An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode

Mario Méndez

Department of Computer Science
University of New Mexico
Albuquerque, NM, USA

Jorge Navas

Department of Computer Science
University of New Mexico
Albuquerque, NM, USA

Manuel V. Hermenegildo

Departments of Computer Science
University of New Mexico, Albuquerque, NM (USA)
and Technical University of Madrid, Madrid (Spain).

Abstract

Abstract interpretation has been widely used for the analysis of object-oriented languages and, in particular, Java source and bytecode. However, while most existing work deals with the problem of finding expressive abstract domains that track accurately the characteristics of a particular concrete property, the underlying fixpoint algorithms have received comparatively less attention. In fact, many existing (abstract interpretation based–) fixpoint algorithms rely on relatively inefficient techniques for solving inter-procedural call graphs or are specific and tied to particular analyses. We also argue that the design of an efficient fixpoint algorithm is pivotal to supporting the analysis of large programs. In this paper we introduce a novel algorithm for analysis of Java bytecode which includes a number of optimizations in order to reduce the number of iterations. The algorithm is *parametric* -in the sense that it is independent of the abstract domain used and it can be applied to different domains as "plug-ins"-, multivariant, and flow-sensitive. Also, is based on a program transformation, prior to the analysis, that results in a highly uniform representation of all the features in the language and therefore simplifies analysis. Detailed descriptions of decompilation solutions are given and discussed with an example. We also provide some performance data from a preliminary implementation of the analysis.

Email: mario@cs.unm.edu
Email: jorge@cs.unm.edu
Email: herme@unm.edu

1 Introduction

Analysis of the Java language (either in its source version or its compiled bytecode [19]) using the framework of abstract interpretation [8] has been the subject of significant research in the last decade (see, e.g., [20] and its references). 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 (e.g., [3,31]) or statically verify certain properties about the run-time behavior of the code (e.g., [13,17]). In contrast to this concentration and progress on the development of new, refined domains there has been comparatively little work in the underlying fixpoint algorithms and frameworks. In fact, many existing abstract interpretation-based analyses use relatively inefficient fixpoint algorithms. In other cases, the fixpoint algorithms are specific and/or tied to particular analyses and cannot easily be reused for other domains.

Instead, interesting progress on fixpoint algorithms has been made for example in functional and logic programming, where a number of solutions have been proposed to speed up analysis fixpoint convergence (see, e.g., [24,6,14,29] and its references). However, the formulation of these algorithms is strongly tied to the operational semantics of those languages. As a result, their adaptation to Java and Java bytecode is not straightforward, since fundamental aspects of the semantics of object-oriented programming such as virtual calls, object instantiation, static methods and variables, destructive update, etc. are not dealt with, at least directly.

We argue that the design of an efficient fixpoint algorithm is pivotal to supporting the analysis of large programs. In this paper we propose and describe in detail a novel algorithm for analysis of Java bytecode which includes a number of optimizations in order to reduce the number of iterations as well as other unique characteristics. In particular, dependencies are kept 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 *memoing*. The proposed algorithm is *parametric* in the sense that it is independent of the abstract domain used and it can be applied to different domains. The algorithm specifies a reduced number of basic operations that each domain must implement. This allows having a single implementation to which the designer of new analyses can add new domains as "plug-ins." The algorithm is also *multivariant*: abstract calls to a given method that represent different input patterns are automatically analyzed separately. This is both more precise and efficient than alternative techniques such as cloning methods for each call site, since cloning can produce either too many versions of methods (if two call sites are determined to use the same input pattern) or too few (if two different, separate input patterns arise from a single call site). The algorithm is also top-down/flow-sensitive, in order to allow modeling properties that depend on the data flow characteristics of the program.

Finally, another interesting characteristic of the algorithm is that it is preceded by a program transformation, prior to the analysis, that results in a highly uniform representation of all the features in the language and therefore simplifies analysis. This program transformation includes a certain level of decompilation of the

bytecode which recovers part of the original code structure lost in the bytecode representation. Our decompilation process is based in part on existing tools [23,35] to which we add a number of steps (normalizing the intermediate representation which is actually analyzed, representing different classes of statements in a unified way, automatically introducing relational information between initial and final states on methods calls, etc.) which we argue greatly simplify the burden of designing new analyses and abstract operations. While not the subject of this paper, the algorithm can also be applied to Java source code, applying a similar transformation.

Java programs rely heavily on libraries and analysis thus usually expands to many imported classes. Thus, modular analysis is definitely an important issue in this context. However, and in order to concentrate on the description of the fixpoint algorithm, we will not deal with modular analysis issues in this paper. Instead, we assume that methods exported by libraries are annotated in an assertion language that describes which output abstract states are provided for certain input abstract states (we use a particular assertion language based on [28] but adapted to resemble the Java Modeling Language [16], however we omit also a detailed description of this assertion language from the description for brevity). A solution for modular analysis in the context of Java can be found for example in [27], and, more specifically relevant to our algorithm, in [4,7].

Regarding other related work, as mentioned before, most published analyses based on abstract interpretation for Java or Java bytecode do not provide much detail regarding the implementation of the fixpoint algorithm. Also, most of the published research (e.g., [3,5]) focuses on particular properties and therefore their solutions (abstract domains) are tied to them, even when they are explicitly multipurpose [18]. In [25] the authors mention a choice of several univariant and multivariant computations, but no further information is given. The more recent and quite interesting Julia framework [33] is intended to be generic and targets bytecode as in our case. Their fixpoint techniques are based on prioritizing analysis of non-recursive components over those requiring fixpoint computations and using abstract compilation [15]. However, few implementation details are provided. Also, this is a *bottom-up* framework, while our objective is to develop a top-down, multivariant framework. While it is well-known that bottom-up analysis can be adapted to perform top-down analyses by subjecting the program to a "magic-sets"-style transformation [30], the resulting analyzers typically lack some of the characteristics that are the objective of our proposal, and, specially, multivariance. Finally, in [21] a generic static analyzer for the modular analysis and verification of Java classes is presented. The algorithm presented is also *bottom-up*, and only a naive version of it (which is not efficient for mutually recursive call graphs) is presented.

2 Intermediate program representation

We start by describing the first phase of the analysis: the translation of the Java bytecode into an intermediate representation. In order to concentrate on the fixpoint algorithm, which is the main objective of the paper, this description is summarized, concentrating on the characteristics of the transformation and illustrating it with a relatively complete example (the full description can be found in [22]). The

$$
\begin{array}{lll}
prog & ::= & (\{meta_1, \ldots, meta_m\}, \{or-tp_1, \ldots, or-tp_n\}) \\
meta & ::= & \texttt{subclass}(k_1, k_2) \mid \texttt{implements}(k, \{par_1, \ldots, par_n\}) \\
or-tp & ::= & \texttt{method}(name, \{par_1, \ldots, par_n\}, k, attr, \texttt{body}(stmt)) \\
par & ::= & (name, t) \\
stmt & ::= & assignStmt \mid invokeStmt \\
 & & condStmt \mid returnStmt \\
 & & nopStmt \mid \{stmt_1, \ldots, stmt_n\} \\
assignStmt & ::= & \texttt{assign}(var, rvalue) \mid \texttt{assign}(var, var, field) \\
invokeStmt & ::= & \texttt{invoke}(name, \{par_1, \ldots, par_n\}, k) \\
condStmt & ::= & \texttt{guard}(imm_1\ condop\ imm_2) \\
condop & ::= & \langle \mid \rangle \mid = \mid != \mid \leq \mid \geq \\
returnStmt & ::= & \texttt{return} \\
nopSmt & ::= & \texttt{nop} \\
imm & ::= & var \mid constant \\
rvalue & ::= & concreteRef \mid imm \mid expr \\
concreteRef & ::= & field \mid var.field \\
expr & ::= & imm1\ binop\ imm2 \mid invokeExpr \mid \texttt{new}\ type \\
binop & ::= & + \mid - \mid = \mid \neq \mid \leq \mid \geq \mid * \ldots
\end{array}
$$

Fig. 1. Internal representation of the bytecode.

translation process produces a structured, decompiled representation of the Java bytecode and is based on the SOOT framework [35] which has been successfully used in previous analyses [9,2]. However, instead of analyzing directly the Jimple representation –based on `gotos`– it is processed further in order to build a control flow graph (CFG) in a similar way to the Dava tool [23]. The idea is also analogous to the approach of [13,33] but the graph obtained is somewhat different since we do not distinguish between stack and local variables, and all the operands are explicit in the expressions. The actual internal representation used is described by the grammar in Fig. 1. [5] In our current implementation we deal only with the fundamental features of the language such as inheritance, virtual calls, and method visibility.

Here and in the rest of the paper, we will denote by $\mathcal{V}$ the set of variables in the program and by $\mathcal{M}$ the set of method names. The types $\mathcal{T}$ of the application include classes $\mathcal{K}$ and atomic types. The decompilation process represents methods as *OR-tuples* $(name, fp, k_{callee}, body) \in \mathcal{M} \times \mathcal{P}(\mathcal{V} \times \mathcal{T}) \times \mathcal{K} \times \mathcal{P}(Stmt)$. The domain of OR-tuples is denoted by $\mathcal{O}$ and therefore a program P is just an element of $\mathcal{P}(\mathcal{O})$. A first key idea in the transformation is to have a single representation for all types of loops, as well as for conditional structures and standard methods, which are all transformed into OR-tuples. For example, an unconditional jump in the bytecode is first decompiled as a conditional block, which is further converted into a "*pseudo*" method. This label refers to the fact that those methods did not exist in the original bytecode. Given a statement `if` $cond_1$ $stmt_1$ `else if` $cond_2$ $stmt_2 \ldots$ `else` $stmt_n$ in the context of a class k, n OR-tuples are obtained of the form $\{(name\_if, \{(v_1, k_1), \ldots, (v_n, k_n)\}, k, [cond_1, stmt_1]), \ldots, (name\_if, \{(v_1, k_1), \ldots, (v_n, k_n)\}, k, [\overline{cond_1}, \ldots, \overline{cond_{n-1}}, cond_n, stmt_n])\}$. The tag $name\_if$ uniquely identifies the set of OR-tuples. The formal parameters (v_i, k_i) are the variables (and their classes) referenced inside the intermediate `if` block.

A second important aspect in the representation of the code is the *meta-information* stored about it. Although that information could be indirectly retrieved from intermediate data structures, a more convenient approach is to maintain a ta-

[5] This grammar has been simplified slightly for better understanding. An intuition of its complete form can be derived from Fig. 2.

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

class Vector{
    Element first;

    public void append(Vector v){
1       Element e = first;
2       if (e == null)
3           first = v.first;
4       else{
5           while (e.next != null)
6              e = e.next;
7           e.next = v.first;}
    }
    public void add(Element element){
        Element e = new Element();
        e.value = element.value;
        Vector v  = getNewVector();
        v.first = e;
        append(v);
    }
}
class ZipVector extends Vector{

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

(a)

```
class Element extends java.lang.Object{
    int value;
    Element next;
    [...]
}
class Vector extends java.lang.Object{
    Element first;
    public void append(Vector){
        Vector r0, r1;
        Element r2, $r3, $r4, $r5;

        r0 := @this: Vector;
        r1 := @parameter0: Vector;
1       r2 = r0.<Vector: Element first>;
2       if r2 != null goto label0;
3       $r3 = r1.<Vector: Element first>;
3       r0.<Vector: Element first> = $r3;
        goto label2;
     label0:
5       $r4 = r2.<Element: Element next>;
5       if $r4 == null goto label1;
6       r2 = r2.<Element: Element next>;
        goto label0;
     label1:
7       $r5 = r1.<Vector: Element first>;
7       r2.<Element: Element next>= $r5;
     label2:
        return;
    }
    [...]

public class ZipVector extends Vector
    [...]
```

(b)

```
subclass('user:vector',java.lang.object,[]).
subclass('user:zipvector','user:vector',[]).
subclass('user:element',java.lang.object,[]).
implements('user:vector','add',['user:vector','user:element','void']).
implements('user:zipvector','add',[user:zipvector','user:element','void']).

[...]

method('user:vector:append','user:vector','void',recursive(not),
       formal([(R0,'user:vector'),(R1,'user:vector')]),
       local([(R2,'user:element'),(R3,'user:element'),
              (R4,'user:element'),(R5,'user:element')]),
  body([
1      staticinvoke('check_not_null',[(R0,'user:vector')],java.lang.object),
1      assign(R2,R0,first,'user:element'),
       conditionalinvoke('user:vector:append_if00',
              [(R0,'user:vector'),(R1,'user:vector'),(R2,'user:element'),
               (R3,'user:element'),(R4,'user:element'),(R5,'user:element')])
  ])).


method('user:vector:append_if00','user:vector:append','user:vector','void',
       formal([(R0,'user:vector'),(R1,'user:vector'),(R2,'user:element'),
               (R3,'user:element'),(R4,'user:element'),(R5,'user:element')]),
  body([
2       guard(R2==null),
3       staticinvoke('check_not_null',[(R1,'user:vector')],java.lang.object),
3       assign(R3,R1,first,'user:element'),
3       staticinvoke('check_not_null',[(R0,'user:vector')],java.lang.object),
3       setfield(R0,first,R3,'user:element'),
        return('user:vector:append')
  ])).

method('user:vector:append_if00','user:vector:append','user:vector','void',
       formal([(R0,'user:vector'),(R1,'user:vector'),(R2,'user:element'),
               (R3,'user:element'),(R4,'user:element'),(R5,'user:element')]),
  body([
4       guard(not(R2==null)),
5       assign(R4,R2,next,'user:element'),
        loopinvoke('user:vector:append_if00_while00',[(R1,'user:vector'),
                 (R2,'user:element'),(R4,'user:element'),(R5,'user:element')])
  ])).


method('user:vector:append_if00_while00,','user:vector:append','user:vector','void',
       formal([(R1,'user:vector'),(R2,'user:element'),(R4,'user:element'),
               (R5,'user:element')]),
  body([
5       guard([R4==null]),
6       staticinvoke('check_not_null',[(R1,'user:vector')],java.lang.object),
6       assign(R5,R1,first,'user:element'),
6       staticinvoke('check_not_null',[(R1,'user:element')],java.lang.object),
6       setfield(R2,next,R5,'user:element')
  ])).


method('user:vector:append_if00_while00','user:vector:append','user:vector','void',
       formal([(R1,'user:vector'),(R2,'user:element'),(R4,'user:element'),
               (R5,'user:element')]),
  body([
5       guard(not([R4==null])),
7       staticinvoke('check_not_null',[(R2,'user:element')],java.lang.object),
7       assign(R2,R2,next,'user:element'),
7       staticinvoke('check_not_null',[(R2,'user:element')],java.lang.object),
7       assign(R4,R2,next,'user:element'),
        loopinvoke('user:vector:append_if00_while00',[(R1,'user:vector'),
                 (R2,'user:element'),(R4,'user:element'),(R5,'user:element')])
  ]) ).
```

(c)

Fig. 2. Vector example

ble containing which classes implement which methods, as well as the hierarchy, interface relations, etc. In this way, we can easily determine (for example) the set of classes in which a virtual call might take place without having to resort every time to an abstract syntax tree transversal.

A third key idea is to expose the internal structure of the more complex bytecode instructions. Java bytecodes are sometimes high-level instructions that encode relatively complex operations. Instead of delegating the treatment of such complexities to the abstract domain, we make these aspects of the operational semantics explicit in the intermediate representation itself using program transformations as in [13]. In the same way, a pivotal aspect in languages with destructive updates is the stor-

age of relational information about the formal parameters in a method invocation, so that on method exit we can distinguish whether the parameter state should be propagated back to the caller or it refers to a new, fresh instance. In [25,32] the solution is based on the framework by altering call semantics. Instead we introduce explicit assignments to temporal variables which are undone at the end of the method's body. We argue that the solutions that we apply result in simple domain implementations (important for our parametric approach), as well as increased portability of the domains: analysis of similar languages (e.g., C#) can (almost) reuse existing abstractions, provided that the compilation phase decompiles in this way the language-dependent features. We also argue that the representation proposed greatly facilitates later analyses.

Example 2.1 Figure 2 shows three representations of the same code, an alternative implementation of the JDK `Vector` class. We include the original source in Fig. 2a for better understanding of the example. Figure 2b is the output of the SOOT (de-)compiler, in Jimple format, for the `Vector` bytecode. Stack and local elements have been converted into named variables and all the expressions are typed, but the presence of *gotos* complicates later analyses. Meta-information about class hierarchies, overwritten methods, etc. is also implicit in the code.

The data structure that represents the Control Flow Graph that is the input to our fixpoint algorithm is shown in Fig. 2c. The meta-information part (first five lines) states that `ZipVector` is a direct descendant of the user-defined `Vector` class. Both implement an `add` method that receives an `Element` object and returns nothing. We now focus on the `append` method. Most of the statements in the Jimple representation are kept in a very similar format (the line numbers will help the reader identify the correspondences) except for `goto`s and `if`s which are now OR-tuples. For example, the `if` block starting at line 2 corresponds to the two OR-tuples named `user:vector:append_if00`, which have as formal parameters all the variables of the container method because they are referenced in their bodies. The `while` loop in lines 5-6 is constructed in a similar way, although recursive calls are inserted by the compiler. Space limitations prevent us from showing how the relational information is copied at the beginning and end of every method.

3 Top-down Approach to Bytecode Analysis

The program transformations of Sect. 2 greatly simplify our bytecode analysis since we only have two possible flows in the CFG: the branching invocations of OR-tuples or serial execution of all other statements. For the first case we will not distinguish in analysis between real (existing in the source) and pseudo (generated via program transformation) methods, which are semantically equivalent. In the event of an invocation $i = invoke(mname, ap, k\_caller) \in \mathcal{M} \times \mathcal{P}(\mathcal{V} \times \mathcal{T}) \times \mathcal{K}$ the semantics of both is computed by calculating the least upper bound of the semantics of all possible OR tuples compatible with such invocation: $\mathcal{SS}[\![invoke(mname, ap, k_{caller})]\!]\sigma = \sqcup(\mathcal{SS}[\![stmt_i]\!]\sigma)$ if $(name, fp, k_{callee}, stmt_i) \in \mathcal{O}$ and $comp(i, o)$. The function *comp* returns a boolean value indicating if a particular implementation $o = (name, fp, k_{callee}, stmt_i)$ is compatible with the invocation: i.e., if their names are identical and their signatures and the class where they are

defined are compatible according to a partial order for Java classes $\leq_{\mathcal{T}}$ like the one described in [17].

$$comp(i,o) = \begin{cases} true \quad \text{if } name = mname \text{ and } k_{caller} \leq_{\mathcal{T}} k_{callee} \text{ and} \\ \qquad\qquad |ap| = |fp| \text{ and } ap_i.k \leq_{\mathcal{T}} fp_i.k \quad i = 1 \ldots n \\ false \text{ otherwise} \end{cases}$$

However, this high-level description of the semantics of an invocation does not take into account implementation issues like the particular strategy (bottom-up or top-down) followed or fixpoint calculations. We now develop a refined approach to the problem, which in fact handles the two types of flows in a uniform fashion.

A particularly useful and efficient way of controlling the interpretation process is to follow a top-down strategy starting from the program main entry point and an abstraction of the input data (or a *topmost value*, if such abstraction is not available). The top-down strategy proposed implicitly creates a graph during analysis where nodes (statements) with several descendants correspond to branches in the concrete execution (conditionals, virtual calls, loops), all of them abstracted as invocations of OR-tuples. Nodes with one descendant indicate serial execution and are abstracted by recursively applying the process to the child node. More precisely, an invocation is an *OR-node* whose children are the bodies of all the OR-tuples whose signature matches that of the call, and each body is an *AND-node* where the semantics of each statement (possibly containing further OR-nodes) are composed.

Given a *call state* CA prior to a statement *stmt*, the *exit state* CP is computed by the function $\mathcal{SS}[\![stmt]\!] : \mathcal{D} \mapsto \mathcal{D}$, with three subcases:

(i) If the statement is a invocation $i = invoke(mname, ap, k_{caller})$, let $o_1, \ldots, o_n$ be the OR-tuples such that $comp(i, o_i) = true$. First we restrict the actual state to those variables that are in *ap*. This is performed by means of the *project* operation described below and results in a new state $\lambda = CA|_{ap}$. The description is further modified to rename the variables so they work in each context of the callee: $\beta_i = \lambda|_{ap}^{fp}$. Then we call recursively $\mathcal{SS}[\![stmt_i]\!]\beta_i$ in order to obtain an exit state for the callee β_i'. Now we proceed in the opposite direction, first by renaming back all variables so that each abstraction is described in terms of the variables in the caller and then by lubbing their partial results: $\lambda' = \bigsqcup \beta_i'|_{fp}^{ap}$. The last step implies conjoining λ' with the initial description via the *extend* operation described below: $CP = extend(CA, \lambda')$.

(ii) If the statement is a concatenation of statements $\{stmt_1, \ldots, stmt_n\}$, the output state is calculated as the composition of the semantics of each element in the list, starting with the initial state: $CP = \mathcal{SS}[\![stmt_n]\!](\ldots \mathcal{SS}[\![stmt_1]\!](CA))$

(iii) If the statement is atomic (does not include further statements) we have a base case that is resolved directly by the domain: $CP = \mathcal{SS}[\![stmt]\!](\sigma_i)$.

The interprocedural, top-down approach requires the designer of the domain to provide two extra operations in addition to the standard [8] lattice functions such as least upper bound or ordering. The $project : \mathcal{D} \times \mathcal{P}(\mathcal{V}) \mapsto \mathcal{D}$ operator restricts the current abstraction to the set of variables specified. The intuition behind it is the removal of irrelevant information in the actual state, in the sense that it does

not relate to the actual parameters of the invocation, reflecting the scoping rules of the blocks being analyzed. The second operation is *extend* : $\mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$, which updates an abstract state CA based on another description λ' that involves only variables in CA. The purpose of *extend* is somehow symmetric to the projection, because after returning from a method invocation we need to reconcile the result of the call (affecting only a few variables within the scope of the caller) with the previous state (affecting all the variables in such scope).

Example 3.1 A pair-sharing domain approximates pairs of variables that might point to the same location in memory [32]. An abstract state like $\{\{X,Y\},\{X,X\},\{Y,Y\},\{Z,Z\}\}$ is an abstraction of a particular heap configuration where variables X and Y might point to the same object, while Z definitely references another position in memory. Projection $\sigma|_{\mathcal{V}}$ is defined as $\{S \mid S = S' \cap V, S' \in \sigma\}$. In the example of Fig. 2c, assume that the actual state before the call to `vector:append_if00_while00` is $CA = \{\{R_0,R_1\},\{R_0,R_2\},\{R_1,R_2\},\{R_0,R_0\},\{R_1,R_1\},\{R_2,\ R_2\}\}$. Since the invocation involves only variables $V = \{R_1,R_2,R_4,R_5\}$ we get $\lambda = CA|_V = \{\{R_1,R_2\},\ \{R_1,R_1\},\{R_2,R_2\}\}$.

The *extend* operation is less straightforward. Assume the existence of a method `foo`(R_0,R_1) called in state $CA = \{\{R_0,R_0\},\{R_0,R_2\},\{R_1,R_1\},\{R_2,R_2\}\}$. After analyzing the body of `foo` the resulting state is $\lambda' = \{\{R_0,R_1\},\{R_0,R_o\},\{R_1,R_1\}\}$, probably because some field in R_0 has been assigned to R_1 or to any of its non null fields (or vice versa) within the method. The information discovered is propagated back to the caller and, thus, $\mathit{extend}(CA,\lambda') = \{\{R_0,R_1\},\{R_0,R_2\},\{R_1,R_2\},\ \{R_0,R_0\},\{R_1,R_1\},\{R_2,R_2\}\}$.

Note that precision can be further improved if, for example, the abstraction captures the run-time class of the objects invoked. Our solution to this issue makes use in the implementation of object orientation by allowing specialization of the base framework through subclassing. For the particular example in hand, domains containing class analysis information [1,10] would just overwrite the implementation of the *comp* predicate in order to obtain smaller sets of candidate methods to analyze.

In addition to the points above, there is one more issue that needs to be addressed. The overall abstract interpretation framework scheme described works in a relatively straightforward way if the (transformed) program has no recursion (i.e., there are no loops or recursion in the original bytecode). Consider, on the other hand, a recursive OR-tuple. If there are two OR-nodes for the tuple in the tree such that the actual parameters *apars* and input state CA are identical, and one node is a descendant of the other, then the tree is infinite and analysis does not terminate. In order to ensure termination, some sort of fixpoint computation is needed. This is the subject of the following section.

4 Generic Top-Down Analysis Algorithm

We now describe our generic top-down analysis algorithm. The algorithm computes the least fixed point making use of *memo tables* [12,36,11]. A memo table contains the results of computations already performed and it is typically used to avoid needless recomputation. However, in our context it is also used to store results

```
Analyze(P, Stmt, CA, MT, Set)
  case Stmt of
    conditional:
      return AnalyzeCond(P, Stmt, CA, MT, Set)
    recursive:
      return AnalyzeLoop(P, Stmt, CA, MT, Set)
    no_recursive:
      return AnalyzeNoLoop(P, Stmt, CA, MT, Set)
    special:
      return AnalyzeSpecial(P, Stmt, CA, MT, Set)
    builtin:
      return AnalyzeBuiltin(Stmt, CA)
  end

AnalyzeCond(P, I, CA, MT, Set)
  λ:=CA
  I = (N, _, _)
  entry:=Find(MT, ⟨N, λ⟩, complete)
  if entry ≠ ∅ then
    entry = ⟨λ', _⟩
  else
    λ' := ⊥
    M:=Lookup(I)
    foreach m ∈ M
      m = (N, _, _, Stmts)
      ⟨λ'_m, MT, Set⟩:=
        EntrytoExit(P, λ, Stmts, MT, Set)
      λ' := λ' ⊔ λ'_m
    end
    Let ID be an unique identifier
    MT:=Insert(MT, ⟨N, λ, λ', complete, ID⟩)
  end
    CP:=λ'
  return ⟨CP, MT, Set⟩

AnalizeNoLoop(P, I, CA, MT, Set)
  I = (N, Ap, _)
  apars = vars(Ap)
  λ:=Project(CA, apars)
  entry:=Find(MT, ⟨N, λ⟩, complete)
  if entry ≠ ∅ then
    entry = ⟨λ', _⟩
  else
    λ' := ⊥
    λ:=λ|^{apars}_{{R0,...,Rn}}
    M:=Lookup(I)
    foreach m ∈ M
      m = (N, Fp, _, Stmts)
      fpars:=vars(Fp)
      V:=vars(Stms)
      β:=Project(λ, fpars)
      β:=Augment(β, V)
      ⟨β', MT, Set⟩:=EntrytoExit(P, β, Stmts, MT, Set)
      λ'_m:=Project(β', apars)
      λ'_m:=λ'_m|^{apars}_{{R0,...,Rn}}
      λ' := λ' ⊔ λ'_m
    end
    Let ID be an unique identifier
    MT:=Insert(MT, ⟨N, λ, λ', complete, ID⟩)
  end
  CP:=Extend(CA, λ')
  return ⟨CP, MT, Set⟩
```

Fig. 3. The fixpoint algorithm (A)

obtained from an earlier round of iteration and whether a certain entry represents final, stable results for the method, or intermediate approximations obtained half way during the convergence of fixpoint computations. An $entry : \mathcal{M}\times\mathcal{D}\times\mathcal{S}\times\mathcal{D}\times\mathcal{I}^+$ in the memo table has the following fields: method name, its projected call state (λ), its status, its projected exit state (λ') and a unique identifier. $find : \mathcal{MT} \times \mathcal{M} \times \mathcal{D} \times \mathcal{S} \mapsto \mathcal{D} \times \mathcal{I}^+$ returns a tuple (λ', ID) corresponding to an entry from the memo table if there exists a renaming such that this entry matches with the given method name and its λ. Other memo table operations are: $findStatus : \mathcal{MT} \times \mathcal{M} \times \mathcal{D} \mapsto \mathcal{D} \times \mathcal{I}^+ \times \mathcal{S}$, $updStatus : \mathcal{MT} \times \mathcal{M} \times \mathcal{D} \times \mathcal{S} \mapsto \mathcal{MT}$, $updLambdaPrime : \mathcal{MT} \times \mathcal{M} \times \mathcal{D} \times \mathcal{D} \mapsto \mathcal{MT}$, and $insert : \mathcal{MT} \times \mathcal{E} \mapsto \mathcal{MT}$. We also assume a procedure called $lookup : \mathcal{M} \mapsto \mathcal{P}(\mathcal{M})$ which given a method description returns all methods that implement it.

The actual analysis algorithm is shown in pseudocode in Figs. 3 and 4.[6] There are three major subcases. If the statement is an invocation of a non recursive method, `AnalyzeNoLoop` handles the call. It first checks whether there is an entry in the memo table for the name of the invoked method and its λ. In that case the stored value of λ' is immediately passed to the `Extend` operation to yield the exit state. Otherwise, the variables of its λ are renamed to the set of variables $\{R_0, \ldots, R_n\}$ and for each method m returned by the `Lookup` procedure the following actions are carried out: a projection of λ onto the m variables and addition of the variables of

[6] This description does not include the abstract operation of widening. It is straightforward to modify the algorithm to include widening of call and answer patterns, we omit it for simplicity.

```
AnalyzeLoop(P, I, CA, MT, Set)
  I = (N, Ap, _)
  apars:=vars(Ap)
  λ:=Project(CA, apars)
  entry:=FindStatus(MT, ⟨N, λ⟩)
  λ:=λ|^{apars}_{{R0,...,Rn}}
  if entry ≠ ∅ then
    entry = ⟨λ'1, ID, status⟩
    case status of
      complete:
        λ'2 :=λ'1
      fixpoint:
        λ'2 :=λ'1
        Set:=Set ∪ {ID}
      approximate:
        MT:=UpdStatus(MT, ⟨N, λ⟩, fixpoint)
        ⟨λ'2, MT, Set⟩:=
          CompFixpo(P, I, λ, MT, Set)
    end
  else
    λ':=⊥
    M:=Lookup(I)
    foreach non-recursive m ∈ M
      m = (N, Fp, _, Stms)
      fpars:=vars(Fp)
      V:=vars(Stms)
      β:=Project(λ, fpars)
      β:=Augment(β, V)
      ⟨β', MT, Set⟩:=EntrytoExit(P, β, Stms, MT, Set)
      λ'm:=Project(β', apars)
      λ'm:=λ'm|^{apars}_{{R0,...,Rn}}
      λ':=λ' ⊔ λ'm
    end
    MT:=Insert(MT, ⟨N, λ, λ', fixpoint, ID⟩)
    ⟨λ'2, MT, Set⟩:=CompFixpo(P, I, λ, MT, Set)
  end
  CP:=Extend(CA, λ'2)
  return ⟨CP, MT, Set⟩

EntrytoExit(P, β, Stmts, MT, Set)
  CA:=β
  foreach Stmt ∈ Stmts until Stmt = return
    ⟨CP, MT, Set⟩:=Analyze(P, Stmt, CA, MT, Set)
    CA:=CP
  end
  β':=CP
  return ⟨β', MT, Set⟩

CompFixpo(P, I, λ, λ', MT, Set)
  I = ⟨N, Ap, _⟩
  apars:=vars(Ap)
  entry:=Find(MT, ⟨N, λ⟩, _)
  set_I:=∅
  changed:=false
  repeat
    fixpoint:=true
    entry = ⟨λ', ID⟩
    M:=Lookup(I)
    foreach m ∈ M
      m = (N, Fp, _, Stmts)
      if N is recursive or changed
        fpars:=vars(Fp)
        V:=vars(Stmts)
        β:=Project(λ, fpars)
        β:=Augment(β, V)
        ⟨β', MT, set_Stmts⟩:=
          EntrytoExit(P, β, Stmts, MT, ∅)
        λ'm:=Project(β', apars)
        λ'm:=λ'm|^{apars}_{{R0,...,Rn}}
        λ'old:=λ'
        λ':=λ'old ⊔ λ'm
        if λ'old ≠ λ' then
          fixpoint:=false
          changed:=true
          MT:=UpdLambdaPrime(MT, ⟨N, λ⟩, λ')
        end
        set_I:=set_I ∪ set_Stmts
      end
    end
  until (fixpoint = true)
  if set_I \ {ID} = ∅ then
    status:=complete
  else
    status:=approximate
  end
  MT:=UpdStatus(MT, ⟨N, λ'⟩, status)
  Set:=Set ∪ set_I \ {ID}
  return ⟨λ', MT, Set⟩
```

Fig. 4. The fixpoint algorithm (B)

the m body to yield its corresponding β. Then, each statement in the body of m is analyzed by calling the `EntrytoExit` procedure resulting in a set of exit states which are "lubbed." These states have been previously projected onto the variables of the invoked method and renamed in terms of these variables. This "lubbed" state is inserted as an entry in the memo table and characterized as `complete`. Finally, the `Extend` operation is applied in order to produce the exit state.

In *conditional* methods the decompilation ensures that the formal parameters of the method are indeed named as in the caller. Furthermore, caller and callee have an identical scope so in an invocation $I = \langle N, Ap, \_\rangle$ to a conditional method, all the compatible tuples $m = \langle N, Fp, \_, Stmts\rangle$ verify $vars(Stmts) = vars(Fp)$ (i.e., they have no extra local variables) and $vars(CA) = vars(Ap) = vars(Fp) = \{R_0, \ldots, R_n\}$. This property is used in `AnalyzeCond` to speed up analysis, since the `Project` and `Extend` operations can be skipped.

Finally, when a method is recursive the fixpoint computation defined by the `AnalyzeLoop` procedure in Fig 4 is required since analysis needs to be repeated until fixpoint is reached for the abstract and-or 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. Firstly, `AnalyzeLoop` begins analyzing those non-recursive instances of the invoked method in the same way as `AnalyzeNoLoop`. With this, we are able to yield a possible λ' different from $\bot$ which will accelerate the further fixpoint computation, and then an entry in the memo table is inserted with this information and characterized as `fixpoint`. After this, the `CompFixpo` procedure (also defined in Fig. 4) is called. At each iteration, a similar process to that described in `AnalyzeNoLoop` is performed. However, between the end of one iteration and the beginning of the next one, the values of the *previous* λ' and the *new* λ' are compared. If they are the same, then fixpoint has been reached and the procedure finishes ensuring that the least fixed point has been computed. Otherwise, the least fixed point has not been reached yet and a new iteration will be performed.

Dealing with Mutually Recursive Methods. For the sake of simplicity, the description of the analysis so far has omitted some details which are needed in order to support mutually recursive methods. In this case, our algorithm operates as follows. Firstly, we need to use new values for the *status* field in memo table entries. `fixpoint` is used when the fixpoint has not been reached yet. `approximate` represents when the fixpoint has been reached for a method m_1 in this entry 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). Finally, `complete` is used when fixpoint has been reached for this method. Furthermore, we also need to use the *ID* field in order to detect occurrences of mutual recursion. We also need to use a set of ID's to keep track of the recursive methods during the analysis. When a fixpoint computation is started, the analysis searches for an entry in the memo table. Given a method and its λ, if there exists an entry characterized as `complete`, then the λ' is obtained from it. If the entry is characterized as `fixpoint` means that the method is recursive and thus we add its ID in the set of ID's. If the entry is `approximate`, then the method or one of its successors in the and-or tree has an approximate value of its exit state. Thus, we need to mark it as `fixpoint` and start its fixpoint computation again. Finally, after a fixpoint computation is reached we need to verify the ID's contained in the set of ID's. If this set contains only the ID corresponding to the method which is being analyzed, then the value of its λ' is complete. Otherwise, the method depends on other ID's (i.e., methods) and so, we mark its output abstract value as `approximate`. In both cases, we eliminate the method's ID from the set of ID's.

Example 4.1 We now illustrate how the fixpoint algorithm described in Sect. 4 works for the program in Fig. 2. The domain used will be pair sharing. The objective is to analyze the semantics of the `append` method in the context of the `Vector` and `ZipVector` classes.

Space limitations obviously prevent us from showing the entire process in detail. We will instead assume that the starting program point for analysis is right before

var byt	var src	*line*
R_0	*this*	–
R_1	*v*	–
R_2	*e*	1
R_3	*this.first*	3
R_4	*e.next*	5
R_5	*v.first*	7

Fig. 5. Equivalence of variables between source code and internal representation

the call to `append` in the `Vector` implementation of `add`. Note that the method creates a vector V which contains a shallow copy of *Element* so that the three objects (*This*, *Element* and, V) cannot point to the same location in memory and $CA^{Vector}_{append} = \{\{This, This\}, \{Element, Element\}, \{V, V\}\}$.

The invocation is classified as non recursive and handled by `AnalyzeNoLoop`. We now have to project CA^{Vector}_{append} over the two actual parameters and then rename these to the equivalent formal parameters.[7] Since R_0 is *This* and R_1 is V we get $\lambda_{append} = \{\{R_0, R_0\}, \{R_1, R_1\}\}$. To simplify notation we will denote `append_if00` and `append_if_while00` by `if` and `while` respectively. Analysis of the `append` body results in a call to `AnalyzeCond`, since the last statement is an invocation to `if`. At that point $CA_{if} = \{\{R_0, R_0\}, \{R_0, R_2\}, \{R_1, R_1\}, \{R_2, R_2\}\}$ because e (R_2) points to a field of *this* (R_0).

Conditional invocations are simpler to handle: no *project*, *extend*, or rename operations are required. Instead, we directly examine the two methods corresponding to `if`. The first branch implies that R_2 is null and that a R_0's field and R_3 point to the vector passed as argument R_1, Thus, $\lambda'_{if,1} = \{\{R_0, R_1\}, \{R_0, R_3\}, \{R_1, R_3\}, \{R_0, R_0\}, \{R_1, R_1\}, \{R_3, R_3\}\}$. The second compatible method with the invocation implies $R_2 \neq null$ but its semantics depends on a loop call to `while`. Control of the algorithm is passed to the `AnalyzeLoop` subroutine which projects and renames $CA_{while} = \{\{R_0, R_2\}, \{R_2, R_4\}, \{R_0, R_0\}, \{R_1, R_1\}, \{R_2, R_2\}, \{R_4, R_4\}\}$ again yielding $\lambda_{while} = \{\{R_2, R_4\}, \{R_1, R_1\}, \{R_2, R_2\}, \{R_4, R_4\}\}$. The non recursive part is then analyzed first. Since termination depends on R_4 being null and the final assignment (line 7 in the source) forces R_1 and R_2 to share through intermediate variable R_5 we have $\lambda'_{while,1} = \{\{R_1, R_2\}, \{R_2, R_5\}, \{R_1, R_5\}, \{R_1, R_1\}, \{R_2, R_2\}, \{R_5, R_5\}\}$. A new entry e_1 =(`while`,λ_{while},fixpoint,$\lambda'_{while,1}$,id_1) is inserted in the memo table.

Fixpoint computation starts by analyzing (recursive) methods that are compatible with the invocation. The only tuple found (last in Fig. 2c) is processed in a straightforward manner until the self-invocation, which triggers a search in the memo table with return value e_1 (`AnalyzeLoop` subroutine). We use the current approximation of the `while` semantics, derived from the base case. On return to the fixpoint routine, we will calculate a $\lambda'_{while,2}$ which is identical to $\lambda'_{while,1}$, because the statements in the body of the recursive tuple do not really alter any information about variables in λ_{while}. The relation $(\lambda_{while}, \lambda'_{while})$ did not change after one single iteration and the process can be considered as *complete* for the `while` method.

[7] For better understanding of the variable equivalence check Fig. 5.

		PS			
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t
dyndisp	71	68	3	114	30
clone	41	38	3	42	52
dfs	102	98	4	103	68
passau	167	164	3	296	97
qsort	185	142	43	182	125
intgrqsort	191	148	43	159	110
pollet01	154	126	28	276	196
zipvector	272	269	3	513	388
cleanness	314	277	37	360	233

Fig. 6. Analysis times, number of program points, and number of abstract states.

The memo table status of the e_1 tuple is updated accordingly.

Coming back to the semantics of the second branch of the `if` method, we observe that it has to be identical to $extend(CA_{if}, \lambda'_{while,1})$, which forces further sharings with the R_0 object to produce $\lambda'_{if,2} = \{\{R_0, R_1\}, \{R_0, R_3\}, \{R_1, R_3\}, \{R_0, R_0\}, \{R_1, R_1\}, \{R_3, R_3\}\}$. We now write a new entry in the memo table: (`if`, CA_{if}, complete, $\lambda'_{if,1} \sqcup \lambda'_{if,2}$,$id_2$). This entry, projected over the formal parameters of `append` results in yet another entry (`append`, $\{\{R_0, R_0\}, \{R_1, R_1\}\}$, complete, $\{\{R_0, R_1\}, \{R_0, R_0\}, \{R_1, R_1\}\}$,$id_3$). This semantics is congruent with the concatenation that takes place inside the method.

We are now in the position of inferring the abstract semantics of `add` in class `Vector`. Remember that $CA^{Vector}_{append} = \{\{This, This\}, \{Element, Element\}, \{V, V\}\}$ and that the call to `append` results (after renaming) in $\{\{This, V\}, \{This, This\}, \{Element, Element\}, \{V, V\}\}$. We repeat the same process of projecting over the formal parameters thus $CP^{Vector}_{add} = \{\{This, This\}, \{Element,\ Element\}\}$. In the *ZipVector* there is a different call state prior to `append` invocation, derived from the insertion of the `element` in `v` (instead of copying its fields, like in `Vector`): $CA^{ZipVector}_{append} = \{\{Element, V\}, \{This, This\}, \{Element, Element\}, \{V, V\}\}$. Nevertheless, `AnalyzeLoop` will find the λ entry already in the memo table, since $CA^{Vector}_{append}|_{This,V} = CA^{ZipVector}_{append}|_{This,V}$ thus $\lambda^{Vector}_{append} = \lambda^{ZipVector}_{append}$. We can reuse the computed semantics to get the same λ'_{append} for the call. On extension with $CA^{ZipVector}_{append}$ it results in $CP^{ZipVector}_{add} = \{\{This, Element\}, \{This, This\}, \{Element, Element\}\}$. If we repeat the process for a call state CA_{append} where *This* and V share, CP_{append} will remain the same on exit, but the memo table now contains two entries for the same method reflecting the two different call contexts (multivariance).

5 Some Experimental Results

We have completed a preliminary implementation of our framework, and coded a pair sharing (PS) analysis extending the operations described in [32] in order to handle some additional cases required by our benchmark programs such as primitive variables, visibility of methods, etc. The benchmarks used have been adapted from previous literature on either abstract interpretation for Java or points-to analysis [32,26,25,34]. Our experimental results are summarized in Fig. 6. The first

column ($\#tp$) 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 program points that the analysis explores, while $\#up$ represents the ($\#tp - \#rp$) points that are not analyzed because the analysis determines that they are unreachable. Since our framework is multivariant 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 of abstract states is typically larger than the number of reachable program points. Column $\#\sigma$ provides the total number of these abstract states inferred by analysis. The level of multivariance is the ratio $\#\sigma/\#rp$. In general, such a larger number for $\#\sigma$ tends to indicate more precise results. The t column in Fig. 6 provides preliminary results regarding running times for the different benchmarks, in milliseconds, on a Pentium III 2.0Ghz, 1Gb of RAM, and averaging several runs after eliminating the best and worst values.

6 Conclusions

We have presented a novel algorithm for analysis of Java bytecode which includes a number of optimizations in order to reduce the number of iterations. The algorithm is *parametric* in the sense that it is independent of the abstract domain used. The algorithm is also multivariant and top-down/flow-sensitive. Also, the algorithm uses a program transformation, prior to the analysis, that results in a highly uniform representation of all the features in the language and which simplifies analysis.

References

[1] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA*, pages 324–341, 1996.

[2] 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.

[3] 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, Denver, Colorado, November 1999.

[4] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

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

[6] B. Le Charlier, O. Degimbe, L. Michael, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, pages 15–26. Springer-Verlag, September 1993.

[7] J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS. Springer-Verlag, April 2006.

[8] 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.

[9] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, January14-16 2004. ACM Press, New York, NY.