A Generic Approach to Bytecode Analysis

by

Mario Méndez-Lojo

B.S., Computer Science, University of La Coruna, 2000
M.S., Computer Science, University of New Mexico, 2006
M.B.A., Anderson School, University of New Mexico, 2008

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Dedication

To my mother, Inna, every single cyclist on Earth, and all the people who were part of this experience.

Acknowledgments

First of all, I want to express my gratitude to my thesis advisor, and director of the CLIP Research Group, Manuel Hermenegildo, for all his time, energy, and resources that have allowed me to finish this thesis. In addition to his availability at any time, he has encouraged me to continue with this work in the hardest moments and when I needed it the most. It has been a great honor to be part of this research group, and I hope I have achieved the required scientific level for participating in the activities developed by the group. I also gratefully acknowledge the support of the Prince of Asturias Chair in Information Science and Technology at UNM funded by Iberdrola.

I also want to thank Jorge Navas for his collaboration and friendship, that have resulted in a fruitful series of common work that are a fundamental part of this thesis, and that I hope we can continue in other future projects.

I have to mention also the members of the CLIP group, with whom I have shared good times. I do not have enough space here to refer to all of them without missing any one, but I want to thank especially Amadeo Casas, with whom I had many useful scientific discussions and shared plenty of good moments in Albuquerque.

Together with them, the rest of the co-authors of the papers in which I have been involved have contributed to my scientific education: Mark Marron (University of New Mexico), and Ondřej Lhoták (University of Waterloo). Their different perspective about research problems we had in common enlighted my critical thinking.

Finally, I will like to refer to all the people who might not have contributed to this thesis directly, but without their participation it would have been impossible to make it: Guille, Marcos, Xaquin, Miguelo, Annette, Fernando, Roberto, Tiki, Pulse, Sergio and his blog, Marie-Michele, Sarah, Eric, Marion, Inna, Andras, Salvador, Japji...

A Generic Approach to Bytecode Analysis

by

Mario Méndez-Lojo

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

A Generic Approach to Bytecode Analysis

by

Mario Méndez-Lojo

B.S., Computer Science, University of La Coruna, 2000

M.S., Computer Science, University of New Mexico, 2006

M.B.A., Anderson School, University of New Mexico, 2008

Ph.D., Computer Science, University of New Mexico, 2008

Abstract

Analysis of the Java language (either in its source version or its compiled bytecode) using the framework of abstract interpretation has been the subject of significant research in the last decade. Most of this research concentrates on finding new abstract domains that better approximate a particular concrete property of the program analyzed in order to optimize compilation or statically verify certain properties about the run-time behavior of the code. In contrast to this concentration and progress on the development of new, refined domains, there has been comparatively little work in the development of extensible, generic frameworks, an issue which is at the core of the analysis. The first component in such a generic framework is a standard representation of the program that facilitates later analyses or optimizations. Although

many times the description of that Control Flow Graph is omitted, we show that a uniform, compact representation is fundamental in order to manipulate similar constructions of the language in a uniform way. The Horn clause-based representation chosen is general enough to represent not only object-oriented programs, but also logic programming applications, or multi-language applications that combine both paradigms.

In the context of the abstract interpretation framework, the fixpoint algorithm that lies at its very core has been demonstrated to have dramatic impact in the efficiency and precision of any analysis. A particularly optimal combination of the two attributes can be achieved by a flow-sensitive, context-sensitive fixpoint algorithm, provided certain optimizations are included. We present a detailed description of such an algorithm, which handles mutually recursive nodes in the Control Flow Graph and uses memoization for further efficiency.

Generic abstract interpretation frameworks work in conjunction with abstract domains. Each domain captures a particular property of the program. A very interesting characteristic to analyze is whether a set of variables share, i.e., whether they might reach the same memory location. The information gathered by a sharing domain is used for parallelization and/or for optimizing the compilation. What we present is a combination of domains (sharing, nullity, and types) which can work together to refine their results, i.e., be more precise. The approach is shown to achieve better results than a previous sharing analysis.

The combinatorial nature of the set sharing domain has been the subject of intense debate within the analysis community. The exponential complexity of some of the operations (e.g., abstract unification in Prolog or field load/store in Java) seemed to be a big obstacle that prevented the domain from being widely used. In this thesis, we present a more efficient implementation of set sharing, which is based on a special type of Binary Decision Diagrams, called Zero-supressed Binary Decision Diagrams

(ZBDDs). By representing sets of sets with this data structure, we not only improve the memory requirements of the analysis but also achieve better efficiency in terms of the overall running time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Bytecode Analysis

Analysis of the Java bytecode [LY97] using the framework of abstract interpretation [CC77] has been the subject of significant research in the last decade (see, e.g., [LC05] and its references). The abstract interpretation approach brings a useful combination of characteristics: it is automatic and practical, producing useful results for a good number of applications, while at the same time being rigorous and semantics-based. On the other hand, a low-level, object-oriented language such as Java bytecode is interesting because it introduces novel challenges that complicate the application of existing (for logic or imperative languages) analysis techniques: 1) its unstructured control flow, e.g., the use of `goto` statements rather than recursive structures; 2) its object-oriented features, like virtual invocations or static methods; and 3) its stack-based model, in which stack cells store intermediate values.

Most of the research focused on Java bytecode concentrates on finding new abstract domains that better approximate a particular concrete property of the program analyzed in order to optimize compilation (e.g., [Bla99, Ruf00]) or statically verify

certain properties about the run-time behavior of the code (e.g., [GS05, Ler01]). In contrast to this concentration and progress on the development of novel domains, there has been comparatively little work in the underlying representations, fixpoint algorithms, and frameworks. In fact, many existing abstract interpretation-based analyses do not actually describe the particular Control Flow Graph (representation of the program) format used [Log07], use relatively inefficient fixpoint algorithms [Spo05], or use solutions tied to particular analyses and that cannot easily be reused for other domains. We believe this can be improved upon: one of the stated advantages of abstract interpretation-based tools is that they can be made highly independent of the particular analysis, but the limitations mentioned make most of the work done in the area comparatively difficult to reuse in other contexts.

A particular property of interest that has received much attention is the memory state, i.e., the layout of the memory at any program point. This classic problem has been approached using standard flow analyses, abstract interpretation, or hybrids for many different languages. Different categorizations of the existing solutions are 1) alias analysis [WL95, BCCH94, HBCC99, LR92], which identifies variables referring to the exact memory location; 2) shape analysis [GH96, SRW99, RS01], which aims at providing a relatively accurate, graph-based abstraction of the memory layout at each program point, normally at some efficiency cost; 3) points-to analysis [Ste96, BLQ$^+$03, EGH94], which determines an approximation of the set of variables and heap cells that are reachable from every variable of the program, including those that are dynamically allocated; 4) sharing analysis [JL89, MH89, SS05], which detects which variables do not share in memory, i.e., do not point (transitively) to the same location. Another possible classification attends to properties of the framework itself rather than the domain: flow-insensitive analyses do not take into account the sequential order of the statements in the program, resulting in lower running times but less accurate results when compared to flow-sensitive counterparts. An analysis is context-sensitive if it differentiates among different calls

(calling points in the program, or, in some cases, calls from different states) to a procedure. Context-insensitive analyses mix all such calls providing summary information which is valid for all of them but of course also less precise. The additional precision obtained from context sensitivity has been shown to be important in practice in, e.g., the heap analysis of logic programs [BdlBH99] and, more recently, in object-oriented programs [WL04].

There has been extensive work in recent years on the use of Binary Decision Diagrams [Zhu02, BLQ+03, WL04, ZC04] to represent points-to information. The work presented in [Lho06] represents a step forward towards a standard representation of set-based domains using BDDs. Standard BDDs are a data structure frequently used for representing large amounts of data within the static analysis and the model checking (see [iM96] and its references) communities. However, BDDs do not perform well in all problems. In particular, a variant of the standard BDD structure is needed when confronted with the problem of how to represent sets of sets of elements. In that environment, Zero-supressed BDDs seem to be a reasonable solution, assuming certain properties about the sparsity of the data to be represented. ZBDDs were introduced by Minato [iM93] and applied to a great diversity of problems in model checking (e.g., [YHTiM96, Cou97, iM01]). More recently, Lhoták *et al.* have applied ZBDDs to the exploration of infinite state spaces [LSJ07] in the context of points-to analysis. Their main contribution -apart from the introduction of ZBDDs for static analysis- is an experimental evaluation of the performance of existing points-to analyses when the representation uses ZBDDs, and not BDDs. However, there is still an open question about the use of ZBDDs for implementating domains that do not have a natural representation using standard BDDs, as is the case of combinatorial domains (type analyses that associate variables with sets of types, sharing sets, etc.).

1.2 Thesis Objectives

The first objective of the work presented in this thesis is to present a generic framework for analysis of Java bytecode (although many concepts might be applied to similar, object-oriented languages) by using abstract interpretation. The word *generic* means in this context that a) the internal representation chosen for the program isolates later analyses from intricate constructions of the language, and b) we can perform different analyses without any change other than plugging in the different abstractions. The second objective of our work is to show the applicability of such a generic framework by implementing a more precise and efficient sharing analysis on top of it, that benefits from both the more precise results obtained by complementary abstract domains, and the compact intermediate representation chosen.

1.3 Overview of the Main Contributions

1. The first contribution is an intermediate representation used to represent any bytecode program. Our solution is based on the use of Horn clauses, which represent naturally and in a simple way alternative program flows (conditionals, virtual invocations, etc.), blocks, control, etc. The resulting graph is compact and uniform, characteristics that translate into a more uniform view of the original program for the analysis, and that facilitate performing compiler optimizations at that level. This work has been done in collaboration with Jorge Navas (University of New Mexico) and has been published at the International Symposium on Logic-based Program Synthesis and Transformation in 2007 [MLNH07a]. I am the main contributor to this work.

2. We also propose an optimized fixpoint algorithm that receives a Control Flow Graph and computes an approximation of the state of the program (in terms

of the abstraction chosen) for all the statements in that graph. Precision is ensured by using a context-sensitive approach, while efficiency relies on memoizing past computations. A first description [MLNH07b] of the algorithm was published at the ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07) in 2007; an improved version [NMLH07] was later published at the 9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007. In both cases I collaborated with Jorge Navas, from the University of New Mexico. We have both contributed in approximately equal amounts to this work.

3. We also introduce a sharing abstract domain that works in conjunction with the fixpoint algorithm mentioned above. The abstract operations are shown to improve the precision of the results, when compared to recent, related work. The corresponding paper [MLH08] has been published at the 9th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2008. I am the main contributor in this work.

4. Finally, we also present a novel representation of the set sharing abstraction, based on Zero-supressed Binary Decision Diagrams (ZBDDs). To the best of our knowledge this is the first link provided between set sharing and ZBDDs. This work has been done in collaboration with Ondřej Lhoták (University of Waterloo, Canada) and has been published [MLLH08] at the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC 2008). I am the main contributor in this work.

Contribution 1 is the object of study in Chapter 2. The fixpoint algorithm (contribution 2) is the main topic of Chapter 3. The set sharing abstract domain (contribution 3) is presented in Chapter 4, while Chapter 5 provides an in-depth look at the work performed in relating set sharing with ZBDDs (contribution 4). Finally, Chapter 6 presents the conclusions and suggests the future work to be made.

Chapter 2

Intermediate Representation

2.1 Motivation

The gap between programs and semantics is greater in the case of object-oriented languages than in, for example, declarative languages. For this reason, static analyses of object-oriented programs usually rely on intermediate languages that respect the original semantics while having a more uniform and basic syntax (e.g., block-based representations) and a more declarative semantics (e.g., static single assignment transformations). Some significant concrete examples which have been proposed of such intermediate representations for object-oriented programs are Jimple [VRHS+99] for Java or BoogiePL [DL05] for .NET.

In this chapter we propose the use of a Horn clause-based representation as an intermediate language. This is useful for at least two reasons. The first advantage derived from our approach is that we can then adapt and extend a mature logic programming framework (in our case, CiaoPP [HPBLG03]) and apply the resulting system to the analysis of Java bytecode. A second strength is that the new framework can now be easily adapted to the analysis of other languages without having to

redefine the fixpoint algorithm, as we will see in Chapter 3. In fact, using the transformation and intermediate representation that we propose, from the analyzer point of view an object-oriented program is indistinguishable from, e.g., a Prolog one (although of course different abstract domains and definitions of pseudo-builtins are used). This brings in the additional advantage of being able to analyze multiple languages within the same framework.

We start the chapter by looking at the main components of the analyzer (Section 2.2). The transformation process is described in detail in Section 2.3. We then illustrate the application of our approach to other languages, such as C# (Section 2.5). Intermediate representations used by related abstract interpretation-based frameworks are discussed in Section 2.6, and Section 2.7 presents the chapter conclusions.

2.2 Overview of the analysis framework

Our framework is composed of a front-end preprocessor and a back-end analyzer, as shown in Figure 2.1. The preprocessor transforms an input in Java bytecode into a set of Horn clauses that represent a safe approximation of its standard semantics (Section 2.3). The decompilation process is based on a postprocessing of the Jimple representation returned by the Soot [VRHS+99] tool.

The resulting Horn-clause intermediate representation is then analyzed using a framework based on the CiaoPP analyzer [HPBLG03], thus benefiting from its advanced features: efficient computation of fixpoints using memoization, context-sensitivity, modularity, etc. The programmer needs only to implement the particular abstract domain of interest, which includes also defining the abstract meaning of a set of "built-in" predicates that represent the language-dependent semantics of the basic operations of the source language. This has been the case with the set shar-

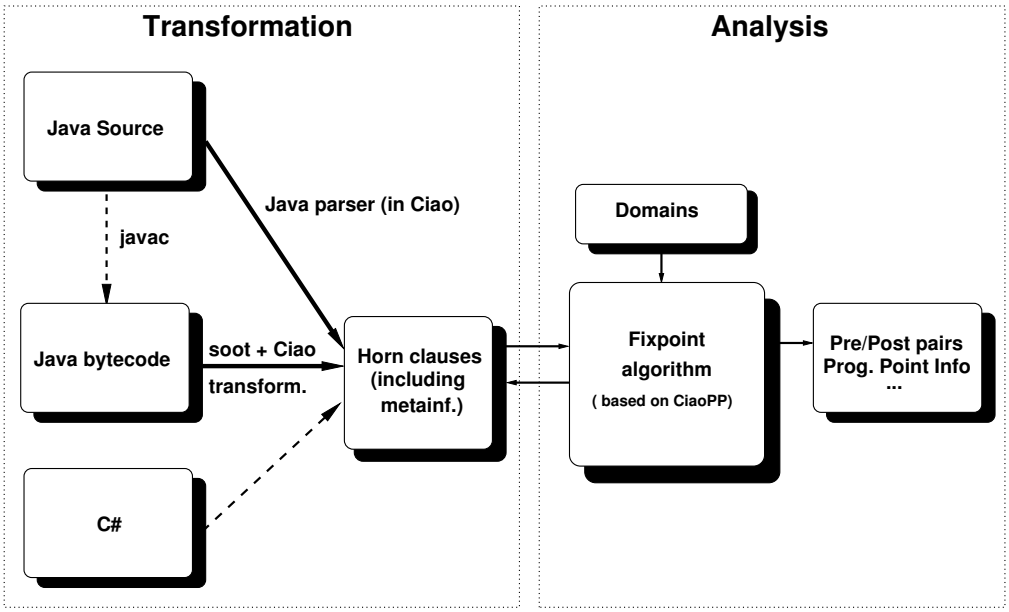

Figure 2.1: Transformation and analysis pipeline.

ing (main topic of Chapters 4 and 5) domain and resource usage (please refer to Chapter 6) analysis.

On the other hand, our approach does liberate the designer of an analysis from the burden of coding a fast, reliable, and efficient abstract interpretation platform. Analysis results are computed in the standard form (p, σ), where p uniquely identifies a program point and σ is an abstract state which safely approximates all the possible states at that program point during run time (more precisely, a set of abstract states, one for each different abstract entry state into each block –see Chapter 3). Information stored during the transformation process allows relating those line numbers with the ones of the original bytecode or source program, making it possible to reflect back the results on the original program text (as JML-like assertions [LBR06]), pinpoint errors in the original program, or implement compiler optimizations.

Other languages can be incorporated into the framework (i.e., analyzed) by pro-

viding a correct transformation for them. For example, support for other object-oriented languages like C#, that share many syntactic and semantics features with Java, is easily achievable as illustrated in Section 2.5. In addition, programs written in Ciao [HC94, HBC$^+$08, BCC$^+$06] are also accepted by the system as input, provided there is an appropriate abstract domain for them.

2.3 Transformation of Java bytecode

Analysis of a Java bytecode program normally requires its translation into an intermediate representation that is easier to manipulate. In particular, our decompilation (assisted by the Soot [VRHS$^+$99] tool) involves elimination of stack variables, conversion to three-address statements, static single assignment (SSA) transformation, and generation of a Control Flow Graph (CFG) that is ultimately the subject of analysis. Our ultimate objective is to support the full Java language but the current transformation has some limitations: it does not yet support reflection, threads, generics or runtime exceptions. The rest of the 1.5 specification is supported.

The compiler receives as input a Control Flow Graph, expressed in the Jimple language, and outputs another Control Flow Graph. The following grammar describes that final intermediate representation; some of the elements in the tuples are named so we can refer to them as $node.\mathsf{name}$.

$$
\begin{array}{lcl}
CFG & ::= & Block^+ \\
Block & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{fpars}{:}Id^+,\mathsf{annot}{:}expr^*,\mathsf{body}{:}Stmt^*) \\
Sig & ::= & (\mathsf{class}{:}Type,\mathsf{name}{:}Id,\mathsf{pars}{:}Type^+) \\
Stmt & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{apars}{:}(Id|Ct)^+) \\
Var & ::= & (\mathsf{name}{:}Id,\mathsf{type}{:}Type)
\end{array}
$$

The main rules followed by the compiler during the transformation are:

- Every Jimple block is transformed into a *block method*. A block method is similar to a Java method, with some particularities: a) if the program flow reaches it, every statement in it will be executed, i.e, it contains no branching; b) its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types. A block method is the equivalent of a Horn *clause* in a logic program, while a set of block methods with the same signature (name, number of formal parameters, and types of the formal parameters) is the equivalent of a *predicate* in a logic program.

- There is no branching within a block method. Instead, each conditional `if` *cond* $stmt_1$ `else` $stmt_2$ in the Jimple program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of decompiling *cond*, while we attach $\overline{cond}$ to its sibling.

- We add an extra formal parameter to any non-static method call, in order to include a reference to the callee object.

- We add an extra formal parameter to any non-void method, representing the value returned.

- For every formal parameter (*input* formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the block method that contains its final version in the SSA transformation (*output* formal parameter).

- Variable declarations are removed; instead, all the statements in the intermediate representation are typed, thus the analysis can retrieve the type of a variable from them.

```
import java.net.URLEncoder;
public class CellPhone {
 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
  if (smsPk != null) {
    String newSms = enc.format(smsPk.sms);
    stm.send(newSms);
    smsPk.next=sendSms(smsPk.next,enc,stm);
    smsPk.sms = newSms;
  }
  return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  native void send(String data);
}
```

Figure 2.2: Motivating example: Java source code

- Return statements are removed. If they are nested within a non-void method, the return statement is replaced by an assignment to the formal parameter that represents the method output.

- Virtual invocations are replaced by static calls to all the block methods that could possibly be invoked at runtime. A set of block methods with the same signature sig can be retrieved by the function $getBlocks(CFG, sig)$.

- In the Jimple representation, the control flow is implicit by each block having a set of successors. In our IR, that relation is made explicit, since the non-terminal block methods contain a static invocation to their successors. Therefore, there are many more invocations in our IR than in the Jimple CFG.

- For every Jimple statement, there is a corresponding statement in our IR. For instance, a Jimple `AssignStmt` is converted into a `Builtin.asg` statement.

- Every statement in a block method is an invocation, including builtins (assignment `asg`, field dereference `gtf`, etc.), which are understood as block methods of the class `Builtin`.

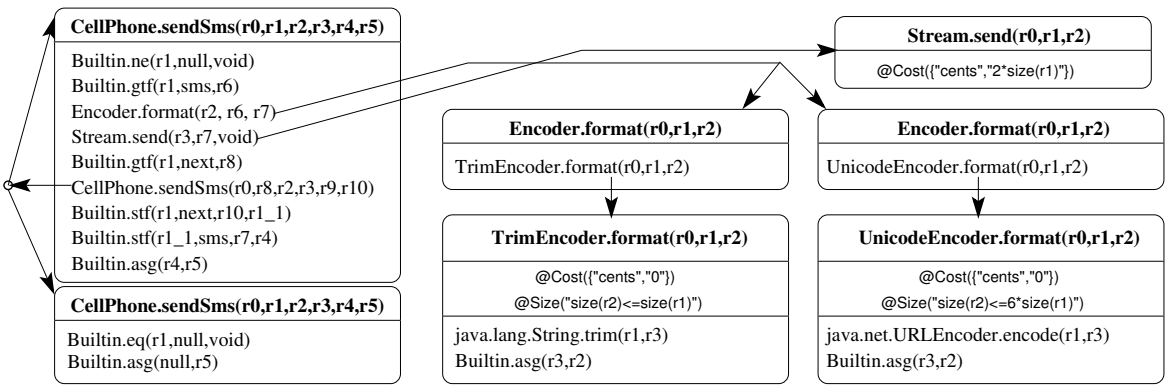

Figure 2.3: Motivating example: resulting CFG

- Java 1.5 annotations are preserved as metainformation attached to each method or class, even when the current version of Soot does not preserve annotations. This is done instead directly by our tool.

Example 2.3.1 We now consider the source code for the `CellPhone.sendSms` method listed in Figure 2.2. Its transformation results in the two leftmost *clauses* of the set of block methods depicted in Figure 2.3. Input formal parameters r_0, r_1, r_2, r_3 correspond to $this$, $smsPk$, enc, and stm, respectively. In the case of r_1, the contents of its fields *next* and *sms* are altered by invoking the `stf` (abbreviation for `setfield`) builtin block method. The output formal parameter r_4 contains the final state of r_1 after those modifications. The value returned by the block methods is contained in r_5. The type information has been omitted from Fig. 2.3, although we mentioned before that every statement contains that information. In the case of `Encoder.format`, for example, we say that there are two blocks with the same signature because they are both defined in class `Encoder`, have the same name (`format`) and list of types of formal parameters {`Encoder,String,String`}.

```
package examples;

public class Vector {

    Element first;

    public void add(int value){
        Element e = new Element();
        e.value = value;
        Vector v = new Vector();
        v.first = e;
        append(v);
    }
    public void append(Vector v){
        Element e = first;

        if (e == null)
            first = v.first;
        else{
             while (e.next != null)
                e = e.next;

             e.next = v.first;
          }
    }
}
```

```
class SubVector extends Vector{

    public void append(Vector v){
        //...
    }
}
```

class	ancestor
Vector	Object
SubVector	Vector
Element	Object

method	entry
Vector$init	y
Vector$add	y
Vector$dyn*append	y
Vector$append	y
Vector$append#1#2	n
Vector$append#3#4	n
SubVector$init	y
SubVector$append	y
Element$init	y

Figure 2.4: Vector example: source code and corresponding metainformation.

2.4 Metainformation

The addition of metainformation during the transformation, although not strictly required, can aid the *fixpoint algorithm* that is described in the next chapter, if some characteristics related to the original source are known. In other cases, the abstract domain can use certain information about the program not directly encoded in the Horn clauses. Both demands are solved via the addition of *metainformation* to the transformation. We illustrate this point with the example in Figure 2.4, which shows an alternative version of the JDK `Vector` class. The descendant `SubVector` contains an alternative version of the `append` method. The corresponding Horn clauses, represented as a Control Flow Graph, are shown in Figure 2.5. We omitted the constructor (`init`) clauses for simplicity.

Hierarchy and method type tables contained in the metainformation are shown

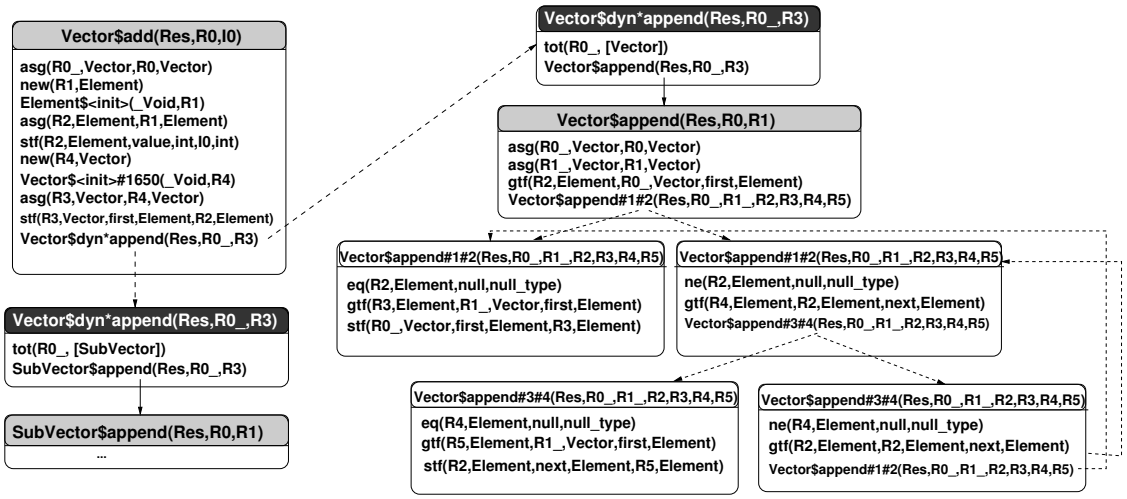

Figure 2.5: Call Graph for the example in Figure 2.4.

in Figure 2.4 (such tables are represented as sets of facts). In the case of the parent-child relations, the purpose is to provide the abstract domain code with access to the class tree, the more obvious application being class analysis [BS96]. The second table contains a classification for each method, which can be y (entry) or n (internal). It is used to optimize the performance of the fixpoint engine, by avoiding the *projection* and *extension* operations [Bru91] that are sometimes computationally expensive. These operations are domain dependent, and invoked from the fixpoint engine, as shown in Chapter 3.

2.5 Transformation of other OO languages

Our framework can be adapted to other languages apart from Java, especially for those like C# that share similar syntax and statement semantics to Java. The examples in Figures 2.6 and 2.7 illustrate this point. The first snippet, in Figure 2.6, is written in Java. The value returned by the `getDefaultLanguage` invocation in the

```
public class Lang{

  public void foo(Location loc){
    String lang = loc.getDefaultLanguage();
    ...
  }
}

class Location {
    public String getDefaultLanguage(){
      return "English";
    }
}

class China extends Location{
    public String getDefaultLanguage(){
      return "Mandarin";
    }
}

class Sichuan extends China{
}
```

```
Lang$foo(Res,R0,R1):-
    asg(R0_,Lang,R0,Lang),
    asg(R1_,Location,R1,Location),
    Location$dyn*getDefaultLanguage(R4,R1_),
    ret.

Location$getDefaultLanguage(Res,R0):-
    asg(R0_,Location,R0,Location),
    asg(Res,java.String,"English",java.String),
    ret.

China$getDefaultLanguage(Res,R0):-
    asg(R0_,China,R0,China),
    asg(Res,java.String,"Mandarin",java.String),
    ret.

Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [China,Sichuan]),
    China$getDefaultLanguage(Res,R1_).
Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [Location]),
    Location$getDefaultLanguage(Res,R1_).
```

Figure 2.6: Transformation of a virtual invocation in Java.

```
namespace Lang{

public class Lang{
  public void foo(Location loc){
    string lang = loc.getDefaultLanguage();
    ...
  }
}
class Location {
  public string getDefaultLanguage(){
    return "English";
  }
}
class China:Location{
  private string getDefaultLanguage(){
    return "Mandarin";
  }
}
class HongKong:China{}
}
```

```
Lang$foo(Res,R0,R1):-
    asg(R0_,Lang,R0,Lang),
    asg(R1_,Location,R1,Location),
    Location$dyn*getDefaultLanguage(R4,R1_),
    ret.

Location$getDefaultLanguage(Res,R0):-
    asg(R0_,Location,R0,Location),
    asg(Res,string,"English",string),
    ret.

China$getDefaultLanguage(Res,R0):-
    asg(R0_,China,R0,China),
    asg(Res,string,"Mandarin",string),
    ret.

Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [China]),
    China$getDefaultLanguage(Res,R1_).
Location$dyn*getDefaultLanguage(Res,R1_):-
    tot(R1_, [Location,HongKong]),
    Location$getDefaultLanguage(Res,R1_).
```

Figure 2.7: Transformation of a virtual invocation in C#.

`foo` method returns `English` if `loc` has runtime type `Location` and `Mandarin` if the runtime type is `China` or `Sichuan`, since this last class inherits the implementation of `getDefaultLanguage` from `China` according to standard Java semantics [GJSB05]. The C# language is quite similar in most aspects, but polymorphic invocations have

been further refined (and complicated). In Figure 2.7, snippet written in C#, only class `China` overshadows the default definition for the `getDefaultLanguage` method given in the superclass; `HongKong` inherits the `Location` implementation. Therefore, an invocation like `(new Hong Kong()).getDefaultLanguage()` returns `English`.

When analyzing a virtual invocation like the one in the first line of `foo`, we could have implemented internal mechanisms in the analyzer for differentiating the two possible interpretations that the call might have in each language. That implies an undesirable, double implementation of either the fixpoint algorithm or the abstract domains, since the analyzer would then be language-dependent. To bypass this problem, we introduce additional pseudo-builtins that contain language-dependent features. We can see in Figures 2.6 and 2.7 how the Horn clause representation is almost identical in both cases, except for the bodies of the two `Location$dyn*getDefault Language` clauses. In the case of Java, we indicate that the first clause is executed if the runtime type of `this` (`tot`) is either `China` or `Sichuan`, while the second requires that variable to be of runtime type `Location`. The situation is reversed in the C# example, in which instances of `Location` and `HongKong` share the implementation `Location$getDefaultLocation` while invocations on objects of (exactly) class `China` are redirected to `China$getDefaultLocation`.

In this particular case, the abstract domain is not required to know which actual language is to be analyzed, but only to provide a common, correct transfer function for the `tot` builtin, which will return as output state the same input state if the instance happens to have a runtime type included in the list of accepted classes, and $\perp$ if not.

2.6 Related work

In [PJC06] the authors describe how to automatically derive Prolog versions of Java programs that share the same operational semantics. However, the compilation applies to a smaller subset of Java than that supported in our work and no experimental results are provided (in Chapter 3 we show some statistics for the compilation phase of medium-sized programs).

More closely related to ours is the work presented in [AGZHP07], which draws in part on the ideas of [PGS98]. The authors focus on how to reuse existing logic programming tools, in order to analyze Java bytecode. The approach is based on encoding an interpreter of the Java Virtual Machine bytecode in a logic language, Ciao [BCC+04], and then partially evaluating this interpreter with respect to the concrete program to be analyzed. This results in a *residual* program which has the same semantics as the original one but is often easier to analyze than the original set of bytecode+interpreter.

While the approach of [AGZHP07] is obviously very interesting, it also has the shortcoming that it is quite dependent on the quality of the results obtained by the partial evaluator. Given the state of the art in partial evaluation, this may clearly vary significantly depending on the input program. The approach presented herein is based instead on a direct translation from the Java program into a Horn clause representation, which obviates this problem, at the cost of having to write and prove correct the transformer. Also, in this translation we do not try to mimic the operational semantics of the Java program in the Horn clause version (i.e., the resulting program if run, e.g., on a Prolog system, would not necessarily produce equivalent results to those of the Java program). Instead, the aim is to *safely approximate* the semantics of the Java program in the Horn clause representation by taking advantage of the collecting SLD semantics assumed by the analyzer. This

allows flexibility in the translation and eliminates the burden of having to simulate exactly the operational semantics of the source language since we do not want to execute the program but only to obtain safe results by analyzing it. The flexibility and directness of this approach also allows supporting a much larger subset of the language than in [AGZHP07], including exceptions, inheritance, interfaces, etc.

Another work that is closely related to ours is [AAG+07], in which a block-based representation is presented. The main differences between [AAG+07] and our compilation process are a) we deal with a bigger subset of Java b) in our approach, each block in the CFG is treated as a method *per se*, thus all the different program constructions (loops, if-then-else, etc) are transformed into the same intermediate representation, while in the approach of [AAG+07] there is a clear distinction between a method and a block.

In most of the (non CLP-based) abstract interpretation frameworks for analysis of Java (e.g., [Bla99, CL05]) the authors prefer to focus on particular properties and therefore their solutions (abstract domains and analysis algorithms) are tied to them, even when if they may be explicitly labeled as multipurpose [LAS00]. In [Pol04] the authors use a framework that is closely related to Gaia [LV94]. However, the intermediate representation is not described and the semantics of the interprocedural operations is again tied to the Java language. The more recent Julia framework [Spo05] is intended to be generic from the point of view of domains but once more also targets Java as unique source language. Finally, in [Log07] another interesting generic static analyzer for the modular analysis and verification of Java classes is presented. The algorithm presented is tailored specifically to Java source.

2.7 Chapter conclusions

The framework introduced in this chapter consists of a two-step process: a transformation of the program into a set of Horn clauses, and a fixpoint algorithm. We claim that our intermediate representation is flexible in the sense that is decoupled from any language-dependent features. This characteristic allows us to design a fixpoint algorithm, described in Chapter 3, that is also generic.

Chapter 3

The Fixpoint Algorithm

3.1 Background and Motivation

As mentioned in Chapter 1, there has been relatively little work on the underlying fixpoint algorithms used by abstract interpretation-based frameworks. In fact, most of the existing analyzers use fixpoint algorithms that are relatively inefficient or specific to a particular source language or analysis, thus cannot easily be reused in other contexts.

In this chapter, we introduce a novel, efficient fixpoint algorithm, which accepts as input the intermediate representation described in Chapter 2, and is therefore largely independent from language-specific characteristics. The efficiency of the algorithm relies on keeping dependencies between different methods during analysis so that only the really affected parts need to be revisited after a change during the convergence process. The algorithm deals thus efficiently with mutually recursive call graphs. In addition, recomputation is avoided using *memoization*.

The proposed algorithm is also *parametric* with respect to the abstract domain,

specifying a reduced number of basic operations that it must implement. Another characteristic is that it is *context sensitive* –abstract calls to a given method that represent different input patterns are automatically analyzed separately – and follows a top-down approach, in order to allow modeling properties that depend on the data flow characteristics of the program. To our knowledge, this is the first concise and precise description of a top-down, context sensitive, and parametric fixpoint algorithm for object oriented programs.

3.2 The Top-Down Analysis Algorithm

We now describe our top-down analysis algorithm, which calculates the least fixed point given a control flow graph (equivalent to the intermediate representation introduced in Chapter 2), and an initial abstract state. Intermediate results are stored in a memo table, which contains the results of computations already performed and is typically used to avoid needless recomputation. In our context it is used to store results obtained from an earlier round of iteration, and also to track whether a certain entry represents final, stable results for the block, or intermediate approximations obtained half way during the convergence of fixpoint computations. An entry in the memo table has the following fields: block name, its projected call state (λ), its status, its projected exit state (λ') and a unique identifier. Along with the memo table we assume operations which allow to query the status of an entry, retrieve the projected exit state, and add or update an entry.

The pseudocode for the fixpoint algorithm is shown in Figs. 3.1 and 3.2. Builtins are treated directly by each domain; external invocations can be handled in two different ways: 1) using a *worst-case assumption*, in which any reference to an external method returns the top-most element in the domain for all the variables involved in the call; b) using the results of a previous phase of the analysis performed over that

```
topDownAnalyze(CFG, method, dom, in, mt, set)
  case classify(CFG,method) of
    not_recursive:return analyzeNonRecMethod(CFG, method, dom, in, mt, set)
    recursive:return analyzeRecMethod(CFG, method, dom, in, mt, set)
    builtin : return dom.analyzeBuiltin(method, in, mt)
    external : return dom.analyzeExternal(method, in, mt)

analyzeNonRecMethod(CFG, method, dom, in, mt, set)
  name:=getName(method)
  actPars:=getActualParams(method)
  λ:=dom.project(in, actPars)
  if mt.isComplete(⟨name, λ⟩) then
    λ':=mt.getOutput(⟨name, λ⟩)
  else ⟨λ', mt, set⟩:=
      analyzeNonRecBlocks(CFG, name, dom, actPars, λ, complete, mt, set)
  out:=dom.extend(in, actPars, λ')
  return ⟨out, mt, set⟩

analyzeNonRecBlocks(CFG, name, dom, actPars, λ, st, mt, set)
  λ:=λ|{res,r0,...,rm} / {actPar0,...,actParm}
  blocks:=getNonRecBlocks(name)
  λ':=⊥
  foreach block ∈ blocks
    body:=getBody(block)
    ⟨β', mt, set⟩:=analyzeBody(CFG, β, dom, body, mt, set)
    λb':=dom.project(β', {res, r0, ..., rm})
    λ':=λ' ⊔ λb'
  λ':=λ'|{actPar0,...,actParm} / {res,r0,...,rm}
  mt.insert(⟨name, λ, λ', st⟩)
  return ⟨λ', mt, set⟩

analyzeBody(CFG, β, body, dom, mt, set)
  in:=β
  foreach stmt ∈ body
    ⟨out, mt, set⟩:= topDownAnalyze(CFG, stmt, dom, in, mt, set)
    in:=out
  return ⟨out, mt, set⟩
```

Figure 3.1: The top-down fixpoint algorithm

external code.

Invocations of non-recursive methods are handled by `analyzeNonRecMethod`.

```
analyzeRecMethod(CFG, method, dom, in, mt, set)
    name:=getName(method)
    actPars:=getActualParams(method)
    λ:=dom.project(in, actPars)
    if mt.isComplete(⟨name, λ⟩) then
        λ′:=mt.getOutput(⟨name, λ⟩)
    elseif mt.isFixpoint(⟨name, λ⟩) then
        λ′:=mt.getOutput(⟨name, λ⟩)
        set:=set ∪ {getUniqueID(name)}
    elseif mt.isApproximate(⟨name, λ⟩) then
        mt.update(⟨name, λ⟩, fixpoint)
        ⟨λ′, mt, set⟩:=analyzeRecBlocks(CFG, method, dom, λ, mt, set)
    else
        ⟨λ′, mt, set⟩:=analyzeNonRecBlocks(CFG, name, dom, actPars, λ, fixpoint, mt, set)
        set:=set ∪ {getUniqueID(name)}
        ⟨λ′, mt, set⟩:=analyzeRecBlocks(CFG, method, dom, λ, λ′, mt, set)
    out:=dom.extend(in, actPars, λ′)
    return ⟨out, mt, set⟩

updateDeps(method, mt, set_method, set)
    id:=getUniqueID(method)
    if set_method \ {id} = ∅ then
        status:=complete
        foreach id′ such that id′ depends on id
            remove dependence between id′ and id
            if id′ is independent then
                let ⟨name_id′, λ′_id′⟩ be associated with id′
                mt.update(⟨name_id′, λ′_id′⟩, complete)
        status:=approximate
        make id dependent from set_method \ {id}
    mt.update(⟨name, λ′⟩, status)
    set:=set ∪ set_method \ {id}
    return ⟨mt, set⟩
```

Figure 3.2: The top-down fixpoint algorithm (continuation)

This procedure first checks if there is an entry in the memo table for the name of the invoked method and its λ. In that case, we reuse the previously computed value for λ'. Otherwise, the variables of its λ are renamed to the set of variables $\{res, r_0, \ldots, r_m\}$ (we will assume a standard naming for the formal parameters of the

```
analyzeRecBlocks(CFG, method, dom, λ, λ', mt, set)
    name:=getName(method)
    actPars:=getActualParams(method)
    λ:=λ|{res,r0,...,rm}_{actPar0,...,actParm}
    blocks:=getRecBlocks(name)
    set_method:=∅
    fixpoint:=true
    repeat
        foreach block ∈ blocks
            body:=getBody(block)
            ⟨β', mt, set_body⟩:=
                analyzeBody(CFG, β, dom, body, mt, ∅)
            dom.project(β', actPars)
            λ'_old:=λ'
            λ':=λ'_old ⊔ β'|{actPar0,...,actParm}_{res,r0,...,rm}
            if λ'_old ≠ λ' then
                fixpoint:=false
                mt.update(⟨N, λ⟩, λ')
            set_method:=set_method ∪ set_body
    until (fixpoint = true)
    ⟨mt, set⟩:=updateDeps(method, mt, set_method, set)
    return ⟨λ', mt, set⟩
```

Figure 3.3: The top-down fixpoint algorithm (continuation)

form $res, r_0, \ldots, r_m$) and an exit state is calculated for each block the method is built of. The results are then merged through the lub (least upper bound, usually denoted by the symbol $\sqcup$) operation, renamed back to the scope of the callee, and inserted as an entry in the memo table characterized as `complete`. Finally, λ' is reconciled with the calling state through the `extend` operation, yielding the exit state.

When a method is recursive, the `analyzeRecMethod` procedure in Fig 3.2 repeats the analysis until a fixpoint is reached for the abstract execution tree, i.e., until it remains the same before and after one round of iteration. In order to do this, we keep

track of a flag to signal the termination of the fixpoint computation. The procedure starts the analysis in the non-recursive blocks of the invoked method, thus accelerating convergence since the initial λ' is different from $\perp$. An entry in the memo table is inserted with that tentative abstract state and characterized as `fixpoint`. The remaining, recursive blocks are analyzed within `analyzeRecBlocks`, which repeats their analysis until the value of λ' does not change between two consecutive iterations.

This basic scheme requires two extra features in order to work also for mutually recursive calls. One is the addition of new possible values for the *status* field in memo table entries. If the fixpoint has not been reached yet for a entry (m_1, λ), we saw that it is labeled as `fixpoint`; if it has been reached, but by using a possibly incomplete value of λ' of some other method m_2 (i.e., a value that does not correspond yet to a fixpoint), we tag that entry as `approximate` . The second required artifact is a table with dependencies between methods. Note that the fixpoint computation can involve two or more mutually recursive methods, which will indefinetely wait for the other to be `complete` before reaching that status. This deadlock scenario can be avoided by pausing analysis in method m_2 if it depends of a call to a method m_1 which is already in `fixpoint` state; we will use the current approximation λ' for m_1 and wait until it reaches `complete` status and notifies (via `updateDeps`) all the methods depending on it.

Computation of that fixpoint can be sometimes computationally expensive or even prohibitive, so in order to speed it up we use a combination of techniques. The first is *memoization* [Die87, HWD92, Mut91] since the memo table acts as a cache for already computed tuples. Efficiency of the computation can be further improved by keeping track of the dependencies between methods. In the above scenario, during subsequent iterations for m_1, the subtree for m_2 is explored every time and its entry in the memo table labeled as `approximate`. After the last round of iteration for m_1,

its entry in the memo table will be tagged as `complete` but the row for m_2 remains as `approximate`. The subtree for m_2 has to undergo an unnecessary exploration, since it has already used the `complete` value of the exit state of m_1. In order to avoid this redundant work, after each fixpoint iteration all those methods depending only on another m that just changed its status to `complete` are automatically tagged with the same status.

Another major feature of our algorithm is its accuracy. Although precision remains in general a domain-related issue, our solution possesses inherent characteristics that help yield more precise results. First, the algorithm offers results of the analysis at each program point due to its top-down condition. Second, and more relevant, the algorithm is fully context sensitive: every new encountered abstract state for the set of formal parameters is independently stored in the memo table. Moreover, different caller contexts will use the same entry as long as the state of their actual parameters is identical. Although not present in the pseudo-code, our current implementation also supports path-sensitivity [DLS02], which allows independent reasoning about different branches.

Example 1 *We show how an example of mutual recursion (*`Vector$append`*) described in Fig 3.4 is handled by the fixpoint algorithm defined in Figs. 3.1 and 3.2. Both the source code and the CFG are identical to the ones presented in the previous chapter; we reproduce them here for convenience. Also for simplicity, the abstract domain used is nullity, capable of approximating which variables are definitely null and which ones definitely point to a non-null location. The objective is not to fully understand each of the entries of the memo table in Figure 3.5, which would require a complementary explanation of the domain transfer functions and going through a vast amount of intermediate states, but to illustrate how some interesting dependencies and status change in a very specific subset of those states. The method names have been shortened to fit into the tables.*

```
package examples;

public class Vector {

    Element first;

    public void add(int value){
        Element e = new Element();
        e.value = value;
        Vector v = new Vector();
        v.first = e;
        append(v);
    }
}
```

```
    public void append(Vector v){
        Element e = first;
        if (e == null)
            first = v.first;
        else{
             while (e.next != null)
                e = e.next;
             e.next = v.first;
        }
    }

class SubVector extends Vector{
    public void append(Vector v){
        //...
}
```

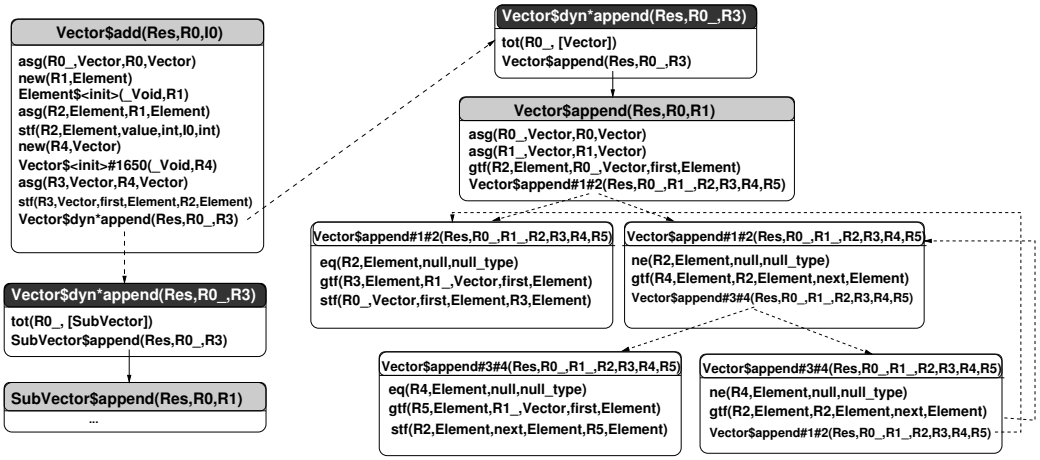

Figure 3.4: Java source code and corresponding CFG.

In step 1 it is assumed that the non-recursive blocks for app_{34} *and* app_{12} *have already been analyzed. Both entries for these blocks are marked as* fixpoint *since they correspond to recursive methods whose analyses have not converged to a fixpoint yet. Note that there exist two different entries corrresponding to method* app_{12} *which has been analyzed* twice *with different abstract call patterns: one when called from app and another when called from* app_{34} *yielding* $\langle app_{12}, \lambda_1, \lambda'_{11}\rangle$ *and* $\langle app_{12}, \lambda_3, \lambda'_{31}\rangle$*, respectively. In step 2, the analysis corresponding to the entry* $\langle app_{12}, \lambda_3, \lambda'_{31}\rangle$ *has converged to a fixpoint but using the incomplete value of* $\langle app_{34}, \lambda_2, \lambda'_{21}\rangle$*. Therefore, the entry is forced to* approximate *changing its exit state to* λ'_{32}*. In step 3, the analysis for the method* app_{34} *reaches a fixpoint and since it does not depend on other methods,*

step	method	λ	λ'	st	dep
1	app_{12}	λ_1	λ'_{11}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{21}	fix	$\{app_{34}\}$
	app_{12}	λ_3	λ'_{31}	fix	$\{app_{12}\}$
2	app_{12}	λ_1	λ'_{11}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{21}	fix	$\{app_{34}\}$
	app_{12}	λ_3	λ'_{32}	**app**	$\{app_{12}, app_{34}\}$
3	app_{12}	λ_1	λ'_{11}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{22}	**com**	$\emptyset$
	app_{12}	λ_3	λ'_{32}	app	$\{app_{12}\}$
4	app_{12}	λ_1	λ'_{12}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{22}	com	$\emptyset$
	app_{12}	λ_3	λ'_{32}	**com**	$\emptyset$
5	app	λ_0	λ'_{0}	**com**	$\emptyset$
	app_{12}	λ_1	λ'_{12}	com	$\emptyset$
	app_{34}	λ_2	λ'_{22}	com	$\emptyset$
	app_{12}	λ_3	λ'_{32}	**com**	$\emptyset$

Figure 3.5: Fixpoint calculation for `Vector$append`

the entry $\langle app_{34}, \lambda_2, \lambda'_{21}\rangle$ *is marked as* complete *and updated to* $\langle app_{34}, \lambda_2, \lambda'_{22}\rangle$*. After this step, the algorithm notices that* $\langle app_{12}, \lambda_3, \lambda'_{32}\rangle$ *is* approximate *and waiting for a complete value of* $\langle app_{34}, \lambda_2, \lambda'_{22}\rangle$ *which has been already produced. Thus, the entry* $\langle app_{12}, \lambda_3, \lambda'_{32}\rangle$ *is marked directly as* complete *and no extra iteration is required. This change is illustrated in step 4. Finally, the analysis characterizes also the entry* $\langle app_{12}, \lambda_1, \lambda'_{12}\rangle$ *as* complete *and terminates the semantics computation of app.*

3.3 Experimental Results

We have completed a preliminary implementation of our framework and tested it by using two abstractions. The first one is a nullity domain (denoted by N, see Chapter 4) which associates every variable in the program with one out of three

name	k	m	b	pp
`Health`	8	30	620	833
`BH`	9	70	1198	1473
`Voronoi`	6	73	988	1108
`MST`	6	36	443	304
`Power`	6	32	997	1143
`TreeAdd`	2	12	193	217
`Em3d`	4	22	444	566
`Perimeter`	10	45	543	770
`BiSort`	2	15	323	467
`All`	42	287	5168	6432

Table 3.1: General statistics about the benchmarks utilized.

possible states: null, non-null, or any. The second is a Class Hierarchy domain (CHA, see also Chapter 4) [BS96], which uses the combination of the statically declared type of an object and the class hierarchy of the program to determine the set of possible targets of a virtual invocation. The use of a CHA shows the scalability of our framework for a domain with non-linear worst-case complexity in its operations. Our experimental results are summarized in Tables 3.1 and 3.2; the benchmarks belong to the JOlden suite [jol]. The first table contain basic metrics about the application: number of classes (k), methods (m) and bytecodes (b). Since those numbers really correspond to the Jimple representation of the code, we also list how many program points (pp) are present in the Control Flow Graph analyzed. This metric differs slightly from the number of bytecodes in the sense that extra blocks and builtins make it slightly larger; pp also provides a better approximation for the size of the program analyzed because the semantics of the Java bytecodes are made explicit, as seen in Chapter 2. The data in Table 3.2 strictly corresponds to the analysis phase. Since our framework is context sensitive and can thus keep track of different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number

name	pt	st_N	ast_N	at_N	st_{CHA}	ast_{CHA}	at_{CHA}
`Health`	1.1	3.5	2964	4.7	3.8	3542	52.1
`BH`	3.2	4.2	6266	24.3	2.7	4757	59.4
`Voronoi`	2.2	2.6	2850	5.9	3.8	5147	81.3
`MST`	0.1	2.7	844	0.7	2.6	1609	11.6
`Power`	2.1	4.2	4743	12.1	2.3	2908	32.7
`TreeAdd`	2.0	2.2	468	0.3	2.6	529	6.1
`Em3d`	0.1	5.5	3100	4.8	4.9	3320	49.5
`Perimeter`	0.1	3.1	2366	1.8	4.5	3731	25.0
`BiSort`	0.1	4.2	1934	2.9	3.4	1614	15.6
`All`	10.5	6.1	39159	163.7	4.1	29586	39.2

Table 3.2: Compilation and analysis times on a Pentium M 1.73Ghz with 1Gb of RAM.

of abstract states is typically larger than the number of reachable program points. Column *ast* provides the total number of these abstract states inferred by analysis. The level of precision is the ratio ast/pp, presented in column *st*. In general, such a larger number for *st* tends to indicate more precise results.

Running times are listed in columns *pt* (time invested in preprocessing the program and produce the corresponding CFG) and *at* (analysis); both are given in seconds. The benchmarks have been tested on a Pentium M 1.73Ghz with 1Gb of RAM. We chose to divide the total time because we expect the decompilation process to be fully run only once; posterior executions can use incremental compilation for those files that changed, thus the preprocessing phase is almost negligible in medium and large programs. Although the same approach can be taken for the analysis [PH96], we do not support incrementality at that level in the current implementation. The results seem to support the feasibility of the approach, even if further work is certainly required to see how applicable the technique is for large programs.

3.4 Related work

Most published analyses based on abstract interpretation for Java (source or bytecode) do not provide much detail regarding the implementation of the fixpoint algorithm. Also, most of the related research (e.g., [Bla99, CL05]) focuses on particular properties and therefore their solutions (abstract domains) are tied to them, even when they are explicitly multipurpose, like TVLA [LAS00]. In [Pol04] the authors mention a choice of several context insensitive and sensitive computations, but no further information is given. The more recent and quite interesting Julia framework [Spo05] is intended to be generic and targets bytecode as in our case. Its fixpoint technique is based on prioritizing analysis of non-recursive components over those requiring fixpoint computations and using abstract compilation [HWD92]. However, few implementation details are provided. Also, this is a *bottom-up* framework, while our objective is to develop a top-down, context sensitive framework. While it is well-known that bottom-up analyses can be adapted to perform top-down analyses by subjecting the program to a "magic-sets"-style transformation [Ram91], the resulting analyzers typically lack some of the characteristics that are the objective of our proposal, and, specially, context sensitive results. Cibai [Log07] is another generic static analyzer for the modular analysis and verification of Java classes. The algorithm presented is *top-down*, and only a naive version of it (which is not efficient for mutually recursive call graphs) is presented.

Finally, our work is inspired by the ideas in [MH92], where a fixpoint engine for Prolog is proposed. Although our algorithm is based on the ideas already present in that work, we modified the algorithm in non-trivial ways so it is adapted to our general intermediate representation, thus eliminating all those aspects that were dependent of Prolog.

3.5 Chapter conclusions

In this chapter, we have completed the presentation of our abstract interpretation framework by introducing a novel fixpoint algorithm. The algorithm benefits from acceleration techniques like memoization or dependency tracking, considerably reducing the number of iterations. We also claim that the analysis has the potential to be very accurate because of the top-down, context sensitive approach adopted.

Chapter 4

The Set Sharing Abstract Domain

4.1 Background and Motivation

Sharing analysis [JL89, MH91, SS05] aims to detect which variables do not share in memory, i.e., do not point (transitively) to the same location. It can be viewed as an abstraction of the graph-based representations of memory used by certain classes of alias analyses (see, e.g., [WL95, BCCH94, HBCC99, LR92]). Obtaining a safe (over-approximation) of which instances might share allows parallelizing segments of code, improving garbage collection, reordering execution, etc. Also, sharing information can improve the precision of other analyses.

As mentioned in Chapter 3, *nullity* analysis is aimed at keeping track of null variables. This allows for example verifying properties such as the absence of null-pointer exceptions at compile time. In addition, by combining sharing and null information it is possible to obtain more precise descriptions of the state of the heap. As also mentioned in Chapter 3, in type-safe, object-oriented languages *class* analysis [Age95, BS96, DGC95, PS91] (sometimes called *type* analysis) focuses on determining, in the presence of polymorphic calls, which particular implementation

of a given method will be executed at run-time, i.e., what is the specific class of the called object in the hierarchy. Multiple compilation optimizations benefit from having precise class descriptions: inlining, dead code elimination, etc. In addition, class information may allow analyzing only a subset of the classes in the hierarchy, which may result in additional precision.

We propose a novel analysis which infers in a combined way *set sharing*, *nullity*, and *class* information for a subset of Java. The objective of using a reduced cardinal product [CC79] of these three abstract domains is to achieve a good balance between precision and performance, since the information tracked by each component helps refine that of the others. While in principle these three analyses could be run separately, because they interact (we provide some examples of this), this would result in a loss of precision or require an expensive iteration over the different analyses until an overall fixpoint is reached [CMB$^+$95, CC79]. In addition, note that since our analysis is multivariant, and given the different nature of the properties being tracked, performing analyses separately may result in different sets of abstract values (contexts) for each analysis for each program point. This makes it difficult to relate which abstract value of a given analysis corresponds to a given abstract value of another analysis at a given point. At the other end of things, we prefer for clarity and simplicity reasons to develop directly this three-component domain and the operations on it, rather than resorting to the development of a more unified domain through (semi-)automatic (but complex) techniques [CMB$^+$95, CFR$^+$97]. The final objectives of our analysis include verification, static debugging, and optimization.

4.2 Standard Semantics

Figure 4.1 shows the motivating example for this chapter, an alternative implementation for the `java.util.Vector` class of the JDK in which vectors are represented

```
class Element {
  int value;
  Element next;}

class Vector {
  Element first;

  public void add(Element el) {
    Vector v  = new Vector();
    el.next  = null;
    v.first = el;
    append(v);
  }
}
```

```
public void append(Vector v) {

  if (this != v) {
    Element e = first;
      if (e == null)
        first = v.first;
      else {
        while (e.next != null)
          e = e.next;
       e.next = v.first;
     }
  }
}
```

Figure 4.1: Vector example.

as linked lists. It is important to note that we provide the source code so the reader can get a better understanding of the example; the actual analysis works in the corresponding bytecode by transforming it into the intermediate representation described in Chapter 2. In the same fashion, we will refer to some Java source expressions and statements during the rest of the chapter (as in `new`), even when the framework deals with their transformed representation in the actual analysis.

4.2.1 Basic Notation

We first introduce some notation and auxiliary functions used in the rest of the chapter. By $\mapsto$ we refer to total functions; for partial ones we use $\rightarrow$. The powerset of a set s is $\mathcal{P}(s)$; $\mathcal{P}^+(s)$ is an abbreviation for $\mathcal{P}(s) \setminus \{\emptyset\}$. The *dom* function returns all the elements for which a function is defined; for the codomain we will use *rng*. A substitution $f[k_1 \mapsto v_1, \ldots, k_n, \mapsto v_n]$ is equivalent to $f(k_1) = v_1, \ldots, f(k_n) = v_n$. We will overload the operator for lists so that $f[K \mapsto V]$ assigns $f(k_i) = v_i$, $i = 1, \ldots, m$,

assuming $|K| = |V| = m$. By $f|_{-S}$ we denote removing S from $dom(f)$. Conversely, $f|_S$ restricts $dom(f)$ to S. For tuples $(f_1, \ldots, f_m)|_S = (f_1|_S, \ldots, f_m|_S)$. Renaming in the set s of every variable in S by the one in the same position in T ($|S| = |T|$) is written as $s|_S^T$. This operator can also be applied for renaming single variables. We denote by $\mathcal{B}$ the set of Booleans.

4.2.2 Program State and Sharing

With $\mathcal{M}$ we designate the set of all method names defined in the program. For the set of distinct identifiers (variables and fields) we use $\mathcal{V}$. We assume that $\mathcal{V}$ also includes the elements *this* (instance where the current method is executed), and *res* (for the return value of the method). In the same way, $\mathcal{K}$ represents the program-defined classes. We do not allow `import` declarations but assume as member of $\mathcal{K}$ the predefined class `Object`.

$\mathcal{K}$ forms a lattice implied by a subclass relation $\downarrow$: $\mathcal{K} \rightarrow \mathcal{P}(\mathcal{K})$ such that if $t_2 \in \downarrow t_1$ then $t_2 \leq_{\mathcal{K}} t_1$. The semantics of the language implies $\downarrow \texttt{Object} = \mathcal{K}$. Given $def : \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$, that determines whether a particular class provides its own implementation for a method, the Boolean function $redef : \mathcal{K} \times \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$ checks if a class k_1 redefines a method existing in the ancestor k_2: $redef(k_1, k_2, \mathrm{m}) = true$ iff $\exists k$ s.t. $def(k, \mathrm{m})$, $k_1 \leq_{\mathcal{K}} k <_{\mathcal{K}} k_2$.

Static types are accessed by means of a function $\pi : \mathcal{V} \mapsto \mathcal{K}$ that maps variables to their declared types. The purpose of an *environment* π is twofold: it indicates the set of variables accessible at a given program point and stores their declared types. Additionally, we will use the auxiliary functions $F(k)$ (which maps the fields of $k \in \mathcal{K}$ to their declared type), and $type_\pi(expr)$, which maps expressions to types, according to π.

The description of the memory state is based on the formalization in [SS05,

HPS06]. We define a frame as an element of $Fr_\pi = \{\phi \mid \phi \in dom(\pi) \mapsto Loc \cup \{\text{null}\}\}$, where $Loc = \mathbb{I}^+$ is the set of memory locations. A frame represents the first level of indirection and maps variable names to locations except if they are null. The set of all objects is $Obj = \{k \star \phi \mid k \in \mathcal{K}, \phi \in Fr_{F(k)}\}$. Locations and objects are linked together through the memory $Mem = \{\mu \mid \mu \in Loc \mapsto Obj\}$. A new object of class k is created as $new(k) = k \star \phi$ where $\phi(f) = null \ \ \forall f \in F(k)$. The object pointed to by v in the frame ϕ and memory μ can be retrieved via the partial function $obj(\phi \star \mu, v) = \mu(\phi(v))$. A valid heap configuration (concrete state $\phi \star \mu$) is any element of $\Sigma_\pi = \{\sigma \mid \phi \in Fr_\pi, \mu \in Mem\}$. We will sometimes refer to a pair σ with δ.

The set of locations $R_\pi(\phi \star \mu, v)$ reachable from $v \in dom(\pi)$ in the particular state $\phi \star \mu \in \Sigma_\pi$ is calculated as $R_\pi(\phi \star \mu, v) = \cup\{R^i_\pi(\phi \star \mu, v) \mid i \geq 0\}$, the base case being $R^0_\pi(\phi \star \mu, v) = \{(\phi(v))|_{Loc}\}$ and the inductive one $R^{i+1}_\pi(\phi \star \mu, v) = \cup\{rng(\mu(l).\phi))|_{Loc} \mid l \in R^i_\pi(\phi \star \mu, v)\}$. Reachability is the basis of two fundamental concepts: sharing and nullity. Distinct variables $V = \{v_1, \ldots, v_n\}$ *share* in the actual memory configuration δ if there is at least one common location in their reachability sets, i.e., $share_\pi(\delta, V)$ is true iff $\cap^n_{i=1} R_\pi(\delta, v_i) \neq \emptyset$. A variable $v \in dom(\pi)$ is *null* in state δ if $R_\pi(\delta, v) = \emptyset$. Nullity is checked by means of $nil_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{B}$, defined as $nil_\pi(\phi \star \mu, v) = true$ iff $\phi(v) = null$.

The *run-time type* of a variable in scope is returned by $\psi_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{K}$, which associates variables with their dynamic type, based on the information contained in the heap state: $\psi_\pi(\delta, v) = obj(\delta, v).k$ if $\overline{nil_\pi}(\delta, v)$ and $\psi_\pi(\delta, v) = \pi(v)$ otherwise. In a type-safe language like Java runtime types are congruent with declared types, i.e., $\psi_\pi(\delta, v) \leq_\mathcal{K} \pi(v) \ \ \forall v \in dom(\pi), \forall \delta \in \Sigma_\pi$. Therefore, a correct approximation of ψ_π can always be derived from π. Note that at the same program point we might have different run-time type states ψ^1_π and ψ^2_π depending on the particular program path executed, but the static type state is unique.

Denotational (compositional) semantics of sequential Java has been the subject of previous work (e.g., [AF99]). In our case we define a simpler version of that semantics for the subset defined in Section 4.2, described as transformations in the frame-memory state. The descriptions are similar to [SS05]. Expression functions $\mathcal{E}^I_\pi[\![]\!] : expr \mapsto (\Sigma_\pi \mapsto \Sigma_{\pi'})$ define the meaning of Java expressions, augmenting the actual scope $\pi' = \pi[res \mapsto type_\pi(exp)]$ with the temporal variable res. Command functions $\mathcal{C}^I_\pi[\![]\!] : com \mapsto (\Sigma_\pi \mapsto \Sigma_\pi)$ do the same for commands; semantics of a method `m` defined in class k is returned by the function $I(k.\mathtt{m}) : \Sigma_{input(k.m)} \to \Sigma_{output(k.m)}$. The definition of the respective environments, given a declaration in class k as t_{ret} `m`$(this : k, p_1 : t_1 \ldots p_n : t_n)$ `com`, is $input(k.m) = \{this \mapsto k, p_1 \mapsto t_1, \ldots, p_n \mapsto t_n\}$ and $output(k.m) = input(k.m)[out \mapsto t_{ret}]$.

Example 2 *Assume that, in Figure 4.1, after entering in the method* `add` *of the class* `Vector` *we have an initial state* $(\phi_0 \star \mu_0)$ *s.t.* $loc_1 = \phi_0(el) \neq null$. *After executing* `Vector v = new Vector()` *the state is* $(\phi_1 \star \mu_1)$, *with* $\phi_1(v) = loc_2$, *and* $\mu_1(loc_2).\phi(first) = null$. *The field assignment* `el.next = null` *results in* $(\phi_2 \star \mu_2)$, *verifying* $\mu_2(loc_1).\phi(next) = null$. *In the third line,* `v.first = el` *links* loc_1 *and* loc_2 *since now* $\mu_3(loc_2).\phi(first) = loc_1$. *Now* v *and* el *share, since their reachability sets intersect* at least *in* $\{loc_1\}$. *Finally, assume that* `append` *attaches* v *to the end of the current instance this resulting in a memory layout* $(\phi_4 \star \mu_4)$. *Given* $loc_3 = obj((\phi_4 \star \mu_4)(this)).\phi(first)$, *it should hold that* $\mu_4(\ldots \mu_4(loc_3).\phi(next) \ \ldots).\phi(next) = loc_2$. *Now this shares with* v *and therefore with* el, *because* loc_1 *is reachable from* loc_2.

4.3 Abstract Semantics

An abstract state $\sigma \in \mathcal{D}$ in an environment π approximates the sharing, nullity, and run-time type characteristics (as described in Section 4.2.2) of set of concrete states

$$\mathcal{SE}[\![\texttt{null}]\!](sh, nl, \tau) = (sh, nl', \tau')$$
$$nl' = nl[res \mapsto null]$$
$$\tau' = \tau[res \mapsto \downarrow object]$$
$$\mathcal{SE}[\![\texttt{new } k]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = sh \cup \{\{res\}\}$$
$$nl' = nl[res \mapsto nnull]$$
$$\tau' = \tau[res \mapsto \{\kappa\}]$$
$$\mathcal{SE}[\![v]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = (\{\{res\}\} \uplus sh_v) \cup sh_{-v}$$
$$nl' = nl[res \mapsto nl(v)]$$
$$\tau' = \tau[res \mapsto \tau(v)]$$
$$\mathcal{SE}[\![v\,.\,\texttt{f}]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ (sh', nl', \tau') \text{ otherwise} \end{cases}$$
$$sh' = sh_{-v} \cup \bigcup\{\mathcal{P}^+(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh_v\}$$
$$nl' = nl[res \mapsto unk, v \mapsto nnull]$$
$$\tau' = \tau[res \mapsto\downarrow F(\pi(v)(\texttt{f}))]$$
$$\mathcal{SE}[\![v\,.\,\texttt{m}(v_1, \ldots, v_n)]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ \sigma' \text{ otherwise} \end{cases}$$
$$\sigma' = \mathcal{SE}[\![\texttt{call}(v, m(v_1, \ldots, v_n))]\!](sh, nl', \tau)$$
$$nl' = nl[v \mapsto nnull]$$

Figure 4.2: Abstract semantics for the expressions.

in Σ_π. Every abstract state combines three abstractions: a sharing set $sh \in \mathcal{DS}_\pi$, a nullity set $nl \in \mathcal{DN}_\pi$, and a type member $\tau \in \mathcal{DT}_\pi$, i.e., $\mathcal{D} = \mathcal{DS}_\pi \times \mathcal{DN}_\pi \times \mathcal{DT}_\pi$.

The sharing abstract domain $\mathcal{DS}_\pi = \{\{v_1, \ldots, v_n\} \mid \{v_1, \ldots, v_n\} \in \mathcal{P}(dom(\pi)) , \cap_{i=1}^{n} C_\pi(v_i) \neq \emptyset\}$ is constrained by a class reachability function which retrieves those classes that are reachable from a particular variable: $C_\pi(v) = \cup\{C_\pi^i(v) \mid i \geq 0\}$, given $C_\pi^0(v) = \downarrow\pi(v)$ and $C_\pi^{i+1}(v) = \cup\{rng(F(k)) \mid k \in C_\pi^i(v)\}$. By using class reachability, we avoid including in the sharing domain sets of variables which cannot share in practice because of the language semantics. The partial order $\leq_{\mathcal{DS}_\pi}$ is set inclusion.

We define several operators over sharing sets, standard in the sharing literature [JL89, MH89]. The binary union $\uplus : \mathcal{DS}_\pi \times \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$, calculated as $S_1 \uplus$

$S_2 = \{Sh_1 \cup Sh_2 \mid Sh_1 \in S_1, Sh_2 \in S_2\}$ and the closure under union $* : \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$ operators, defined as $S^* = \{\cup SSh \mid SSh \in \mathcal{P}^+(S)\}$; we later filter their results using class reachability. The relevant sharing with respect to v is $sh_v = \{\, s \in sh \mid\ v \in s\}$, which we overloaded for sets. Similarly, $sh_{-v} = \{\, s \in sh \mid\ v \notin s\}$. The projection $sh|_V$ is equivalent to $\{S \mid S = S' \cap V, S' \in sh\}$.

The nullity domain is $\mathcal{DN}_\pi = \mathcal{P}(dom(\pi) \mapsto \mathcal{NV})$, where $\mathcal{NV} = \{null, nnull, unk\}$. The order $\leq_{\mathcal{NV}}$ of the nullity values ($null \leq_{\mathcal{NV}} unk$, $nnull \leq_{\mathcal{NV}} unk$) induces a partial order in $\mathcal{DN}_\pi$ s.t. $nl_1 \leq_{\mathcal{DN}_\pi} nl_2$ if $nl_1(v) \leq_{\mathcal{NV}} nl_2(v)\ \forall v \in dom(\pi)$. Finally, the domain of types maps variables to sets of types congruent with π: $\mathcal{DT}_\pi = \{(v, \{t_1, \ldots, t_n\}) \in dom(\pi) \mapsto \mathcal{P}(\mathcal{K}) \mid\ \{t_1, \ldots, t_n\} \subseteq \downarrow\pi(v)\}$.

We assume the standard framework of abstract interpretation as defined in [CC77] in terms of Galois insertions. The concretization function $\gamma_\pi : \mathcal{D} \mapsto \mathcal{P}(\Sigma_\pi)$ is $\gamma_\pi(sh, nl, \tau) = \{\delta \in \Sigma_\pi \mid\ \forall V \subseteq dom(\pi), share_\pi(\delta, V)$ and $\nexists W,\ V \subset W \subseteq dom(\pi)$ s.t. $share_\pi(\delta, W) \Rightarrow V \in sh$, and $R_\pi(\delta, v) = \emptyset$ if $nl(v) = null$, and $R_\pi(\delta, v) \neq \emptyset$ if $nl(v) = nnull$, and $\psi_\pi(\delta, v) \in \tau(v)\ , \forall v \in dom(\pi)\}$.

The abstract semantics of expressions and commands is listed in Figs. 4.2 and 4.3. As their concrete counterparts, they take an expression or command and map an input state $\sigma \in \mathcal{D}$ to an output state $\sigma' \in D^{\sigma}_{\pi'}$ where $\pi = \pi'$ in commands and $\pi' = \pi[res \mapsto type_\pi(expr)]$ in expression $expr$.

The semantics of a method call is explained in Section 4.3.1. The use of set sharing (rather than pair sharing) in the semantics prevents possible losses of precision, as shown in Example 3.

Example 3 *In the* `add` *method (Figure 4.1), assume that* $\sigma = (\{\{this, el\}, \{v\}\}, \{this/nnull, el/nnull, v/nnull\})$ *right before evaluating* `el` *in the third line (we skip type information for simplicity). The expression* `el` *binds to* res *the location of* el*, i.e., forces* el *and* res *to share. Since* $nl(el) \neq null$ *the new sharing is* $sh' =$

$$\mathcal{SC}[\![v\texttt{=}expr]\!]\sigma = ((sh'|_{-v})|^v_{res}, nl'|^v_{res}, \tau''|_{-res})$$
$$\tau'' = \tau'[v \mapsto (\tau'(v) \cap \tau'(res))]$$
$$(sh', nl', \tau') = \mathcal{SE}[\![expr]\!]\sigma$$

$$\mathcal{SC}[\![v\texttt{.f=}expr]\!]\sigma = (sh'', nl'', \tau')|_{-res}$$
$$sh'' = \begin{cases} \bot & \text{if } nl'(v) = null \\ sh' & \text{if } nl'(res) = null \\ sh^y \cup sh'_{-\{v,res\}} & \text{otherwise} \end{cases}$$
$$nl'' = nl'[v \mapsto nnull]$$
$$sh^y = (\bigcup\{\mathcal{P}(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh'_v\} \cup \bigcup\{\mathcal{P}(s|_{-res} \cup \{v\}) \uplus \{\{res\}\} \mid s \in sh'_{res}\})^*$$
$$(sh', nl', \tau') = \mathcal{SE}[\![expr]\!]\sigma$$

$$\mathcal{SC}[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!]\sigma = \begin{cases} \sigma'_1 & \text{if } nl(v) = null \\ \sigma'_2 & \text{if } nl(v) = nnull \\ \sigma_1 \sqcup \sigma_2 & \text{if } nl(v) = unk \end{cases}$$
$$\sigma'_i = \mathcal{SC}[\![com_i]\!]\sigma$$
$$\sigma_1 = \mathcal{SC}[\![com_1]\!](sh|_{-v}, nl[v \mapsto null], \tau[v \mapsto \downarrow\pi(v)])$$
$$\sigma_2 = \mathcal{SC}[\![com_2]\!](sh, nl[v \mapsto nnull], \tau)$$

$$\mathcal{SC}[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](sh, nl, \tau) = \begin{cases} \sigma'_1 & \text{if } nl(v) = nl(w) = null \\ \sigma'_2 & \text{if } sh|_{\{v,w\}} = \emptyset \\ \sigma'_1 \sqcup \sigma'_2 & \text{otherwise} \end{cases}$$
$$\sigma'_i = \mathcal{SC}[\![com_i]\!](sh, nl, \tau)$$

$$\mathcal{SC}[\![com_1\,\texttt{;}\,com_2]\!]\sigma = \mathcal{SC}[\![com_2]\!](\mathcal{SC}[\![com_1]\!]\sigma)$$

Figure 4.3: Abstract semantics for the commands.

$(\{\{res\}\} \uplus sh_{el}) \cup sh_{-el} = (\{\{res\}\} \uplus \{\{this, el\}\}) \cup \{\{v\}\} = \{\{res, this, el\}, \{v\}\}$. *In the case of pair-sharing, the transfer function [SS05] for the same initial state* $sh = \{\{this, el\}, \{v, v\}\}$ *returns* $sh'_p = \{\{res, el\}, \{res, this\}, \{this, el\}, \{v, v\}\}$, *which translated to set sharing results in* $sh'' = \{\{res,\ el\}, \{res, this\}, \{res, this,\ el\}, \{this, el\}, \{v\}\}$, *a less precise representation (in terms of* $\leq_{\mathcal{DS}_\pi}$*) than* sh'.

Example 4 *Our multivariant analysis keeps two different call contexts for the* `append` *method in the* `Vector` *class (Figure 4.1). Their different sharing information shows how sharing can improve nullity results. The first context corresponds to external*

calls (invocation from other classes), because of the `public` *visibility of the method:* $\sigma_1 = (\{\{this\}, \{this, v\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. *The second corresponds to an internal (within the class) call, for which the analysis infers that this and v do not share:* $\sigma_2 = (\{\{this\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. *Inside* `append`*, we avoid creating a circular list by checking that this* $\neq v$. *Only then is the last element of this linked to the first one of v. We use* `com` *to represent the series of commands* `Element e = first; if (e==null)...else..` *and* `bdy` *for the whole body of the method. Independently of whether the input state is* σ_1 *or* σ_2 *our analysis infers that* $\mathcal{SC}[\![\texttt{com}]\!]\sigma_1 = \mathcal{SC}[\![\texttt{com}]\!]\sigma_2 = (\{\{this, v\}\}, \{this/nnull,\ v/nnull\},\ \{this/\{vector\}, v/\{vector\}\}) = \sigma_3$. *However, the more precise sharing information in* σ_2 *results in a more precise analysis of* `bdy`*, because of the guard* `(this!=v)`. *In the case of the external calls,* $\mathcal{SC}[\![\texttt{bdy}]\!]\sigma_1 = \mathcal{SC}[\![\texttt{com}]\!]\sigma_1 \sqcup \mathcal{SC}[\![\texttt{skip}]\!]\sigma_1 = \sigma_1 \sqcup \sigma_3 = \sigma_1$. *When the entry state is* σ_2*, the semantics at the same program point is* $\mathcal{SC}[\![\texttt{bdy}]\!]\sigma_2 = \mathcal{SC}[\![\texttt{com}]\!]\sigma_2 = \sigma_3 < \sigma_1$. *So while the internal call requires* $v \neq null$ *to terminate, we cannot infer the final nullity of that parameter in a public invocation, which might finish even if v is null.*

4.3.1 Method Calls

The semantics of the expression `call`$(v,\ m(v_1, \ldots, v_n))$ in state $\sigma = (sh, nl, \tau)$ is calculated by implementing the top-down methodology described in [NMLH07]. We will assume that the formal parameters follow the naming convention F in all the implementations of the method; let $A = \{v, v_1, \ldots, v_n\}$ and $F = dom(input(k.m))$ be ordered lists. We first calculate the projection $\sigma_p = \sigma|_A$ and an entry state $\sigma_y = \sigma_p|_A^F$. The abstract execution of the call takes place only in the set of classes $K = \tau(v)$, resulting in an exit state $\sigma_x = \bigsqcup\{\mathcal{SC}[\![k'.m]\!]\sigma_y \,|\, k' = lookup(k, m), k \in K\}$, where *lookup* returns the body of k's implementation of m, which can be defined in k or inherited from one of its ancestors. The abstract execution of the method in a

subset $K \subseteq \downarrow\pi(v)$ increases analysis precision and is the ultimate purpose of tracking run-time types in our abstraction. We now remove the local variables $\sigma_b = \sigma_x|_{F\cup\{out\}}$ and rename back to the scope of the caller: $\sigma_\lambda = \sigma_b|^{A\cup\{res\}}_{F\cup\{out\}}$; the final state σ_f is calculated as $\sigma_f = extend(\sigma, \sigma_\lambda, A)$. The $extend : \mathcal{D}\times\mathcal{D}\times\mathcal{P}(dom(\pi)) \mapsto \mathcal{D}$ function is described in Algorithm 1.

Algorithm 1: Extend operation

input : state before the call σ, result of analyzing the call σ_λ
and actual parameters A

output: resulting state σ_f

if $\sigma_\lambda = \bot$ **then**
$\sigma_f = \bot$

else
let $\sigma = (sh, nl, \tau)$, and $\sigma_\lambda = (sh_\lambda, nl_\lambda, \tau_\lambda)$, and $AR = A \cup \{res\}$

$$
\begin{aligned}
star &= (sh_A \cup \{\{res\}\})^* \\
sh_{ext} &= \{s \mid s \in star, s|_{AR} \in sh_\lambda\} \\
sh_f &= sh_{ext} \cup sh_{-A} \\
nl_f &= nl[res \mapsto nl_\lambda(res)] \\
\tau_f &= \tau[res \mapsto \tau_\lambda(res)] \\
\sigma_f &= (sh_f, nl_f, \tau_f)
\end{aligned}
$$

end

In Java references to objects are passed by value in a method call. Therefore, they cannot be modified. However, the call might introduce new sharing between actual parameters through assignments to their fields, given that the formal parameters they correspond to have not been reassigned. We keep the original information by copying all the formal parameters at the beginning of each call, as suggested in [Pol04]. Those copies cannot be modified during the execution of the call, so a meaningful correspondence can be established between A and F.

We can do better by realizing that analysis might refine the information about the actual parameters within a method and propagating the new values discovered back to σ_f. For example, in a method `foo(Vector v){if v!=null skip else throw_null}`, it is clear that we can only finish normally if $nl_x(v) = nnull$, but in the actual semantics we do not change the nullity value for the corresponding argument in the call, which can only be more imprecise. Note that the example is different from `foo(Vector v){v = new Vector}`, which also finishes with $nl_x(v) = nnull$. The distinction over whether new attributes are preserved or not relies on keeping track of those variables which have been assigned inside the method, and then applying the propagation only for the unset variables.

Example 5 *Assume an extra snippet of code in the* `Vector` *class of the form* `if (v2!=null) v1.append(v2) else com`*, which is analyzed in state* $\sigma = (\{\{v_1\}, \{v_2\}\}, \{v_1/nnull, v_2/nnull\}, \{v_1/\{vector\}, v_2/\{vector\}\})$*. Since we have nullity information, it is possible to identify the block* `com` *as dead code. In contrast, sharing-only analyses can only tell if a variable is definitely null, but never if it is definitely non-null. The call is analyzed as follows. Let* $A = \{v_1, v_2\}$ *and* $F = \{this, v\}$*, then* $\sigma_p = \sigma|_A = \sigma$ *and the entry state* σ_y *is* $\sigma|_A^F = (\{\{this\}, \{v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\})$*. The only class where* `append` *can be executed is* `Vector` *and results (see Example 4) in an exit state for the formal parameters and the return variable* $\sigma_b = (\{\{this, v\}\}, \{this/nnull, v/nnull, out/null\}, \{this/\{vector\}, v/\{vector\}, out/\{void\}\})$*,*
which is further renamed to the scope of the caller obtaining $\sigma_\lambda = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$*. Since the method returns a* `void` *type we can treat* res *as a primitive (null) variable so* $\sigma_f = extend(\sigma, \sigma_\lambda, \{v_1, v_2\}) = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$*.*

Example 6 *The extend operation used during interprocedural analysis is a point*

where there can be significant loss of precision and where set sharing shows its strengths. For simplicity, we will describe the example only for the sharing component; nullity and type information updates are trivial. Assume a scenario where a call to `append(v1,v2)` *in sharing state* $sh = \{\{v_0, v_1\}, \{v_1\}, \{v_2\}\}$ *results in* $sh_\lambda = \{\{v_1, v_2\}\}$. *Let* A *and* AR *be the sets* $\{v_1, v_2\}$ *and* $\{v_1, v_2, res\}$ *respectively. The extend operation proceeds as follows: first we calculate star as* $(sh_A \cup \{\{res\}\})^* = (sh \cup \{\{res\}\})^* = (\{\{v_0, v_1\}, \{v_1\}, \{v_2\}, \{res\}\})^* = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_1\}, \{v_1, v_2\}, \{v_1, v_2, res\}, \{v_1, res\}, \{v_2\}, \{v_2, res\}, \{res\}\}$, *from which we delete those elements whose projection over* AR *is not included in* sh_λ, *obtaining* $sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$. *The resulting sharing component is the union of that* sh_{ext} *with* $sh_{-A} = \emptyset$, *so* $sh_{f1} = sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$.

When the same sh *and* sh_λ *are represented in their pair sharing versions* $sh^p = \{\{v_0, v_1\}, \{v_o, v_0\}, \{v_1, v_1\}, \{v_2, v_2\}\}$ *and* $sh^p_\lambda = \{\{v_1, v_2\}, \{v_1, v_1\}, \{v_2,\ v_2\}\}$, *the extend operation in [SS05] introduces spurious sharings in* sh_f *because of the lower precision of the pair-sharing representation. In this case,* $sh^p_{f2} = (sh \cup sh^p_\lambda)^*_A = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_0, v_0\}, \{v_1, v_1\}, \{v_2, v_1\}\}$. *This information, expressed in terms of set sharing, results in* $sh_{f2} = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0,\ v_1, v_2\}, \{v_1, v_2\}, \{v_0\}, \{v_1\}, \{v_2\}\}$, *which is much less precise that* sh_{f1}.

4.4 Proofs

We have to prove that $\alpha_\pi(\mathcal{E}^I_\pi[\![expr]\!](\gamma_\pi(\sigma)) \leq \mathcal{SE}[\![expr]\!]\sigma$ (in the case of commands, $\alpha_\pi(\mathcal{C}^I_\pi[\![com]\!](\gamma_\pi(\sigma)) \leq \mathcal{SC}[\![com]\!]\sigma$). We denote by LHS the left-hand side of the equation, which will be further rewritten until showing that it is approximated by the right-hand side (RHS), the semantics described in Fig. 4.2 and 4.3. The abstraction function for the sharing component is $\alpha_\pi(S) = \{V \subseteq dom(\pi) \mid \exists \delta \in$

S s.t. $\bigcap_{v_i \in V} R_\pi(\delta, v_i) \neq \emptyset$ and $\nexists W \subseteq dom(\pi)$ s.t. $V \subset W$ and $\bigcap_{w_i \in W} R_\pi(\delta, w_i) \neq \emptyset\}$. For the nullity component the abstraction is $\alpha_\pi(S) = \{v_i/null \in dom(\pi) \times \mathcal{DN}_\pi \mid \forall \delta \in S,\ R_\pi(\delta, v_i) = \emptyset\} \cup \{w_i/nnull \in dom(\pi) \times \mathcal{DN}_\pi \mid \forall \delta \in S,\ R_\pi(\delta, w_i) \neq \emptyset\} \cup \{y_i/unk \in dom(\pi) \times \mathcal{DN}_\pi \mid y_i \notin V, y_i \notin W\}$. Finally, types in the set of states S are abstracted as $\alpha_\pi(S) = \{v/T \in dom(\pi) \times \mathcal{P}(\mathcal{K}) \mid \forall \delta \in S,\ \psi_\pi(\delta, v) \in T\}$.

`null`

LHS $= \alpha_\pi(\{\phi[res \mapsto null] \star \mu | \phi \star \mu \in \gamma_\pi(\sigma)\})$. However, the addition of null variables cannot affect the sharing (from the definition of α_π) but only the nullity component. Therefore, LHS $= \alpha_\pi(\{\phi \star \mu \mid \phi \star \mu \in \gamma_\pi(\sigma)\}).nl[res \mapsto null] = \alpha_\pi(\gamma_\pi(\sigma)).nl[res \mapsto null] = (sh, nl[res \mapsto null], \tau) \leq \mathcal{SE}[\![\texttt{null}]\!]\sigma$. The nullity value for res is trivially correct; the rest of the variables are unaffected. The type value of res is the most general one and therefore correct.

`new` k

LHS $= \alpha_\pi(\{\phi[res \mapsto l] \star \mu[l \mapsto o] \mid \phi \star \mu \in \gamma_\pi(\sigma)\})$. Since l is a fresh location, res cannot reach any location already pointed to by another variable, so we can separate the memory state after the expression in two independent parts. By semantics of the language, l is a non-null location and therefore the nullity value for res correctly approximates the standard semantics; the type value for res is just the one of the class constructor invoked; the rest of the variables see no changes and their current values for nl and τ remain correct. LHS$= \alpha_\pi(\{\phi \star \mu \mid \phi \star \mu \in \gamma_\pi(\sigma)\}) \cup (\{\{res\}\}, \{res/nnull\}, \{res/\{k\}\}) = \alpha_\pi(\gamma_\pi(sh)) \cup (\{\{res\}\}, \{res/nnull\}, \{res/\{k\}\})$ $= \mathcal{SE}[\![\texttt{new}\ k]\!]\sigma$

v

LHS $= \alpha_\pi(\{\phi[res \mapsto \phi(v)] \star \mu \mid \phi \star \mu \in \gamma_\pi(\sigma)\})$. We will call the new frame ϕ'.

Since res is removed after evaluating an expression, we only have to check whether its addition to the frame is properly approximated. The new nullity and type values correctly approximate the effect of evaluating the expression, since v was correctly approximated by nl and τ and now res is a synonym of v; the rest of the variables remain unchanged so $(nl[res \mapsto nl(v)], \tau[res \mapsto \tau(v)])$ is a correct approximation for them.

If $nl(v) = null$ the semantics is the same as in `null`; if not, in the new state $\phi' \star \mu$ there is a subset of variables which did not reach any location reachable from v. Those variables are unaffected and their previous approximation sh_{-v} is correct. For the rest of the variables, if sh_v approximated their reachability then $sh_v \uplus \{\{res\}\}$ is the minimal approximation for $\phi' \star \mu$, since $R_\pi(\phi' \star \mu, v) = R_\pi(\phi' \star \mu, res)$ and therefore there cannot be any sharing in which v is included but res is not.

`v.f`

LHS $= \alpha_\pi(\{\phi[res \mapsto l] \star \mu \mid l = (obj(\phi \star \mu, v).\phi)(f)\ , \phi \star \mu \in \gamma_\pi(\sigma)\})$=$\alpha_\pi(\{\phi' \star \mu\})$. In a normal execution all those variables which did not reach a location reachable from v cannot be reached from res, and therefore they are correctly approximated by sh_{-v}. Variables $\{w_1, \ldots, w_n\}$ in any σ verifying $R_\pi(\phi \star \mu, w_i) \cap R_\pi(\phi \star \mu, v) \neq \emptyset$ might reach the l location or be reached from it. However, the only definitive information is that $R_\pi(\phi' \star \mu, v) \cap R_\pi(\phi' \star \mu, res) \neq \emptyset$, information captured by applying the $\uplus$ operator between $\{\{v\}\}$ and any sharing set where res appears. The remaining possibilities (including those already existing in $\phi \star \mu$) are correctly abstracted by $\{\{\{v\}\} \uplus \mathcal{P}(s|_{-v} \cup \{res\}) \mid s \in sh_v\}$, since we create a set for every possibility in a sharing set of sh_v but without introducing impossible sharings: for example, if $\{\{v, a\}, \{v, b\}\}$ was the starting state, the expression `v.f` cannot introduce sharing between a and b and the result is $\{\{v, a\}, \{v, a, res\}, \{v, b\}, \{v, b, res\}, \{v, res\}\}$. The nullity value for res is correct since it is the most general one; type information

is also trivially correct.

`call`$(v, m(v_1, \ldots, v_n))$

(We provide here an informal proof; the reader interested in the how the fixpoint is calculated in the presence of method calls can refer to Chapter 3).

In Java method calls cannot alter the caller frame (Fr_π), but just subsequent levels of indirection: fields of variables in the scope of the caller. The only exception to this is the returned value res. Hence, an analysis of the call which strictly computes the most general sharing for the actual parameters and res starting at the caller state, and that assumes the most general nullity and type values for res, is always correct.

An initial approximation for σ_f is therefore $star = ((sh_A \cup \{\{res\}\})^* \cup sh_{-A}, nl[res \mapsto unk], \tau[res \mapsto (\downarrow\pi(res))])$. We can improve precision by using the semantics of the callees σ_λ. That semantics is correct since it approximates the call in all the class hierarchy of $\pi(v)$, including $\psi_\pi(v)$.

The nullity and type value are trivial, since they cannot change during the call (but we can possibly find a more precise value for them, see Sect.4.3.1) except for res: we just copy the new values for that variable to nl_f and τ_f, which for the rest are clones of nl and τ, respectively. The sharing case is more complicated. On one hand we have sh_{star}, which (provably) is an over-approximation for sh_f, and on the other sh_λ, which describes the final state exclusively about the actual parameters. We now filter out from the former those elements such that their information about the actual parameters and res is incompatible with sh_λ, regardless of the other elements in the set, obtaining $sh_{ext} = \{s \mid s \in sh_{star}, s|_{AR} \in sh_\lambda\}$. All the sharings not related to the actual parameters are preserved, resulting in $sh_f = sh_{ext} \cup sh_{-A}$.

$v = expr$

LHS $=(\alpha_\pi(\{\phi'[v \mapsto \phi'(res)])|_{-res}) \star \mu' \mid \phi \star \mu \in \gamma_\pi(\sigma)\})$. The proof is analogous to the one of the v expression. Assume that the semantics $\mathcal{E}^I_\pi[\![expr]\!]$ is correct, the concrete semantics of the assignment is identical to that of expression evaluation, just exchanging the res and v variables. In the case of nullity and types, the resulting state just replaces res by v, which is the result of overwriting v values with those of res and then removing any occurrence of res.

The sharing component is more complex. First, all previous sharings of v are deleted ($sh' = sh|_{-\{v\}}$) and it now appears in all sharing groups where res was, approximated by $(sh'_{-res} \cup (sh_{res} \uplus \{\{v\}\}))|_{-res} = sh'_{-res} \cup (sh'_{res}|^v_{res}) = sh'|^v_{res} = (\mathcal{SC}[\![v{=}expr]\!]\sigma).sh$.

`v.f`$= expr$

Analogous to the `v.f` proof, but taking into account that res might share with other variables (and has to be removed after the assignment).

`if` v`==`w com_1 `else` com_2

If $sh|_{\{v,w\}} = \emptyset$, then $\nexists\delta \in \gamma_\pi(\sigma)$ s.t. $\phi(v) = \phi(w)$ by definition of $\gamma_\pi(\sigma)$. Therefore, LHS$=\alpha_\pi(\mathcal{C}^I_\pi[\![\texttt{if...}]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq \phi(w)\}))=\alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq \phi(w)\}))=\alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\}))\leq \mathcal{SC}[\![com_2]\!]\sigma =$RHS.

If $sh|_{\{v,w\}} \neq \emptyset$, then we might have $\phi(v) = \phi(w)$ and LHS$= \alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi\star\mu \in \gamma_\pi(\sigma) \mid \phi(v) = \phi(w)\})) \sqcup \alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq \phi(w)\})) \leq \alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!] (\{\phi \star \mu \in \gamma_\pi(\sigma)\})) \sqcup \alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\})) \leq \mathcal{SC}[\![com_1]\!]\sigma \sqcup \mathcal{SC}[\![com_2]\!]\sigma$=RHS.

`if` v`==null` com_1 `else` com_2

If $\sigma.nl[v] = null$, the concretization function ensures $\phi(v) = null\ \forall\phi \star \mu \in \gamma_\pi(\sigma)$

thus LHS=$\alpha_\pi(\mathcal{C}^I_\pi[\![\texttt{if}\ldots]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = null\}))$ =$\alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = null\}))$= $\alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!](\{\phi \star \mu \in \gamma_\pi(\sigma)\})) \leq \mathcal{SC}[\![com_1]\!]\sigma$ =RHS. A similar reasoning can be applied for the case where $nl[v] = nnull$.

If $nl[v] = unk$, LHS=$\alpha_\pi(\mathcal{C}^I_\pi[\![\texttt{if}\ldots]\!](\{\phi\star\mu \in \gamma_\pi(\sigma)|\phi(v) = null\}))\sqcup\alpha_\pi(\mathcal{C}^I_\pi[\![if...]\!](\{\phi\star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = nnull\}))$. The first term is equivalent to $\alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!])(\{\phi \star \mu \in \gamma_\pi(\sigma) \mid \phi(v) = null\})) = \alpha_\pi(\mathcal{C}^I_\pi[\![com_1]\!])(\{\phi \star \mu \in \gamma_\pi(sh|_{-v}, nl[v \mapsto null])\})$ (by definition of γ_π), which is $\leq \mathcal{SC}[\![com_1]\!](sh|_{-v}, nl[v \mapsto null])$. In an analogous way, the second term is $\alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!](\{\phi\star\mu \in \gamma_\pi(\sigma) \mid \phi(v) \neq null\})) = \alpha_\pi(\mathcal{C}^I_\pi[\![com_2]\!])(\{\phi\star\mu \in \gamma_\pi(sh, nl[v \mapsto nnull])\}) \leq \mathcal{SC}[\![com_2]\!](sh, nl[v \mapsto nnull])$. Therefore, the left-hand side of the equation is approximated by the semantics given.

$com_1\,;com_2$

True by correctness of the composition of correct operations.

4.5 Experimental results

We now provide some precision and cost results obtained from the implementation in the framework described in the Chapters 2 and 3 of our set-sharing, nullity, and class ($SSNlTau$) analysis. In order to be able to provide a comparison with the closest previous work, we also implemented the pair sharing (PS) analysis proposed in [SS05]. We have extended the operations described in [SS05], enabling them to handle some additional cases required by our benchmark programs such as primitive variables, visibility of methods, etc. Also, to allow direct comparison, we implemented a version of our $SSNlTau$ analysis, which is referred to simply as SS, that tracks set sharing using only declared type information and does not utilize the (non-)nullity component. In order to study the influence of tracking run-time types we have implemented a version of our analysis with set sharing and (non-)nullity, but

		PS				SS				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta_t$
dyndisp*	71	68	3	114	30	68	3	114	29	-2
clone	41	38	3	42	52	38	3	50	81	55
dfs	102	98	4	103	68	98	4	108	68	0
passau*	167	164	3	296	97	164	3	304	120	23
qsort	185	142	43	182	125	142	43	204	165	32
intqsort	191	148	43	159	110	148	43	197	122	10
pollet01*	154	126	28	276	196	126	28	423	256	30
zipvector*	272	269	3	513	388	269	3	712	1029	164
cleanness*	314	277	37	360	233	277	37	385	504	116
overall	1497	1330	167	2045	1299	1330	167	2497	2374	82.75

Figure 4.4: Analysis times, number of program points, and number of abstract states.

again using only the static types, which we will refer to as $SSNl$. In these versions without dynamic type inference only declared types can affect τ and thus the dynamic typing information that can be propagated from initializations, assignments, or correspondence between arguments and formal parameters on method calls is not used. Note however that the version that includes tracking of dynamic typing can of course only improve analysis results in the presence of polymorphism in the program: the results should be identical (except perhaps for the analysis time) in the rest of the cases. The polymorphic programs are marked with an asterisk in the tables.

The benchmarks used have been adapted from previous literature on either abstract interpretation for Java or points-to analysis [SS05, PLC01, Pol04, SS00]. We added two different versions of the `Vector` example of Figure 4.1. Our experimental results are summarized in Tables 4.4, 4.5, and 4.6.

The first column ($\#tp$) in Tables 4.4 and 4.5 shows the total number of program points (commands or expressions) for each program. Column $\#rp$ then provides, for each analysis, the total number of *reachable* program points, i.e., the number of pro-

		SSNl					SSNlTau				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta_t$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta_t$
dyndis*	71	61	10	103	53	77	61	10	77	20	-33
clone	41	31	10	34	100	92	31	10	34	90	74
dfs	102	91	11	91	129	89	91	11	91	181	166
passau*	167	157	10	288	117	18	157	10	270	114	17
qsort	185	142	43	196	283	125	142	43	196	275	119
intqsort	191	148	43	202	228	107	148	43	202	356	224
pollet01*	154	119	35	364	388	98	98	56	296	264	35
zipvect*	272	269	3	791	530	36	245	27	676	921	136
cleanss*	314	276	38	383	276	38	266	48	385	413	77
overall	1497	1294	203	2452	2104	61	1239	258	2227	2634	102

Figure 4.5: Analysis times, number of program points, and number of abstract states.

gram points that the analysis explores, while $\#up$ represents the $(\#tp - \#rp)$ points that are not analyzed because the analysis determines that they are unreachable. It can be observed that tracking (non-)nullity (Nl) reduces the number of reachable program points (and increases conversely the number of unreachable points) because certain parts of the code can be discarded as dead code (and not analyzed) when variables are known to be non-null. Tracking dynamic types (Tau) also reduces the number of reachable points, but, as expected, only for (some of) the programs that are polymorphic. This is due to the fact that the class analysis allows considering fewer implementations of methods, but obviously only in the presence of polymorphism.

Since our framework is multivariant and thus tracks many different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number of abstract states inferred is typically larger than the number of reachable program points. Column $\#\sigma$ provides the total number of these abstract states inferred by the analysis. The level of multivariance is the ratio $\#\sigma/\#rp$. It can be observed that the simple set sharing

analysis (SS) creates more abstract states for the same number of reachable points. In general, such a larger number for $\#\sigma$ tends to indicate more precise results (as we will see later). On the other hand, the fact that addition of Nl and Tau reduces the number of reachable program points interacts with precision to obtain the final $\#\sigma$ value, so that while there may be an increase in the number of abstract states because of increased precision, on the other hand there may be a decrease because more program points are detected as dead code by the analysis. Thus, the $\#\sigma$ values for $SSNl$ and $SSNlTau$ in some cases actually decrease with respect to those of PS and SS.

The t column in Tables 4.4 and 4.5 provides the running times for the different analyses, in milliseconds, on a Pentium M 1.73Ghz, 1Gb of RAM, running Fedora Core 4.0, and averaging several runs after eliminating the best and worst values. The $\%\Delta t$ columns show the percentage variation in the analysis time with respect to the reference pair-sharing (PS) analysis, calculated as $\Delta_{dom}\%t = 100 * (t_{dom} - t_{PS})/t_{PS}$. The more complex analyses tend to take longer times, while in any case remaining reasonable. However, sometimes more complex analyses actually take less time, again because the increased precision and the ensuing dead code detection reduces the amount of program that must be analyzed.

Table 4.6 shows precision results in terms of sharing, concentrating on the SP and SS domains, which allow direct comparison. A more usage-oriented way of measuring precision would be to study the effect of the increased precision in an application that is known to be sensitive to sharing information, such as, for example, program parallelization [BdlBH99]. On the other hand this also complicates matters in the sense that then many other factors come into play (such as, for example, the level of intrinsic parallelism in the benchmarks and the parallelization algorithms) so that it is then also harder to observe the precision of the analysis itself. Such a client-level comparison is beyond the scope of this chapter, and we concentrate here instead on

	PS		SS	
	$\#sh$	$\%sh$	$\#sh$	$\%sh$
dyndisp (*)	640	60.37	435	73.07
clone	174	53.10	151	60.16
dfs	1573	96.46	1109	97.51
passau (*)	5828	94.56	3492	96.74
qsort	1481	67.41	1082	76.34
integerqsort	2413	66.47	1874	75.65
pollet01 (*)	793	89.81	1043	91.81
zipvector (*)	6161	68.71	5064	80.28
cleanness (*)	1300	63.63	1189	70.61
overall	20363	73.39	15439	80.24

Figure 4.6: Sharing precision results.

measuring sharing precision directly.

Following [CMB+95], and in order to be able to compare precision directly in terms of sharing, column $\#sh$ provides the sum over all abstract states in all reachable program points of the cardinality of the sharing *sets* calculated by the analysis. For the case of pair sharing, we converted the pairs into their equivalent set representation (as in [CMB+95]) for comparison. Since the results are always correct, a smaller number of sharing sets indicates more precision (recall that $\top$ is the power set). This is of course assuming σ is constant, which as we have seen is not the case for all of our analyses. On the other hand, if we compare PS and SS, we see that SS has consistently more abstract states than PS and consistently lower numbers of sharing sets, and the trend is thus clear that it indeed brings in more precision. The only apparent exception is *pollet01* but we can see that the number of sharing sets is similar for a significantly larger number of abstract states.

An arguably better metric for measuring the relative precision of sharing is the ratio $\%_{Max} = 100 * (1 - \#sh/(2^{\#vo} - 1))$ which gives $\#sh$ as a percentage of its

$\mathcal{SE}[\![\texttt{null}]\!](SH)$
$SH' = SH$
$\mathcal{SE}[\![\texttt{new } k]\!](SH)$
$SH' = SH \cup \{\{res\}\}$
$\mathcal{SE}[\![v]\!](SH)$
$SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$
$\mathcal{SE}[\![v\,.\,\texttt{f}]\!](SH)$
$SH' = \begin{cases} \bot \text{ if } \mathbf{mustBeNull}(SH, v) \\ SH \cup (\{\{v, res\}\} \uplus \bigcup_{S \in SH_v} \mathcal{P}(S\|_{-v})) \text{ else} \end{cases}$

Figure 4.7: Abstract semantics for the expressions as set operations

maximum possible value, where $\#vo$ is the total number of object variables in all the states. The results are given in column $\%sh$. In this metric 0% means all abstract states are $\top$ (i.e., contain no useful information) and 100% means all variables in all abstract states are detected not to share. Thus, larger values in this column indicate more precision, since analysis has been able to infer smaller sharing sets. This relative measure shows an average improvement of 7% for SS over PS.

4.6 A more precise set of abstract operations

In this section, we review the abstract set sharing semantics that was defined in Section 4.3, and improve the precision for two of the operations: the field load and the field store. Figs. 4.7 and 4.10 contain the semantics of expressions and commands, respectively. They represent the transition from an initial abstract state SH to a final abstract state SH'.

Our abstract state has now been reduced to a pair composed of an abstract set

sharing and a type component τ. The latter helps in determining which variables are non null and which ones may be null, thus the nullity component of the abstract domain in Section 4.3 can be eliminated. If we consider *null* as another type [Ler03], then a variable may be null if *null* is one of its possible types: $\mathbf{mayBeNull}(\tau, v) = (\overline{\mathbf{mustBeNull}}(SH, v)$ and null $\in \tau(v))$. For clarity, we omitted the type component from the transfer functions in Figure 4.7 and 4.10.

4.6.1 Semantics of Expressions (refinement)

Null, New and Variable Load: The `null` expression loads the null constant into the special variable res, so it has no effect on the abstract state, since res does not point to any object, and therefore does not share with any variable (including itself), both before and after evaluating the expression. The `new` expression adds the singleton $\{res\}$ to the current set sharing, since it creates a fresh object that cannot be reached from any of the existing variables. A variable load `v` forces res to be an alias of v, and therefore res shares with all those variables with which v shares. Sharings in SH_{-v} remain unaffected, since the addition of res cannot change the reachability set of any variable not reachable from v. For instance, given $SH = \{\{v_0, v_1, v_2\}, \{v_3\}\}$, the variable load v_0 results in $SH' = SH_{-v} \cup (\{\{res\}\} \uplus SH_v) = \{\{v_3\}\} \cup (\{\{res\}\} \uplus \{\{v_0, v_1, v_2\}\}) = \{\{v_3\}\} \cup \{\{res\} \cup \{v_0, v_1, v_2\}\} = \{\{v_0, v_1, v_2, res\}, \{v_3\}\}$.

Field Load: In the case that v`.f` is null, there is no change in the existing set sharing. Because the expression of SH' includes SH, that case is correctly approximated. When v`.f` is not null, we know that the object being assigned to res is reachable from v. The other variables that share with v in SH may or may not share with res in SH'. In the state G_0 of Figure 4.8, although v_2 shares with v_0 in the initial and final states, it does not share with res in the final state; however, v_1 will

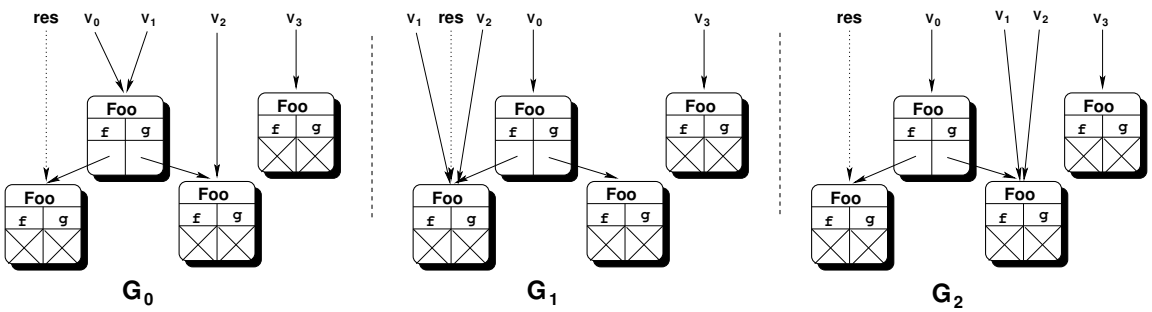

Figure 4.8: Three concrete states.

share with both res and v_0 after the load. We write $\{\{v, res\}\} \uplus \bigcup_{S \in SH_v} \mathcal{P}(S|_{-v}))$ to account for objects reachable from v which become also reachable from res, and may be reachable from any subset of the variables that shared with v in SH. Objects not reachable from v (SH_{-v}) are accounted for by the union with SH. For instance, in the same state G_0, if $\{v_3\} \in SH$, then the load of v_0`.f` does not alter that particular element, which has to also be present in SH'.

Example 7 *The graphs in Figure 4.8 illustrate three different memory states before the evaluation of* $v_0.\mathtt{f}$. *The initial set sharing is identical in all cases:* $sh(G_0) = sh(G_1) = sh(G_2) = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. *However, the evaluation results in a different set sharing for each resulting graph* G'_i*:* $sh(G'_0) = \{\{v_0, v_1, v_2\}, \{v_0, v_1, res\}, \{v_3\}\}$, $sh(G'_1) = \{\{v_0, v_1, v_2, res\}, \{v_3\}\}$, *and* $sh(G'_2) = \{\{v_0, v_1, v_2\}, \{v_0, res\}, \{v_3\}\}$. *Assume that the abstract state that approximates all the initial concrete states is also* $SH = \{\{v_0, v_1, v_2\}, \{v_3\}\}$. *The transfer function for* $v_0.\mathtt{f}$ *results in a final abstract state* $SH' = SH \cup (\{\{v_0, res\}\} \uplus \mathcal{P}(\{v_1, v_2\})) = \{\{v_0, v_1, v_2\}, \{v_3\}\} \cup (\{\{v_0, res\}\} \uplus \{\{\}, \{v_1\}, \{v_2\}, \{v_1, v_2\}\}) = \{\{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_0, v_2, res\}, \{v_0, res\}, \{v_3\}\}$. *As required, all the sharings* $sh(G'_0)$, $sh(G'_1)$, *and* $sh(G'_2)$ *are included in* SH'.

4.6.2 Semantics of Commands (refinement)

Variable Store: For a store of the form v=$expr$, the semantics comprises three steps. First, the expression on the right-hand side is evaluated. Second, all ocurrences of v are removed from the current abstract state, since the value of v is being overwritten. Finally, all appearances of res are replaced by v, which deletes res from the abstract state.

Field Store: First, we evaluate the expression whose result is being stored; SH_1 contains that intermediate value. Sharings in SH_1 unrelated to v or res are unaffected by the store and contained in $SH_2 = SH_{1_{-\{v,res\}}}$, which is a subset of the final state. For each sharing in SH_{1_v}, the store might affect the reachability set of each variable involved and result in many smaller sharings. For example, in a memory state like G in Figure 4.9, an assignment to v_0`.f` destroys any sharing between v_0 and v_1 (note that res does not share with v_1), but not the one between v_0 and v_2. All the possible combinations for the final sharings that have to do with v are contained in $SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$.

Now, for every sharing in SH_3 that contains v we have two possibilities: all the variables share also with res (and therefore, with $SH_{1_{res}}$), or none of them does. Note that every possible intermediate case in which just a few of the variables share with $SH_{1_{res}}$ is represented by a smaller subset in SH_3 containing only those variables. While $SH_4 = SH_{1_{res}} \uplus SH_{3_v}$ includes the combinations in which all the variables do share with $SH_{1_{res}}$, SH_3 approximates the situations in which none of them do share with res.

Example 3. Assume an initial state (after evaluating the expression) G depicted in Figure 4.9. The dotted edge indicates where v_0`.f` will point after the execution of v_0`.f`$=$ $expr$. The initial set sharing is $sh(G) = \{\{v_0, v_1\}, \{v_0, v_2\}, \{res\}\}$. After the

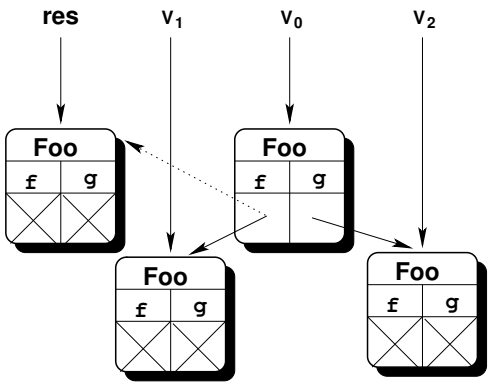

Figure 4.9: Graph G.

load, $sh(G') = \{\{v_0, v_2\}, \{v_0, res\}, \{v_1\}\}$. Assume that the starting abstract state, after the evaluation of the expression $expr$, is also $SH_1 = \{\{v_0, v_1\}, \{v_0, v_2\}, \{res\}\}$. Since there is no sharing unrelated to v or res, $SH_2 = \emptyset$. The next step is to calculate $SH_3 = \mathcal{P}(\{v_0, v_1\}) \cup \mathcal{P}(\{v_0, v_2\}) \setminus \{\{\}\} = \{\{v_0\}, \{v_0, v_1\}, \{v_1\}\} \cup \{\{v_0\}, \{v_0, v_2\}, \{v_2\}\}$ $= \{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1\}, \{v_2\}\}$. Since $SH_{1_{res}} = \{\{res\}\}$ and $SH_{3_{v0}} = \{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}\}$, $SH_4 = \{\{v_0, res\}, \{v_0, v_1, res\}, \{v_0, v_2, res\}\}$. The final abstract state $SH' = \{\{v_0\}, \{v_0, v_1\}, \{v_0, v_2\}, \{v_1\}, \{v_2\}\}$ is the union of $SH_3|_{-res} = SH_3$ and $SH_4|_{-res} \subset SH_3$. As required, $sh(G') \subseteq SH'$ holds after the removal of the auxiliary variable res from G'.

Conditional Statements: In the case where the guard is (v`==null`), the type component may contain definite information about whether a variable v is not null ($null \notin \tau(v)$). If we cannot determine exactly the nullity of v (i.e., **mayBeNull**(τ, v) is true), then the final state is the least upper bound of the resulting set sharing for the two branches. In particular, $SH_1 \sqcup SH_2 = SH_1 \cup SH_2$.

In the case where the condition is v`==`w, the sharing information may be enough to tell that the two variables are definitely equal, because they are both null and there-

$\mathcal{SC}[\![v\texttt{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}[\![expr]\!](SH)$ $SH_2 = SH_1\|_{-v}$ $SH' = SH_2\|^v_{res}$
$\mathcal{SC}[\![v\texttt{.f=}expr]\!](SH)$
$SH_1 = \mathcal{SE}[\![expr]\!](SH)$ $SH_2 = SH_{1_{-\{v,res\}}}$ $SH_3 = \bigcup_{S \in SH_{1_v}} \mathcal{P}(S) \setminus \{\{\}\}$ $SH_4 = SH_{1_{res}} \uplus SH_{3_v}$ $SH' = \begin{cases} \bot \text{ if } \mathbf{mustBeNull}(SH_1, v) \\ SH_2 \cup (SH_3 \cup SH_4)\|_{-res} \text{ else} \end{cases}$
$\mathcal{SC}[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}[\![com_1]\!](SH\|_{-v})$ $SH_2 = \mathcal{SC}[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \mathbf{mustBeNull}(SH, v) \\ SH_1 \cup SH_2 \text{ if } \mathbf{mayBeNull}(\tau, v) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}[\![com_1]\!](SH)$ $SH_2 = \mathcal{SC}[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \mathbf{mustAlias}(SH, v, w) \\ SH_1 \cup SH_2 \text{ if } \mathbf{mayAlias}(SH, v, w) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}[\![com_1\,\texttt{;}\,com_2]\!](SH)$
$SH' = \mathcal{SC}[\![com_2]\!](\mathcal{SC}[\![com_1]\!](SH))$

Figure 4.10: Abstract semantics for the commands.

fore $\mathbf{mustAlias}(SH, v, w) = (\mathbf{mustBeNull}(SH, v)$ and $\mathbf{mustBeNull}(SH, w))$. On the other hand, v and w do not share if they do not appear together within a subset of SH. Therefore $\mathbf{mayAlias}(SH, v, w) = (\overline{\mathbf{mustAlias}}(SH, v, w)$ and $SH_{\{v,w\}} \neq \emptyset)$.

It is important to see that sharing information does not imply equality: a set sharing like $\{\{v, w\}\}$ indicates that v and w might reach a common object, not that they must be aliases.

Example 9 *Given a command like* `if (cond)` $v0 = v1$ `else` $\{v0 =$ `null;` $v1 =$ `null`$\}$*, and assuming an initial abstract state* $SH = \emptyset$ *that does not contain enough information to determine cond, the set sharing corresponding to the* `if` *branch is* $SH_1 = \{\{v_0, v_1\}\}$*. The abstract state after simulating the* `else` *branch is* $SH_2 = \{\}$*. Therefore, the final state is* $SH' = SH_1 \cup SH_2 = \{\{v_0, v_1\}\}$*. However,* SH' *does not imply that* v_0 *necessarily shares with* v_1*, even when they appear together in* SH'*, but that* v_0 might *reach an object reachable from* v_1 *in some of the concrete states approximatted by* SH'*; in the example, if cond would be false, both variables are null and do not share.*

4.7 Related Work

The closest related work is that of [SS05] which develops a *pair*-sharing [Søn86] analysis for object-oriented languages and, in particular, Java. Our description of the set sharing part of our domain is in fact based on their elegant formalization. The fundamental difference is that we track *set* sharing instead of *pair* sharing, which provides increased accuracy in many situations and can be more appropriate for certain applications, such as detecting independence for program parallelization. Also, our domain and abstract semantics track additionally nullity and classes in a combined fashion which, as we have argued above, is particularly useful in the presence of multivariance. In addition, we deal directly with a larger set of object features such as inheritance and visibility. Finally, we have implemented our domains (as well as the pair sharing domain of [SS05]), integrated them in our multivariant analysis and verification framework, and benchmarked the system. Our experimental

results are encouraging in the sense that they seem to support that our contributions improve the analysis precision at reasonable cost.

In [Pol04, PLC01], the authors use a *distinctness* domain in the context of an abstract interpretation framework that resembles our sharing domain: if two variables point to different abstract locations, they do not share at the concrete level. Their approach is closer to shape analysis [SRW99] than to sharing analysis, which can be inferred from the former. Although information retrieved in this way is generally more precise, it is also more computationally demanding and the abstract operations are more difficult to design. We also support some language constructs (e.g., visibility of methods) and provide detailed experimental results, which are not provided in their work.

Most recent work [SB06, MRR02, WL04] has focused on context-sensitive approaches to the points-to problem for Java. These solutions are quite efficient, but flow-insensitive and overly conservative. Therefore, a verification tool based on the results of those algorithms may raise spurious warnings. In our case, we are able to express sharing information in a safe manner, as invariants that all program executions verify at the given program point.

4.8 Chapter conclusions

We have proposed an analysis based on abstract interpretation for deriving precise sharing information for a Java-like language. Our analysis is multivariant, which allows separating different contexts, and combines Set Sharing, Nullity, and Classes: the domain captures which instances definitely do not share or are definitively null, and uses the classes to refine the static information when inheritance is present. We have implemented the analysis, as well as previously proposed analyses based on pair sharing, and obtained encouraging results: for all the examples the set sharing

domains (even without combining with Nullity or Classes) offer more precision than the pair sharing counterparts while the increase in analysis times appears reasonable. In fact the additional precision (also when combined with nullity and classes) brings in some cases analysis time reductions. This seems to support that our contributions bring more precision at reasonable cost.

Chapter 5

Using ZBDDs to represent Set Sharing

5.1 Background and Motivation

In the previous chapter we have shown that set sharing is a more precise alternative than pair sharing. However, some of the intrinsic operations of the set sharing domain are exponential in the number of local variables being tracked, which can become a problem for certain programs and has limited so far its wider application. This intrinsic complexity can be dealt with in part by introducing widenings, i.e., simplifying the sharing sets conservatively when they become too large, but of course at the expense of losing precision [CC92, Fec96, ZBH99, NBH06]. Finding significantly more efficient implementations reduces the need for resorting to such lossy solutions and consequently improves practicality.

To this end, we introduce a new, efficient implementation of the set sharing domain using Zero-supressed Binary Decision Diagrams (ZBDDs). ZBDDs were designed to represent sets of combinations (i.e., sets of sets), so they can represent very

naturally the elements of the set sharing domain. To the best of our knowledge this is the first link provided between set sharing and ZBDDs. We start by showing how to express the operations needed for implementing the set sharing transfer functions in terms of basic ZBDD operations. Also, for some of the operations, we propose custom ZBDD algorithms that are more appropriate for these particular cases than those in the standard ZBDD libraries. In particular we provide a design for native ZBDD operations that emulate non-standard set manipulations. The introduction of ZBDDs is done at the implementation level and does not alter the definition of the domain operations, so that the domain designer does not need to be aware of their presence. Due to the new, underlying data structure, we were able to scale from the small set of experiments in Chapter 4 (involving at most 50 elements at a time) up to thousands of sharings and still get reasonable times.

Additionally, we provide performance results comparing two implementations of the set-sharing domain: an efficient, compact, bitset-based alternative (representing a highly-tuned version of the traditional approach) and our ZBDD-based implementation. The results show that the ZBDD version performs better in terms of both memory usage and running time. Our custom ZBDD algorithms are also shown to perform better in practice than the stock ones.

5.2 Semantics as ZBDD operations

Zero-suppressed BDDs (ZBDDs) [iM93, iM96] are a data structure similar to binary decision diagrams (BDDs) [Bry92], but designed to encode sets of combinations (i.e., sets of sets of primitive elements). To encode the set sharing domain using ZBDDs, we define the primitive elements to be the variables in the program being analyzed. ZBDDs have been demostrated to perform better [MT98, LSJ07] than standard BDDs when encoding sets of combinations that are sparse in the sense that

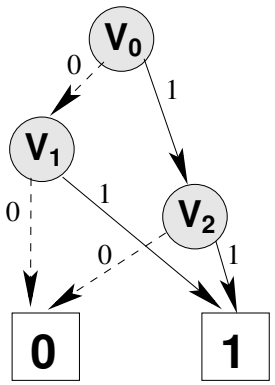

Figure 5.1: ZBDD representing $\{\{v_0, v_2\}, \{v_1\}\}$

a) the set contains just a small fraction of all the possible combinations, and b) each combination contains just a few literals. A ZBDD is a rooted directed acyclic graph (DAG) of non-terminal and terminal nodes. Each non-terminal ZBDD node is labeled with a variable, and has two outgoing edges to other nodes, called the zero-edge and the one-edge. There are two terminal nodes, the zero node and the one node. They do not have variables or outgoing edges. The universe of all variables is totally ordered, and the order of the variables appearing on the nodes of any path through the ZBDD is consistent with the total order. Each path through the ZBDD that ends at the one terminal node defines a set of variables. The set contains a variable v if the path passes through a node labeled with v, and leaves the node along its one edge. Assuming the variable ordering is fixed, the smallest ZBDD representing a given set of sets is unique.

Example 5. Assume a set of variables $Var = \{v_0, v_1, v_2\}$ and the variable ordering v_0, v_1, v_2. The unique smallest ZBDD representing the set of sets $\{\{v_0, v_2\}, \{v_1\}\}$ is the ZBDD shown in Figure 5.1. There are two paths from the root of the ZBDD to the one terminal node. On the path containing the v_0 and v_1, only the node labeled v_1 is exited through the one edge; thus, this path represents the set $\{v_1\}$. On the

path containing v_0 and v_2, both nodes are exited through their one edges; thus, this path represents the set $\{v_0, v_2\}$.

Efficient algorithms exist for common operations on the set of sets encoded by a ZBDD, including union (denoted +), intersection, set difference, product ($SH_1 * SH_2 = \{ S_1 \cup S_2 \mid S_1 \in SH_1 \text{ and } S_2 \in SH_2 \}$), and rest ($SH\%v = \{ S \in SH \mid v \notin S \}$), and division ($SH/v = \{ S \setminus \{v\} \mid S \in SH \text{ and } v \in S \}$).

A set sharing like $SH = \{\{v_0, v_2\}, \{v_1\}\}$ is expressed in ZBDD notation as $SH = v_0v_2 + v_1$. Note that we will denote single literal sets by a single lower case letter (like v), while generic ZBDDs will be referred to with double upper case (normally, SH). For instance, given the set sharings $SH = v_0v_2 + v_1$ and v_0 , an expression like $SH * v_0 = v_0v_1 + v_0v_2$ is legal. The empty set is written as 0, and the set containing only the empty set is written as 1.

5.2.1 Expressions and Commands; Native Operations

Figs. 5.2 and 5.3 show the ZBDD version of the transfer functions in Figure 4.7 and 4.10. For most of the set operations, there is an equivalent native ZBDD operation. For instance, $SH_1 \uplus SH_2$ is equivalent to $SH_1 * SH_2$ and SH_{-v} is equivalent to $SH\%v$. This correspondence is useful because it results in no gap between the denotational semantics of Section 4.6 and the implementation. However, we added a number of non-standard ZBDD operators to improve the readability of the equations. The set of elements in SH containing v (SH_v, in set notation) is obtained via $SH//v = SH/v * v$. We delete all the ocurrences of v in SH using $\textbf{projOut}(SH, v) = SH/v + SH\%v - 1$. The unit set 1 (which represents the set containing the empty set) has to be deleted because SH might contain the single literal v, as we did in the corresponding *project out* set operator $SH|_{-v}$.

In other occasions, we created new ZBDD operators because of efficiency reasons.

$\mathcal{SE}[\![\texttt{null}]\!](SH)$
$SH' = SH$
$\mathcal{SE}[\![\texttt{new } k]\!](SH)$
$SH' = SH + res$
$\mathcal{SE}[\![v]\!](SH)$
$SH' = \mathbf{setResEqTo}(SH, v)$
$\mathcal{SE}[\![v\,.\,\texttt{f}]\!](SH)$
$SH' = \begin{cases} \bot \text{ if } \mathbf{mustBeNull}(SH, v) \\ SH + v * res * \mathbf{powUnion}(SH/v) \text{ else} \end{cases}$

Figure 5.2: Abstract semantics for the expressions as ZBDD operations

For instance, the variable load set equation $SH' = (\{\{res\}\} \uplus SH_v) \cup SH_{-v}$ can be expressed as $SH' = res*(SH//v)+SH\%v$. This combination of standard operators, while intuitive, has the disadvantage of being inefficient in practice. Since we expect this function to be invoked with high frequency (every time a variable is on the right hand side of an assignment), we devised a dedicated (i.e., an algorithm that manipulates the graph itself instead of relying on a combination of the primitive operators) ZBDD algorithm that computes the same result, $\mathbf{setResEqTo}(SH, v)$. The algorithm, shown in Figure 5.4, uses the same notation as in [iM96]: P_0 and P_1 for the graph reachable through the zero-edge and one-edge, respectively, *P.top* for the current variable, and $Getnode(v, P_0, P_1)$ for the procedure that generates a node with the variable v and subgraphs P_0 and P_1. The correctness of $\mathbf{setResEqTo}(SH, v)$ is based on a variable order in which res is always the last variable, the one closer to the leaves. Given this precondition, we only need to find v in the graph, and then multiply its one-edge child by res, which will preserve the variable order.

With the basic ZBDD operators and **setResEqTo** we can understand the transfer functions of the null, new, and variable load expressions. The field load, on the other

$\mathcal{SC}[\![v\texttt{=}expr]\!](SH)$
$SH_1 = \mathcal{SE}[\![expr]\!](SH)$ $SH_2 = \mathbf{projOut}(SH_1 \% res, v)$ $SH' = SH_1 / res * v + SH_2$
$\mathcal{SC}[\![v\texttt{.f=}expr]\!](SH)$
$SH_1 = \mathcal{SE}[\![expr]\!](SH)$ $SH_2 = SH_1 \% v \% res$ $SH_3 = \mathbf{projOut}($ $\quad \mathbf{powUnion}(SH_1 // v) - 1, res)$ $SH_4 = (SH_1 / res) * (SH_3 // v)$ $SH' = \begin{cases} \bot \text{ if } \mathbf{mustBeNull}(SH_1, v) \\ SH_2 + SH_3 + SH_4 \text{ else} \end{cases}$
$\mathcal{SC}[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}[\![com_1]\!](\mathbf{projOut}(SH, v))$ $SH_2 = \mathcal{SC}[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \mathbf{mustBeNull}(SH, v) \\ SH_1 + SH_2 \text{ if } \mathbf{mayBeNull}(\tau, v) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](SH)$
$SH_1 = \mathcal{SC}[\![com_1]\!](SH)$ $SH_2 = \mathcal{SC}[\![com_2]\!](SH)$ $SH' = \begin{cases} SH_1 \text{ if } \mathbf{mustAlias}(SH, v, w) \\ SH_1 + SH_2 \text{ if } \mathbf{mayAlias}(SH, v, w) \\ SH_2 \text{ else} \end{cases}$
$\mathcal{SC}[\![com_1\,\texttt{;}\,com_2]\!](SH)$
$SH' = \mathcal{SC}[\![com_2]\!](\mathcal{SC}[\![com_1]\!](SH))$

Figure 5.3: Abstract semantics for the commands as ZBDD operations.

hand, depends on the ZBDD version of the predicate that determines whether a variable is null: $\mathbf{mustBeNull}(SH, v) = (SH/v = 0)$. It also requires computing

```
setResEqTo(P) {
  if (P = 0 or P = 1 or P.top > v)
    return P;
  if (P.top < v))
    return Getnode(P.top, P0, P1);
  return Getnode(P.top, P0, res*P1);
}


powUnion(P) {
  if (P = 0 or P = 1)
    return P;
  R0 ← powUnion(P0);
  R1 ← powUnion(P1);
  return Getnode(P.top, Ro + R1, 1 + R1);
}
```

Figure 5.4: Native operations **setResEqTo** and **powUnion**

the union of the powersets of the elements of a set sharing SH: $\{\mathcal{P}(S) \mid S \in SH\}$. Although this seems to be a complex operation, it has a very natural description in terms of an algorithm in ZBDDs. We have devised a native ZBDD algorithm, **powUnion**(SH), shown in pseucode in Figure 5.4. This native implementation will prove to be fundamental for the efficiency of the analysis (Section 5.3). The correctness proof of the algorithm follows:

Proof 1 ***powUnion****(SH) correctly computes* $\bigcup_{S \in SH} \mathcal{P}(S)$*:*

$\textbf{\textit{powUnion}}(ZBDD(a, P_0, P_1)) = \textbf{\textit{powUnion}}(P_0 + a * P_1) = \textbf{\textit{powUnion}}(P_o) + \textbf{\textit{powUnion}}(a * P_1) = \bigcup_{S \in P_0} \mathcal{P}(S) \cup \bigcup_{S \in P_1} (\mathcal{P}(S \cup \{a\}) = \bigcup_{S \in P_0} \mathcal{P}(S) \cup \{\{a\}\} \cup \bigcup_{S \in P_1} (\mathcal{P}(S) \uplus \{\{\}, \{a\}\}) = \bigcup_{S \in P_0} \mathcal{P}(S) \cup \bigcup_{S \in P_1} \mathcal{P}(S) \cup \{\{a\}\} \cup \bigcup_{S \in P_1} (\mathcal{P}(S) \uplus \{\{a\}\}) = \bigcup_{S \in P_0 \cup P_1} \mathcal{P}(S) \cup \{\{a\}\} \cup (\{\{a\}\} \uplus \bigcup_{S \in P_1} \mathcal{P}(S)) = ZBDD(a, \textbf{\textit{powUnion}}(P_0 + P_1), 1 + \textbf{\textit{powUnion}}(P_1))$.

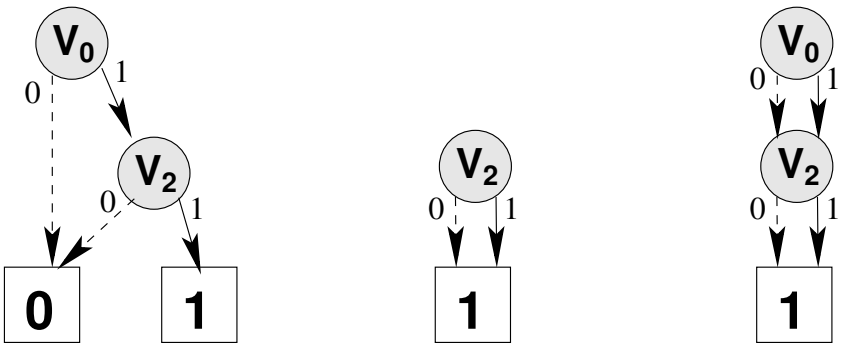

Figure 5.5: ZBDDs representing v_0v_2, $1 + v_2$, and $1 + v_0 + v_0v_2 + v_2$.

Example 11 *We show how the native algorithm computes* ***powUnion****(v_0v_2). Figure 5.5 contains the initial ZBDD representing v_0v_2 (left). To compute* ***powUnion*** *for the original ZBDD, we first recursively compute* ***powUnion*** *for the node labeled v_2. When* ***powUnion*** *is applied to the node labeled v_2, which represents the set v_2, we have $R_0 = P_0 = 0$ and $R_1 = P_1 = 1$. The result is a node labeled v_2 with zero successor $R_0 + R_1 = 1$ and one successor $1 + R_1 = 1 + 1 = 1$, shown in the center of the figure. This ZBDD represents the powerset of v_2, namely $1 + v_2$. We will call this ZBDD N. When we compute* ***powUnion*** *of the original ZBDD, $R_0 = P_0 = 0$, and $R_1 = N$. This step generates a node with value v_0, zero successor $R_0 + R_1 = 0 + N = N$, and one successor $1 + R_1 = 1 + N = N$. Because both nodes are identical (*reduction rule *applied within Getnode), we can delete one of them and change both edges of v_0 to lead to just one N, as shown in the right ZBDD in Figure 5.5. The resulting graph represents $1 + v_0 + v_0v_2 + v_2$.*

The command semantics (Figure 5.3) is described in terms of the operators listed before. We only add a new predicate, used when checking if two variables might be aliases: $\mathbf{mayAlias}(SH, v, w) = (\overline{\mathbf{mustAlias}}(SH, v, w)$ and $SH/(v * w) \neq 0)$. The following example shows how the field store from Example 3 would be calculated using ZBDDs.

Example 12 *Assume we start evaluating* v_0`.f=` *expr in an abstract set sharing* $SH_1 = v_0v_1+v_0v_2+res$. *Because all the sharings in* SH_1 *contain* v_0 *or* res, $SH_2 = 0$. *The union of the powersets of* $SH_1//v_0 = v_0v_1 + v_0v_2$ *is calculated in a very similar fashion to the last example, and results in a set sharing* $1 + v_0 + v_0v_1 + v_0v_2 + v_1 + v_2$. *Therefore,* $SH_3 = \textbf{\textit{projOut}}(v_0 + v_0v_1 + v_0v_2 + v_1 + v_2, res) = v_0 + v_0v_1 + v_0v_2 + v_1 + v_2$. *The last component of the result is* $SH_4 = (SH_1/res) * (SH_3//v_0) = 1 * (SH_3//v_0) = v_0+v_0v_1+v_0v_2$. *The result is* $SH' = 0+SH_3+SH_4 = SH_3 = v_0+v_0v_1+v_0v_2+v_1+v_2$, *which is the same result obtained in the set example.*

5.3 Experiments

To evaluate the efficiency (in terms of memory usage and running time) of the ZBDD approach, we compared it to an alternative representation for set sharings based on sets of bitsets. Bitsets are a fast, light representation compared to other ways of representing a set sharing. In a bitset, each bit b_i indicates if the variable v_i is in the sharing ($b_i = 1$) or not ($b_i = 0$). Our first implementation used the Java library where a `BitSet` is an array of double words. However, our first experiments showed that this approach does not scale beyond set sharings with more than a few thousand elements. For this reason, we replaced the library implementation by a lightweight version, which only requires a single word to represent each sharing. This effectively limits the number of variables to be not more than 32 for the bitset approach, which is reasonable when confronted with powerset operations. In all the experiments we assume that the number of variables n is bounded by 32, but note that the ZBDD implementation performs well for larger set sharings, and could handle bigger values of n. Our ZBDD implementation of set sharing is based on the JDD library [Vah08]. In our experiments we assumed a fixed variable ordering: the variables are numbered, and the smaller the number, the closer to the root they are.

Several characteristics of set sharings influence the memory usage and the performance of the data structure representing them. Although the number of variables n seems to be important, our two representations are independent of this parameter. In the case of the bitsets, because we use 32 bits to store every sharing, independently of the number of variables. In the case of ZBDDs, only the statistical distribution of the sharings (i.e., their sparsity) influences the number of nodes required to represent the information, and therefore the memory usage and performance of the ZBDD. For the same reason, the behavior of the two data structures is independent of the *sharing density* of SH, i.e., the proportion of the number of sharings over the maximum possible: $SH_d = |SH|/2^n$.

The most decisive factor is the number of sharings $|SH|$. Because we allocate a new bitset every time a new sharing is added, the performance of the set of bitsets approach is inversely proportional to $|SH|$. In the case of ZBDDs we also have to

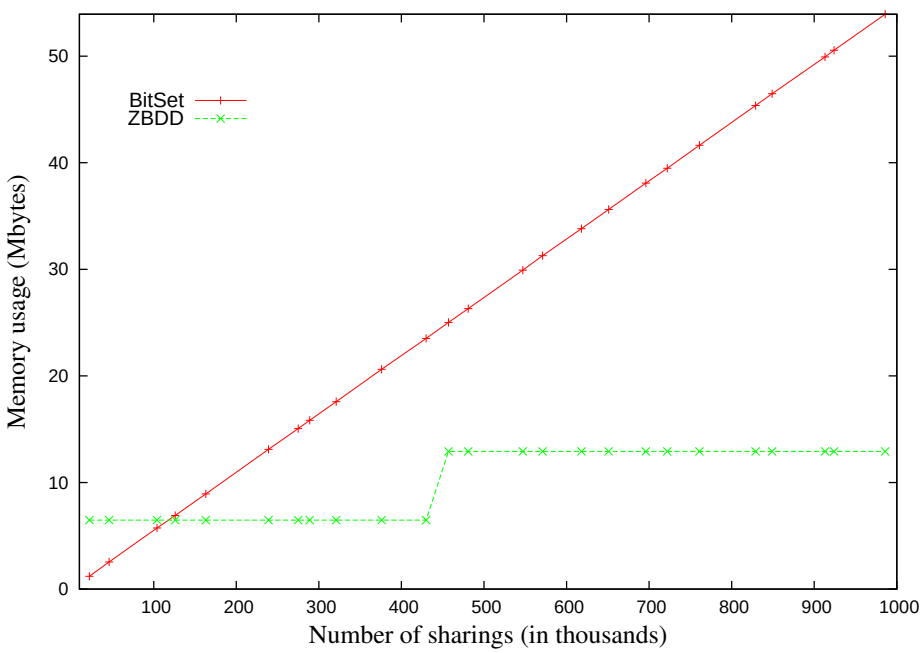

Figure 5.6: Memory usage experiments. Over 25 runs.

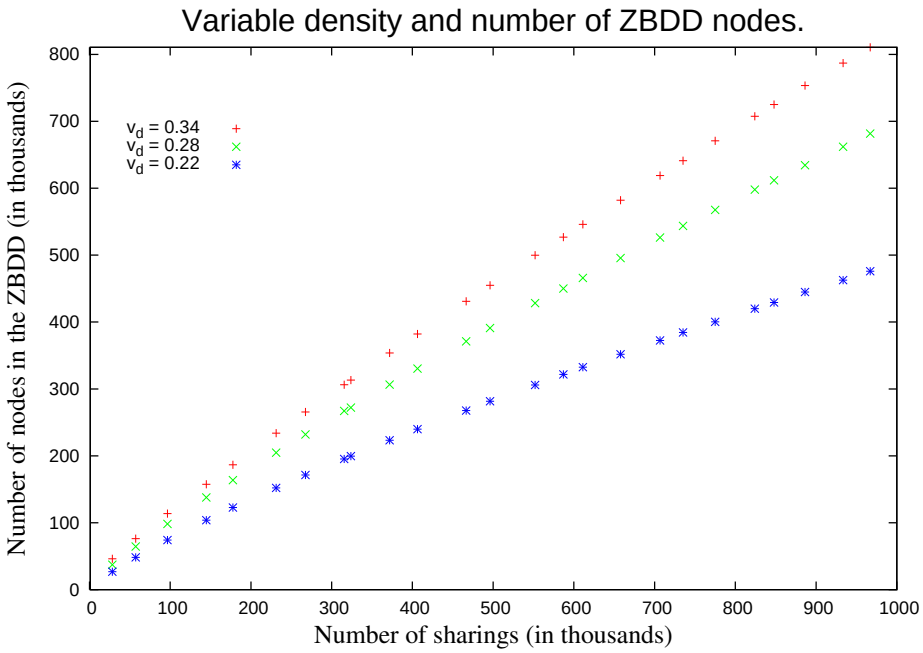

Figure 5.7: Relationship between variable density and number of ZBDD nodes. Over 25 runs.

take into account the *variable density*. This metric is the average number of variables per sharing: $v_d = \frac{1}{n*|SH|} * \sum_{S \in SH} |S|$. A small variable density is synonymous with a sparse set sharing, and therefore we can expect the ZBDD to perform inversely proportional to the metric. We now examine how the number of sharings and the variable density relate to memory consumption and execution times in our experiments.

Memory Usage: We generated random set sharings and measured the space requirements for the Java objects backing the set of bitsets and ZBDD as reported by a profiler [JPr]. The different memory usages are shown on the left of Figure 5.6. The plot shows that the ZBDD performs better than the bitset solution. The differences are more significant (a factor of 5) for large values of $|SH|$. A set of bitsets uses 56 bytes per sharing, less than the 80 required by a set of the JDK 1.5 `BitSet` class.

At one million sharings, the set of bitsets requires more than 56Mb, while the same information occupies 12Mb in the ZBDD version ($v_d = 0.28$). The staircase behavior of the ZBDD memory usage function is due to the *capacity* of the array storing the node list (ZBDDs are represented as arrays in JDD), which doubles when the load exceeds a certain threshold.

In the leftmost graph in Figure 5.6 we did not take into account the effect of variable density. The other plot in that figure demonstrates how ZBDDs benefit from sparse variable distributions. This time we do not show the number of Kbytes in the y-axis, but rather the number of nodes in the binary decision diagram. As expected, sparse sharings require fewer nodes than those that are more dense in terms of v_d. In the experiments, the number of nodes goes down by an average 38.2% from $v_d = 0.34$ to $v_d = 0.22$.

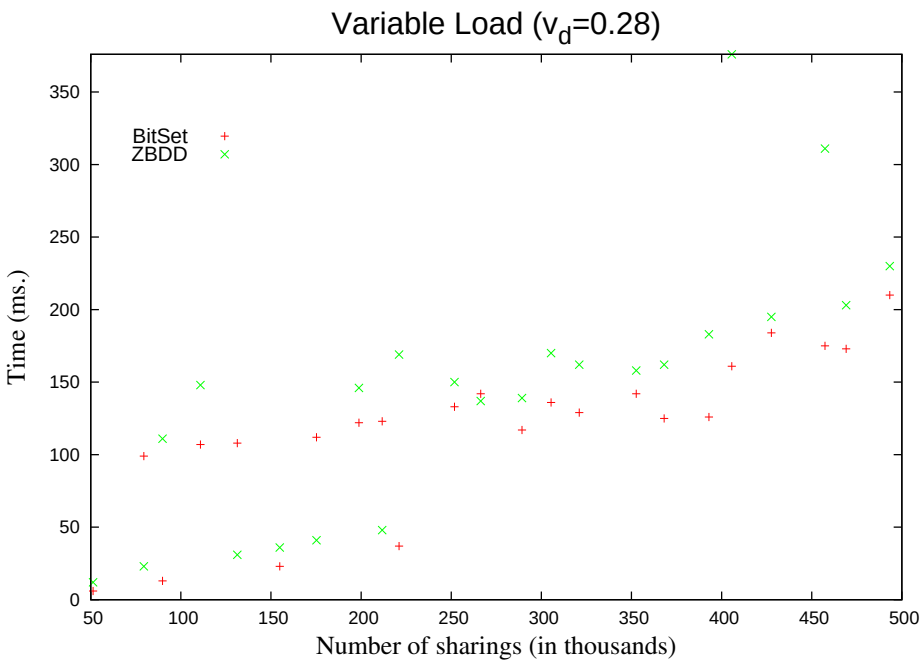

Figure 5.8: Performance of a set of bitsets vs ZBDD (variable load). Over 25 runs..

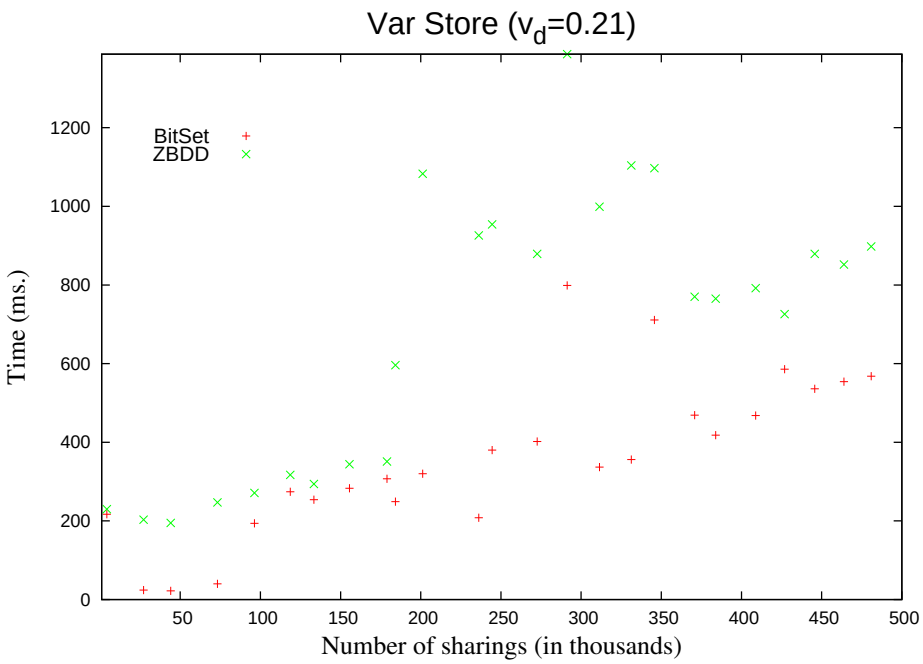

Figure 5.9: Performance of a set of bitsets vs ZBDD (variable store). Over 25 runs..

Speed: We measured the number of milliseconds required to compute the semantics of the most significant operations (variable load/store, and field load/store), given a random initial set sharing. We disabled the JDD cache for the experiments. All the measurements were done on a Pentium M 1.73Ghz with 1Gb of RAM. The virtual machine was Sun's JVM 1.5.0 running on Ubuntu 6.06. The results are in Figs. 5.8- and 5.11.

The time required to simulate a variable load presents a similar, linear behavior in both cases; the bitset version is 14.6% faster in the average. Although not reflected in Figure 5.10, the native operation **setResEqto** takes half the time of the equivalent composition of ZBDD operations (see Section 5.2). For the variable store, both running times are roughly linear in the number of sharings. However, the lack of a native ZBDD implementation results in running times noticeably slower than those of the set of bitsets. It remains an open question whether a dedicated ZBDD algorithm

can be devised for this command.

The powerset operation is a major obstacle for a feasible implementation of set sharing using the sets of bitsets. Both the field load and field store transfer functions depend on this operation. While the ZBDD **powUnion** algorithm requires reasonable times for calculating the union of many powersets, the bitset implementation presents exponential growth with respect to the number of sharings. For example, it needs half a minute to compute the output state for a field load in which the initial sharing has 5,000 elements. The ZBDD implementation finishes the same operation in less than 600ms. The field store (Figure 5.11), which is a more complex operation, presents a similar pattern, although the running times are always significantly larger than for the field load.

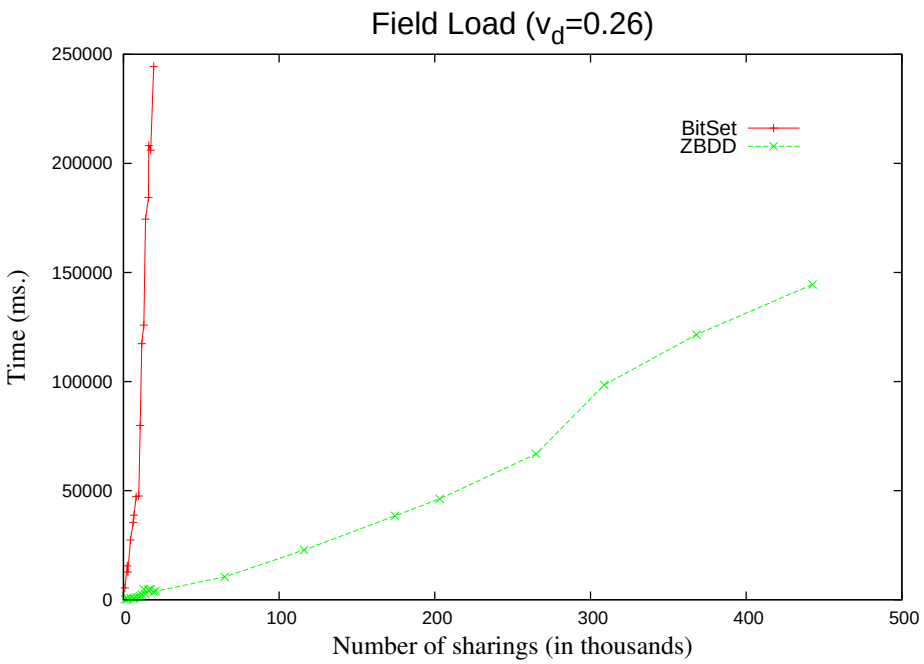

Figure 5.10: Performance of a set of bitsets vs ZBDD (field load). Over 25 runs..

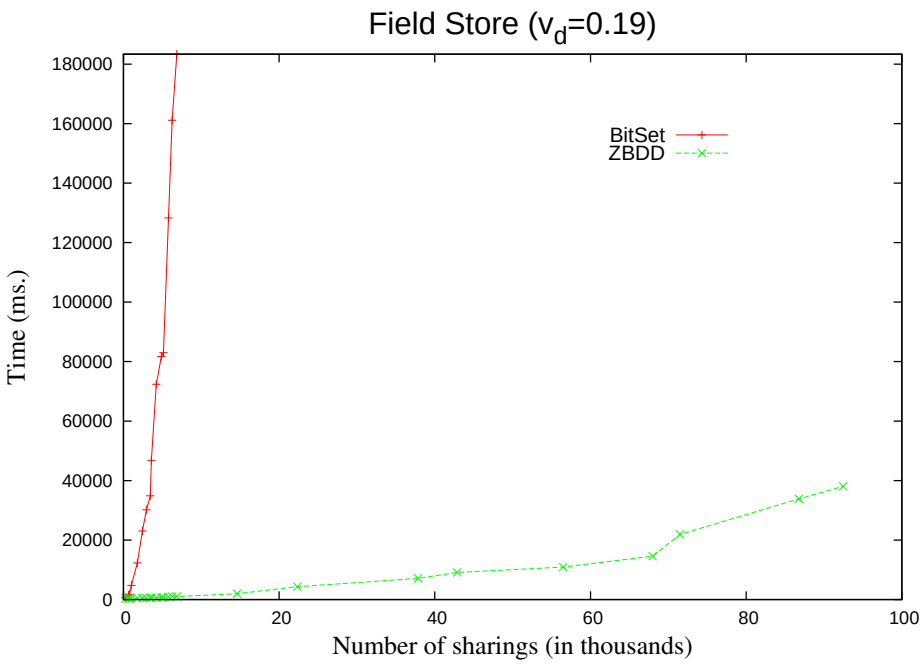

Figure 5.11: Performance of a set of bitsets vs ZBDD (field store). Over 25 runs..

5.4 Related Work

There has been extensive work in recent years on the use of BDDs [Zhu02, BLQ+03, WL04, ZC04] to represent (abstract) *points-to* information. In these abstractions, information is stored in the form of (v, a) pairs, where each such pair indicates that v may point to the allocation site a. As mentioned before, set sharing information can be interpreted as an *abstraction* of points-to information where instead of representing which exact objects can be pointed to by a variable, the domain captures only which sets of variables may point transitively to the same object. Thus, our analysis works at a different level since the set sharing encoding can result in some loss of precision, but offers the advantage of more compact representation.

ZBDDs were introduced by Minato [iM93] and applied to a great diversity of problems in model checking (e.g., [YHTiM96, Cou97, iM01]). More recently, Lhoták

et al. have applied ZBDDs to the exploration of infinite state spaces [LSJ07] in the context of points-to analysis. The main differences between this work and [LSJ07] are one hand the abstraction used (set sharing vs. points-to pairs) and on the other that in the approach proposed the domain does not require relational information, i.e., we can use existing ZBDD libraries [Som05, Vah08] directly in our implementation.

To the extent of our knowledge, this is the first work that relates set sharing analysis with ZBDDs or presents implementation results for the set-sharing domain using any type of binary decision diagram. In the logic programming realm, there has been a significant amount of work related to set sharing-based analysis for the automatic parallelization of Prolog programs (e.g., [JL89, MH89, MH91, BdlBH99]). However, the abstract operations show significant differences with the ones required for an imperative/OO language. Furthermore, to the best of our knowledge, all existing implementations use lists of lists to represent set sharings. In [CSS99] a connection between the set sharing domain and standard BDDs is suggested, but no implementation or experimental results are provided and there is no mention of ZBDDs. More recent work [PS07] for Java presents results for a BDD-based implementation of the less precise pair sharing domain. Because in this case the abstraction is a set of pairs (and not a set of sets), the representation used is quite different from ours.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Static analysis frameworks usually comprise many interrelated phases; the final complexity, generality, and accuracy of the analyzer depends on all of them. Our objective has been to show that we can achieve a high degree of abstraction when facing the problems of how to represent the source to be analyzed, how to efficiently compute a safe upper approximation of the semantics of that program, and how to determine certain sharing characteristics that might allow later compiler optimizations such as scheduling, load/store elimination, etc.

The first part of this thesis (Chapters 2 and 3) has dealt with two common analysis problems: the intermediate representation and the fixpoint engine. In Chapter 2, we showed how a Horn clause-based language is a suitable and convenient IR that permits an unique framework to analyze a variety of languages. In particular, programs written in Java and Prolog are accepted as input, even though the particular abstract domain in use does change depending on the source language.

In Chapter 3, we provided a description of the algorithm that makes the analysis possible, since in most of the existing abstract interpretation literature this component is taken for granted, even when it is responsible for an important percentage of the final efficiency and accuracy of any analysis that sits on top of it. Memoization, abstract state summaries, a top-down approach, and assertions are the fundamental characteristics of the described fixpoint engine.

The second part of this work (Chapters 4 and 5) has focused on the design and implementation of an abstract domain that can accurately track certain properties of the program. We have chosen the sharing property (i.e., whether two or more variables definitely do not point to the same memory location), as it represents a summary of the points-to information of the program, and opens the possibility for many compiler optimizations. We selected the *set* sharing abstraction, as it is able to express more complicated predicates about the program variables than the existing *pair* sharing alternative. Our initial work showed how that theoretical, intuitive precision gain also translates into practical gains in our set of experiments (%7 improvement).

In many occasions, the set sharing domain has been considered inefficient, given its intrinsic combinatorial nature. In fact, there are quite efficient implementations of the pair sharing abstraction using Binary Decision Diagrams (BDDs). Our challenge was to come up with the adecuate data structure to manipulate sets of sets of variables; the answer found is Zero-supressed Binary Decision Diagrams (ZBDDs). This is the first time (to the extent of our knowledge) that a compact representation of sharing sets is made possible, either in the logic programming or the object-oriented realms. Chapter 5 describes the abstract operations of Chapter 4 in terms of ZBDD operators.

Although not included in this thesis, we also worked in the development of analyses which extend the classic concept of resource consumption (which generally refers

to execution steps, time, memory), in order to cover more application-dependent notions of resources such as bytes sent or received by an application, files left open, etc. We created a fully automated analysis [NMLH08] for inferring upper bounds on the usage that a Java bytecode program makes of a set of *application programmer-definable* resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the program makes of that resource for any given set of input data sizes. The analysis proposed is independent of the particular resource. In [NMLH08] we also present experimental results covering an ample set of interesting resources.

6.2 Future Work

Set sharing is considered to be a *storeless* abstraction of the heap, i.e., we only abstract information relative to the local variables themselves. More elaborate *store* models consider also tracking information about the program that takes into account the shape of memory at a program point (i.e., their internal representation is closer to the actual graph structure of the heap). Although there are many interesting properties that are tracked in this type of *shape analyses* (such as, e.g., those of Marron et al. [MSHK07] or Sagiv et al. [LAS00]), we have focused our attention on the sharing attributes of the program being analyzed. The ultimate goal is to show how *deep* sharing is abstracted by the domain, thus allowing parallelization of loops that manipulate lists and arrays, if no sharing occurs.

Another interesting problem is the empirical study of the performance gains obtained by using the described ZBDD-based set sharing implementation. Our experiments in Chapter 5 just showed memory and performance gains in comparison with

a BitSet alternative for individual abstract operations; it remains to determinate what the real gain in performance is in the actual set sharing analysis. For that reason, we plan to compare the two versions within two contexts: a) the set sharing analysis presented in Section 4; b) the logic programming analysis with the same name (and abstract representation), using the CiaoPP analyzer, in which previous implementations using lists of lists or the clique domain [NBH06] are available for comparison.

The ultimate purpose of the framework presented in this dissertation is the use of the analysis information provided by the particular abstract domain for optimization, debugging or verification purposes. Additional work needs to be completed in order to assess the usefulness of the set sharing domain. In particular, we need to choose a client application and show how the information inferred by analysis helps in improving the results of that particular application. For instance, we would like to show that the sharing information can be used in the automatic parallelization of interesting applications, and that their parallel versions run faster.

Bibliography

[AAG+07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[AF99] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.

[Age95] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[AGZHP07] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, number 4354 in LNCS. Springer-Verlag, 2007.

[BCC+04] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM), 2004. Available at `http://www.ciaohome.org`.

[BCC+06] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[BCCH94] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC*, pages 234–250, 1994.

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study

in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[Bla99] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.

[BLQ+03] Marc Berndl, Ondřej 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*, pages 103–114. ACM Press, 2003.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[CC92] 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.

[CFR+97] Agostino Cortesi, Gilberto File, Francesco Ranzato, Roberto Giacobazzi, and Catuscia Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, number 3385 in LNCS, pages 147–163. Springer, 2005.

[CMB+95] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

[Cou97] O. Coudert. Solving Graph Optimization Problems with ZBDDs, 1997.

[CSS99] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*, pages 77–101, 1995.

[Die87] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

[DL05] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[EGH94] 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.

[Fec96] Christian Fecht. An efficient and precise sharing domain for logic programs. In Herbert Kuchen and S. Doaitse Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 469–470. Springer, 1996.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[GS05] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of VMCAI*, LNCS. Springer-Verlag, 2005.

[HBC+08] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In Jose Meseguer Pierpaolo Degano, Rocco De Nicola, editor, *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

[HBCC99] Michael Hind, Michael G. Burke, Paul R. Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[HC94] M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.

[HPBLG03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.

[HPS06] Patricia M. Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[iM93] Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993.

[iM96] Shin ichi Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer, Norwell/MA, USA, 1996.

[iM01] Shin ichi Minato. Zero-suppressed BDDs and their Applications. *STTT*, 3(2):156–170, 2001.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[jol] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[JPr] JProfiler. `http://www.ej-technologies.com/products/jprofiler/`.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS*, number 1824 in LNCS, pages 280–301. Springer, 2000.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[LC05] F. Logozzo and A. Cortesi. Abstract interpretation and object-oriented languages: quo vadis? In *Proc. of the 1st. Int'l. Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL'05)*, ENTCS. Elsevier Science, January 2005.

[Ler01] Xavier Leroy. Java bytecode verification: An overview. In *CAV'01*, number 2102 in LNCS, pages 265–285. Springer, 2001.

[Ler03] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[Log07] Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*, number 4349 in LNCS. Springer, Jan 2007.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI*, pages 473–489. ACM, 1992.

[LSJ07] O. Lhoták, S.Curial, and J.N.Amaral. Using ZBDDs in Points-to Analysis. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, 2007.

[LV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

[MLH08] M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[MLLH08] M. Méndez-Lojo, O. Lhoták, and M. Hermenegildo. Efficient Set Sharing using ZBDDs. In *21st International Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008. To appear.

[MLNH07a] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

[MLNH07b] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *ISSTA*, pages 1–11, 2002.

[MSHK07] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap Analysis in the Presence of Collection Libraries. In *ACM WS on Program Analysis for Software Tools and Engineering (PASTE'07)*. ACM, June 2007.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[Mut91] Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs*. PhD thesis, University of Texas at Austin, August 1991.

[NBH06] J. Navas, F. Bueno, and M. Hermenegildo. Efficient top-down set-sharing analysis using cliques. In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[NMLH07] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[NMLH08] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. Technical Report UNM TR-CS-2008-02 - CLIP1/2008.0, University of New Mexico, Department of Computer Science, UNM, January 2008. Submitted for publication.

[PGS98] J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[PJC06] J. Peralta and J.Cruz-Carlon. From static single-assignment form to definite programs and back. Extended abstract in International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR), July 2006.

[PLC01] Isabelle Pollet, Baudouin Le Charlier, and Agostino Cortesi. Distinctness and sharing domains for static analysis of java programs. In *ECOOP '01: 15th European Conference on Object-Oriented Programming*, London, UK, 2001.

[Pol04] Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java*. PhD thesis, Catholic University of Louvain, 2004. Dept. of Computer Science.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.

[PS07] É Payet and F. Spoto. Magic-Sets Transformation for the Analysis of Java Bytecode. In H. R. Nielson and G. Filé, editors, *Proceedings of the 14th International Static Analysis Symposium (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 452–467, Kongens Lyngby, Denmark, August 2007. Springer.

[Ram91] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *The Journal of Logic Programming*, 11(3 & 4):189–216, October/November 1991.

[RS01] Noam Rinetzky and Shmuel Sagiv. Interprocedural shape analysis for recursive programs. In *CC*, 2001.

[Ruf00] Erik Ruf. Effective synchronization removal for java. *PLDI'00, SIGPLAN Notices*, 35(5):208–218, 2000.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[Som05] F. Somenzi. *CUDD: CU Decision Diagram Package*, 2005. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[Søn86] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[Spo05] F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[SS00] M. Streckenbach and G. Snelting. Points-to for java: A general framework and an empirical comparison. Technical report, University Passau, November 2000.

[SS05] S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *Static Analysis Symposium (SAS)*, pages 320–335, 2005.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Vah08] A. Vahidi. *JDD: A Pure Java BDD Library*, 2008. `http://javaddlib.sourceforge.net/jdd/index.html`.

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

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, pages 1–12, 1995.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

[YHTiM96] Tomohiro Yoneda, Hideyuki Hatori, Atsushi Takahara, and Shin ichi Minato. BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of petri nets. In *FMCAD*, pages 435–449, 1996.

[ZBH99] E. Zaffanella, R. Bagnara, and P. M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–432. Springer-Verlag, Berlin, 1999.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic Pointer Analysis Revisited. In *PLDI*, pages 145–157, 2004.

[Zhu02] Jianwen Zhu. Symbolic Pointer Analysis. In *ICCAD*, pages 150–157, 2002.