an auditing student at ENS, Paris. Thanks to Eric Bodden, Laurie Hendren, Patrick Lam and others who replied quickly and clearly to my questions on McGill's Soot mailing list. Thanks to Roberto Bagnara, Enea Zaffanella and other PPL development team members who helped me to use their library.

My parents, as well as other family members of mine, have supported me at all times in my life. I owe them life-long, infinitely.

At last, I wish to express my deepest gratitude to Fang, my wife, for her unconditional love and support. "You and I have memories $\sim$ longer than the road that stretches out ahead..." (The Beatles – Two of Us)

Résumé

Le rythme rapide et furieux de l'évolution de la technologie informatique, tant pour hardware et les logiciels, est devenu un article de foi pour beaucoup. Le problème de la fiabilité se pose depuis la construction d'ENIAC, le premier calculateur électronique qui est Turing complet : ses tubes à vide brûlés chaque jour, laissant l'ordinateur fréquemment non-opérationnel. Depuis les années 70, les systèmes informatiques deviennent plus en plus complexe et empiètent massivement sur notre vie quotidienne à travers toutes sortes de systèmes embarqués, ordinateurs portables, téléphones mobiles et réseaux informatiques. Aujourd'hui, la fiabilité du logiciel est un attribut essentiel de la qualité du logiciel. Si la production de logiciel fiable est depuis longtemps la préoccupation d'ingénieurs, elle devient à ce jour une branche de sujets de recherche riche en applications, dont l'analyse statique.

Nous considérons le problème de l'analyse statique des propriétés numériques en présence d'accèss au tas. L'objectif n'est pas d'étudier une nouvelle analyse statique mais de combiner l'anlayses numérique et l'analyses de pointeur. Afin d'exploiter des analyses existantes, nous avons définit un nouveau domain abstrait, NumP, qui consiste de produit d'une abstraction standard de pointeurs et une abstraction numérique qui extend domaines abstraits numériques pour permettre d'exprimer des relations entre des éléments du tas, des champs et de variables numériques. Le domaine abstrait NumP est développé à l'aide de la théorie d'interprétation abstraite.

Une deuxème contribution porte sur l'amélioration d'analyse de type *points-to*. Cette étude est motivée par l'observation que des graphes résultant d'analyses *points-to* contiennent souvent des liens qui ne peuvent pas

correspondre à une vraie relation de référene pendant l'execution. Nous considérons des situations dans lesquelles un pointeur est un alias d'un second qui limite les cibles du premier à un sous-ensemble strict des cibles identifiées par l'analyse *points-to*. À cette fin, nous proposons l'utilisation d'une analyse de *must alias*es afin d'éliminer les liens correspondant à des références redondantes. Partant d'une définition sémantique de ces liens, les résultats principaux de ce travail consistent en une condition nécessaire caractérisant ces liens. Nous avons formalisé cette combinaison en nous appuyant sur la notion de bisimulation, bien connue en vérification de modèles ou théorie de jeu. Un algorithme de complexité polynomiale est proposé et prouvé correct. Nous avons aussi démontré que le problème pour trouver la conditon nécessaire et suffisante est NP-dur.

Sur la partie de validation, nous présentons notre approche dans le contexte de l'analyse de programme Java. Cette analyse est implémentée par combiner SOOT qui fournit des analyses de pointeurs, et bibliothèque PPL (Parma Polyhedra Library) qui est une librarie de domaines abstraits numériques. Notre expériences sur l'algorithme Bellman-Ford explorent les combinaisons de quatre analyses existantes, deux analyses numériques sur des intervalles et des polyèdres de la PPL et deux analyses de points-to fournies par SOOT. Les résultats expérimentaux sur les benchmarks de Dacapo montrent notamment que notre anlayse permet d'inférer un nombre significativement plus grand d'invariants que l'utilisation des deux analyses existantes (29 fois plus d'invariants en moyenne). En outre, le surcôut en temps de la nouvelle analyse est limité (13% en moyenne).

Contents

1 Introduction **12**

1.1 Context of the Problem 12

1.2 Objectives and Methods 15

1.2.1 $\mathsf{Num} \times \mathsf{Pter}$ 15

1.2.2 $\mathsf{Pter} \times \mathsf{Must}$ 16

1.3 Contributions 17

1.4 Plan 18

2 Background **19**

2.1 Some Definitions in Lattice Theory 19

2.2 The Languages WHILE_{np}, WHILE_{n} and WHILE_{p} 21

2.3 Elements of Abstract Interpretation 22

2.4 Static Numerical Analysis 25

2.5 Points-to Analysis 27

3 Lifting Numerical Abstract Domains **32**

3.1 Introduction 32

3.1.1 Objectives and Contributions 34

3.2 Semantics Abstraction 35

3.2.1 An Isomorphic Operational Semantics 36

3.2.2 Cartesian Abstraction 37

3.2.3 The Abstract Domain $NumP$ 38

3.3 Transfer Functions 39

3.4 Proof of Soundness 43
3.4.1 Preliminaries 43
3.4.2 Proof 45
3.5 Related Work 47
3.6 Conclusion 49

4 Enhancing Points-to Analysis 50
4.1 Introduction 50
4.1.1 Motivating Example 50
4.1.2 Backward-simulation 52
4.1.3 Contribution 58
4.2 Redundancy Elimination of Points-to Graph 59
4.2.1 Must-graph and the Non-standard points-to graph . . 59
4.2.2 Redundancy Elimination in a Semantics-based View . . 60
4.2.3 Toward the Soundness Condition 63
4.3 Backward-simulation and Fuzzy Nodes 66
4.4 Algorithm of Redundancy Elimination 70
4.5 Incompleteness 73
4.6 Comparison with Related work 78
4.7 Conclusions 79

5 Prototyping NumP 81
5.1 Design issues 81
5.1.1 Reused components 81
5.1.2 Precision/cost trade-off 82
5.2 Implementation 83
5.2.1 Architecture 83
5.2.2 Work-flow 84
5.3 Experimental Results 89
5.3.1 `Bellman-Ford` 89
5.3.2 Dacapo 91

6 Conclusions **93**

Bibliography **95**

List of Figures

1.1 Illustration of alias. 14

1.2 Example program (left) and the semantics actions for its analyse(right). The variable "delta" is the symbolic variable. . . . 16

2.1 The formal syntax of WHILE_{np} (left), and an example program (right). 21

3.1 Structural Operational semantics $\widetilde{\longrightarrow}^{\flat} : \mathsf{WHILE}_{np} \to (\widetilde{State} \times \widetilde{State})$ 36

3.2 $[\![\cdot]\!]^{\natural} : \mathsf{WHILE}_{np} \to (A^{\natural} \to A^{\natural})$. The notation $\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d$ means that $d \in \{(\mathsf{p}(y_p), f_n) \mid \mathsf{p} \in \tilde{\mathsf{p}}\}$ 37

3.3 Semantics abstraction of memory states at the loop entry of the example program in Fig. 2.1 (right, l. 3). Heap locations are depicted as rectangles labeled by references. The value of each pointer variable is depicted as an arrow from the variable name to the referenced rectangle. The symbol $\diamond$ is for the `null` pointer. We have omitted the range $1 \leqslant k \leqslant 8$ of the script k occurring in the first three rows. The label for the field "next" on the directed edges is not drawn for the first three rows. . . 40

3.4 Semantics abstraction toward $NumP$ takes three steps. 41

4.1 The example analyzed code. The program first creates two linked lists (from $\ell 10$ to $\ell 40$ where `List` has a field f), non-deterministically assigns to variable x the references of the two lists (from $\ell 50$ to $\ell 90$). At last, an instruction accessing the heap is performed under the condition that x, y hold the same reference value. 53

4.2 Compare standard points-to analyzer and ours. From the first column to the third column: line number, standard points-to analyzer, our analyzer and must-alias analyzer. The graph corresponds to the result before the indicated line number. Labels of the arcs with the same pair of source and targets are grouped together. 54

4.3 Possible concrete environments for points-to graph in the first row of Fig. 4.2. Here, we have assumed that: r_{10}, r_{15} are abstracted as h_1; r_{20}, r_{25} are abstracted as h_2; r_{30}, r_{35} are abstracted as h_3; and r_{40} is abstracted as h_4. 55

4.4 Backward-simulation 58

4.5 Points-to graph (left) and must-graph (right). Redundant arcs are $h_1 \xrightarrow{f} h_2$ and $h_2 \xrightarrow{f} h_3$. 58

4.6 Incompleteness: points-to graph (left) and must-graph (right). 73

4.7 A Hamilton graph G 75

4.8 From left to right, the points-to graph Θ and the must graph μ constructed from the graph G. 75

4.9 The reduced minimal points-to graph $\bar{\Theta}$. 76

5.1 The architecture of the prototype represented as class diagram of UML 84

5.2 The work-flow of the prototype represented as action diagram of UML 84

5.3 An example in `Java`. The program passes an array of integers to a list of `Unsigned` numbers. `Unsigned` is a superclass of `Pos` and `Neg`. It has one field `val` of integer type. The class `List` has two fields, `item` of type `Unsigned`, and `next` of type `List`. 86

List of Tables

3.1 Post-conditions of $a.val = b.val + c.val$, assuming $b.val \in [3, 6], c.val \in [4, 8]$. Columns 2-6 show 5 aliasing relations between 3 variables. Column 7 joins the results. 33

5.1 Case study on `Bellman-Ford` 90

5.2 Performance test of NumP for the `Dacapo-2006-MR2` benchmark 92

Chapter 1

Introduction

1.1 Context of the Problem

The fast and furious pace of change in computing technology, both for hardware and software, has become an article of faith for many. The reliability of the first Turing-complete electronic computer ENIAC was more about a question of hardware. Its vacuum tubes burned out every day, leaving the computer frequently non-operational. In 70 years, the computer systems are booming exponentially, with their performance and programs size multiplied by millions. These systems are becoming increasingly complex and massively impinge on our daily life through all kinds of embedded systems, laptops, mobile phones and computer network. To date, *software reliability* is a key attribute of software quality.

Software reliability is defined by ANSI[1] as the probability of failure-free software operation for a specified period of time in a specified environment. The term *failure* in this definition means any departure from the *required* function of the system. In safety-critical systems, software failures are fatal. Some disasters are caused by infamous computer arithmetic errors. The Patriot missile failed to intercept a *scud* missile because of the inaccuracy in a floating point calculation. Patriot measured time in tenths of second and its internal computing system calculated the measured time by 1/10 to produce the time in seconds. However, using its 24-bit register the non-terminating

[1]Standard Glossary of Software Engineering Terminology, ANSI/IEEE, 1991

binary expansion of 1/10 has to be truncated, which introduced an error of about 0.000000095 decimal. The tiny rounding error when accumulated with more than 100 running hours, was large enough to miss the incoming scud. The Patriot failure cost 28 lives.[2]

Another well-known computer-made tragedy was the destruction of Ariane 501 in 1996. It is caused by wrong conversion of a 64-bit floating point number relating to the horizontal velocity of the rocket to a 16 bit signed integer. The velocity was recorded by a number larger than $32,767$ which is the largest number in a 16-bit signed integer. By consequence, the conversion failed due to the run-time error of *integer overflow*. The failure resulted in a loss of more than 370 million U.S dollars [29].

In computer systems that are not safety-critical, a certain failure rate may be tolerable. Still, this is a question of money and service quality. Poor services may be the primary complaints of disgruntled clients. Significant financial consequences can be caused for the manufacturers because correct systems are essential for their survival. For example, Intel's highly promoted Pentium chip *P5* is found inaccurate when dividing floating-point numbers that occur within a specific range. This design woe, known as the *Pentium FDIV bug* [3], caused Intel a loss of approximately 475 million U.S. dollars to replace faulty processors, and severely damaged its reputation as reliable chip-maker.

To sum up, the reliability of computer-based systems crucially depend on the correctness of its computing. Can man, who created the computer, be capable of preventing machine-made misfortune? The theory of *Static analysis* strives to achieve this ambition.

While a large range of properties can be considered by methodologies of static analyses, this thesis focuses on the static analysis of numerical properties. A simple example can be the automatic discovery of the signs of program variables. A more advanced example would be the discovery of linear relations of program variables, or even non-linear relations. These kinds of

[2] GAO. 1992. Patriot Missile Software Problem. Report of the Information Management and Technology Division. Available at http://www.fas.org/spp/starwars/gao/im92026.htm.

[3] Discovered by Thomas Nicely, more information about the bug can be found at http://www.trnicely.net/pentbug/pentbug.html

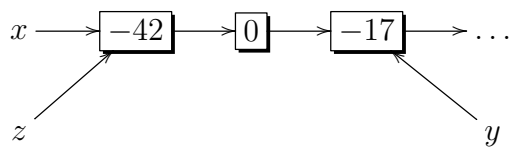


Figure 1.1: Illustration of alias.

analyses are usually necessary to verify program safety conditions involving numerical computations, such as *division by zero*, *array index out-of-bound*, or *buffer overflow*.

Dated back to 1978, Cousot and Halbwachs showed how to determine at compile-time linear relations among program variables [26]. This kind of endeavor, technically called *static numerical analysis*, keeps continuing with further development of abundant numerical abstract domains [56, 17] that vary with different precision/efficiency trade-off.

These analyses have been successfully used in practice. Some libraries of abstract domains have been developed, such as PPL, Polyglot or Apron, to list a few. On the other hand, the more and more complex data structures in high-level programming languages make the usage of pointer ubiquitous. Pointers change the way we look at static numerical analysis.

Pointers may introduce *alias problems*. An alias occurs when a storage location is pointed-to by pointers of different names. For instance, Fig. 1.1 represents the data structure of a linked list. Each node has a field *val* that stores an integer and a field *next* that stores the reference of its successor node. In the figure, both variables x and z point to the first node; y points to the third node. The alias relation contains (x, z), $(x.next.next, y)$, $(z.next.next, y)$, etc.

To a programmer, this alias problem sounds to be a frequent issue: an operation of program modifies the properties of a target and unintentionally changes something that does not appear in the operation. For example, a *store* statement may appear to only modify the value of $x.f$, but each field reference $y.f$ such that y and x hold equal reference before the operation will also be affected.

1.2 Objectives and Methods

To analyze numerical properties in the presence of pointers, we have integrated pointer analyses with traditional static numerical analysis.The challenge lies in how to combine the component analyses. We have extensively employed the semantics approximation framework *abstract interpretation* [24] to uniformly combine pointer analyses and traditional numerical analysis. Two pointer analyses are considered in this context. They are points-to analysis and must-alias analysis. Below we write Num (sans-serif font) for traditional static numerical analysis, Pter for points-to analysis, and Must for must-alias analysis. The long-term objective of this work is to design a new analysis A that integrates the traditional static numerical analysis with the two aforementioned pointer analyses.

$$A = \mathsf{Num} \times \mathsf{Pter} \times \mathsf{Must}$$

This thesis presents how we combine Num with Pter, and Pter with Must, the combination of the three components are left as future work. Below we show the basic ideas through two examples.

1.2.1 $\mathsf{Num} \times \mathsf{Pter}$

The example in Fig. 1.2 allocates a list of memory cells on the heap to store integers ranging from -5 to 2. Suppose that we want to infer the property holding at the loop entry:

- The value of $hd.next^k.val$ for any k is in the range of -5 to 2

Traditional numerical abstract domains are able to reason on properties between scalar variables. They can by no way infer the properties of the quantified $hd.next^k.val$ which does not even appear literally in the program. The solution is to correlate these $hd.next^k.val$ semantically with some program identifier that is syntactically contained in the program. This is where a pointer analysis comes in. In words, we use the pointer analysis to treat the concrete program as a sequence of abstract semantics actions specified on the right of Fig. 1.2. The symbolic variable δ represents $elem.val$ and is treated in a similar way as scalar variable. However, the semantic action

```
1 int i = -5;
2 A hd = null, elem = null;
3 while (i < 3) {
4     elem = new Node();//h
5     elem.val = i;
6     elem.next = hd;
7     hd = elem;
8     i = i + 1;
9 }
```

```
1 int i = -5;
2 while (i < 3) {
3   δ = i;
4   i = i + 1;
5 }
```

Figure 1.2: Example program (left) and the semantics actions for its analyse(right). The variable "delta" is the symbolic variable.

$\delta = i$ is to be considered as accumulating, rather than updating, the values of i to δ. Using traditional numerical analysis, we are able to obtain $\delta \in [-5, 2]$ at the entry of the loop. The final step consists in correlating the values of $hd.next^k.val$ with δ. This step relies on both points-to analysis and the semantics with regard to δ: δ represents all the concrete references associated with h and labeled by f. In addition, the points-to analysis tells that each $hd.next^k$ only holds references allocated at h. The correctness of these steps can be guaranteed using the theory of abstract interpretation.

1.2.2 Pter × Must

In practice, pointer analyses are mostly flow-insensitive. This category of pointer analyses do not distinguish program flows and may cause imprecision for the combined numerical analysis presented above.

Consider the Java snippet below. Assume that f is a class field. The flow-insensitive points-to analysis would reason that f may be equal to a or b for all the program points. This is imprecise because at l. 5 and l. 6 we have f and b must not be equal, and f and a must not be equal at l. 9 and l. 10. Due to this imprecision, the static numerical analysis presented above would tell that $f.val$ would be in the range of $[10, 20]$ at l. 6 and at l. 10.

This imprecision can be recovered using must-alias analysis. Consider the case at l. 6. We have f and a must be equal before the line. This information, combined with the fact that a and b can not be equal for the whole program (obtained from points-to analysis) ensures that f and b can not be equal at l. 6. The using of must-alias analysis allows us to refine the flow-insensitive analysis, which in turn makes the static numerical analysis more precise. Here, the refined points-to analysis combined with traditional numerical analysis infers that the possible values of $f.val$ will be 10 at l .6 and 20 at l .10.

```
1   a = new A();
2   b = new A();
3   if (...){
4      f = a;
5      f.val=10;
6   }
7   else{
8      f = b;
9      f.val=20;
10  }
```

1.3 Contributions

We have developed a new abstract domain that combines traditional abstract domain with points-to analysis. This new abstract domain allows us to express a new category of numerical properties that cannot be expressed by traditional numerical domain, such as $x.f + y.g < a[i]$. This abstract domain has a modular design and is built from its component abstract domains in a black-box manner. This is meaningful because the soundness of the component analyses can be proved based on the soundness of its components, and the implementation of the combined analysis can be achieved effortlessly using the existing implementations of its components .

Our second contribution is an algorithm to remove a part of redundancy in flow-insensitive points-to analysis using must-alias analysis. The algorithm computes the reduced product of the domains of must-alias analysis and

points-to analysis. This allows us to refine the points-to analysis *a posterior*.

We have experimented with several prototypes to test the effectiveness of these approaches. Our first implementation consists of a wrapping of abstract domains in PPL, using SOOT [70] as the front-end. This implementation scales up to program with more than 350 KLoc. Although many existing numerical domains have been developed in the form of libraries, like PPL, NEWPOLKA and APRON etc, to the best of our knowledge little relevant work has been done for Java communities and scale up to real-life programs. Based on this implementation, we integrate points-to analysis with traditional numerical domain. The prototype shows that the combined analysis discovers significantly more numerical invariants than traditional static numerical analysis. In addition, the time overhead of the combined analyses is little and thus makes it scalable to large program as long as its component analyses are scalable.

1.4 Plan

This thesis is organized as follows. Chapter 2 is the background for understanding this thesis. Chapter 3 gives the theoretical framework of numerical analysis in the presence of pointers. Chapter 4 presents our theoretical study of partial redundancy elimination of points-to graph in the presence of must-alias. Conclusion and future work are shown in Chapter 5.

Chapter 2

Background

We use standard notations in *predicate calculus* (e.g. Enderton's [31]) and *set theory* (e.g. Bourbaki's [10]). Preliminary concepts on *lattice theory* will be first introduced. A standard reference is Birkhoff's book [6]. Basic notations on the theory of *abstract interpretation* will then be covered. The references for this subject can be Cousot's thesis in 1978 [20] or [25] of Cousot and Cousot. At last, we present the *static numerical analysis* and *points-to analysis* as two instances of the abstract interpretation framework.

2.1 Some Definitions in Lattice Theory

Let U be a *set*. The set of all subsets of U will be denoted by $\wp(U)$. The set of all integers, will be denoted by $\mathbb{Z}$. The cardinal of a set U is denoted by $|U|$. Given two sets A and B, a *relation* R is a subset of $A \times B$. We write $a\ R\ b$ for $(a, b) \in R$. The relation R is called a *function*, if for each $a \in A$, there exists a unique $b \in B$ such that aRb, *i.e.,* $\forall a, a' \in A, b \in B : b\ R\ a \wedge b\ R\ a' \implies a = a'$. This function is said to have *type* $A \to B$.

Definition 2.1.1. *The* post-image *(resp.* pre-image*) of a relation $R \subseteq A \times B$ is a function of type $\wp(A) \to \wp(B)$ (resp. $\wp(B) \to \wp(A)$):*

$$post[R](A_1) \triangleq \{b \in B \mid \exists a : (a, b) \in R \wedge a \in A_1\} \qquad \text{(post-image)}$$

$$pre[R](B_1) \triangleq \{a \in A \mid \exists b : (a, b) \in R \wedge b \in B_1\} \qquad \text{(pre-image)}$$

Definition 2.1.2 (Semilattice)**.** *A* semilattice $(D, \sqcup)$ *is the set* D *equipped with a* join *operation* $\sqcup$ *such that for all* $a, b, c \in D$

$$a \sqcup a = a \qquad \text{(idempotent)}$$
$$a \sqcup b = b \sqcup a \qquad \text{(commutative)}$$
$$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup a \qquad \text{(associative)}$$

The concept of *partially ordered* set can then be derived from that of semilattice.

Definition 2.1.3 (Partial order)**.** *Given a semilattice* $(D, \sqcup)$*, the partial order* $\sqsubseteq$ *is defined to be the maximal relation on* D *s.t. for each* $a, b \in D$*,* $a \sqsubseteq b$ *if and only if* $a \sqcup b = b$*. A set* D *equipped with a partial order* $\sqsubseteq$ *is said to be a* poset*. An element* $d \in D$ *is an* upper bound *of* D_1 *iff each* $d_1 \in D_1$ *satisfies* $d_1 \sqsubseteq d$*;* d *is called the* supremum *of* D_1 *if* d *is an upper bound, and for any upper bound* d' *of* D_1*, the condition* $d \sqsubseteq d'$ *holds.*

The dual definitions of *lower bound* and *infimum* are omitted. It can be verified that the relation $\sqsubseteq$, is *reflexive* ($\forall a : a \sqsubseteq a = a$), *transitive* ($\forall a, b : a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b$), and *anti-symmetric* ($\forall a, b, c : a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c$).

The complete lattice is a semilattice with a "complete join":

Definition 2.1.4 (Complete lattice)**.** *A semilattice* $(D, \sqcup)$ *is* complete *if and only if any subset of* D *has supremum. The complete lattice can be denoted as* $(D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$*, where* $\sqsubseteq$ *is the partial order derived from* $\sqcup$ *as above. The complete meet* $\sqcap$ *is defined as the supremum of the lower bounds:* $\sqcap D_1 \triangleq \sqcup\{a \in D \mid \forall d, a \sqsubseteq d\}$ *At last,* $\bot$ *and* $\top$ *are the infimum and supremum of* D*.*

Definition 2.1.5 (Fixpoints)**.** *Given a function* f *defined over a poset* $(D, \sqsubseteq)$*, an element* $d \in D$ *is* fixpoint*,* post-fixpoint*, or* pre-fixpoint *of* f*, if* $f(d) = d$*,* $f(d) \sqsubseteq d$ *or* $d \sqsubseteq f(d)$ *respectively. The least fixpoint of* f *is denoted by* lfp f*.*

$$
\begin{aligned}
s_n &::= x_n = k \mid x_n = y_n \\
&\mid x_n = y_n \diamond z_n \mid x_n \bowtie y_n \\
s_p &::= x_p = \texttt{new} \mid x_p = x_p.f_p \\
&\mid x_p = y_p \mid x_p.f_p = y_p \\
s_{np} &::= x_p.f_n = y_n \mid x_n = y_p.f_n \\
s &::= s_n \mid s_p \mid s_{np}
\end{aligned}
$$

```
1 int i = -5;
2 A hd = null, elem = null;
3 while (i < 3) {
4     elem = new Node();//h
5     elem.val = i;
6     elem.next = hd;
7     hd = elem;
8     i = i + 1;
9 }
```

Figure 2.1: The formal syntax of WHILE_{np} (left), and an example program (right).

A main result from Tarski [68] is that a monotonic function defined over a complete lattice admits a least fixpoint:

Theorem 2.1.1 (Tarski's fixpoint theorem)**.** *Given a monotone function f over a complete lattice $(D, \sqsubseteq, \bot, \sqcup)$, the set $F = \{d \in D | f(d) = d\}$ is a non-empty complete lattice w.r.t. the order $\sqsubseteq$. Furthermore* lfp $f = \sqcap\{d \in D | f(d) \sqsubseteq d\}$

2.2 The Languages WHILE_{np}, WHILE_n and WHILE_p

Consider an imperative language that mixes pointer and numerical operations. The language will be denoted by WHILE_{np}. The left of Fig. 2.1 gives its formal syntax. In the figure, we write k for constant values, $\diamond$ for an arithmetic operator$+$, $-$, $*$ and $\backslash$, and the symbol $\bowtie$ belonging to $\{<, >, \leqslant, \geqslant, ==, \neq\}$ denotes a comparison operator. We assume two sets of variables: numerical variables and pointer variables. Variables x, y, z and field f are subscripted with n or p to indicate whether they are numerical values or pointers. An example program is shown on the right of Fig. 2.1.

We shall use WHILE_n to refer to basic statements only involving numerical variables and use the meta-variables s_n to range over those statements. Sim-

ilarly, we let WHILE_p be the statements that only use pointer variables and let s_p range over those statements . Thus, the basic statements of WHILE_{np} include those in WHILE_n and WHILE_p, and two more statements in the forms of $x_p.f_n = y_n$ and $x_n = y_p.f_n$, where x_p, y_p are pointer variables, x_n, y_n are numerical variables and f_n is a numerical field. We let s_{np} range over the two extra assignments statements belonging to WHILE_{np}. Finally, we use meta-variable s to range over all the statements of WHILE_{np}, *i.e.*, s_n, s_p and s_{np}.

The following syntactical categories of these languages will be used in this thesis. We write Var_n, Var_p, Fld_n and Fld_p for the variables and fields of type n or p. We write *Ref* for the set of concrete references of program memories. It is supposed to be an infinite enumerate set.

2.3 Elements of Abstract Interpretation

Abstract interpretation, introduced in the late 1970's [24] by P. Cousot and R. Cousot, is a framework of semantics approximation. We briefly review the major terminology of this theory.

Informally, abstract interpretation aims to construct two different meanings for a programming language where the first gives the usual meaning of programs in the language, and the second can be used to answer certain questions about the runtime behavior of programs in the language. The standard meaning of programs, called *concrete semantics*, can be typically described by their input-output function, and the standard interpretation will then be a function which maps programs to their input-output functions. The abstract meaning, called *abstract semantics* will be defined by a function which maps programs to mathematical objects of a particular universe, called *abstract semantics domain*.

Mathematically, the semantics of a program P can often be expressed by a least fixpoint $\mathrm{lfp}\ \mathsf{t}\,[\![P]\!]$ that is the least solution to a *constraint system* $X = \mathsf{t}\,[\![P]\!]\,(X)$ computed on a complete lattice.

Example 2.3.1. Consider a loop that increments the value of x:

```
1   x=0;
```

```
2    while (x<10){
3            x = x+1;
4    }
```

To infer possible values of x before each program point (from 1 to 5), we can construct the following constraint system.

$$
\begin{aligned}
X_1 &= \varnothing \\
X_2 &\supseteq \{0\} \cup X_4 \\
X_3 &\supseteq X_2 \cap (-\infty, 10) \\
X_4 &\supseteq \{x+1 | x \in X_4\} \\
X_5 &\supseteq X_2 \cap [10, \infty)
\end{aligned}
$$

The analysis of this problem amounts to solving the least fixpoint of the constraints system on the domain of $\Pi_{i=1}^{5}(X_i \to \mathit{Intv})$, in which Intv is the set of intervals.

The soundness of the abstract semantics is described using a *concretization function* $\gamma : A^\sharp \to A^\flat$, giving the meaning of the abstract elements in terms of concrete elements. We say that the abstract semantics $\mathsf{lfpt}^\sharp [\![P]\!]$ is sound with respect to the concrete semantics $\mathsf{lfpt}^\flat [\![P]\!]$, or say that the latter is approximated by the former, if $\mathsf{lfpt}^\flat [\![P]\!] \sqsubseteq^\flat \gamma(\mathsf{lfpt}^\sharp [\![P]\!])$. In this paper, we frequently verify a stronger *soundness condition* in the form of

$$\mathsf{t}^\flat [\![P]\!] \circ \gamma \sqsubseteq^\flat \gamma \circ \mathsf{t}^\sharp [\![P]\!] \tag{2.1}$$

By "being sound", we always refer to *partial soundness*, *i.e.*, if P terminates, then (2.1) holds.

We introduce the concept of *Galois connection*.

Definition 2.3.1 (Galois connection)**.** *Consider two posets* $(A^\flat, \sqsubseteq^\flat)$ *and* $(A^\sharp, \sqsubseteq^\sharp)$*. If functions* $\alpha : A^\flat \to A^\sharp$ *and* $\gamma : A^\sharp \to A^\flat$ *satisfy, for each* $a^\flat \in A^\flat$ *and* $a^\sharp \in A^\sharp$*,*

$$a^\flat \sqsubseteq^\flat \gamma(a^\sharp) \quad \textit{iff.} \quad \alpha(a^\flat) \sqsubseteq^\sharp a^\sharp$$

then the quadruple

$$(A^\flat, \alpha, \gamma, A^\sharp)$$

is called a Galois connection*.*

In terms of abstract interpretation, the sets $A^\flat$, $A^\sharp$ are often called *concrete domain* and *abstract domain* respectively, and the functions $t^\flat \in A^\flat \to A^\flat$, $t^\sharp \in A^\sharp \to A^\sharp$ are called (global) concrete transfer function and (global) abstract transfer function.

Example 2.3.2 (Cartesian abstraction)**.** Given 2 posets A and B, then we have the Galois connection $(\wp(A \times B), \alpha^\times, \gamma^\times, \wp(A) \times \wp(B))$ where

$$\alpha^\times \triangleq \lambda R.(\mathsf{post}[\mathsf{fst}]\ R, \mathsf{post}[\mathsf{snd}]\ R) \tag{2.2}$$

$$\gamma^\times \triangleq \lambda(A_0, B_0).A_0 \times B_0 \tag{2.3}$$

Example 2.3.3 (Composition of Galois connections)**.** Given 2 Galois connections $(A^\flat, \alpha_1, \gamma_1, A^\natural)$ and $(A^\natural, \alpha_2, \gamma_2, A^\sharp)$, then $(A^\flat, \alpha_3, \gamma_3, A^\sharp)$ is also a Galois connection, with $\alpha_3 \triangleq \alpha_2 \circ \alpha_1$ and $\gamma_3 \triangleq \gamma_1 \circ \gamma_2$.

Theorem 2.3.1 (Approximation of Fixpoint [25])**.** *Given two complete lattices $(A^\flat, \sqsubseteq^\flat)$ and $(A^\sharp, \sqsubseteq^\sharp)$ and the Galois Connection $(A^\flat, \alpha, \gamma, A^\sharp)$. Let $t^\flat$ and $t^\sharp$ be monotonic functions defined respectively on $A^\flat$ and $A^\sharp$. If the condition*

$$t^\flat \circ \gamma \sqsubseteq^\flat \gamma \circ t^\sharp$$

holds, then we have an approximation of the least fix point of $t^\flat$ by the least fix point of $t^\sharp$:

$$\mathrm{lfp} t^\flat \sqsubseteq \gamma(\mathrm{lfp} t^\sharp)$$

The computation of $\mathrm{lfp} t^\sharp$ is problem-dependent: if the iterates $t^{\sharp k}(\bot^\sharp)$ for $k = 0, 1 \ldots$, started from some initial $\bot^\sharp$ become eventually stable ($A^\sharp$ is said to enjoy the *ascending chain condition*), then $\mathrm{lfp} t^\sharp$ can be computed using brute force. This is a typical case for data-flow analysis. In case that the iterates converge slowly or do not converge, the algorithm to compute the fix point of $t^\sharp$ may involve an extrapolation strategy. In [24], Cousot introduced an operator called *widening* to guarantee fast termination of fix point computation.

Definition 2.3.2. *A widening $\triangledown$ is an operator of type $A^\sharp \times A^\sharp \to A^\sharp$ such that*

$$\forall a_1^\sharp, a_2^\sharp \in A^\sharp : a_1^\sharp \sqsubseteq^\sharp a_1^\sharp \triangledown a_2^\sharp \wedge a_2^\sharp \sqsubseteq^\sharp a_1^\sharp \triangledown a_2^\sharp$$

and for all increasing chains $a_0^\sharp \sqsubseteq^\sharp a_1^\sharp \sqsubseteq^\sharp \ldots$, *the increasing chain defined by*

$$w_0 = a_0^\sharp, w_1 = w_0 \triangledown a_1^\sharp \ldots w_{i+1} = w_i \triangledown a_{i+1}^\sharp$$

is not strictly increasing.

Theorem 2.3.2 (Kleene iteration with widening [24])**.** *The following iteration sequence*

$$X_0 = \bot^\sharp$$
$$X_{i+1} = \begin{cases} X_i, & \text{if } t^\sharp(X_i) \sqsubseteq^\sharp X_i \\ X_i \triangledown t^\sharp(X_i) & \text{otherwise} \end{cases}$$

is ultimately stationary and its limit is a post-fixpoint for $t^\sharp$.

2.4 Static Numerical Analysis

The target language of this static analysis is WHILE_n. The tracked information is called *numerical properties*. We distinguish two kinds:

- *Global numerical properties* refer to properties related to the whole program, including program execution time, consumed memories. An example is the static worst-case execution time (WCET) analysis. It is remarkably difficult to determine tight WCET bounds due to hardware complications and architectural features like instruction pipelines. A well-known WCET analyzer is aiT by AbsInt[1].

- *Local numerical properties* are those associated with program identifiers, in particular program variables. This category of analysis is demanded for the automatic detection of some well-known run-time errors like *division by zero* or *array index out of bound*. The algorithm developed by Karr in 1976 computes for each program control point the affine relations that hold among the program variables whenever the control point is reached [44]. An affine relation is a property of the form

[1] http://www.absint.com

$\Sigma_{i=1}^{k} c_i\ x_i = c$ where x_i are program variables and c_i, c are constant. In 1978, Cousot and Halbwachs [26] presented an eminent generalization of Karr's approach. They introduced the theory of *abstract interpretation*, and brought the designing of various *numerical abstract domains* into the mainstream. By using polyhedra instead of affine relations as space of approximation, their analysis allows us to specify programs with affine inequalities $\Sigma_{i=1}^{k} c_i\ x_i \leqslant c$.

This thesis considers the second category of numerical properties. We use the term *numerical property*, for any conjunction of formulae in some decidable theory of arithmetic. A numerical property can be loosely seen as a geometric shape. For example, the numerical property $\{x^2 + y^2 \leqslant 1, \quad x \leqslant 0, \quad y \leqslant 0\}$ is composed of the conjunction of three arithmetic formulae, representing a quart of the unit disc. Each formula of a numerical property is assumed to be quantifier-free. The constant values in the formula are integers.

Certain classes of numerical properties with a uniform geometric feature are called *abstract numerical domains*. The "interval", "octagon", or "polyhedral" abstract domains are thus named after their represented geometric shapes. In this paper, an abstract numerical domain is considered as a subset of the universe of numerical properties.

As usual, an *environment* is a partial mapping from program variables to their associated values. In our context, we consider *numerical environment* of integer values,

$$Num \triangleq Var_n \rightarrow \mathbb{Z}_\perp$$

where Var_n is the set of scalar variables holding numerical values.

The relationship between a numerical environment n and a numerical property $\bar{\mathsf{n}}$ is formalized by the concept of *valuation*. We say that n is a valuation of $\bar{\mathsf{n}}$, denoted by $\mathsf{n} \models \bar{\mathsf{n}}$, if $\bar{\mathsf{n}}$ becomes a tautology after each of its free variables, if any, has been replaced by its corresponding value in n.

Definition 2.4.1 (Interface of the traditional numerical analyzer)**.**

$$(\mathsf{WHILE}_n, \wp(Num), [\![\cdot]\!]_n^\natural, \gamma_n, Num^\sharp, [\![\cdot]\!]_n^\sharp)$$

The concrete numerical domain *and the* abstract numerical domain *for the language* WHILE_n *are respectively* $\wp(Num)$ *and* $Num^\sharp$*. They are related by the* concretization function $\gamma_n : Num^\sharp \to \wp(Num)$ *defined by*

$$\gamma_n(\bar{\mathsf{n}}) = \{n \in Num \mid n \models \bar{\mathsf{n}}\} \tag{2.4}$$

The partial order $\sqsubseteq$ *is consistent with the monotonicity of* γ_n, i.e., $\bar{\mathsf{n}}_1 \sqsubseteq \bar{\mathsf{n}}_2$ *implies* $\gamma_n(\bar{\mathsf{n}}_1) \subseteq \gamma_n(\bar{\mathsf{n}}_2)$*. For each statement* s_n *of* WHILE_n*, the concrete semantics* $[\![s_n]\!]_n^\natural$ *is assumed to be the powerset lifting* $[\![s_n]\!]_n^\natural \triangleq \mathsf{post}[\xrightarrow{Num}(s_n)]$ *of some standard operational semantics:*

$$\xrightarrow{Num} : \mathsf{WHILE}_n \to \wp(Num \times Num) \tag{2.5}$$

The abstract semantics $[\![\cdot]\!]_n^\sharp$ *satisfies the soundness condition:*

$$[\![\cdot]\!]_n^\natural \circ \gamma_n \subseteq \gamma_n \circ [\![\cdot]\!]_n^\sharp \tag{2.6}$$

At last, we assume the availability of a join operator $\sqcup$ *and a widening operator* $\triangledown$*. The join operator is assumed to be sound with regard to the partial order* $\sqsubseteq$*, and* $\triangledown$ *is assumed to be sound as specified in Sect. 4 of [23].*

2.5 Points-to Analysis

The imperative language WHILE_p provides basic pointer operations like dynamic allocation, pointer assignments, field store and field load. Classical *store-based semantics* models the memory as the *environment* and the *store*. Roughly speaking, variable assignment modifies the environment and the store is modified by indirect access of memory. The environment is most commonly thought of as a partial mapping from program variables to locations, and the store is specified by a partial mapping from locations to values. Conventionally, the model also needs to know the usage status of allocated locations. Each state of the store-based semantic domain used in this thesis is assumed to be garbage-free, namely, each allocated location is *reachable* in a sense that we shall make precise below.

The points-to analysis [30] is a dataflow analysis widely used for the static pointer analysis. The essential idea of points-to analysis is to partition

the concrete memory references *Ref* into a finite set of abstract references H, and then summarize the run-time pointer relations via elements of H and program variables. The result of the analysis is often expressed by a graph-like structure, called *points-to graph*. The memory partition process mentioned above is sometimes called a *naming scheme*. A popular naming scheme, known as k-CFA [63], is based on the k most recent call sites on the stack of the thread creating the object. Pointer analyses have been surveyed by numerous authors [37, 57, 59]. The 5-page survey of Hind and Pioli [41] is mostly cited; different axes balancing between efficiency and effectiveness are identified, with so called *equality-based* [66], *subset-based* [1] and *flow-sensitive* [18] variations. several directions for the then- future research are also discussed: How to improve the efficiency without affecting scalability or vice-versa, how to design an analysis for a client's needs, are flow-sensitive or context-sensitive analyses worth more investigation, which heap modeling shall we choose, etc.

For type-safe languages like Java, the flow-insensitive analysis is of polynomial complexity [13], but the analysis is difficult in general. The NP-hardness of a flow-insensitive analysis is shown by Horwitz [42] for programs without dynamic memory allocation and when all the variables are scalars and arbitrary number of dereferencing is allowed. Many techniques have been proposed to optimize points-to analyses. The online cycle elimination of Fahndrich *et al.* [32] represents points-to analysis as a graph problem and collapse cycles into single nodes since each element of the cycle has the same points-to information. *Lazy Cycle Detection* proposed by Hardekopt and Lin [38, 39] find the cycles using heuristics, so that the complexity overhead of Fahndrich can be greatly reduced.

Another dimension that improves points-to analysis is by using efficient data structures. In particular, BDD[11] was found to be much more space-efficient than traditional storage of points-to information[75]. This finding was then exploited by Berndl *et al.* [4] and Whaley and Lam [73] for efficient points-to analysis algorithms using BDDs for Java.

The challenge of points-to analysis, as in other static analysis, is to improve the precision of analysis without sacrificing the scalability. Lhotak and Chung [49] propose a *Strong Update analysis* combining both features: it is efficient like flow-insensitive analysis, with the same worst-case bounds,

yet its precision benefits from strong updates like flow-sensitive analysis. The key insight is that strong updates are applicable when the dereferenced points-to set is a singleton, and a singleton set is cheap to analyze. Hence the analysis focuses on flow sensitivity on singleton sets. Larger sets, which will not lead to strong updates, are modeled flow insensitively to maintain efficiency. De and D'Souza [27] propose to represent points-to information as maps, rather than *points-to graph* from access paths to sets of abstract objects.Their approach is similar to the classic *k-limiting* approach which truncate analysis targets by a predefined bound k: Their method finally leads to a flow-sensitive pointer analysis algorithm for Java that can perform strong updates on heap-based pointers. Recently, Khedker, Mycroft and Rawat [45] propose a *lazy* points-to analysis based on *liveness analysis*. They argue that the vast majority of points-to pairs calculated by existing algorithms are never used by any client analysis or transformation because they involve dead variables. They reformulate a flow- and context- sensitive points-to analysis in terms of a joint points-to and liveness analysis so that potentially unused points-to relations will not be computed.

Concrete semantics We assume that a *naming scheme* can be interfaced with a function

$$\triangleright \in Ref \rightarrow H \tag{2.7}$$

In this presentation, we use a simple and standard naming scheme to name heap elements after the program point of the statement that allocates them (which is typical for the *context-insensitive* variant of points-to analysis). The elements of H will also be called *allocation sites* or *abstract references*.

Let Var_p, Ref, and Fld_p be the set of pointer variables, references, and fields for pointer references. A *state* σ of the store-based semantics is a pair of partial mappings ρ from Var_p to Ref, called *environments*, and partial mappings $\hbar$ from $Ref \times Fld_p$ to Ref, called *stores*. The store-based semantics domain will be denoted by $Pter$.

$$Pter \triangleq \{(\rho, \hbar) \mid \rho \in Var_p \rightarrow Ref_\perp, \hbar \in Ref \times Fld_p \rightarrow Ref_\perp\}$$

Given $(\rho, \hbar) \in Pter$, we say $r \in Ref$ is *reachable* if there exists $x \in Var_p$ such that $\rho(x) = r$, or there exists some reachable $r' \in R$ and $f \in Fld_p$

s.t. $\hbar(r', f) = r$. The state $(\rho, \hbar)$ is called *garbage-free* if each *reference* in $\{r \in Ref \mid (r, f) \in dom(\hbar)\}$ is reachable.

The concrete semantics domain is defined to be the collection of subsets of garbage-free states in $Pter$.

The effect of a statement s_p of WHILE_p can be modeled as the operational semantics on $Pter$. We write $\langle s_p, \sigma\rangle \xrightarrow{Pter} \sigma'$ if σ is the state before s_p then σ' can be a state after s_p *under the condition that* s_p *terminates*. Since we are only interested in garbage-free states, below we assume the operator of garbage collection is available, denoted by gc

$$\langle x = \texttt{null}, (\rho, \hbar)\rangle \xrightarrow{Pter} gc(\rho[x \to \bot], \hbar) \tag{2.8}$$

$$\langle x = \texttt{new}, (\rho, \hbar)\rangle \xrightarrow{Pter} gc(\rho[x \to r_{fresh}], \hbar) \textit{ for } r_{fresh} \notin \textit{reachable}(\rho, \hbar) \tag{2.9}$$

$$\langle x = y, (\rho, \hbar)\rangle \xrightarrow{Pter} gc(\rho[x \to \rho(y)], \hbar) \tag{2.10}$$

$$\langle x = y.f, (\rho, \hbar)\rangle \xrightarrow{Pter} gc(\rho[x \to \hbar(\rho(y), f)], \hbar) \textit{ for } (\rho(y), f) \in dom(\hbar) \tag{2.11}$$

$$\langle x.f = y, (\rho, \hbar)\rangle \xrightarrow{Pter} gc(\rho, \hbar[(\rho(x), f) \to \rho(y)]) \textit{ for } \rho(x) \neq \bot \tag{2.12}$$

$$\langle x == y, (st, \hbar)\rangle \xrightarrow{Pter} gc(\rho, \hbar) \textit{ for } \rho(x) = \rho(y) \tag{2.13}$$

$$\langle x \neq y, (st, \hbar)\rangle \xrightarrow{Pter} gc(\rho, \hbar) \textit{ for } \rho(x) \neq \rho(y) \tag{2.14}$$

The concrete semantics is defined to be the powerset lifting of the operational semantics.

Abstract semantics Let $\mathsf{p}^\sharp$ be a graph-like data structure composed of two kinds of arcs: $x \to h$ and $h' \xrightarrow{f} h$, where x and f range over variables and fields in WHILE_p and h', h range over abstract references of the underlined naming scheme. Let us call this data structure *points-to graph* and denote by $\texttt{arc}(\mathsf{p}^\sharp)$ for the set of its arcs.

Given a set of concrete states $\tilde{\mathsf{p}} \in \wp(Pter)$, its abstraction can be processed as follows: whenever there exists an $(\rho, \hbar) \in \tilde{\mathsf{p}}$ s.t. $\rho(x) = r$ and $r \rhd h$ for some variable x, concrete reference r and abstract reference h, there must be an arc $x \to h$ in the points-to graph; if $\hbar(r', f) = r$ for some concrete state $(\rho, \hbar)$ and $r' \rhd h'$ and $r \rhd h$ for abstract references h, h' , then there must be an arc $h' \xrightarrow{f} h$ in the points-to graph. Because H, Fld_p, Var_p are assumed

to be finite set, there esists a *smallest* points-to graph that abstracts a given subset of $Pter$. The set of these points-to graphs can be defined as

$$Pter^\sharp \triangleq (Var_p \times Fld_p) \times (H \times Fld_p \times H)$$

It can be shown that this smallest, or called *best* abstraction in the terminology of abstract interpretation, consists of a points-to graph without garbage. By abuse of language, we write $Pter^\sharp$ for garbage-free points-to graphs.

The relationship between $Pter^\sharp$ and its concrete counterpart $\wp(Pter)$ can be formalized with the concretization function γ_p defined as

$$\begin{aligned}\gamma_p(\mathsf{p}^\sharp) \triangleq \{(\rho, \hbar) \in Pter \mid \rho(x) = r, r \rhd h \Rightarrow x \to h \in \mathtt{arc}(\rho, \hbar)\\ \hbar(r', f) = r \wedge r' \rhd h' \wedge r \rhd h \Rightarrow h' \xrightarrow{f} h \in \mathtt{arc}(\rho, \hbar)\}\end{aligned} \quad (2.15)$$

The *abstract semantics* of points-to analysis is usually specified in the style of constraints system. Below, let $\mathsf{p}^\sharp \ni a$ a shortcut for $a \in \mathtt{arc}(\mathsf{p}^\sharp)$. The constraint system can be specified as

$$\text{x=y}\frac{\mathsf{p}^\sharp \ni y \to h}{\mathsf{p}^\sharp \ni x \to h} \qquad \text{x=y.f}\frac{\mathsf{p}^\sharp \ni y \to h' \quad \mathsf{p}^\sharp \ni h' \xrightarrow{f} h}{\mathsf{p}^\sharp \ni x \to h} \quad (2.16)$$

$$\text{x=}\mathtt{new}\frac{}{\mathsf{p}^\sharp \ni x \to r_{fresh}} \qquad \text{x.f=y}\frac{\mathsf{p}^\sharp \ni x \to h' \quad \mathsf{p}^\sharp \ni x \xrightarrow{f} h}{\mathsf{p}^\sharp \ni h' \xrightarrow{f} h} \quad (2.17)$$

$$(2.18)$$

We write $[\![\cdot]\!]_p^\sharp$ for the abstract transfer function derived from the constraint system above.

Definition 2.5.1. *The points-to analysis can be defined as the tuple*

$$(\mathsf{WHILE}_p, \wp(Pter), [\![\cdot]\!]_p^\natural, \gamma_p, Pter^\sharp, [\![\cdot]\!]_p^\sharp)$$

Chapter 3

Lifting Numerical Abstract Domains to Heap-manipulating Programs

3.1 Introduction

The static analysis of numerical properties of program variables can draw on a rich body of techniques including abstract domains of intervals [21], polyhedron [26], octagons [56] which have found their way into mature implementations. In a similar way, the analysis of properties describing the shape of data structures in the heap has flourished into a rich set of points-to and alias analyses which also have provided a range of production-quality analyzers. However, these two types of analyses do not always integrate so well. Numerical properties such as $x.v + y.w \leqslant a[i]$ (where $x.v$ and $y.w$ are Java field references of type `int` and $a[i]$ is an array reference of type `int`) are *alien* [**?**] to traditional numerical domains and would thus be coarsely over-approximated as *unknown*, representing no information.

When extending numerical analyses to entities such as $x.v$ we are immediately faced with the problem that pointers introduce *aliases* which make program reasoning harder. As an example, consider the effect of the assign-

Table 3.1: Post-conditions of $a.val = b.val + c.val$, assuming $b.val \in [3,6], c.val \in [4,8]$. Columns 2-6 show 5 aliasing relations between 3 variables. Column 7 joins the results.

	$a/b/c$	ab/c	ac/b	bc/a	abc	join
a.val	[7,14]	[7,14]	[7,14]	[8,12]	[8,12]	[7,14]
b.val	[3,6]	[7,14]	[3,6]	[4,6]	[8,12]	[3,14]
c.val	[4,8]	[4,8]	[7,14]	[4,6]	[8,12]	[4,14]

ment

$$a.val = b.val + c.val \tag{3.1}$$

The variables a, b, c are bound to objects with the numerical field val. Assuming that $b.val \in [3,6]$ and $c.val \in [4,8]$ hold before the statement, we can derive different properties for their values after the assignment depending on the knowledge we have about aliasing between the references a, b and c.

In particular, the values of $b.val$ or $c.val$ may be updated if the condition $a = b$ or $a = c$ holds before the statement. The following approach considers the potential aliases among variables a, b and c. There are five possible alias relations, as shown on columns 2-6 of the first row in Tab. 3.1, where we use '/' to mean the partitions of variables induced by aliasing. For example, in the case of ab/c, the alias relation is $a = b \neq c$, and thus $a.val$, $b.val$ are two names that must be updated simultaneously. We obtain $a.val \in [7,14], b.val \in [7,14]$ and $c.val \in [4,8]$. The last column of the table shows the post-conditions by joining the results in columns 2-6.

Analyzing the statement for every possible alias relation between variables in turn and taking the conservative join of obtained results gives a sound result. However, this naive approach is not feasible as the number of aliasing relations among N variables quickly becomes large[1]. A better solution is to combine traditional static numerical analysis with points-to analyses that can provide information about aliasing relations and hence rule out some

[1]The number of aliasing relations is the number of partitions of a n-element set (known as the *Bell number*) and is asymptotically $O(\epsilon^n n!)$ for any positive ϵ [12]

spurious aliases. This chapter is concerned with developing a theoretical foundation for combining pointer analysis with static numerical analysis.

3.1.1 Objectives and Contributions

The goal is not to define new pointer and numerical analyses but to provide the necessary theory for interfacing existing analyses with each other. We shall be following the methodology of abstract interpretation [25] when constructing the theory. The contributions of the paper are both theoretical and practical. On the theoretical side, we propose a new abstract domain combining traditional static numerical domains and points-to analysis. The abstract domain is constructed in three steps:

1. the first is a lattice isomorphism in which the references in the heap part of the state are re-injected into (and hence made explicit in) the numerical part of the state,

2. the second is a Cartesian (attribute-independent) abstraction [25] of the numerical and the heap part of the state,

3. the third is the application of the abstractions of the existing domains.

Thus, it is the first step that makes the combination possible, by preparing the re-use of the abstract pointer values when extending the numerical domains to cover properties about heap values. We define and prove the correctness of the transfer functions for this new combined domain.

On the practical side, we have experimented with the combination of several existing domains by implementing a combined static numerical and pointer analysis, using the Java Optimization Framework SOOT [70] as the front-end, and relying on the abstract domains from existing static analysis libraries such as the Parma Polyhedra Library PPL [2] and the SOOT Pointer Analysis Research Kit SPARK [50]. This prototype analyzer, called NumP, has been run on programs in the `Dacapo-2006-MR2` [7] benchmark suite. The largest among them, *chart*, has several hundreds of KLOC in Jimple [71]. Our experiments confirm that a combined analysis is feasible even for large-sized programs and that it discovers significantly more program properties than what is possible by pure numerical analysis, and this

at a cost that is comparable to the cost of running the numerical and pointer analysis separately.

In addition, the goal of modular re-use of static analyses has been attained as the implementation of our prototype is mainly based on the existing implementations of traditional numerical and pointer analyses. We have instanced NumP with a context-insensitive and a context sensitive points-to analyses on one side, and an interval and a polyhedral abstract domains on the other side.

Notation Let A, B be two sets. Given a relation $R \subseteq A \times B$, we write $\mathsf{post}[R] \in \wp(A) \to \wp(B)$ for the function $\lambda A_1.\{b \mid \exists a \in A_1 : (a,b) \in R\}$. We use fst and snd as the operators that extract the first and the second components of a pair respectively. For a given set U, the notation $U_\perp$ means the disjoint union $U \cup \{\perp\}$. Given a mapping $m \in A \to B_\perp$, we express the fact that m is undefined in a point x by $m(x) = \perp$. The set of integers is denoted by $\mathbb{Z}$. We write "$\triangleq$" for "defined as".

3.2 Semantics Abstraction

The store-based semantics for heap reasoning is standard. We follow the notations of [60], in which a state keeps track of the allocated references $A \in \wp(Ref)$, and a pair of an environment ρ and a heap hp.

$$\begin{aligned}(A, \rho, hp) \in State = \wp(Ref) \times & \overbrace{(Var_n \to \mathbb{Z}_\perp) \times (Var_p \to Ref_\perp)}^{Env} \\ & \times \underbrace{((Ref \times Fld_n) \to \mathbb{Z}_\perp) \times ((Ref \times Fld_p) \to Ref_\perp)}_{Heap}\end{aligned}$$

Write $A^\flat$ for the powerset of $State$. In the context of WHILE_{np}, the variables and fields are typed. Let $\longrightarrow^\flat$: $S \to \wp(State \times State)$ denote its *structural operational semantics* (omitted).

$$\frac{\langle s_n, \mathsf{n}\rangle \xrightarrow{Num} \mathsf{n}'}{\langle s_n, (\mathsf{n}, \mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}', \mathsf{p})} \qquad \frac{\mathsf{p}(y_p) = r \quad d = (r, f_n) \quad \langle d = x_n, \mathsf{n}\rangle \xrightarrow{Num} \mathsf{n}'}{\langle y_p.f_n = x_n, (\mathsf{n}, \mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}', \mathsf{p})}$$

$$\frac{\langle s_p, \mathsf{p}\rangle \xrightarrow{Pter} \mathsf{p}'}{\langle s_p, (\mathsf{n}, \mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}, \mathsf{p}')} \qquad \frac{\mathsf{p}(y_p) = r \quad d = (r, f_n) \quad \langle x_n = d, \mathsf{n}\rangle \xrightarrow{Num} \mathsf{n}'}{\langle x_n = y_p.f_n, (\mathsf{n}, \mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}', \mathsf{p})}$$

Figure 3.1: Structural Operational semantics $\widetilde{\longrightarrow}^{\flat} : \mathsf{WHILE}_{np} \to (\widetilde{State} \times \widetilde{State})$

3.2.1 An Isomorphic Operational Semantics

The lemma below shows that we can express the structural operational semantics (SOS for short) of WHILE_{np} in terms of the SOSs of WHILE_n and WHILE_p.

Lemma 3.2.1 (Isomorphic store-based semantics of WHILE_{np})**.** *Let Num and Pter be the sets of concrete states of* WHILE_n *and* WHILE_p*, as specified in Chap. 2. Let D denote the set of the pairs of concrete references in Ref and the numerical fields in* Fld_n*.*

$$D \triangleq Ref \times Fld_n \tag{3.2}$$

Let $Num[D \cup Var_n]$ *be the set that extends Num over* $D \cup Var$*,* i.e.*,* $Num[D \cup Var_n] \triangleq (D \cup Var_n) \to \mathbb{Z}_\perp$*. Then a concrete state of* WHILE_{np} *is also an element in* $\widetilde{State}$*, with*

$$\widetilde{State} \triangleq Num[D \cup Var_n] \times Pter \tag{3.3}$$

In addition, The SOS of WHILE_{np}*, denoted by* $\widetilde{\longrightarrow}^{\flat}$*, can be expressed by the SOSs of* WHILE_n *and* WHILE_p*. (Fig. 3.1, in which* $\xrightarrow{Pter}$ *is the SOS of* WHILE_p*, and* $\xrightarrow{Num}$ *is the SOS of* WHILE_n *over* $D \cup Var_n$*).*

Let $\widetilde{A^{\flat}}$ be the power-set of $\widetilde{State}$, we define the *collecting semantics* [22] as the lifting of the operational semantics $\widetilde{\longrightarrow}^{\flat}(s)$ to power-sets, *i.e.*, $[\![s]\!]^{\tilde{\flat}} \triangleq \mathsf{post}[\widetilde{\longrightarrow}^{\flat}(s)]$.

3.2.2 Cartesian Abstraction

Lemma 3.2.2 (Cartesian Abstraction)**.** *Let* $A^\natural \triangleq \wp(Num[D \cup Var_n]) \times \wp(Pter)$ *and* $(\widetilde{A^\flat}, \alpha^\times, \gamma^\times, A^\natural)$ *be the Cartesian abstraction [25],* i.e., $\alpha^\times \triangleq \lambda R : \widetilde{A^\flat}.(\mathsf{post}[\mathsf{fst}]\ R, \mathsf{post}[\mathsf{snd}]\ R)$ *and* $\gamma^\times \triangleq \lambda(A_0, B_0) : A^\natural.A_0 \times B_0$.

The transfer functions $[\![\cdot]\!]^\natural : \mathsf{WHILE}_{np} \to (A^\natural \to A^\natural)$ *defined in Fig. 3.2 is the* best transformer *[25] of* $[\![\cdot]\!]^{\tilde{\flat}}$*, that is,* $\forall s \in \mathsf{WHILE}_{np} : [\![s]\!]^\natural = \alpha^\times \circ [\![s]\!]^{\tilde{\flat}} \circ \gamma^\times$.

$$[\![s_p]\!]^\natural(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq (\tilde{\mathsf{n}}, [\![s_p]\!]^\natural_p(\tilde{\mathsf{p}})) \qquad [\![s_n]\!]^\natural(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq ([\![s_n]\!]^\natural_n(\tilde{\mathsf{n}}), \tilde{\mathsf{p}})$$

$$[\![x_n = y_p.f_n]\!]^\natural(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq \left(\left(\bigcup_{\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d} [\![x_n = d]\!]^\natural_n(\tilde{\mathsf{n}})\right), \tilde{\mathsf{p}}\right)$$

$$[\![y_p.f_n = x_n]\!]^\natural(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq \left(\left(\bigcup_{\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d} [\![d = x_n]\!]^\natural_n(\tilde{\mathsf{n}})\right), \tilde{\mathsf{p}}\right)$$

Figure 3.2: $[\![\cdot]\!]^\natural : \mathsf{WHILE}_{np} \to (A^\natural \to A^\natural)$. The notation $\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d$ means that $d \in \{(\mathsf{p}(y_p), f_n) \mid \mathsf{p} \in \tilde{\mathsf{p}}\}$

Proof for the case of $y_p.f_n = x_n$.

$$\begin{aligned}
&\alpha^\times \circ [\![y_p.f_n = x_n]\!]^{\tilde{\flat}} \circ \gamma^\times(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \\
=&\alpha^\times \circ [\![y_p.f_n = x_n]\!]^{\tilde{\flat}}(\tilde{\mathsf{n}} \times \tilde{\mathsf{p}}) && \text{(Def. } \gamma^\times) \\
=&\alpha^\times(\{\widetilde{\longrightarrow}^\flat(y_p.f_n = x_n)(\mathsf{n}, \mathsf{p}) \mid (\mathsf{n}, \mathsf{p}) \in \tilde{\mathsf{n}} \times \tilde{\mathsf{p}}\}) && \text{(Def. } [\![\cdot]\!]^{\tilde{\flat}}) \\
=&\alpha^\times(\{(\xrightarrow{Num}(d = x_n)(\mathsf{n}), \mathsf{p}) \mid (\mathsf{n}, \mathsf{p}) \in \tilde{\mathsf{n}} \times \tilde{\mathsf{p}}, d = (\mathsf{p}(y_p), f_n)\}) && \text{(Def. } \widetilde{\longrightarrow}^\flat) \\
=&(\{\xrightarrow{Num}(d = x_n)(\mathsf{n}) \mid (\mathsf{n}, \mathsf{p}) \in \tilde{\mathsf{n}} \times \tilde{\mathsf{p}}, \tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d\}, \tilde{\mathsf{p}}) && \text{(Def. } \alpha^\times) \\
=&[\![y_p.f_n = x_n]\!]^\natural(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) && \text{(Def } [\![\cdot]\!]^\natural)
\end{aligned}$$

□

3.2.3 The Abstract Domain *NumP*

Definition 3.2.1 (Symbolic variable)**.** *A symbolic variable δ is a pair of abstract reference $h \in H$ and numeric field $f_n \in Fld_n$. The set of symbolic variables is denoted by Δ.*

$$\Delta \triangleq H \times Fld_n \tag{3.4}$$

The role of *symbolic variables* is formalized via the notion of *instantiation*.

Definition 3.2.2 (Instantiation)**.**

$$\mathtt{Ins}_{\triangleright} \triangleq \{\sigma : \Delta \to D \mid \sigma(h, f_n) = (r, g_n) \Rightarrow h = \triangleright(r) \wedge f_n = g_n\} \tag{3.5}$$

Notation Let $Num^\sharp[\Delta \cup Var_n]$ and $Num^\sharp[D \cup Var_n]$ denote the extensions of $Num^\sharp$ over $\Delta \cup Var_n$ and $D \cup Var_n$ respectively. Given $\sigma \in \mathtt{Ins}_{\triangleright}$, we denote by $[\sigma]$ the capture-avoiding substitution operator of type $Num^\sharp[\Delta \cup Var_n] \to Num^\sharp[D \cup Var_n]$ that replaces all the free occurrences of δ in $\mathsf{n}^\sharp \in Num^\sharp[\Delta \cup Var_n]$ with $\sigma(\delta)$.

Definition 3.2.3. *The semantics of elements in $Num^\sharp[\Delta \cup Var_n]$ is defined by the concretization function $\gamma_\delta : Num^\sharp[\Delta \cup Var_n] \to \wp(Num[D \cup Var_n])$*

$$\gamma_\delta(\mathsf{n}^\sharp) \triangleq \{\mathsf{n} \in Num[D \cup Var_n] \mid \forall \sigma \in \mathtt{Ins}_{\triangleright}, \mathsf{n} \in \gamma_n \circ [\sigma](\mathsf{n}^\sharp)\}$$

Read it as follows: a numerical environment n over $D \cup Var_n$ is in the concretization of the numerical property $\mathsf{n}^\sharp$ over $\Delta \cup Var_n$ if and only if n is in the concretization of each instance of $\mathsf{n}^\sharp$.

Example 3.2.1. Consider $\mathsf{n}^\sharp \in Num^\sharp[\Delta \cup Var_n]$ to be a system of linear inequalities $AX \leqslant B$ with A and B being a numerical matrix and a vector respectively, and X is a vector on $\Delta \cup Var_n$. Without loss of generality, we write X as the vector $(\delta_1 \ldots \delta_m, z_1 \ldots z_l)$ for $\delta_i \in \Delta$ and $z_j \in Var_n$. Then $AX \leqslant B$ represents the conjunction of all $A\bar{X} \leqslant B$ in which $\bar{X}$ can be any $(d_1 \ldots d_m, z_1 \ldots z_l)$ in which z_i remains the same as in X and there exists an instantiation $\sigma \in \mathtt{Ins}_{\triangleright}$ s.t. $\sigma(\delta_i) = d_i$ for any $1 \leqslant i \leqslant m$.

Definition 3.2.4 (The abstract domain *NumP* and its concretization)**.** *The abstract domain NumP is defined to be*

$$NumP \triangleq Num^{\sharp}[\Delta \cup Var_n] \times Pter^{\sharp} \tag{3.6}$$

The concretization function $\gamma^{\shortparallel}$ *of type* $NumP \to A^{\natural}$ *is defined as* $\lambda(\mathsf{n}^{\sharp}, \mathsf{p}^{\sharp}).(\gamma_{\delta}(\mathsf{n}^{\sharp}), \gamma_p(\mathsf{p}^{\sharp}))$ *where* γ_p *is the concretization function of the underlying points-to analysis.*

Example 3.2.2. Revisit the program in Fig. 2.1 (right). A list of integers ranging from -5 to 2 is stored iteratively on the heap. At each iteration, a memory cell, bound to variable *elem*, is allocated. The cell consists of a numerical field *val* and a reference field *next*. The head of the list is always pointed to by variable *hd*.

Fig. 3.3 shows the memory states that arise at the loop entry (l. 3 of the source code) as well as the process of the semantics abstraction in three steps. The first row illustrates the concrete heap states. The second row is an isomorphic version that separates numerical and pointer information. The third row is the abstract state obtained by performing Cartesian abstraction over the second row. The last row shows the abstract state of our abstract domain. Note that each state, $(\mathsf{n}_k, \mathsf{p}_k)$ of the second row is a concretization of the abstract state $(\mathsf{n}^{\sharp}, \mathsf{p}^{\sharp})$. In particular, the $(h, val) \to [-5, 2]$ part is to be interpreted as: the numerical values stored at (r, val) must be in the range from -5 to 2, whenever the memory cell referred by r is allocated at h.

The semantics abstraction process is summarized in Fig. 3.4. Starting from the standard concrete store-based domain $A^{\flat}$, we find an isomorphic form $\widetilde{A^{\flat}}$. Then the Cartesian abstraction gives rise to a pair of well studied concrete domains of traditional numerical and points-to analyses. We then "plug in" the existing abstract domains and reuse those abstractions as black-boxes.

3.3 Transfer Functions

Let $(\mathsf{n}^{\sharp}, \mathsf{p}^{\sharp})$ be a state of *NumP*. We are concerned with how it should be updated by statements of WHILE_{np}. Let $[\![s]\!]^{\sharp}(\mathsf{n}^{\sharp}, \mathsf{p}^{\sharp})$ be the state just after

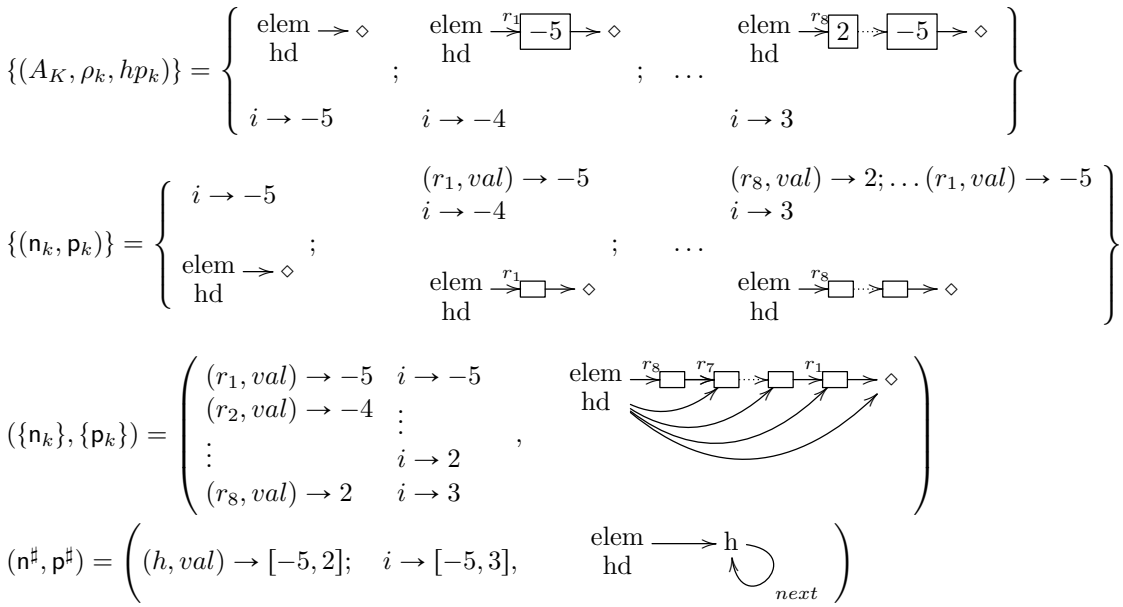


Figure 3.3: Semantics abstraction of memory states at the loop entry of the example program in Fig. 2.1 (right, l. 3). Heap locations are depicted as rectangles labeled by references. The value of each pointer variable is depicted as an arrow from the variable name to the referenced rectangle. The symbol $\diamond$ is for the `null` pointer. We have omitted the range $1 \leqslant k \leqslant 8$ of the script k occurring in the first three rows. The label for the field "next" on the directed edges is not drawn for the first three rows.

the execution of some statement s. Below, we explain how their abstract semantics should be defined following the three categories s_n, s_p, and s_{np} (See Chap. 2 for the three categories). Note that the points-to component of our abstraction is described in a flow-sensitive style but is relatively easy to be adapted for a flow-insensitive points-to analysis. The proof of soundness is sketched at the end of the section.

$$
\begin{array}{cc}
A^\flat \triangleq \wp(\wp(Ref) \times Env \times Heap) & \\
\downarrow\uparrow & \text{Isomorphism} \\
\widetilde{A^\flat} \triangleq \wp(Num[D \cup Var_n] \times Pter) & \\
\downarrow\uparrow & \text{Cartesian abstraction} \\
A^\natural \triangleq \wp(Num[D \cup Var_n]) \times \wp(Pter) & \\
\downarrow\uparrow & \text{Reuse existing domains} \\
NumP \triangleq Num^\sharp[\Delta \cup Var_n] \times Pter^\sharp &
\end{array}
$$

Figure 3.4: Semantics abstraction toward $NumP$ takes three steps.

Transfer function for s_n

It is sound to assume that assignments or assertions of numerical variables have no effect on the heap. If s_n is an assignment in WHILE_n, it can be treated in the same way as in traditional numerical analysis using its abstract transfer function $[\![\cdot]\!]^\sharp_n$ The transfer function for updating $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with s_n can be defined as:

$$[\![s_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq [\![s_n]\!]^\sharp_n\,\mathsf{n}^\sharp, \mathsf{p}^\sharp \tag{3.7}$$

If s_n is an assertion in WHILE_n, $\mathsf{p}^\sharp$ may be refined. For example, consider the *compound statement*[2] `if (a > 0) p = q` where p and q are reference variables and a is a numerical variable. Although it should be possible to perform a dead-code elimination using inferred numerical relations, similar to Pioli's conditional constant propagation [58], we still use the Eq. (3.7) for the ease of implementation.

Transfer function for s_p

It is also sound to assume that s_p has no effect upon $\mathsf{n}^\sharp$. Yet the reasoning is different from the above case. For example, if $\mathsf{n}^\sharp, \mathsf{p}^\sharp$ is the state shown on the last row of Fig. 3.3, how can we tell whether an assignment of pointers modifies $\mathsf{n}^\sharp$ or not? Recall that the intended semantics of $(h, val) \rightarrow [-5, 2]$

[2]This term is used here to be distinguished from basic statements as s_n, s_p or s_{np}. Note that s_n is the assertion, not the whole if-statement.

is that every values stored in each (r, val) satisfying $r \rhd h$ must be in the range of $[-5, 2]$. That is to say, $\mathsf{n}^\sharp$ represents a fact about the *numerical content* stored in the corresponded concrete references. A pointer assignment can by no means modify any numerical values stored in the heap. The algorithm to update $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with s_p can be written as:

$$[\![s_p]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \mathsf{n}^\sharp, [\![s_p]\!]^\sharp_p\,\mathsf{p}^\sharp \tag{3.8}$$

Transfer function of s_{np}

The transfer function for s_{np} is more interesting. Consider an assignment $x_n = y_p.f_n$. Assume that the state before the assignment is $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with $\mathsf{n}^\sharp = \{(h1, val) \to [0, 5], (h_2, val) \to [10, 20]\}$ and $\mathsf{p}^\sharp = \{(y_p, h_1), (y_p, h_2)\}$. Since y_p points to h_1, h_2 and thus $y_p.f_n$ is bound with a subset of values stored at (r, f_n) so that $r \rhd h_1$ or $r \rhd h_2$, we know that at run-time the assignment updates x_n to a value that is either in $[0, 5]$ or in $[10, 20]$. In the context of the non-relational abstract domain, the semantics of $x_n = y_p.f_n$ can be approximated by the join of the effects of the assignment of symbolic variables, $x_n = (h, f_n)$, for all h such that y_p points to h.

$$[\![x_n = y_p.f_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \left(\left(\bigsqcup_{\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta} [\![x_n = \delta]\!]^\sharp_n\,(\mathsf{n}^\sharp)\right), \mathsf{p}^\sharp\right) \tag{3.9}$$

where we write $\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta$ to mean that δ is some symbolic variable (h, f_n) with h pointed to by x_p. *i.e.*, $\exists h : \delta = (h, f_n) \wedge h \in \mathsf{p}^\sharp(x_p)$. Assume that $\mathsf{p}^\sharp(y_p.f_n)$ is a singleton $\{\delta\}$.

Now, consider an assignment $y_p.f_n = x_n$ with y_p pointing to the abstract h of the points-to graph. We regard $y_p.f_n = x_n$ as an assignment to symbolic variable $(h, f_n) = x_n$. By $(h, f_n) = x_n$, we actually mean that the field f_n of *one of the concrete objects represented by* h is to be updated to the value of x_n, while the other concrete objects represented by h remain unchanged. In practice, we rewrite the symbolic variable (h, f_n) as some (fictitious) scalar variable, say δ, and symbolically execute $\lambda c : c \sqcup [\![\delta = x_n]\!]^\sharp_n\,(c)$ using traditional numerical analyses, e.g. interval analysis, equipped with the abstract

semantics $[\![\cdot]\!]^\sharp_n$ of assignment and the abstract operator of join $\sqcup$.

$$[\![y_p.f_n = x_n]\!]^\sharp(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \left(\left(\bigsqcup_{\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta} \mathsf{n}^\sharp \sqcup [\![\delta = x_n]\!]^\sharp_n(\mathsf{n}^\sharp)\right), \mathsf{p}^\sharp\right) \tag{3.10}$$

Note that it is not necessary to compute transfer functions for assertions involving field expressions for they are transformed beforehand by our front-end SOOT to assertions in WHILE_n or in WHILE_p. For instance, a source code `if (x.f>0) ...`, is transformed to `a = x.f; if (a>0) ...` . before our analysis.

Join and Widening

The join of two facts is defined as the set of all facts that are implied independently by both. The join of $(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp{}_1)$ and $(\mathsf{n}^\sharp_2, \mathsf{p}^\sharp{}_2)$ is the join of $\mathsf{n}^\sharp_1$ and $\mathsf{n}^\sharp_2$, paired with the join of $\mathsf{p}^\sharp{}_1$ and $\mathsf{p}^\sharp{}_2$.

$$(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp{}_1) \sqcup^\sharp (\mathsf{n}^\sharp_2, \mathsf{p}^\sharp{}_2) = (\mathsf{n}^\sharp_1 \sqcup \mathsf{n}^\sharp_2, \mathsf{p}^\sharp{}_1 \cup \mathsf{p}^\sharp{}_2) \tag{3.11}$$

When computing the fixpoint, the iterates of our numerical points-to domain do not necessarily converge because of its numerical components. We perform a piecewise widening for the numerical part.

$$(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp{}_1) \triangledown^\sharp (\mathsf{n}^\sharp_2, \mathsf{p}^\sharp{}_2) = (\mathsf{n}^\sharp_1 \triangledown \mathsf{n}^\sharp_2, \mathsf{p}^\sharp{}_1 \cup \mathsf{p}^\sharp{}_2) \tag{3.12}$$

Theorem 3.3.1 (Soundness)**.** *The transfer functions* $[\![\cdot]\!]^\sharp : \mathsf{WHILE}_{np} \to (NumP \to NumP)$*, defined in* (3.7)*,* (3.8)*,* (3.9) *and* (3.10)*, are sound with respect to* $[\![\cdot]\!]^\natural$*: for any statement* s *of* WHILE_{np} *and abstract state* $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ *of* $NumP$*,* $[\![s]\!]^\natural \circ \gamma''(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \subseteq \gamma'' \circ [\![s]\!]^\sharp(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$*.*

3.4 Proof of Soundness

3.4.1 Preliminaries

Notations Given $\mathsf{n} \in Num$, its *definition domain* is denoted by $dom(\mathsf{n})$. Given $\mathsf{n}^\sharp \in Num^\sharp$, its *free variable*, denoted by $FV(\mathsf{n}^\sharp)$, is the union of the

free variables of each formula in $\mathsf{n}^\sharp$. The space of bijective functions from A to $\tilde{A}$ is denoted by by $A \leftrightarrow \tilde{A}$.

Definition 3.4.1 (Variable substitution)**.** *Given $n \in \mathsf{Num}$ and a* bijective function $\sigma : dom(n) \leftrightarrow \widetilde{dom}$ *from n's definition domain to some isomorphic $\widetilde{dom}$, we define the operator of* variable substitution*, written as $[\sigma]$, to be a mapping of type $Num \to Num$ defined as*

$$[\sigma] \triangleq \lambda \mathsf{n}.(\mathsf{n} \circ \sigma^{-1}) \tag{3.13}$$

The definition above requires that σ be bijective, and the domain of the mapping σ be the domain of the numerical environment n. This requirement makes the operator $[\sigma]$ a bijection.

Lemma 3.4.1. *Given $\mathsf{n} \in Num$, $\sigma \in dom(n) \leftrightarrow \widetilde{dom}$, we have $[\sigma]^{-1} = [\sigma^{-1}]$*

A substitution of numerical properties is to be understood as the usual capture-avoiding substitution in lambda logic. Again, this substitution will be specified by a bijective function. Although not necessary, we require the definition domain of the specified function be exactly the same as the set of the free variables of the considered numerical property.

Definition 3.4.2 (Substitution of numerical properties)**.** *Let $\mathsf{n}^\sharp \in Num^\sharp$ and $\sigma \in FV(\mathsf{n}^\sharp) \leftrightarrow \widetilde{FV}$ be a bijection. By abuse of language, we denote by $[\sigma]\mathsf{n}^\sharp$ the capture-avoiding substitution using σ of each of its formula.*

It can be seen that Lem. 3.4.1 holds for the overloaded $[\sigma]$ as well. For instance, let $\mathsf{n}^\sharp$ be $\{x + y < 5, z < 10\}$, $\sigma = \{(x, a), (y, b), (z, c)\}$, then $[\sigma]\mathsf{n}^\sharp$ is $\{a + b < 5, c < 10\}$. Applying $[\sigma^{-1}]$ to the latter, we immediately obtain $\mathsf{n}^\sharp$.

The following lemma states that the operation of substitution preserves the relation of valuation. For example, let $\mathsf{n} = \{(x, 2), (y, 3), (z, 5)\}$, $\mathsf{n}^\sharp = \{x + y = z, y \leqslant z\}$, and $\sigma = \{(x, a), (y, b), (z, c)\}$, then $\mathsf{n} \models \mathsf{n}^\sharp$ and $[\sigma]\mathsf{n} \models [\sigma]\mathsf{n}^\sharp$.

Lemma 3.4.2 (Substitution)**.** *Given* $\mathsf{n} \in Num, \mathsf{n}^\sharp \in Num^\sharp$ *and a bijective* σ *s.t.* $dom(\mathsf{n}) = dom(\sigma) = FV(\mathsf{n}^\sharp)$*, then* $n \models \mathsf{n}^\sharp \Rightarrow [\sigma]\mathsf{n} \models [\sigma]\mathsf{n}^\sharp$*.*

This lemma requires that the definition domain of n equals to the set of the free variables in $\mathsf{n}^\sharp$. To apply the lemma of substitution for the case where n has more defined variables than $\mathsf{n}^\sharp$'s free variables and $\mathsf{n} \models \mathsf{n}^\sharp$, we can restrict the definition domain of n to the free variables in $\mathsf{n}^\sharp$ so that the restricted n is a valuation of $\mathsf{n}^\sharp$. This is stated by the lemma below.

Lemma 3.4.3 (Restriction)**.** $\mathsf{n} \models \mathsf{n}^\sharp \Rightarrow \mathsf{n}_{|FV(\mathsf{n}^\sharp)} \models \mathsf{n}^\sharp$*.*

3.4.2 Proof

Proof of Thm. 3.3.1. Take an arbitrary $\mathsf{n}^\sharp \in Num^\sharp[\Delta \cup Var_n]$ and an arbitrary $\mathsf{p}^\sharp \in Pter^\sharp$, we will prove that for all $s \in \mathsf{WHILE}_{np}$,

$$[\![s]\!]^\natural\,(\gamma''(\mathsf{n}^\sharp, \mathsf{p}^\sharp)) \;\dot{\subseteq}\; \gamma''([\![s]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp)) \tag{3.14}$$

The correctness for the case of s_n is an immediate consequence following the assumed soundness of $[\![s_n]\!]^\sharp_n$ with regard to $[\![s_n]\!]^\natural_n$. We also obtain the correctness for the case of s_p because the soundness of $[\![s_p]\!]^\sharp_p$ with regard to $[\![s_p]\!]^\natural_p$ is assumed. The proof for the case of $x_n = y_p.f_n$ is analogous to the case of $x_p.f_n = y_n$ that given below: Denote the left and the right parts of (3.14) `lhs` and `rhs` respectively. By the definition of $[\![\cdot]\!]^\natural$ and $[\![\cdot]\!]^\sharp$, we have

$$\texttt{lhs} = \left(\left(\bigcup_{\gamma_p(\mathsf{p}^\sharp) \vdash x_p.f_n \Downarrow d} [\![d = y_n]\!]^\natural_n\,(\gamma_\delta(\mathsf{n}^\sharp))\right), \gamma_p(\mathsf{p}^\sharp)\right) \tag{3.15}$$

$$\texttt{rhs} = \left(\gamma_\delta\left(\bigsqcup_{\mathsf{p}^\sharp \vdash x_p.f_n \Downarrow \delta} \mathsf{n}^\sharp \sqcup [\![\delta = y_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)\right), \gamma_p(\mathsf{p}^\sharp)\right) \tag{3.16}$$

Take an arbitrary d s.t. $\gamma_p(\mathsf{p}^\sharp) \vdash x_p.f_n \Downarrow d$ and let $\delta = \rhd(d)$. We will prove

$$[\![d = y_n]\!]^\natural_n \circ \gamma_\delta(\mathsf{n}^\sharp) \subseteq \gamma_\delta(\mathsf{n}^\sharp \sqcup [\![\delta = y_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)) \tag{3.17}$$

By the Def. of γ_δ, it suffices to prove a stronger condition:

$$\forall \sigma \in \texttt{Ins}_\triangleright : [\![d = y_n]\!]_n^\natural \circ \gamma_n \circ [\sigma](\mathsf{n}^\sharp) \subseteq \gamma_n \circ [\sigma](\mathsf{n}^\sharp \sqcup [\![\delta = y_n]\!]_n^\sharp (\mathsf{n}^\sharp)) \quad (3.18)$$

Let the left and the right parts of (3.18) denoted by $\texttt{lhs}'$ and $\texttt{rhs}'$ respectively. Take an arbitrary $\sigma \in \texttt{Ins}_\triangleright$, let $\sigma \triangleq \{(\delta_i, d_i)\}_{1 \leqslant i \leqslant |\Delta|}$. Take an arbitrary $\mathsf{n} \in \mathit{Num}[D \cup \mathrm{Var}_n]$ satisfying

$$\mathsf{n} \in \texttt{lhs}' \quad (3.19)$$

we want to show

$$\mathsf{n} \in \texttt{rhs}' \quad (3.20)$$

By Eq. (3.19) and the correctness of $[\![\cdot]\!]_n^\sharp$, we have $\xrightarrow{\mathit{Num}} (d = y_n)(\mathsf{n}) \models [\sigma]\mathsf{n}^\sharp$. Below we will write

$$\mathsf{n}[d \mapsto y_n]$$

namely n updated with $d = y_n$, for $\xrightarrow{\mathit{Num}} (d = y_n)(\mathsf{n})$. We have

$$\mathsf{n}[d \mapsto y_n] \models [\sigma]\mathsf{n}^\sharp \quad (3.21)$$

Thus, by Lem. 3.4.2, we have[3]

$$[\sigma^{-1}](\mathsf{n}[d \mapsto y_n]) \models \mathsf{n}^\sharp \quad (3.22)$$

We continue the proof following whether σ maps δ to d.

- **Case I :** $(\delta, d) \in \sigma$. Since $d \in \mathit{dom}(\sigma^{-1})$, (3.22) implies

  $$([\sigma^{-1}]\mathsf{n})[\delta \mapsto y_n] \models \mathsf{n}^\sharp \quad (3.23)$$

  By the soundness of $[\![\cdot]\!]_n^\sharp$, we have

  $$[\sigma^{-1}]\mathsf{n} \models [\![\delta = y_n]\!]_n^\sharp\, \mathsf{n}^\sharp \quad (3.24)$$

- **Case II** $(\delta, d) \notin \sigma$. Since $d \notin \mathit{dom}(\sigma^{-1})$, (3.22) implies

  $$[\sigma^{-1}]\mathsf{n} \models \mathsf{n}^\sharp \quad (3.25)$$

  due to Lem. 3.4.3.

Combining the two cases, we obtain (3.20) following Lem. 3.4.2.

□

[3] Although σ is not a bijection, it is guaranteed to be injective. Thus σ^{-1} is still well-defined.

3.5 Related Work

While a large number of articles cover issues related to pointer analyses and to numerical abstractions, the program analyses where both pointers and numeric values are taken into account are comparatively few.

Our work was initially inspired by Chang and Leino's congruence-closure abstract domain [**?**]. Their combined abstract domain extends the properties representable by a given abstract domain to schema over arbitrary terms, and not just variables. They deal with alias problem using an ad-hoc *heap succession* abstract domain while we allow to reuse off-the-shell points-to analyses.

Points-to analysis is well known, and many variants have been published (see [40] for a survey). It offers a large spectrum of tradeoffs between precision and scalability with so called *equality-based* [66], *subset-based* [1] and *flow-sensitive* [18] variations. Points-to analyses are relatively imprecise compared to more advanced shape analysis techniques, but they scale well to large programs. Most analyses that combine numerical and pointer information tend to comply with similar simple pointer analyses (TVLA shown below is clearly an exception). Logozzo's Cibai (Class Invariants By Abstract Interpretation) [53] is a modular analysis that combines a type-based pointer analysis and octagons. Sotin and Jeannet [65] extend their generic numerical analyzer Interproc to deal with programs in the presence of pointers to the stack. Miné's [55] shows the power of this simple abstraction by extending it to pointer arithmetic, union types and records of stack variables. The resulted abstraction is integrated to ASTREE [8] and is able to deal with a subset C program that does not have dynamic memory allocation.

The book of Simon [64] gives an extensive study of numeric analysis to avoid buffer-overflows problems in C programs. The author combines ad-hoc numerical domains and a manually refined flow-sensitive points-to analysis. The combination of Simon's work have mutual effect between the heap domain and the numeric domains. His analysis is more precise than that of Miné's and the analysis in this paper, but requires important implementation efforts compare to our modular analysis.

A more sophisticated heap abstraction is *shape analysis* [61]. The TVLA [48] framework based on shape analysis uses *canonical abstraction* to create bounded-

size representations of memory states. The analyses of this family are precise and expressive. TVLA users are demanded to specify the concrete heap using first-order predicates with transitive closure, or user-defined *instrumentation predicates* like `IsNotNull`. Then TVLA automatically derives an abstract semantics based on the users' specification. The numerical abstraction of Gopan *et al.* [36] allows the integration of TVLA with existing numerical domains. The recent TVAL+ [34] uses TVLA on top of SAMPLE (Static Analysis of Multiple LanguagEs), and can be combined with any existing numerical analyses in SAMPLE. The static verifier DESKCHECK [54] combines TVLA and numerical domains. It is sufficiently precise and expressive to check quantified invariants over both heap objects and numeric values. Besides the burden for users to specify the program (a problem that XISA [15] attempts to remedy), the major issue of the shape-analysis-based approaches lies in their scalability. In contrast, our experiments show our capability to run over large programs.

Pioli and Hind [58] show the mutual dependence of *conditional constant analysis* and pointer analysis. The combination is specifically designed for the conditional constant analysis and is not generalized to standard numerical domains. In particular, this approach does not directly cooperate with standard numerical domains because their method relies on the particular feature of conditional constant analysis that is able to partially eliminate infeasible branches.

In a somewhat different strand of work, numerical domains have been used to enhance pointer analysis. Deutsch [28] uses a parametrized numerical domain to improve the accuracy of alias analysis in the presence of recursive pointer data structures. The key idea is to quantify the symbolic field references with integer coefficients denoting positions in data structures. This analysis is able to express properties for cyclic structures such as "for any k, the k-th element of list l of length len, is aliased to its $(k+len)$-th element". Venet [72] develops the structure called the *abstract fiber bundle* to formalize the idea of embedding an abstract numerical lattice within a symbolic structure. The structure enables the using of the large number of existing numerical abstractions to encode a broad spectrum of symbolic properties.

3.6 Conclusion

The primary objective of this work has been the automatic discovery of numerical invariants in Java-like programs, which are generally pointer-aware. We have proposed a methodology for combining numerical analyses and points-to analysis, developed using an approach based on concepts from abstract interpretation. In particular, we have shown how the abstract domain used in points-to analysis can be used to lift a numerical domain to encompass values stored in the heap. The new abstract domain and the accompanying transfer functions have been specified formally. Their correctness are proved. Moreover, the modular way in which the abstract domains are combined via some well-defined interfaces is reflected in the modular construction of a prototype implementation of the analysis framework. This modularity has enabled us to experiment with different choices for the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators. Concretely, the derived abstract semantics allows us to combine existing numerical domains (interval domains, polyhedron etc.) with existing points-to analyses. The modular analyzer uses PPL and SPARK and shows a clear precision enhancement with low time overhead.

Further work will address the issue of how the framework can accommodate analysis features such as strong updates. Also, the analyzer is currently only working intra-procedurally. We would like to develop the theory further so as to be able to build interprocedural analyses using our methodology. Finally, another interesting issue that deserves investigation is the possibility of exploiting other combinations such as points-to and must-alias analysis in order to fine-tune the points-to analysis and *a fortiori* the lifted numerical analysis.

Chapter 4

Enhancing Points-to Analysis by Must-Alias

4.1 Introduction

We find that a part of redundant arcs in points-to graph can be removed in the presence of *must-alias*. In [47], Landi defines *must alias* as the aliases that occur on *all* executions of the program. *Must-alias* analysis tracks a subset of this information. The detected aliases hold for all program executions, but all the must aliases are not detected. Unlike points-to analysis, must-alias analysis is seldom a subject of serious consideration, although it is sometimes used to perform *strong update* that sharpens some dependent analysis, as in the the typestate verification [35]. The focus of this chapter is neither points-to analysis or must-alias analysis, but their combination — we introduce and study the problem of *redundancy elimination of points-to graph in the presence of must-alias*.

4.1.1 Motivating Example

The Java-like snippet shown in Fig. 4.1 is our example program.

Two graph-like structures, *points-to graph* and *must-graph*, will be used to represent memory abstractions. The must-alias will be represented by a data structure called *must-graph*. It is a graph-like structure inspired from

the *e-graph* in [**?**], or the heap abstraction of [33]. The must-graph is a rooted directed graph where a node is an integer, and an arc is labeled by a variable if the arc starts from the root, or fields if the arc starts from a non-root node. The integer nodes of the must-graph are purely symbolic: two access paths are aliased whenever they lead to the same node in the must-graph. The points-to graph used here is similar as, but slightly different from the traditional one introduced earlier in the sense that the points-to graph of this chapter has an extra node, "the root" so that an arc started by variable x to a node h is written as an arc started by the root to h with the variable x as its label.

Below, we illustrate how *must-alias* can be used to perform redundancy elimination on points-to graph. Let us consider the points-to graphs *before* lines $\ell 100$, $\ell 110$ and $\ell 120$.

Before the line $\ell 100$, the standard points-to graph and ours give the same results. From lines $\ell 10$ to $\ell 40$ the program creates two lists of 2 elements linked by the field f, separately assigned to variable y and z. Following lines from $\ell 50$ to $\ell 90$ the program non-deterministically assigns to the variable x the references of the 2 lists. Here, no must-alias is detected.

Then before line $\ell 110$ an if-guard has been passed. Such test is ignored by the standard points-to analysis. Our analyzer, however, will extract the must-alias between x and y. This extraction itself is very simple that should not incur complexity overhead, but the extracted must-alias has sufficient information to remove a redundant arc.

Our analyzer will process in 2 steps. The first step is points-to graph propagation. Our analyzer uses the same *transfer function* as that of standard points-to analysis. The transfer function is a rule of propagation. Clearly, before line $\ell 110$, our analyzer gives the same points-to graph after propagation as the standard analyzer because our analyzer and the standard analyzer have the same points-to graph before line $\ell 100$, but our analyzer goes further by performing the second step that is the redundancy elimination. Our analyzer will detect redundant arcs using must-alias. Here, the redundant arc $\circ \xrightarrow{x} h_3$ will be detected. Intuitively, since x, y must alias, and y does not point-to h_3 (because y points-to h_1 only), we have x cannot point to h_3.

Before line $\ell 120$, the standard points-to analysis will simply add an arc from h_1 to h_3, unaware of the redundant arc from h_1 to h_2 (we will see

why it is redundant below). This analysis reasons conservatively: h_1 might be associated with more than one concrete object, so this analysis will not remove any arc emanating from h_1. By the same reason, it also adds a self-cycle on h_3. Finally, this analysis adds 2 arcs, $h_1 \xrightarrow{f} h_3$ and $h_3 \xrightarrow{f} h_3$. The obtained points-to graph is shown in Fig.4.2 (third row, first column).

Our analyzer processes in 2 steps as aforementioned. The first step is the propagation. It adds $h_1 \xrightarrow{f} h_3$ to the points-to graph. However, it will not add a self-cycle on h_3 because, at this time, our analyzer has already a refined points-to graph as input that does not contain the arc $\circ \xrightarrow{x} h_3$. For the second step, our analyzer performs an extra redundancy elimination using must-alias. The must-alias analyzer detects: at line $\ell 120$, $x.f$, z must alias, and x, y must alias, and by consequence, $y.f$ and z must-alias (Fig. 4.2 third line, third column). This information will guarantee the sound redundancy elimination of $h_1 \xrightarrow{f} h_2$. Intuitively, h_1 is accessible by at most x and y, and both $x.f$ and $y.f$ must point to h_3.

The above reasoning seems to be *ad-hoc*. This heuristic should be formalized and verified. We are faced with 2 questions.

1. What are the exact meanings of the points-to graph, must alias, and the so-called "redundancy"?

2. Under which conditions can a redundancy elimination be safely performed?

In the following, we give a quick overview of our methodology that answers the two questions.

4.1.2 Backward-simulation

The reply to the 1st question requires a semantically-based formalization. We have seen that both points-to graph and must-graph are rooted directed graph. Semantically, they are abstractions of concrete memory information. To formalize the semantics of points-to graph, we will use a *concretization function* γ (in terms of abstract interpretation) that assigns to the points-to graph and must-graph their abstracted concrete environments.

```
 ℓ10  y = new List      //h1
 ℓ20  y.f = new List    //h2
 ℓ30  z = new List      //h3
 ℓ40  z.f = new List    //h4
 ℓ50  if (?) then
 ℓ60      x = y
 ℓ70  else
 ℓ80      x = z
 ℓ90  end if
ℓ100  if (x == y)
ℓ110    x.f = z
ℓ120  end if
```

Figure 4.1: The example analyzed code. The program first creates two linked lists (from $\ell 10$ to $\ell 40$ where `List` has a field f), non-deterministically assigns to variable x the references of the two lists (from $\ell 50$ to $\ell 90$). At last, an instruction accessing the heap is performed under the condition that x, y hold the same reference value.

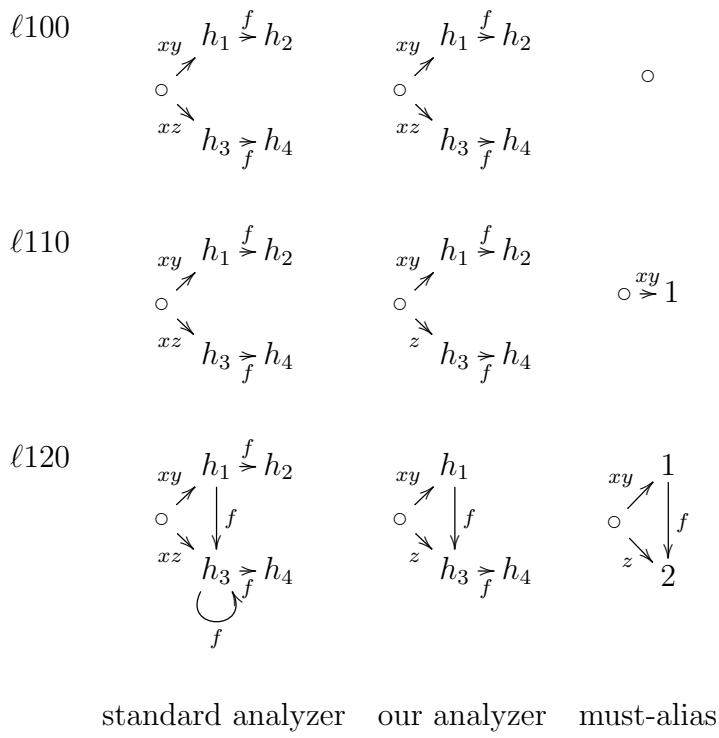


Figure 4.2: Compare standard points-to analyzer and ours. From the first column to the third column: line number, standard points-to analyzer, our analyzer and must-alias analyzer. The graph corresponds to the result before the indicated line number. Labels of the arcs with the same pair of source and targets are grouped together.

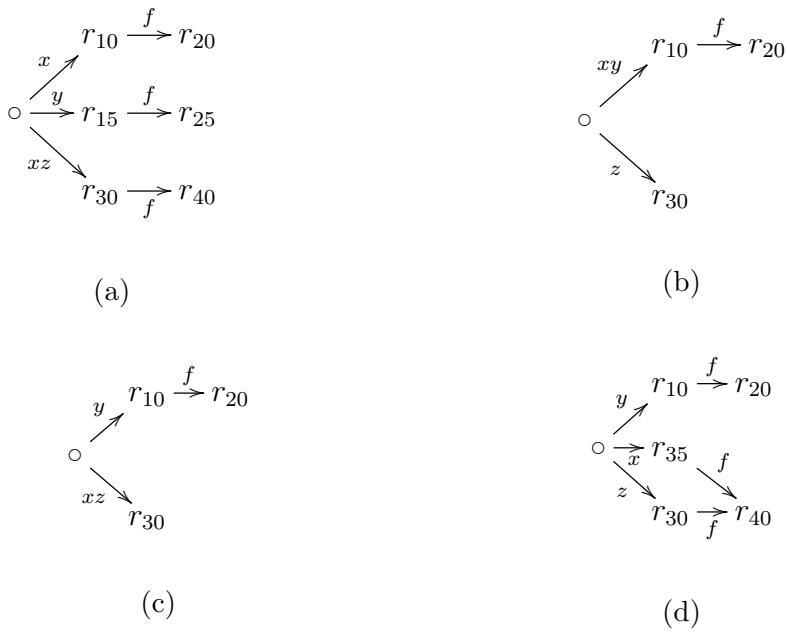


Figure 4.3: Possible concrete environments for points-to graph in the first row of Fig. 4.2. Here, we have assumed that: r_{10}, r_{15} are abstracted as h_1; r_{20}, r_{25} are abstracted as h_2; r_{30}, r_{35} are abstracted as h_3; and r_{40} is abstracted as h_4.

For example, consider the points-to graph at line $\ell 100$ (Fig. 4.2, first row). The concrete environments the points-to graph represents can be any one of Fig. 4.3, in which the concrete reference $r_{c'}$ is abstracted as the abstract reference h_c [1] for c being the first digit of c'. Remark that each abstract reference can represent more than one concrete reference (e.g. Fig 4.3(a)). This semantics view explains why the standard points-to analysis cannot remove the arc $h_1 \xrightarrow{f} h_4$ when `x.f = z` is performed at line $\ell 110$.

Compared with points-to graph, the semantics of must-graph has a determinist characteristic: the variables that reach the same node in the must-graph are guaranteed to share the same reference value at run-time (This can be easily extended to cases with access path, Def. 4.2.3). Further, the must-graph is only partially specifies the concrete memory: it is possible to have two access path that must-alias in the concrete memory whereas their aliasing is not recorded by the must-graph.

Details on our problem formulation can be found in Sect. 4.2. Let μ, Θ be a must-graph and a points-to graph respectively, and $\gamma(\mu), \gamma(\Theta)$ be their represented concrete environments, then the semantics of the points-to graph in the presence of the must-graph is formulated as $\gamma(\mu) \cap \gamma(\Theta)$, denoted by $\gamma(\mu, \Theta)$. Then an arc $h' \xrightarrow{f} h$ is defined *redundant* if and only if $\gamma(\mu, \Theta)$ remains the same if the points-to graph Θ is deprived of the arc $h' \xrightarrow{f} h$. The problem indicated by the chapter's title – redundancy elimination of points-to graph using must-alias – is then formalized as the computation of the minimal sub- points-to graph of Θ, $\bar{\Theta}$ such that $\gamma(\mu, \bar{\Theta}) = \gamma(\mu, \Theta)$.

With this semantics-based formulation, we will be able to reply to the 2nd question mentioned above. The theoretical study on the redundancy elimination is shown in Sect. 4.3. In essence, we aim at a sufficient condition for redundant points-to arcs. We define an arc being `essential` as the exact converse of being redundant. Then our goal is turned to find necessary conditions of an arc being essential. It turns out the found necessary condition is closely related with the concept of backward-simulation. Similar terms like "simulation", "bisimulation" etc., are frequently used in the theory of the Calculus for Communicating Systems (CCS), model checking and game theory.

[1] The `null` pointer is omitted in the drawings of environments.

Definition 4.1.1 (Backward-simulation)**.** *Given a must-graph μ and a points-to graph Θ, a relation R between* $\texttt{node}(\mu)$ *and* $\texttt{node}(\Theta)$ *is a backward-simulation if and only if*

> *For any node n of must-graph and node h of points-to graph, whenever $n\ R\ h$ we have, for any incoming arc of n of label f, written as $n' \xrightarrow{f} n$, we can find a corresponding incoming arc of h with the same label f, written as $h' \xrightarrow{f} h$, such that $n'\ R\ h'$.*

A node $n \in \texttt{node}(\mu)$ is backward-simulated by *$h \in \texttt{node}(\Theta)$, denoted by $n \backsim h$, if and only if there exists a backward simulation R such that $n\ R\ H$.*

$\backsim$ can be equivalently defined to be the greatest fixpoint of its associated functional $F_{\backsim}$ defined as

$$F_{\backsim}(R) \triangleq \{(n, h) \mid \forall n' \xrightarrow{f} n, \exists h' \xrightarrow{f} h : n'Rh'\} \tag{4.1}$$

An important convention is, the root of must-graph is backward-simulated by the root of points-to graph: $\circ \backsim \circ$. [2]

Three observations are immediate.

1. the empty relation is also a backward-simulation.
2. The union of two backward-simulation is still a backward-simulation.
3. $\backsim$ is the union of all backward-simulations.

In this presentation, when we say "compute backward-simulation", we are interested in the maximal one w.r.t. the set inclusion order $\subseteq$.

Illustration of backward-simulation can be found in Fig. 4.4. If n of must-graph is backward-simulated by an h of points-to graph, each incoming arc of n denoted by $n' \xrightarrow{f} n$, must have a corresponding incoming arc of h denoted by $h' \xrightarrow{g} h$ s.t. their labels are the same ($f = g$), and their sources are backward-simulated.

[2] This convention is necessary because the root is not considered as a graph node in this presentation. This convention has no conflict with the above definition of backward-simulation because a root has no incoming arc.

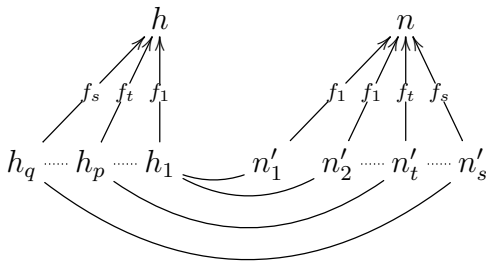


Figure 4.4: Backward-simulation

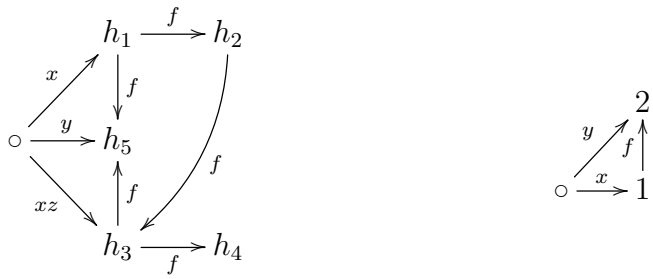


Figure 4.5: Points-to graph (left) and must-graph (right). Redundant arcs are $h_1 \xrightarrow{f} h_2$ and $h_2 \xrightarrow{f} h_3$.

Consider the example in Fig. 4.5. The points-to graph (left) has 2 redundant arcs $h_1 \xrightarrow{f} h_2$ and $h_2 \xrightarrow{f} h_3$ in the presence of the must-graph (right), but $h_3 \xrightarrow{f} h_4$ is *not* redundant because h_3 is shared by z.

Write $n \backsim h$ if n is backward-simulated by h. We have $1 \backsim h_1, 1 \backsim h_3, 2 \backsim h_4, 2 \backsim h_5$. We will show how the information from $\backsim$ can be used to remove the redundant arcs in Sect. 4.3 and 4.4.

4.1.3 Contribution

In this chapter, we introduce the problem of redundancy elimination of points-to graph using must-alias.

- We propose and prove the soundness of a procedure for the problem.

- We show the polynomial complexity of our algorithm is polynomial w.r.t. the size of the input points-to graph and must-graph. This means our approach introduces acceptable complexity overhead.

Outline We formalize the problem in Sect. 4.2. Theoretical results are shown in Sect. 4.3. We give the algorithm in Sect. 4.4 and show its incompleteness in Sect. 4.5.

4.2 Redundancy Elimination of Points-to Graph

In this section, we shall formalize the problem. The syntax of points-to graph slightly differs from the earlier chapters.

We will investigate the mathematical notations and data structures that occur throughout the presentation. Then we specify the semantics of the must-graph and points-to graph. The semantics is defined via *concretization function*. In particular, we introduce the definition of *common environment* and *essential arc*. The latter is an exact converse of redundant arc. In the last subsection, we give a lemma that reveals an equivalent condition of an arc being essential.

4.2.1 Must-graph and the Non-standard points-to graph

We have seen various graph representations in Sect. 4.1. Our problem will be modeled in terms of graph glossary. In general, an *arc-labeled directed graph* (called graph henceforth), G, is defined to be a set of nodes together with a set of labeled arcs joining certain pairs of nodes. An arc labeled f from v' to v is denoted by $v' \xrightarrow{f} v$. The *source* and the *target* of the arc are v' and v respectively. An *incoming* arc (resp. outgoing arc) of node v is an arc that has v as its target (resp. source). Such graph is *determinist* if each node has at most one outgoing arc for a certain label.

In our presentation, a graph is rooted. We assume a single artificial *root*. By convention, we exclude the root to be considered as a node of graph. Each node is assumed to be reachable from one of the roots. An *access path* is

defined to be a sequence of labels. The length of an access path is its number of labels. An empty access path is an access path of length 0.

We will use the following notations. The set of nodes and the set of arcs of a graph G are denoted by $\texttt{node}(G)$ and $\texttt{arc}(G)$ respectively. The universe of labels is denoted by Σ. The root is denoted by $\circ$. We write $v \xrightarrow{f} \top$ to represent the predicate that no outgoing arc of label f exists from v. For an access path $\vec{u} \triangleq f_0.f_1.\ldots.f_n$, $\vec{u}.f$ means $f_0.f_1.\ldots.f_n.f$. Given G, a non-empty access path $\vec{u}$ *evaluates* to the set of nodes it eventually reaches, denoted by $G(\vec{u})$. The result of $G(\vec{u})$ is a set. We write $G(\vec{u}) = \top$ if $\vec{u}$ evaluates to an empty set. We write $G(\vec{u}) = v$ if $\vec{u}$ evaluates to the singleton set $\{v\}$. The empty access path is denoted by ϵ. By convention, the empty access path evaluates to the graph root: $G(\epsilon) = \circ$.

We will use the following primary domains. $Ref \triangleq \{r_1, r_2 \ldots\}$ is an enumerable set of *concrete references*, representing physical memories, ranged over by r. $H \triangleq \{h_1, h_2 \ldots\}$ is an enumerable set of *abstract references*, ranged over by h. Each concrete reference r is abstracted by an abstract reference h, written as $\rhd(r) = h$ or $r \rhd h$. At last, $\mathcal{N} \triangleq \{1, 2 \ldots\}$ is a finite set of *uninterpreted symbols* used for must-graph, ranged over by n .

We can now define the *environment*, the *points-to graph*, and the *must-graph* in terms of the above notations.

Definition 4.2.1. *The environment Env ranged over by ρ, the points-to graph Pto ranged over by Θ, and the must-graph Must ranged over by μ, are defined to be graphs with their nodes belonging to Ref, H and $\mathcal{N}$ respectively. The environment and the must-graph are deterministic: Given a source s and a label l, if $s \xrightarrow{l} t_1$ and $s \xrightarrow{l} t_2$ are two arcs, t_1 and t_2 must be the same node.*

4.2.2 Redundancy Elimination in a Semantics-based View

Semantically, a points-to graph Θ is an over-approximation of the concrete environment: If there is an arc $r' \xrightarrow{f} r$ in the environment ρ, there must be an arc $h' \xrightarrow{f} h$ of points-to graph with $r' \rhd h'$ and $r \rhd h$. Since the nodes are

reachable by root, we represent this semantics by the concretization function $\gamma : Pto \rightarrow \wp(Env)$.

Definition 4.2.2 (Semantics of points-to graph)**.**

$$\gamma(\Theta) \triangleq \{\rho \mid (\rho(\vec{u}) \rhd h' \wedge \rho(\vec{u}.f) \rhd h \implies h' \xrightarrow{f} h \in \texttt{arc}(\Theta)\} \tag{4.2}$$

It is worth noting that the points-to graph thus defined only captures reachable cells.

Definition 4.2.3 (Semantics of must-graph)**.**

$$\gamma(\mu) \triangleq \{\rho \mid (\mu(\vec{u}) = \mu(\vec{v}) \neq \top \implies (\rho(\vec{u}) = \rho(\vec{v})) \vee (\rho(\vec{u}) = \top \wedge \rho(\vec{v}) = \top)\} \tag{4.3}$$

By abuse of language, we have used a single γ to mean the semantics of μ, Θ, and (μ, Θ). A must-graph records an under-approximation of concrete must-alias. Its nodes are purely symbolic. If two access paths evaluate to the same symbolic value, they are either both undefined, or evaluate to the same value in the concrete environments.

Definition 4.2.4 (Semantics of the pair (μ, Θ) and common environment)**.** *Given the pair (μ, Θ), its semantics is the intersection of the semantics of μ and Θ.*

$$\gamma(\mu, \Theta) \triangleq \gamma(\mu) \cap \gamma(\Theta)$$

A common environment is defined to be an environment $\rho \in \gamma(\rho, \Theta)$.

Now we can specify the meaning of redundant arcs. Given a points-to graph Θ and its arc $h' \xrightarrow{f} h$, define $\texttt{reduce}(\Theta, h' \xrightarrow{f} h)$ as an operation that not only eliminates $h' \xrightarrow{f} h$ from the graph, but also eliminates the caused

garbage. For example, in Fig. 4.4, $\texttt{reduce}(\Theta, h_1 \xrightarrow{f} h2)$ is the sub- points-to graph of Θ that does not contain $h_1 \xrightarrow{f} h_2$ and $h_2 \xrightarrow{f} h_3$. The latter is the garbage due to the removal of $h' \xrightarrow{f} h$.

Remark. This step of garbage collection in the definition of `reduce` is necessary. Recall that the nodes of points-to graph are assumed to be reachable from the root. So, if we simply "pull out" an arc from a points-to graph, the resulted graph may contain nodes that are no more reachable from the root. Besides, it is worth noting that `reduce` is only used as a mathematical notation, not for our algorithm.

Definition 4.2.5 (Essential arc)**.** *The points-to arc $h' \xrightarrow{f} h$ of a points-to graph $\vec{u}$ is called* `essential` *in the presence of a must-graph Θ, if the semantics of (μ, Θ), i.e., $\gamma(\mu, \Theta)$, gets changed by removing the considered points-to arc $h' \xrightarrow{f} h$ from the points-to graph Θ.*

$$\texttt{essential}_{\mu,\Theta}(h' \xrightarrow{f} h)) \triangleq h' \xrightarrow{f} h \in \texttt{arc}(\Theta) \wedge \gamma(\mu, \bar{\Theta}) \neq \gamma(\mu, \Theta) \tag{4.4}$$

where $\bar{\Theta}$ is obtained by removing $h' \xrightarrow{f} h$ from Θ.

$$\bar{\Theta} \triangleq \texttt{reduce}(\Theta, h' \xrightarrow{f} h)$$

Our goal is turned to find the necessary conditions of an arc being essential, because the contra-position of such necessary condition will be a sufficient condition to soundly remove arcs of points-to graph. The following lemma gives a such necessary condition of an arc being essential. Intuitively, an arc is essential if it can be "passed through" by some common environment $\rho \in \gamma(\mu, \Theta)$.

Finally, we are able to define our problem indicated by the chapter's title.

Definition 4.2.6 (Problem of redundancy elimination and reduced minimal points-to graph)**.** *Let (μ, Θ) be a pair of must-graph and points-to graph. The problem of redundancy elimination of points-to graph using must-alias is to*

find the minimal sub- points-to graph $\bar{\Theta}$*, called* reduced minimal points-to graph *such that the concretization is preserved.*

$$\texttt{reduced}(\mu, \Theta) \triangleq \min\{\bar{\Theta} \subseteq \Theta \mid \gamma(\mu, \Theta) = \gamma(\mu, \bar{\Theta})\} \tag{4.5}$$

The fact that such minimal points-to graph exists and is unique will be shown in Lem. 4.2.2 below, which gives an equivalent condition that determines whether an arc is essential.

4.2.3 Toward the Soundness Condition

We first define *chain of points-to graph* that will be used to prove Lem. 4.2.2.

Definition 4.2.7 (Chain of points-to arc)**.** *A* chain of points-to arc *from a node* h_0 *to a node* h_m *is a sequence of arcs,* $h_0 \xrightarrow{f_1} h_1$*,* $h_1 \xrightarrow{f_2} h_2, \ldots,$ $h_0 \xrightarrow{f_m} h_m$*, where the target of the edge* $h_{i-1} \xrightarrow{f_{i-1}} h_i$ *equals the source of the edge* $h_i \xrightarrow{f_i} h_{i+1}$*. The arc chain is said to have the source* n_0*, target* n_m*, and length* $m \geqslant 0$*.*

Lemma 4.2.1. *Let* ρ *be an environment of a points-to graph* Θ*,* h *be a node of* Θ*,* $\vec{u} \triangleq f_1.f_2.\ldots f_n$ *be a non-empty access path s.t.*

$$\rho(\vec{u}) \rhd h$$

Then we have a chain of points-to arc from $\circ$ *to* h*,*

$$\circ \xrightarrow{f_1} h_1, h_1 \xrightarrow{f_2} h_2, h_2 \xrightarrow{f_3} h_3 \ldots h_{n-1} \xrightarrow{f_n} h$$

such that $\rho(f_1 \ldots f_k) \rhd h_k$ *for* $1 \leqslant k \leqslant n$*.*

The proof for the above lemma is simple by induction (omitted). The following result plays a kernel role for the next section. The lemma basically states that an arc is essential if and only if it can be "passed through" by a common environment.

Lemma 4.2.2. *Let (μ, Θ) be a pair of must-graph and points-to graph. A points-to arc $h' \xrightarrow{f} h$ is essential if and only if, for some common environment $\rho \in \gamma(\mu, \Theta)$ and an access path $\vec{u}$, we have*

$$\rho(\vec{u}) \rhd h' \wedge \rho(\vec{u}.f) \rhd h \tag{4.6}$$

Proof of Lem. 4.2.2. First, we prove that the $\Longleftarrow$ part by the semantics of points-to graph. Let $\rho \in \gamma(\Theta, \mu)$ be a common environment and $\vec{u}$ be an access path. Assume

$$\rho(\vec{u}) \rhd h' \wedge \rho(\vec{u}.f) \rhd h \tag{A}$$

It suffices to show that ρ is not an environment of $\bar{\Theta}$, recalling the notion of essential arc (Def. 4.2.5) and common environment (Def. 4.2.4). Assume by contradiction that $\rho \in \gamma(\bar{\Theta})$. By (A) and the semantics of points-to graph, we have $h' \xrightarrow{f} h$ as an arc of $\bar{\Theta}$. This contradicts the definition of $\bar{\Theta}$.

Now prove the $\Longrightarrow$ part. Let the points-to arc h$' \xrightarrow{f} h$ be an essential arc, and let $\bar{\Theta} \triangleq \texttt{reduce}(\Theta, h' \xrightarrow{f} h)$. The definition of essential arc (Def. 4.2.5) tells the existence of an environment that belongs to $\gamma(\mu, \bar{\Theta})$ but does not belong to $\gamma(\mu, \Theta)$. Therefore we can find some ρ_* s.t.

$$\rho_* \in \gamma(\Theta) \wedge \rho_* \notin \gamma(\bar{\Theta})$$

.

Recall that $\rho_* \in \gamma(\Theta)$ means, for each $h', f, h, \vec{u}$,

$$\rho_*(\vec{u}) \rhd h' \wedge \rho_*(\vec{u}.f) \rhd h \implies h' \xrightarrow{f} h \in \texttt{arc}(\Theta) \tag{4.7}$$

and $\rho_* \notin \gamma(\bar{\Theta})$ means, for some $h'_*, f_*, h_*, \vec{u}_*$,

$$\rho_*(\vec{u}_*) \rhd h'_* \wedge \rho_*(\vec{u}_*.f_*) \rhd h_* \wedge h'_* \xrightarrow{f_*} h_* \notin \texttt{arc}(\bar{\Theta}) \tag{4.8}$$

Combining the two, we have $h'_* \xrightarrow{f_*} h_* \in \texttt{arc}(\Theta)$ and $h'_* \xrightarrow{f_*} h_* \notin \texttt{arc}(\bar{\Theta})$. Recall that $\bar{\Theta}$ is obtained by removing $h' \xrightarrow{f} h$ and its consequence garbage from Θ, we only need to discuss two cases.

- Case I: The arc $h'_* \xrightarrow{f} h_*$ is exactly $h' \xrightarrow{f} h$. We conclude immediately from (4.8).

- Case II: The arc $h'_* \xrightarrow{f} h_*$ will become the garbage once $h' \xrightarrow{f} h$ is removed from Θ.

  Write[3] $\vec{u}_*$ as $f_1.f_2\ldots$. Due to Lem. 4.2.1 and $\rho_*(\vec{u}_*) \rhd h'_*$, we have a chain C of points-to arc from $\circ$ to h'_* s.t.

  $$\rho_*(f_0.f_1\ldots f_k) \rhd h_k$$

  On the other hand, case II implies that any chain of points-to arcs from $\circ$ to h_*

  $$\circ \xrightarrow{f_1} h_1, h_1 \xrightarrow{f_2} h_2, h_2 \xrightarrow{f_3} h_3.\ldots.h_{n-2} \xrightarrow{f_n} h_* \tag{4.9}$$

  must contain $h' \xrightarrow{f} h$. Otherwise, if $h' \xrightarrow{f} h$ is not one of the arcs in the chain (4.9), then the node h'_* is still reachable through the chain C, noting that the removal of $h' \xrightarrow{f} h$ will not make any arc of the chain become garbage. This contradicts the assumption of case II.

  Thus, we can find a prefix of $\vec{u}$, $\vec{u}_{pre}$, such that

  $$\rho_*(\vec{u}_{pre}) \rhd h' \wedge \rho_*(\vec{u}_{pre}.f) \rhd h$$

  We conclude for case II.

□

Road-map We have expressed an equivalent condition for an arc being essential in the form of

$$\rho(\vec{u}) \rhd h' \wedge \rho(\vec{u}.f) \rhd h \tag{C}$$

The following work is to study the consequence of $\rho(\vec{u}) \rhd h$ by which we will obtain the necessary condition for an arc being essential. Thus, whenever we are asked whether an points-arc is redundant, we verify whether the necessary condition is satisfied. A part of redundant arcs can be eliminated in this way.

It is clear that we are required to find necessary conditions that can be easily checked. Moreover, it is desirable that this approach gives a *complete*

[3] We only prove the case for a non-empty $\vec{u}$.

solution, in the sense that the obtained necessary condition is as strong as its premise (the above C). Here we preview that the desire to be complete should be very difficult to achieve, because we have shown (Sect. 4.5) the NP-hardness of the problem of redundancy elimination of points-to graph. Next, let us go through a theoretical intermezzo.

4.3 Backward-simulation and Fuzzy Nodes

Given (μ, Θ), and $h \in \texttt{node}(\Theta)$, the goal of this section is to find the necessary condition of $\rho(\vec{u}) \rhd h$ for some $\rho \in \gamma(\mu, \Theta)$ and some access path $\vec{u}$. Depending on whether the access path has an evaluation in the must-graph (i.e. whether $\mu(\vec{u}) = \top$), we will consider separately

$$\rho(\vec{u}) \rhd h \wedge \mu(\vec{u}) \neq \top \tag{4.10}$$

and

$$\rho(\vec{u}) \rhd h \wedge \mu(\vec{u}) = \top \tag{4.11}$$

The following lemma gives the necessary condition of (4.10).

Lemma 4.3.1. *A node n of μ is backward-simulated by a node h of Θ, if for some common environment $\rho \in \gamma(\mu, \Theta)$ and access path $\vec{u}$ s.t. $\mu(\vec{u}) \neq \top$, the assertion $\rho(\vec{u}) \rhd h$ holds.*

$$\forall \rho \in \gamma(\mu, \Theta), \forall \mu, \forall n \in \texttt{node}(\mu) : \rho(\vec{u}) \rhd h \wedge \mu(\vec{u}) = n \implies n \backsim h \tag{4.12}$$

The proof for this lemma is tedious. It may be skipped upon a first reading.

Proof. Define

$$R(\rho, \vec{u}) \triangleq \{(n, h) \mid \rho(\vec{u}) \rhd h \wedge \mu(\vec{u}) = n\}$$

The goal is to prove, for any access path $\vec{u} \in \Sigma^*$ and $\rho \in \gamma(\mu, \Theta)$, we have $R(\rho, \vec{u}) \subseteq \backsim$.

Since $\backsim$ is the greatest fixpoint of its associated functional $F_{\backsim}$ by Def. 4.1.1, it suffices to show [4]

$$R(\rho, \vec{u}) \subseteq F_{\backsim}(R(\rho, \vec{u}))$$

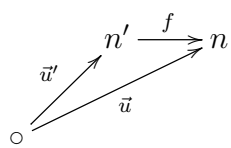
Take an arbitrary access path $\vec{u}$, an environment $\rho \in \gamma(\mu, \Theta)$, a node n of must-graph μ and a node h of points-to graph Θ such that

$$nR(\rho, \vec{u})h \tag{4.13}$$

we are engaged to prove

$$\forall n', f : n' \xrightarrow{f} n \in \texttt{arc}(\mu) \implies \exists h' : h' \xrightarrow{f} h \in \texttt{arc}(\Theta) \wedge n' \; R(\rho, \vec{u}) \; h' \tag{4.14}$$

Recall that we have assumed that each node is reachable from the root. This means, given any n', f s.t. $n' \xrightarrow{f} n$, there exists some $\vec{u}'$ s.t. $\mu(\vec{u}') = n'$. The relationship among $n, n', \vec{u}, \vec{u}'$ can be illustrated by the must-graph



It follows that $\rho(\vec{u}') \neq \top$. This is because, if it is not the case, we have $\rho(\vec{u}'.f) = \top$ and thus

$$\rho(\vec{u}) = \top \tag{4.15}$$

following the semantics of must-graph. But (4.15) contradicts the fact $\rho(\vec{u}) \rhd h$ implied from 4.13.

By Lem.4.2.1, we can find $h' \in \texttt{node}(\Theta)$ s.t. $\rho(\vec{u}') \rhd h'$. Now we verify the conclusion part of (4.14): we have $n' \; R(\rho, \vec{u}) \; h'$ by the definition of R. To show $h' \xrightarrow{f} h$, recall the semantics of points-to graph. It suffices to show $\rho(\vec{u}'.f) \rhd h$. This is true following $\rho(\vec{u}) \rhd h$ and $\mu(\vec{u}) = \mu(\vec{u}'.f)$, since $\rho_* \in \gamma(\mu)$ implies $\rho(\mu'.f) = \rho(\mu)$. This completes the proof for (4.14) under the assumption $nR(\rho, \vec{u})h$. The proof is completed. $\square$

[4] Tarski's fixpoint theorem, or co-inductive proof principle: the greatest fix point of a monotone function over a complete lattice is its greatest pre-fixpoint.

Below we introduce "fuzzy nodes". This will be used to find the necessary condition of (4.11).

Definition 4.3.1 (Fuzzy node)**.** *A fuzzy node h is a points-to node that has an incoming arc $h' \xrightarrow{f} h$ labeled f such that h' is backward-simulated by a must-node n, and this node n does not have an outgoing arc labeled f.*

$$\texttt{fuzzy}(h) \triangleq \exists h' \xrightarrow{f} h, \exists n' \in \texttt{node}(\mu) : n' \backsim h' \wedge n' \xrightarrow{f} \top \tag{4.16}$$

Revisit the example of Fig. 4.5. The node h_3 is a fuzzy node. This is becasue these esists an points-to arc from the root to h_3 labeled z, yet in the must-graph there is no arc labeled by z and orignated from the root (remind that the root in the must-graph is back-simulated with the root in the points-to graph, according to our convention.)

Lemma 4.3.2 (Fuzzy node)**.** *Let $\rho \in \gamma(\mu, \Theta)$ be a common environment, and $\vec{u}$ be an access path such that*

$$\rho(\vec{u}) \rhd h \wedge \mu(\vec{u}) = \top$$

Then h is reachable by a fuzzy node $\tilde{h}$

$$\exists \tilde{h} : \texttt{fuzzy}(\tilde{h}) \wedge \tilde{h} \rightarrow^* h$$

Proof. By $\mu(\vec{u}) = \top$, we have a strict prefix of $\vec{u}$, denoted $u'.f$ such that[5]

$$\mu(\vec{u}') \neq \top \wedge \mu(\vec{u}'.f) = \top \tag{4.17}$$

By Lem. 4.2.1, we have $\tilde{h} \in \texttt{node}(\Theta)$ s.t.

$$\tilde{h} \rightarrow^* h \wedge \rho(\vec{u}') \rhd \tilde{h} \tag{4.18}$$

[5]Here $\vec{u}'$ may be an empty access path which causes $\mu(\vec{u}') = \circ$.

Let $n' \in \texttt{node}(\mu)$ be $\mu(\vec{u}')$ since $\mu(\vec{u}') \neq \top$ by (4.17). Following Lem. 4.2.2, we obtain

$$n' \backsim \tilde{h}$$

Note $n' \xrightarrow{f} \top$ by (4.17). By the definition of fuzzy nodes, we obtain

$$\texttt{fuzzy}(\tilde{h}) \tag{4.19}$$

We conclude combining (4.18) and (4.19). □

Finally, we can give a necessary condition for an arc being essential. This theorem can be directly translated to an algorithm that finds redundant points-to arc in the presence of must-alias.

Theorem 4.3.1. *Let (μ, Θ) be a must-graph and points-to graph. If a points-to arc $h' \xrightarrow{f} h$ is essential, then one of the following assertions must hold.*

$$(a) \quad \exists n', n \in \texttt{node}(\mu) : n' \xrightarrow{f} n \wedge n' \backsim h' \wedge n \backsim h$$

$$(b) \quad \exists n' \in \texttt{node}(\mu) : n' \xrightarrow{f} \top \wedge n' \backsim h'$$

$$(c) \quad \exists \tilde{h} \in \texttt{node}(\Theta) : \texttt{fuzzy}(\tilde{h}) \wedge \tilde{h} \rightarrow^* h'$$

Proof. By Lem. 4.2.2 , we have $\rho(\vec{u}) \rhd h'$ and $\rho(\vec{u}.f) \rhd h$ for some $\rho \in \gamma\langle\mu, \Theta\rangle$ and $\vec{u}$. Logically, we have 3 disjunctive cases:

- *Case 1:* $\exists n', n \in \texttt{node}(\mu), \mu(\vec{u}) = n', \mu(\vec{u}.f) = n$
- *Case 2:* $\exists n' \in \texttt{node}(\mu), \mu(\vec{u}) = n', \mu(\vec{u}.f) = \top$
- *Case 3:* $\mu(\vec{u}) = \top$

Case 1 and case 2 imply (a) and (b) respectively due to the lemma of backward-simulation (Lem. 4.3.1). Case 3 implies (c) due to the lemma of fuzzy node (Lem. 4.3.2).

□

4.4 Algorithm of Redundancy Elimination

A direct implementation following Thm. 4.3.1 is presented in Algorithm 1. Here we describe different parts of the algorithm and show its worst-case complexity. We use these notations: $|\Theta|$ and $|\mu|$ for the number of nodes of Θ and μ respectively; $\|\Theta\|$ and $\|\mu\|$ for the number of arcs of Θ and μ respectively. Denote $\mid \backsim_{\mu,\Theta} \mid$ as the size of the backward simulation. It is clear

$$\mid \backsim_{\mu,\Theta} \mid \leqslant |\mu| \times |\Theta| \tag{4.20}$$

- From line 1 to 11, the algorithm computes the backward-simulation, which is given by *backsim* at the end of the `while` loop (at line 12). To see this, denote $backsim^k$ to be the content of *backsim* when the `while` loop starts its k -th iteration (at line 2) and we have

  $$\begin{aligned} backsim^1 &= \texttt{node}(\mu) \times \texttt{node}(\Theta) \\ backsim^{k+1} &= \{(n,h) \mid \forall n' \xrightarrow{f} n, \exists h' \xrightarrow{f} h : \quad n' \ backsim^k \ h'\} \end{aligned}$$

  Thus *backsim* computes the greatest fix point of $F_{\backsim}$ defined in Def. 4.1.1.

  The worst-case complexity for this part is $O(\|\Theta\| \cdot \|\mu\| \cdot \mid \backsim_{\mu,\Theta} \mid)$. This is because the outside `while` loop (from line 2 to 11) at most iterates $\mid \backsim_{\mu,\Theta} \mid$ times, and its inner loop at most iterates over all the arcs of μ and Θ.

- From line 12 to 18, the algorithm computes the fuzzy nodes of Def. 4.3.1.

  It is clear that the worst-case complexity for this part is $O(\|\Theta\| \cdot \mid \backsim_{\mu,\Theta} \mid)$.

- From line 19 to 24, the algorithm computes the reachable nodes of the fuzzy nodes, i.e., the h satisfying condition (c) of Thm. 4.3.1,

  $$\{h \mid \exists \tilde{h} \in \texttt{node}(\Theta) : \texttt{fuzzy}(\tilde{h}) \wedge \tilde{h} \rightarrow^* h\}$$

  The reachable nodes are marked via the boolean array `visited`. At line 24, we use `traversal(h,visited)` to mean a standard procedure

like depth-first traversal that traverses and marks the nodes of points-to graph reachable from the node h. The procedure is assumed to visit the nodes that are not marked by `visited`.

The worst-case complexity for this part is $O(\|\Theta\| + |\Theta|)$. This is the complexity for a standard traversal like depth-first iteration. Although there may be more than 1 fuzzy node, each traversal will mark the visited nodes so that they will not be visited for the following traversals of the remained fuzzy nodes.

- From line 25 to 31, the algorithm computes redundant arc by putting together the arcs that do not satisfy the conditions (a), (b) or (c) in Thm. 4.3.1.

Finally, we conclude that the worst-case complexity for the algorithm is dominated by the program from line 1 to 11, which is $O(\|\Theta\| \cdot \|\mu\| \cdot | \backsim_{\mu,\Theta} |)$. We have the following conclusion following a conservative estimation of $| \backsim_{\mu,\Theta} |$, cf. (4.20).

Theorem 4.4.1. *The worst-complexity of Algorithm 1 is quartic.*

$$O(\|\Theta\| \cdot \|\mu\| \cdot |\Theta| \cdot |\mu|)$$

At last, as a case study, we review the 2nd example of Sect. 4.1 (Fig. 4.5). We have $1 \backsim h_1, 1 \backsim h_3, 2 \backsim h_4, 2 \backsim h_5$. The fuzzy node is h_3. By Thm. 4.3.1, we immediately obtain the redundant arcs $h_1 \xrightarrow{f} h_2$ and $h_2 \xrightarrow{f} h_3$.

Discussion. In general, must-graph in real programs is very small compared to points-to graph. On the one hand, must-analysis is expensive. The ambition to produce a precise large must-alias may not be realistic; On the other hand, at each merge point of program, must-alias analysis will become smaller because it consists of intersection instead of union at merge points. In the examples in Sect. 4.1, the must-graph has at most 2 nodes. In condition that the must-graph is much smaller than points-to graph, we have an algorithm of complexity $O(\|\Theta\| \cdot |\Theta|)$.

Algorithm 1 Redundancy elimination of points-to graph using must-alias

Input: must graph μ, points-to graph Θ
Output: *redundant* (redundant arcs)

1: $backsim \leftarrow \texttt{node}(\mu) \times \texttt{node}(\Theta) \cup \{(\circ, \circ)\}$
2: **while** *backsim* changes **do**
3: **for each** arc of μ, $n' \xrightarrow{f} n$ **do**
4: $H \leftarrow \emptyset$
5: **for each** arc of Θ labeled by f, $h' \xrightarrow{f} h$ **do**
6: **if** $(n', h') \in backsim$ **then**
7: $H \cup = \{h\}$
8: **end for**
9: $backsim(n) \cap = H$
10: **end for**
11: **end while**

12: $fuzzy \leftarrow \emptyset$
13: **for each** $(n', h') \in backsim$ **do**
14: **for each** outgoing arc of h', $h' \xrightarrow{f} h$ **do**
15: **if** $n' \xrightarrow{f} \top$ **then**
16: $fuzzy \cup = \{h\}$
17: **end for**
18: **end for**

19: **for each** $h \in \texttt{node}(\Theta)$ **do**
20: $visited[h] \leftarrow false$
21: **end for**
22: **for each** $h \in fuzzy$ **do**
23: traversal(h,visited)
24: **end for**

25: $redundant \leftarrow \emptyset$
26: **for each** $h' \xrightarrow{f} h \in \texttt{arc}(\Theta)$ s.t. *not visited*$[h']$ **do**
27: **for each** n' s.t. n' *backsim* h' **do**
28: **if** $not(n' \xrightarrow{f} \top)$ *and not* $((n' \xrightarrow{f} n\ )$ *and* $(n$ *backsim* $h))$ **then**
29: $redundant \cup = \{h' \xrightarrow{f} h\}$
30: **end for**
31: **end for**

4.5 Incompleteness

Consider the following points-to graph (Fig. 4.6 left) and must- graph (Fig. 4.6 right).

Figure 4.6: Incompleteness: points-to graph (left) and must-graph (right).

We obtain the backward-simulation

$$1 \backsim h_1, 1 \backsim h_2, 2 \backsim h_3, 3 \backsim h_4, 4 \backsim h_5$$

None of the nodes is fuzzy. By our algorithm, all arcs should be preserved. However, the arcs $h_3 \xrightarrow{j} h_5$ and $h_4 \xrightarrow{k} h_5$ could have been removed. This is because, the must graph requires $x.f.j = x.g.k$. By consequence, informally, $x.f.j$ must follow the flow $\circ \xrightarrow{x} h_1 \xrightarrow{f} h_3 \xrightarrow{j} h_5$, and $x.g.k$ must follow the flow $\circ \xrightarrow{x} h_2 \xrightarrow{g} h_4 \xrightarrow{k} h_5$, but it is impossible because x cannot simultaneously point to h_1 and h_2. (Remind the concrete environment is modeled to be determinist.)

In the following we show it is NP-hard to find a complete algorithm.

Theorem 4.5.1. *The problem of Redundancy Elimination of Points-to graph with Must-alias (called* $\mathsf{RE}^{\mathsf{PM}}$ *in this section) is NP-hard.*

We will use the following notations and conventions in the section. Given a must-graph μ and a points-to graph Θ, their concretizations are denoted by γ_μ (Def. 4.2.3) and γ_Θ (Def. 4.2.2) respectively. The construction process (to be defined shortly) of the graph G is denoted by $(\mu, \Theta) = \Phi(G)$. The

reduced minimal points-to graph (Def. 4.2.6) of Θ is denoted by $\bar{\Theta}$. Given a graph G, the number of nodes is denoted by $|G|$. For the rooted graphs μ and Θ, their roots are not counted as nodes. For $k \geqslant 0$, we write $x.f^k$ to mean the access path $x.\overbrace{f.f\ldots}^{k}$. By convention, $x.f^0$ means x.

To prove $\mathsf{RE}^{\mathsf{PM}}$ is NP-hard, we will prove that the problem of Hamilton circuit is reducible to the $\mathsf{RE}^{\mathsf{PM}}$. Classically, the problem of Hamilton circuit is, given a graph G with n nodes, to determine if G has a Hamilton circuit. It is known that Hamilton circuit problem is NP-complete for both directed graph and undirected graph. Here we only consider directed graph with at least 2 nodes.

That is to say, given a question, "Does the directed graph G contain a Hamilton circuit?", we answer this question in a polynomial procedure using the oracle machine solving $\mathsf{RE}^{\mathsf{PM}}$.

The process of reduction is as follows. Given a directed graph G with $|G|$ nodes $v_1 \ldots v_{|G|}$, we construct Θ to be the points-to graph started by the arc $\circ \xrightarrow{x} h_1$ where the node h_1 will be connected to an arbitrary node of G, say, v_1; each arc of Θ, except the one connected to the root, is labeled f and the other nodes in G are renamed $h_2, \ldots, h_{|G|}$. We construct μ to be the points-to graph started by the arc $\circ \xrightarrow{x} 1$ where the node 1 will be connected to an arbitrary node of a directed cycle composed of $|G|$ nodes, and the other nodes of this cycle are named $2, \ldots, |G|$; each arc, except the one connected to the root, is labeled f. This process, denoted by $(\mu, \Theta) = \Phi(G)$, guarantees the relation between the nodes numbers: $|G| \ = |\mu \models |\Theta|$.

For example, Fig. 4.7 is the graph G upon which we will determine the existence of a Hamilton circuit. Fig. 4.8 is the corresponding constructed Θ (The label f is not drawn.) and μ.

We will use the following algorithm to determine whether G has a Hamilton circuit, assuming an oracle machine that solves $\mathsf{RE}^{\mathsf{PM}}$.

Input: The directed graph G with at least 2 nodes.
Output: : "Yes/No" if G contains a Hamilton circuit or not.

Construct the points-to graph Θ and the must graph μ using the above mentioned procedure.

Find the reduced minimal points-to graph $\bar{\Theta}$, using the oracle machine that solves $\mathsf{RE}^{\mathsf{PM}}$.

if the nodes number of $\bar{\Theta}$ equals to that of G **then**
  output "Yes".
else
  output "No".

For example, the reduced minimal points-to graph in question is shown in Fig 4.9. The procedure answers "yes" since it contains as many nodes as G.

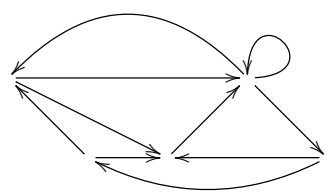


Figure 4.7: A Hamilton graph G

In the following, we assume (μ, Θ) to be the must-graph and points-to graph constructed by the indicated process above, and $\bar{\Theta}$ be the reduced minimal points-to graph.

Lemma 4.5.1. *If G contains a Hamilton circuit, then $\bar{\Theta}$ and G have equal number of nodes: $|\bar{\Theta}| = |G|$.*

Proof. It is clear $|\bar{\Theta}| \leqslant |G|$, since $|\bar{\Theta}| \leqslant |\Theta|$ and $|\Theta| = |G|$. We prove $|\bar{\Theta}| \geqslant |G|$. Assume that G's Hamilton circuit in terms of Θ's nodes is

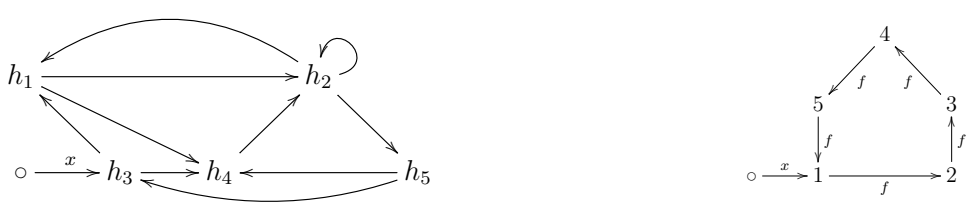


Figure 4.8: From left to right, the points-to graph Θ and the must graph μ constructed from the graph G.

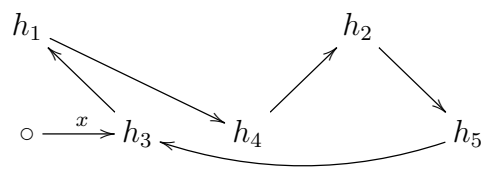


Figure 4.9: The reduced minimal points-to graph $\bar{\Theta}$.

$h_1, h_2, \ldots, h_{|G|}, h_1$. Construct ρ_* to be a graph that substitutes the nodes h_i of $\Theta \in Env$ to an arbitrary r_i such that $r_i \rhd h_i$. The arcs of ρ are chosen to be the circuit

$$r_1 \xrightarrow{f} r_2, r_2 \xrightarrow{f} r_3, \ldots, r_{|G|} \xrightarrow{f} r_1$$

By the definition of γ_{must} and γ_{pto}, we verify that $\rho_* \in \gamma_{must}(\mu)$ and $\rho_* \in \gamma_{pto}(\mu)$. Thus $\rho_* \in \gamma(\mu, \Theta)$. It is also straightforward to establish that, for any access path $\vec{u}$ and $h', h \in nodes(\Theta)$, we have $(\rho_*(\vec{u}) \rhd h' \wedge \rho_*(\vec{u}.f) \rhd h \implies h' \xrightarrow{f}_{\Theta} h)$. By Lem. 4.2.2, we conclude that all the arcs of the Hamilton circuit are essential, thus they must be included in $\bar{\Theta}$. We obtain $|\bar{\Theta}| \geqslant |G|$. $\square$

Lemma 4.5.2. *If G does not contain a Hamilton circuit, then there must be redundant arcs in Θ.*

Proof. Proof by contradiction. Assume that no redundant arc is contained in Θ. By Lem. 4.2.2, each node of Θ can be reached by an environment of $\gamma(\mu, \Theta)$. We have,

$$|G| \ = \#\{h \mid \exists \rho \in \gamma(\mu, \Theta), \exists 0 \leqslant i < |G|, \rho(x.f^i) \rhd h\} \tag{4.21}$$

This is because, for an arbitrary arc $h' \xrightarrow{f} h$ of Θ, there exists an access path $\vec{u}$ and an environment $\rho \in \gamma(\mu, \Theta)$, such that $\rho(\vec{u}) \rhd h'$ and $\rho(\vec{u}.f) \rhd h$. Thus we have (4.21) noting that the space of the considered access paths is $\{x.f^i, i \geqslant 0\}$, and we also obtain the constraints due to the must graph μ: $\rho(x.f^i) = \rho(x.f^{i+|G|})$ for any $i \geqslant 0$ and $\rho \in \gamma(\mu)$.

Since G does not have Hamilton circuit,

$$\forall \rho \in \gamma(\mu, \Theta), \exists n, m \in \mathbf{Z}, 0 \leqslant n < m < |G| \wedge \rhd(\rho(x.f^n)) = \rhd(\rho(x.f^m) \tag{4.22}$$

This is because, the sequence h_{s_i} defined to satisfy $\rho(x.f^i) \rhd h_{s_i}$ for $0 \leqslant i < |G|$ must have repetitive element. Otherwise, the sequence composes a Hamilton circuit. Noting that $\rho(x.f^i) \rhd h_{s_i}$ and $\rho(x.f^{i+1}) \rhd h_{s_{i+1}}$ imply $h_{s_i} \xrightarrow{f} h_{s_{i+1}}$ by the concretization of the points-to graph, and $\rho(x) = \rho(x.f^{|G|})$ by the concretization of the must-graph.

Combining (4.21) and (4.22), we obtain the contradiction $|G| < |G|$. □

Lemma 4.5.3. *If G has at least 2 nodes and does not contain any Hamilton circuit, then the nodes number in $\bar{\Theta}$ is strictly less than that of G, i.e. $|\bar{\Theta}| < |G|$.*

Proof. Define the predicate $P(k)$ for $k \geqslant 2$, which holds if and only if the above proposition is true for the graph G of k nodes, i.e.

$$\begin{aligned} P(k) \triangleq\ & \forall G, \forall (\mu, \Theta), \forall \bar{\Theta} : \Phi(G) = (\mu, \Theta) \wedge \\ & \bar{\Theta} = \texttt{reduced}(\mu, \Theta) \wedge \mid G \mid = k \wedge \neg Hamilton(G) \implies |\bar{\Theta}| < |G| \end{aligned} \tag{4.23}$$

$P(2)$ is trivially true. Assume that $P(k)$ holds for $2 \leqslant k < K$, we need to prove $P(K)$. That is to say, given an arbitrary graph G with k nodes, assume G does not contain a Hamilton circuit, let $\bar{\Theta}$ be the reduced minimal points-to graph of Θ constructed from Φ. We are engaged to prove that $\bar{\Theta}$ contains strictly less nodes than G does.

By Lem. 4.5.2, we have a strict subgraph $\tilde{\Theta} \subsetneq \Theta$ such that $\gamma(\mu, \Theta) = \gamma(\mu, \tilde{\Theta})$. Let $\tilde{G}$ be G with corresponding arcs of $\Theta \backslash \tilde{\Theta}$ removed.

- Case I: $|\tilde{G}| < |G|$, we conclude directly from the two facts $|\tilde{G}| = |\tilde{\Theta}|$ (by construction) and $|\bar{\Theta}| \leqslant |\tilde{\Theta}|$ ($\bar{\Theta}$ is the reduced minimal graph).
- Case II: $|\tilde{G}| = |G|$, we have (a) $\tilde{G}$ does not contain a Hamilton circuit. This is because, otherwise, G will contain a Hamilton circuit as well, noting that G has the same nodes as $\tilde{G}$ and yet more arcs than $\tilde{G}$. We

also have (b) $(\tilde{\Theta}, \mu) = \Phi(\tilde{G})$ (Keep in mind that μ is constructed to contain $|G|$ nodes, thus contains $|\tilde{G}|$ nodes). Therefore, it is eligible to apply the hypothesis of induction. We conclude $|\bar{\Theta}| < |G|$.

□

4.6 Comparison with Related work

Various methods of optimization have been proposed dealing with the precision/efficiency trade-off. We classify these approaches in 3 major categories.

- Semantics abstraction. This category includes the various traditional points-to analyses that are sensitive or insensitive to particular aspects of the program semantics [52, 16, 1]. Examples are context-sensitive /insensitive analysis flow-sensitive /insensitive analysis, or path sensitive /insensitive analysis, etc. The disadvantage of this approach is it sacrifice the precision when it tries to scale up, or vice-verse.

- Data structure designing. This category notably includes the use of binary decision diagram (BDD) as a compact representation [5] of point-to graph.

- Redundancy elimination. Examples are partial on-line cycle elimination[32] and projection merging [67]. Both simplify the points-to graph by detecting redundant points-to relations. The issues of this approach are how to avoid complexity overhead, and how to ensure soundness.

Our work belongs to the 3rd category.

The use of must-alias to refine points-to analysis should date back to the original work [30], in which precise killing information is obtained by definite points-to information. The work is limited to C's stack variables. The heap is abstracted as one cell. Choi *et al* [19] showed how must-alias information can help to yield precise alias. Their analysis is also based on C and the alias information is represented in alias pairs, which is considered inefficient.

Sagiv *et al* [62] gave simultaneous collection of both universal and existential properties of programs, and showed how to use universal assertions to

improve the accuracy of existential assertion. The pointer equality problem is used as an example. Compared to theirs, our work focuses on the pruning of points-to graph. We propose an algorithm to remove redundant arcs in an efficient way.

An algorithm of must-alias is presented in [46], but it handles only single level pointers and cannot be extended to general cases without complexity explosion. In [9], the must-alias is computed based on the concept of instance keys. However, they can only deal with local variables. Must-alias concerns only local variable. Although we do not provide an explicit must-alias analyzer in this paper, the must-graph presented in this paper is similar to the e-graph in [14], or as storeless structure in [43].

4.7 Conclusions

The objective of this work is to refine points-to analysis using must-alias information. This work is theoretical, and is ahead-of-time because of the lack of a must-alias analyzer in practice for now.

We have established an algorithm of polynomial complexity that removes redundant points-to relations with the help of must-alias relation. We start by formalizing the interfaces of the information obtained from points-to analysis and must-alias analysis as rooted directed graph. Then the semantics of the two graph are specified by a concretization function in the sense of abstract interpretation. This semantics-based problem formalization allows us to deduce an algorithm that is proved correct with regard to this semantics. We give the pseudocode of the algorithm which is of polynomial complexity. This argues that the approach has a reasonable complexity overhead.

Pointer analysis has often been presented in an informal way. Numerous studies have been done for the optimization of existing mainstream pointer analysis. Unfortunately, many of these works lack rigorous formalization, which make them less trust-able and more error-prone. The semantics-based approach of this work leads to a fully proved algorithm which provides a

solid way to solve the problem. In particular, the semantics-based approach is mandatory for this problem where one of the component analyses, *i.e.*, must-alias analysis, has not yet been fully studied or implemented in practice.

For future work, we promote a theoretical research for the analysis of must-alias, and a field study for the combination of must-alias with points-to analysis. Experimental results are desired so we can evaluate the benefits of this combination in real-life programs.

Chapter 5

Prototyping NumP

We have implemented a prototype for the abstract domain *NumP*. Below, we write NumP (in sans-serif font) for the prototype. NumP uses SOOT [69] as the front-end. It modularly combines the pointer analyses in SPARK [50], and the abstract numerical domains implemented in PPL [2].

We first implement the traditional static numerical analyzer for Java. The implementation will be denoted by Num. This is done by wrapping abstract domains in PPL. Num either skips unrecognized statements or conservatively approximates them using the operator of *unconstraint* in PPL. The flow-insensitive points-to analyses are directly available in SOOT. They will be denoted by Pter subsequently. The input program is a set of Java classes with a class `main` indicating the entry point used for call-graph construction.

To ensure extensibility and re-usability, the implementation extensively employs standard *object-oriented* technologies. As experimental study, we compare NumP with its component analyses for precision and cost.

5.1 Design issues

5.1.1 Reused components

A modular designing is essential to make the analysis easy to implement and to maintain. In particular, code duplication should be avoided regarding to the two major components of the analysis, namely, pointer analysis and

numerical analysis.

- PPL [3], the Parma Polyhedra Library, is a set of implementations manipulating numerical information that can be represented by points in some n-dimensional vector space. It provides a large amount of numerical abstract domains ready for use, including the non-relational interval abstract domain, the relational polyhedral (convex or not) abstract domains, the weakly-relational octagon abstract domains, and some categories of combined domains, like the powerset construction. The library is written in C++ , and is ported to Java among others languages. We *encapsulate* (in the sense of object-oriented technology) the abstract domains provided by PPL as a back-end to compute transfer functions, widening, narrowing and join, etc.

- SOOT [70] is an open-source toolkit for Java program transformation and optimization. In this implementation, two use cases of SOOT can be found: (1) We use SOOT as the front end to transform code sources or their bytecodes into the Jimple intermediate representation. (2) More than one variants of pointer analyses from SOOT can be borrowed to our implementation, including the field-insensitive/field-sensitive, context-insensitive/context-sensitive, subset-based/union-based, or demand-driven variants.

5.1.2 Precision/cost trade-off

The balance between the precision and the cost is one major concern for the designing of any static analysis.

Regarding to the pointer analysis part of the prototype, we use the flow-insensitive points-to analysis because it can statically infer a relatively precise pointer behavior with affordable time and memory consumption. In the family of flow-insensitive points-to analysis, we may choose to switch on/off the context-sensitivity, field-sensitivity and some other relevant options that influence the performance of the analysis such as subset-based or equality based algorithm. However, we believe it is important to stay with the category of flow-insensitive category, for to the best of our knowledge there has

been no flow-sensitive points-to analyses available that are able to run on large programs in Java.

For the numerical analysis part, while we leave the different numerical domains as possible instance of *NumP*, we will only apply these abstract domains in an intra-procedural way. It is known that the complexity of the operations of numerical domains are mainly decided by the number of variables as well as the line numbers of the analyzed programs. This means, to analyze a large program intra-procedurally has a exponential gain on time in terms of numbers of program procedures/methods.

5.2 Implementation

The abstract domain is implemented to test on its performance. The modular designing has to be respected for the ease of implementation.

5.2.1 Architecture

The structure of the analysis is shown in Fig. 5.1 as a *UML class diagram*: each rectangle represents a Java class, and the arrows indicate the relation between classes. Two kinds of class relations are recorded here: the solid-headed arrow called the *has a* relation, and the hollow-headed arrow standing for inheritance, or the *is a* relation. The analysis depicted as the left-most rectangle takes a generic structure in the sense that it is composed of the abstract domain *NumP* that implements abstract operators and the language parser WHILE_{np}. The kernel architecture is the encapsulation of traditional pointer analysis Pter as its component (the arrow from *NumP* to Pter , and the inheritance (the arrow from *NumP* to Num. PPL and SOOT are used as the back-ends of Num and Pter respectively (the arrows from Num to PPL and the arrow from Pter to SOOT). In addition, SOOT also provides the front-end functionality as the language parser (the arrow from WHILE_{np} to SOOT).

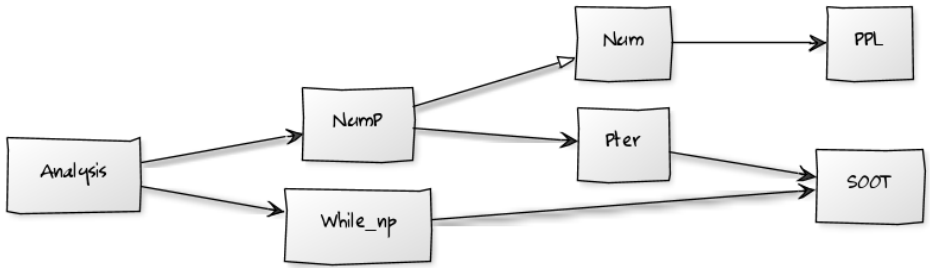


Figure 5.1: The architecture of the prototype represented as class diagram of UML

5.2.2 Work-flow

The analysis runs in three steps. First, it computes a global points-to graph using Pter. Then, it transforms each input class into Jimple codes using SOOT and enriches the intermediary representation with symbolic variables. At last, NumP performs a static numerical analysis of each transformed input class using an extended version of Num that takes symbolic variables into account. In this prototype, the side effects of function calls are not considered. The output of NumP is the invariants before each Jimple statement. The work-flow is shown as the prototype action diagram in Fig. 5.2

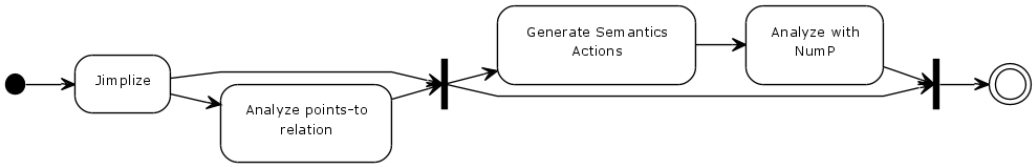


Figure 5.2: The work-flow of the prototype represented as action diagram of UML

As an illustrative example, consider the Java snippet in Listing. 5.1. In this example, an abstract class `Unsigned` uses unsigned numbers to represent both positive and negative values. `Unsigned` has two subclasses `Pos` and `Neg` for this purpose. It is the responsibility of clients to ensure the underlined contract, *i.e.*, objects of type `Unsigned` must hold non-negative values. The Java source code takes an array *buf* and passes the elements to the list *elem*

of type `List`. The `List` has a field *item* for data type `Unsigned` and a field *next* of type `List`. The compound condition structure (l. 7-14 of Listing. 5.1) creates an object of class `Pos` or `Neg` according to whether n is positive or not. For both cases, *data.val* will be assigned to the absolute values of n so that the assumed property of unsignedness can be preserved. From l. 15 to l. 19, the program allocates a new cell to store *data* and link it to the list created from the precedent iteration.

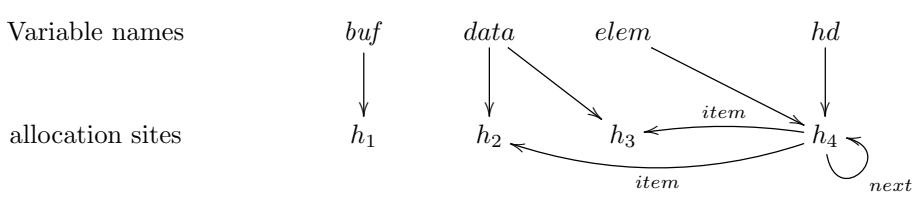
Our analysis is able to infer the following properties at the end of the program (l .21).

- **Prop1** Each list element of *hd* is in the range of 0 to 9:

$$\forall l \geqslant 0, hd.next^{l}.item.val \in [0.9]$$

- **Prop2** Each array element of *buf* is in the range of -9 to 7: $buf[*] \in [-9, 7]$
- **Prop3** The loop index *idx* is equal to or larger than the length of the array buf: $idx \geqslant buf.length$.

We start with a flow-insensitive points-to analysis. A single points-to graph for the whole program can be obtained. The graph has two kinds of arcs. Unlabeled arcs $v \rightarrow h$ from a variable v to an allocation site h, and labeled arcs $h \xrightarrow{f} h'$ between allocation sites h, h' with field f as label.



Semantically, the points-to graph disambiguates the heap and tells what must not alias. In line with this semantics, we derive a symbolic variable $\delta_{h,val}$ for each pair of heap location h and field *val*. The key insight is, numerical values bound to syntactically distinct symbolic variables are guaranteed to be stored at different concrete heap locations. It is therefore reasonable to deal with symbolic variables like with scalar variables.

We associate $buf[i]$ at l. 1 of Listing. 5.1 with a symbolic variable $\delta_{h_1,[*]}$, and $buf.length$ at l. 5 with $\delta_{h_1,length}$. Because variable $data$ points to h_2 and h_3, we associate $data.val$ at l. 9 and l. 13 with both symbolic variables $\delta_{h_2,val}$ and $\delta_{h_3,val}$, reflecting the fact that $data$ may be bound to an object `Pos` or `Neg`. Listing 5.2 illustrates the semantics actions taken by our analysis. It is

```
1  int[] buf = {-9,7,3,-5};//h1
2  Unsigned data = null;
3  List hd = null;
4  int idx = 0;
5  while (idx < buf.length){
6    int n = buf[idx];
7    if (n > 0){
8      data = new Pos();//h2
9      data.val = n;
10   }
11   else{
12     data = new Neg();//h3
13     data.val = -n;
14   }
15   List elem = new List();//h4
16   elem.item = data;
17   elem.next = hd;
18   hd = elem;
19   idx = idx + 1;
20 }
21 return;
```

Listing 5.1: A Java snippet

```
1  δ_{h1,length} ≐ 4;
2  δ_{h1,[*]} ≐ -9;
3  δ_{h1,[*]} ≐ 7;
4  δ_{h1,[*]} ≐ 3;
5  δ_{h1,[*]} ≐ -5;
6  idx = 0;
7  while (idx < δ_{h1,length}){
8    n = δ_{h1,[*]};
9    if (n > 0)
10     δ_{h2,val} ≐ n;
11     δ_{h3,val} ≐ n;
12   else
13     δ_{h2,val} ≐ -n;
14     δ_{h3,val} ≐ -n;
15   idx = idx + 1;
16 }
```

Listing 5.2: Semantics actions

Figure 5.3: An example in Java. The program passes an array of integers to a list of `Unsigned` numbers. `Unsigned` is a superclass of `Pos` and `Neg`. It has one field `val` of integer type. The class `List` has two fields, `item` of type `Unsigned`, and `next` of type `List`.

worth noting that more than one run-time heap locations may be associated with the same symbolic variable, e.g., $\delta_{h_1,[*]}$ corresponds to all heap locations of the array *buf*. By updating the symbolic variable to -9, 7, 3 and -5 successively, we perform a *weak update* (syntactically noted $\doteq$ in Listing 5.2), *i.e.*, *accumulating* values rather than *overwriting* them.

Finally, the analysis of the program in Listing 5.1 can be treated as an *extended* numerical analysis, with its semantics actions specified in Listing 5.2. This analysis is called "extended" because it not only deals with scalar variables, but also deals with symbolic variables. By performing an extended polyhedral analysis, we are able to infer the four invariants at the end of the program: $\delta_{h_2,val} \in [0, 9]$, $\delta_{h_3,val} \in [0, 9]$, $\delta_{h_1,[*]} \in [-9, 7]$ and $\delta_{h_1,length} - idx \leqslant 0$, which imply **Prop1**, **Prop2** and **Prop3** respectively.

Below, we give the analysis results from our analysis. On the right is the intermediate Jimple statement. On the left is the deduced constraints.

```
In                                                      JimpleStmt
---------------                                         -----------------
{true}                                                  r0 := @parameter0: java.lang.String[]
{true}                                                  $r4 = newarray (int)[4]
{true}                                                  #d_0_$r4[0] = -9
{#d_0 = -9}                                             #d_0_$r4[1] = 7
{#d_0 >= -9, -#d_0 >= -7}                               #d_0_$r4[2] = 3
{#d_0 >= -9, -#d_0 >= -7}                               #d_0_$r4[3] = -5
{#d_0 >= -9, -#d_0 >= -7}                               r1 = $r4
{#d_0 >= -9, -#d_0 >= -7}                               n0 = null
{#d_0 >= -9, -#d_0 >= -7}                               r2 = null
{#d_0 >= -9, -#d_0 >= -7}                               i0 = 0
{i0 = 0, #d_0 >= -9, -#d_0 >= -7}                       goto [?= $i3 = #d_1_r1.<LEN>]
{$i3 - #d_1 = 0, -#d_0 >= -7, -#d_5 >= -9,
 -#d_8 >= -9, #d_8 >= 0, #d_5 >= 0, #d_0 >= -9,
 i0 >= 0, -i0 + $i3 > 0}                                i1 = #d_0_r1[i0]
{$i3 - #d_1 = 0, i1 - #d_0 = 0, i0 >= 0, -i0 + #d_1 > 0,
 -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_0 >= -9,
  #d_5 >= 0, #d_8 >= 0} if i1 <= 0                      goto $r7 = new unsigned4.Neg
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0,
 -i1 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0,
  #d_5 >= 0, i1 > 0, i0 >= 0}                           $r5 = new unsigned4.Pos
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0,
 -i1 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0,
 #d_5 >= 0, i1 > 0, i0 >= 0}                            specialinvoke $r5.<unsigned4.Pos: void <init>()>()
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0,
  -i1 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0,
  #d_5 >= 0, i1 > 0, i0 >= 0}                           r6 = $r5
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0,
 -i1 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0,
  #d_5 >= 0, i1 > 0, i0 >= 0}                           #d_5_8_r6.val = i1
{i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0, #d_1 > 0,
  -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0,
  #d_0 > 0, #d_5 >= 0}                                  goto [?= $r8 = new unsigned4.List]
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0,
 -i1 >= 0, -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0,
  #d_5 >= 0, i1 >= -9, i0 >= 0}                         $r7 = new unsigned4.Neg
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0, -i1 >= 0,
 -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0, #d_5 >= 0, i1 >= -9,
 i0 >= 0}                                               specialinvoke $r7.<unsigned4.Neg: void <init>()>()
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0, -i1 >= 0,
 -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0, #d_5 >= 0, i1 >= -9,
 i0 >= 0}                                               r6 = $r7
{$i3 - #d_1 = 0, i1 - #d_0 = 0, -i0 + $i3 > 0, -i1 >= 0,
  -#d_5 >= -9, -#d_8 >= -9, #d_8 >= 0, #d_5 >= 0,
  i1 >= -9, i0 >= 0}                                    $i2 = neg i1
{$i3 - #d_1 = 0, i1 + $i2 = 0, i1 - #d_0 = 0,
 i0 >= 0, -i0 + #d_1 > 0, -i1 >= 0, -#d_5 >= -9,
 -#d_8 >= -9, i1 >= -9, #d_5 >= 0, #d_8 >= 0}           #d_5_8_r6.val = $i2
{i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0, #d_1 > 0,
  -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9,
 #d_5 >= 0, #d_0 >= -9, #d_8 >= 0}                      $r8 = new unsigned4.List
{i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0,
 #d_1 > 0, -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9,
```

```
   #d_5 >= 0, #d_0 >= -9, #d_8 >= 0}                        specialinvoke $r8.<unsigned4.List: void <init>()>()
  {i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0,
   #d_1 > 0, -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9,
   #d_5 >= 0, #d_0 >= -9, #d_8 >= 0}                        r3 = $r8
  {i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0, #d_1 > 0,
   -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_5 >= 0,
   #d_0 >= -9, #d_8 >= 0}                                   r3.<unsigned4.List: unsigned4.Unsigned item> = r6
  {i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0, #d_1 > 0,
   -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_5 >= 0,
   #d_0 >= -9, #d_8 >= 0}                                   r3.<unsigned4.List: unsigned4.List next> = r2
  {i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0, #d_1 > 0,
   -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_5 >= 0,
   #d_0 >= -9, #d_8 >= 0}                                   r2 = r3
  {i1 - #d_0 = 0, i0 >= 0, -i0 + $i3 > 0, #d_1 > 0,
    -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_5 >= 0,
    #d_0 >= -9, #d_8 >= 0}                                  i0 = i0 + 1
  {-#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9, #d_0 >= -9,
  #d_5 >= 0, i0 >= 0, #d_8 >= 0}                            $i3 = #d_1_r1.<LEN>
  {$i3 - #d_1 = 0, -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9,
   #d_5 >= 0, #d_0 >= -9, #d_8 >= 0, i0 >= 0}               if i0 < $i3 goto i1 = #d_0_r1[i0]
  {$i3 - #d_1 = 0, -#d_0 >= -7, -#d_5 >= -9, -#d_8 >= -9,
   #d_8 >= 0, #d_5 >= 0, #d_0 >= -9, i0 >= 0, i0 - $i3 >= 0} return
~end~.
```

5.3 Experimental Results

5.3.1 Bellman-Ford

A case study is carried out on a small program, `Bellman-Ford`, taken from the benchmarking of `Jchord` [51][1]. Its small size (< 500 LOC in Jimple IR) allows us to run different combined analyses, including the expensive polyhedral numerical analysis and the context-sensitive points-to analysis. The objective here is to "plug in" various combinations and evaluate their precision/cost tradeoff.

It is hard to compare the precision between NumP and Num. One metric that we find reasonable is the total number of constraints contained in the inferred invariants. Recall that NumP uses exactly the same transfer functions from Num for statements in WHILE_n, and is able to deal with statements that are in WHILE_{np} but are not in WHILE_n. We count the number of non-trivial invariants (an invariant is trivial if it is `true`) generated by PPL. An

[1]See http://code.google.com/p/jchord/source/browse/trunk/test/bench/bellman-ford/src/BellmanFord.java?r=1550.

Table 5.1: Case study on `Bellman-Ford`

	Analysis	invariants	time (s)
Num	`intv`	984	3.54
	`poly`	1141	5.82
Pter	`spark`	0	54.45
	`geom`	0	114.24
NumP	`intv` + `spark`	1180	77.84
	`intv` + `geom`	1124	112.32
	`poly` + `spark`	1460	92.68
	`poly` + `geom`	1661	115.40

invariant in PPL is a conjunction of unit inequalities. In our context, these invariants may involve symbolic variables. We count $K + 2$ times for an invariant expressed as $\{x \leqslant 3, y \leqslant 4, \delta \leqslant 5\}$ if δ represents K field expressions *that literally appear in the program.*

The first two columns of Tab. 5.1 show the instanced analyses, with `intv` denoting the interval abstract domain (`Int64_Box` from PPL), `poly` denoting the polyhedral abstract domain (`NNC_Polyhedron` from PPL), `spark` denoting the flow-insensitive, context-insensitive points-to analysis used by default in SOOT (from SPARK), and `geom` denoting the flow-insensitive, context-sensitive points-to anlaysis using geometric encoding algorithm [74] (from SPARK).

The last two columns show the number of inferred invariants and the time consumed by each analysis. The results confirm the expectation *viz.*, that the numerical analysis combined with pointer analysis infers more invariants than the numerical analysis only, and is more expensive. The best precision is obtained by polyhedral analysis combined with geometric encoding points-to analysis, which is also the most time-consuming. For this small program, the time spent by interval and polyhedral analyses is negligible compared with pointer analysis. This is because Num is intra-procedural so its complexity depends only on the length and the variable number of the program itself, whereas the points-to analysis is inter-procedural so its complexity depends

on its dependent classes which are more than 10000 for this small program. Also note that the time spent by NumP is not necessarily more than the addition of its component analyses (compare `poly`, `geom` and their combined `poly` + `geom` for example). This might be due to the fact that we are using SOOT front-end which transfers programs to Jimple before each analyses.

Below, we use the invariant number and the consumed time as two performance indicators.

5.3.2 Dacapo

We use `Dacapo-2006-MR2` to evaluate our analysis on real-world programs. Tab. 5.2 gives our experimental results for the combined `intv` +`spark`. We have also tried with polyhedral analysis in which case neither NumP nor Num is able to run over any of the benchmarks. Column 1 of the table gives the eight chosen benchmarks. In column 2, i_{Num} and i_{NumP} are the invariants numbers discovered by Num and by NumP respectively. The invariants number is the total non-trivial invariants collected from each individual method. We use

$$q_i \triangleq i_{\mathsf{NumP}}/i_{\mathsf{Num}} - 1 \tag{5.1}$$

as the indicator for the *precision enhancement*.

In column 3, we show the time consumed by NumP, Num and Pter. t_{NumP}. What we really care about is the time spent by NumP compared with its combined components. The *time overhead* is quantified with q_t defined as

$$q_t \triangleq t_{\mathsf{NumP}}/(t_{\mathsf{Num}} + t_{\mathsf{Pter}}) - 1 \tag{5.2}$$

where t_{Num}, t_{Pter} and t_{NumP} are the total time spent by the analyzers (in seconds).

In Tab. 5.2, NumP gives an average of 29.6× precision enhancement with a time overhead of 0.13. This is meaningful: (1) Traditional Num is shown insufficient to analyze numerical properties in real-world programs because a large number of the numerical invariants involved in the program logic are not expressible by scalar variables. It is with the help of a combined pointer analysis that these alien invariants to Num may be discovered. (2)

Table 5.2: Performance test of NumP for the `Dacapo-2006-MR2` benchmark

Benchmark	Invariants Numbers			Analysis Time (s)			
	i_{Num}	i_{NumP}	q_i	t_{Num}	t_{Pter}	t_{NumP}	q_t
bloat	70238	650091	8.3	16.6	62.6	98.8	0.25
chart	76972	905011	10.8	18.5	137.6	158.1	0.01
eclipse	58377	80875	0.4	14.8	40.5	56.1	0.01
fop	69170	12354926	177.6	23.2	136.3	300.4	0.88
hsqldb	154151	3328080	20.6	30.6	277.9	345.7	0.12
jython	105775	460900	3.4	181.6	134.5	204.4	-0.35
pmd	50023	425933	7.5	15.61	120.0	140.0	0.03
xalan	109147	1050445	8.6	17.02	91.9	122.1	0.12
MEAN	86732	2407033	29.6	39.7	125.2	178.2	0.13

The proposed NumP has the full capability to achieve this ambition because it has little complexity overhead compared to its component analyses.

.

Chapter 6

Conclusions

In this thesis, we presented a static analysis that is able to infer numerical properties in programs with pointers. The analysis has a modular construction which allows us to deal with the tradeoff between efficiency and accuracy by tuning the granularity of abstraction and the complexity of the abstract operators.

As a first contribution, we provided the theoretical framework about the combination of numerical analysis and pointer analysis. We provided formal definition of the combined operators through the operators already introduced in the literature. We proved that the derived abstract operators are correct by construction using the theory of abstract interpretation.

A second contribution of this thesis is the theoretical development of an algorithm of partial redundancy elimination of points-to graph by taking advantage of must alias analysis. By tracking must-alias information, for example, that can be gleaned from conditionals and assignments, it is possible to refine points-to graph during its synthesis. We formalized the algorithm, proved its correctness, and showed its incompleteness.

Through the combination of the pointer analysis and the numerical analyses, we obtained a strictly more accurate analysis and more precise results. The main goal of this refinement consists in the automatic discovery of numerical invariants in Java-like programs, which are in general pointer-aware. This permits to make applicable our analysis to practical cases. Moreover, we afforded a modular construction which allows to deal with the tradeoff

between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators. Notice that further refinement may be possible by enhancing the points-to graph using must-alias.

Another contribution of this thesis is the NumP tool: a new tool which aims at numerical properties of Java-like programs. NumP is written in Java, and to be used for Java. We have successfully combined the pointer analyses in SPARK, and the numerical abstract domains in PPL to modular generate a static numerical analyzer in the presence of pointers. The using of object-oriented technologies and design patterns makes the prototype NumP fully extensible to a large range of numerical domains and pointer analyses engines. The preliminary results gave us the confirmation of theoretical results about efficiency and accuracy.

Possible future works include:

In theory, we need to leverage the current static numerical analysis to enable strong update. Note that precision of the NumP is affected by the weak update algorithm used in this approach. This may be where we can connect the must-alias analysis and NumP.

In practice, the static numerical analyzer NumP needs to be further developed to take side effects of function calls into account This can be either achieved with a fully inter-procedural analysis with context sensitivity taken into account or not following the incurred complexity, or we can adopt a cheaper side-effect analysis to conservatively simulate the side-effects of procedure invocation. As usual, in-lining may be performed prior to the inter-procedural.

Bibliography

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Technical Report 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.

[3] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 213–229. Springer-Verlag, 2002.

[4] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[5] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI*, pages 103–114, 2003.

[6] Garrett Birkhoff. Lattice theory. In *Colloquium Publications*, volume 25. Amer. Math. Soc., 3. edition, 1967.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel,

A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[9] Eric Bodden, Patrick Lam, and Laurie Hendren. Instance keys: A technique for sharpening whole-program pointer analyses with intraprocedural information. Technical Report SABLE-TR-2007-8, October 2007.

[10] Nicolas Bourbaki. *Elments de mathmatique. Thorie des ensembles*. Hermann, Paris, 1970.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[12] Peter J. Cameron. Notes on counting. Available at http://www.maths.qmw. ac.uk/pjc/notes/counting.pdf.

[13] V.T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *ACM SIGPLAN Notices*, volume 38, pages 115–125. ACM, 2003.

[14] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI '05*, pages 147–163, 2005.

[15] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401, 2007.

[16] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

[17] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS*, pages 309–325, 2009.

[18] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[19] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93*, pages 232–245, 1993.

[20] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French).* Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.

[21] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[22] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[23] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[24] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[25] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[26] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[27] Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *ECOOP*, pages 665–687, 2012.

[28] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *ICCL*, pages 2–13, 1992.

[29] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, March 1997.

[30] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.

[31] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[32] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.

[33] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.

[34] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. Tval+ : Tvla and value analyses together. In *SEFM*, pages 63–77, 2012.

[35] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[36] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.

[37] Tobias Gutzmann. *Towards a Gold Standard for Points-to Analysis*. PhD thesis, Linnæus University, 2010.

[38] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[39] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. *SIGPLAN Not.*, 44(1):226–238, January 2009.

[40] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of PASTE 2001*, pages 54–61. ACM, 2001.

[41] Michael Hind and Anthony Pioli. Which pointer analysis should I use. In *In Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[42] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[43] H.B.M Jonkers. Abstract storage structures. *In De Bakker and Van Vliet, editors, Algorithmic languages*, pages 321–343, IFIP, 1981.

[44] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[45] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In *SAS*, pages 265–282, 2012.

[46] W.A. Landi. Interprocedural aliasing in the presence of pointers, 1992. Technical Report LCSR-TR-174 and PhD Thesis.

[47] William Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[48] Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.

[49] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–16, New York, NY, USA, 2011. ACM.

[50] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[51] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM.

[52] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *ESEC / SIGSOFT FSE*, pages 317–326, 2003.

[53] Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI*, pages 283–298, 2007.

[54] Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, pages 71–99, 2010.

[55] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.

[56] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[57] Rupesh Nasre. *Scaling context–sensitive points–to analysis*. PhD thesis, Computer Science and Automation, Indian Institute of Science, BANGALORE 560 012l, 2012.

[58] Anthony Pioli and Michael Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical report, IBM T. J. Watson Research Center, 1999.

[59] Derek Rayside. Points-to analysis, 2005. `www.cs.utexas.edu/ pingali/CS395T/2012sp/lectures/points-to.pdf`.

[60] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 296–309, New York, NY, USA, 2005. ACM.

[61] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[62] Shmuel Sagiv, Nissim Francez, Michael Rodeh, and Reinhard Wilhelm. A logic-based approach to program flow analysis. *Acta Inf.*, 35(6):457–504, 1998.

[63] Olin Shivers. Control-flow analysis of higher-order languages. Technical report, 1991.

[64] A. Simon. *Value-Range Analysis of C Programs*. Springer, August 2008.

[65] Pascal Sotin and Bertrand Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In *ESOP'11 : Proceedings of the 20th European Symposium on Programming*, pages 459–479, 2011.

[66] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. of POPL 1996*, pages 32–41. ACM Press, 1996.

[67] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *POPL*, pages 81–95, 2000.

[68] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[69] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[70] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, page 13, 1999.

[71] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, July 1998.

[72] Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE*, pages 227–236, 2005.

[73] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[74] Xiao Xiao and Charles Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for java. In *ISSTA*, pages 188–198, 2011.

[75] Jianwen Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 150–157, New York, NY, USA, 2002. ACM.