A Generic, Context Sensitive Analysis Framework for Object Oriented Programs

Jorge Navas[1], Mario Méndez-Lojo[1], and Manuel V. Hermenegildo[1,2]

[1] University of New Mexico (USA)
[2] Technical University of Madrid (Spain)

Abstract. Abstract interpreters rely on the existence of a fixpoint algorithm that calculates a least upper bound approximation of the semantics of the program. Usually, that algorithm is described in terms of the particular language in study and therefore it is not directly applicable to programs written in a different source language. In this paper we introduce a generic, block-based, and uniform representation of the program control flow graph and a language-independent fixpoint algorithm that can be applied to a variety of languages and, in particular, Java. Two major characteristics of our approach are accuracy (obtained through a top-down, context sensitive approach) and reasonable efficiency (achieved by means of memoization and dependency tracking techniques). We have also implemented the proposed framework and show some initial experimental results for standard benchmarks, which further support the feasibility of the solution adopted.

Keywords: Fixpoint algorithms; context sensitivity; static analysis; Java bytecode; abstract interpretation.

1 Introduction

Analysis of the Java language (either in its source version or its compiled bytecode [17]) using the framework of abstract interpretation [7] has been the subject of significant research in the last decade (see, e.g., [18] 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., [4, 25]) or statically verify certain properties about the run-time behavior of the code (e.g., [12, 15]). In contrast to this concentration and progress on the development of new, refined domains there has been comparatively little work on the underlying fixpoint algorithms. In fact, many existing abstract interpretation-based analyses use relatively inefficient fixpoint algorithms. In other cases, the fixpoint algorithms are specific to a particular source language or analysis and cannot easily be reused in other contexts.

The proposed framework is generic both in terms of the source language and the abstract domain. Analysis is a two-step process that starts with a program transformation; this phase is language dependent and results in a control flow graph (CFG)-style representation where the operational semantics is made explicit. For example, a virtual call is replaced by a non-deterministic call to all the

possible implementations it can be resolved to. This encoding allows transforming different related idioms of a given language (or from several languages) into a highly uniform representation. We argue that this preliminary (de)compilation process greatly simplifies the burden of designing new analyses and abstract operations.

Although we have generality in mind, for concreteness we implemented a (de)compiler from Java bytecode to our CFG-style representation. This step is partially based in the Soot [21, 27] tool. This has the advantage of automatically providing a way of analyzing certain languages that can be compiled to Java bytecode, like SML [2]. In a similar fashion, we expect the BoogiePL [10] intermediate representation to become more popular and therefore we also target the addition of an alternative compilation phase for that source will allow analysis of CIL programs, written in C#, J#, etc. Our ultimate objective is to support the full Java language but the current implementation has some limitations: it does not support dynamic loading of classes, threads, and runtime exceptions. Also, analysis of the JDK libraries is done under a worst-case assumption.

A second, pivotal piece of the framework is an efficient fixpoint algorithm, introduced in [20]. Herein we improve the description so that it is now decoupled from any 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, ours is the first concise and precise description of a top-down, context sensitive, and parametric fixpoint algorithm for object oriented programs.

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, focusing on the characteristics of the transformation and illustrating it with a relatively complete example. The translation process produces a structured, decompiled representation of the Java bytecode and is based on the SOOT framework [27] which has been successfully used in previous analyses [8, 3]. However, instead of analyzing directly the Jimple representation –based on `gotos`– it is processed further in order to build a control flow graph (CFG). The idea is also analogous to the approach of [12, 26] but the graph obtained is

```
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

Fig. 1. Vector example: source code and corresponding metainformation

somewhat different since we do not distinguish between stack and local variables, and all the operands are explicit in the expressions.

Full independence from the language cannot be achieved only through program transformations. Sometimes, the fixpoint algorithm can be optimized if some characteristics related to the CFG are known. In other occasions, the abstract domain needs information about the program that cannot be found in the flow graph. Both demands are solved via *metainformation* files. We illustrate this point with the example in Figure 1, which shows an alternative version of the JDK `Vector` class. The original Java source has been included for better understanding of the example, although the input to the framework is always in bytecode format. The descendant `SubVector` contains an alternative version of the `append` method. The corresponding CFG is shown in Figure 2; we omitted the constructor (`init`) blocks for simplicity.

Space reasons prevent us from listing the full description of the metainformation; only hierarchy and method type tables are shown in Figure 2. In the case of the parent-child relations, the purpose is to permit the abstract domain access to the class tree, the more obvious application being class analysis [1]. The second table contains a classification for each method, which can be y (entry) or n (internal) and it is used to optimize the performance of the fixpoint engine (avoiding *projection* and *extension* operations, see Section 3). Additionally, this table contains also the method signatures that can be used for the the abstract domain.

An *entry* method corresponds, in the original program, to the first block [13] of the Java method of the same name and shares its signature, except for an

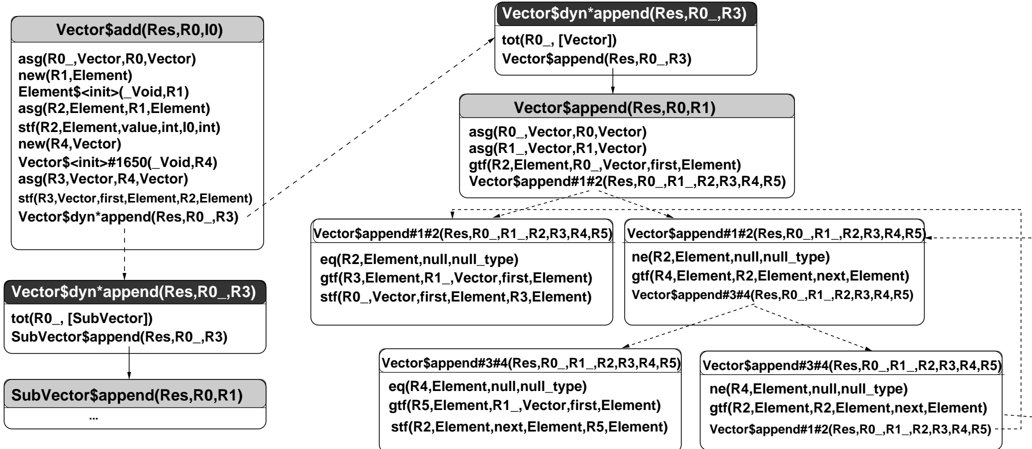

Fig. 2. Control Flow Graph for the example in Figure 1

extra parameter that represents the value returned. The other blocks present in the Java method are compiled into (components of) *internal* methods which share the same set of variables: all the formal parameters and local variables they reference. Examples of constructions converted into internal blocks are `if`, `while` or `for` loops, which are the bytecode level all have the form of labeled sequences of instructions. In the example, we can see how the `if (e==null)...else` conditional in the `Vector` implementation of `append` is converted into two different blocks, one for each branch, which actually share the same name `Vector$append#1#2` (Figure 2). In this case, the internal method is composed of two blocks which are indistinguishable from the caller's point of view, thus causing invocations to the method to be non-deterministic (i.e. causing the execution of one block or another). Entry blocks are marked in grey, internal in white; dotted arrows denote non-deterministic flows while the continuous ones symbolize deterministic calls.

Another flow transformation (*extra* blocks) tries to expose the internal structure of the more complex bytecode instructions, which sometimes encode sophisticated operations. That is the case of a virtual invocation, that triggers a *lookup* in the hierarchy of the instance in order to figure out which particular implementation should be executed. Instead of delegating the treatment of such complexities to the framework, we make these aspects of the operational semantics explicit in the intermediate representation using program transformations as in [12]. Coming back to the example in Figure 1, note that the call to `append` within `add` is polymorphic: it might execute the implementation in `Vector` or the one in `SubVector`. We make this semantics explicit by inspecting the application hierarchy and replacing the virtual invocation with a set of resolved calls, one for each possible implementation. The method acting as a "hub" is called an *extra* block; in the example we have one, `Vector$dyn*append`, marked in black. It behaves in a very similar way to the conditional discussed previously, since the program flow might go through two alternative paths (blocks), one for each implementation of `append`. Each branch contains a guard (`tot`, see the

first statement in each of the `Vector$dyn*append` blocks) listing the acceptable types for the callee.

We believe that the approach adopted for virtual invocations is far more flexible than previous solutions (e.g., [22] or even [20]), in which the semantics of a virtual invocation contains the lookup in the hierarchy, thus complicating its formalization and tying the fixpoint description to the Java language. In our framework, all the calls are resolved *a priori* in the compilation phase and in that sense the fixpoint engine sees no difference between, for example, a virtual call in Java or a function invocation in SML. Checked exceptions [17] are treated by the compiler in a similar way, so analysis is not even aware of their existence.

Finally, the statements themselves correspond to the three-address representation output by Soot: stack and local elements have been converted into named variables and all the expressions are typed. It is interesting how, in an analogous way to the block case, we introduced *extra* statements to further simplify analysis. For example, the `tot` (type of this) builtin filters the execution of subsequent statements when the class of the instance is not listed in the set of possibilities; guard statements have a similar goal in blocks that come from conditional constructions. In Figure 2 the `eq` call at the beginning of the left-most `Vector$append#1#2` block refers to the condition for executing the first branch, while the `ne` call contains its negated version, for the second alternative. Also, those methods that are *entry* but not *extra* contain assignments to shadow variables that simulate the call-by-reference semantics [20].

3 The Top-Down Analysis Algorithm

We now describe our top-down analysis algorithm, which calculates the least fixed point given a control flow graph 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 ($\lambda^{'}$) 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 and 4[3]. Builtins are treated directly by each domain; the same happens for external invocations since we are making, in the current implementation, 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.

[3] It is straightforward to modify the algorithm to include widening, we omit it for simplicity.

```
topDownAnalyze(CFG, method, dom, in, mt, set)
  mflag:=classify(CFG, method)
  case mflag 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)
  end

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)
  end
  out:=dom.extend(in, actPars, λ')
  return ⟨out, mt, set⟩

analyzeNonRecBlocks(CFG, name, dom, actPars, λ, st, mt, set)
  λ:=λ|{actPar_0,...,actPar_m}^{res,r_0,...,r_m}
  blocks:=getNonRecBlocks(name)
  λ':=⊥
  foreach block ∈ blocks
     body:=getBody(block)
     ⟨β', mt, set⟩:=analyzeBody(CFG, β, dom, body, mt, set)
     λ'_b:=dom.project(β', {res, r_0, ..., r_m})
     λ':=λ' ⊔ λ'_b
  end
  λ':=λ'|{res,r_0,...,r_m}^{actPar_0,...,actPar_m}
  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
  end
  β':=out
  return ⟨β', mt, set⟩
```

Fig. 3. The top-down fixpoint algorithm

Invocations of non-recursive methods are handled by `analyzeNonRecMethod`. It 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 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 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` [20] operation, yielding the exit state.

When a method is recursive, the `analyzeRecMethod` procedure in Fig 4 repeats 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

```
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)
    end
    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)
            end
        end
    else
        status:=approximate
        make id dependent from set_method \ {id}
    end
    mt.update(⟨name, λ'⟩, status)
    set:=set ∪ set_method \ {id}
    return ⟨mt, set⟩
```

```
analyzeRecBlocks(CFG, method, dom, λ, λ', mt, set)
    name:=getName(method)
    actPars:=getActualParams(method)
    λ:=λ|^{res,r_0,...,r_m}_{actPar_0,...,actPar_m}
    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'|^{actPar_0,...,actPar_m}_{res,r_0,...,r_m}
            if λ'_old ≠ λ' then
                fixpoint:=false
                mt.update(⟨N, λ⟩, λ')
            end
            set_method:=set_method ∪ set_body
        end
    until (fixpoint = true)
    ⟨mt, set⟩:=
        updateDeps(method, mt, set_method, set)
    return ⟨λ', mt, set⟩
```

Fig. 4. The top-down fixpoint algorithm (continuation)

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 $\lambda^{'}$ 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* [11] 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 [9], which allows independent reasoning about different branches. A final, more elaborate optimization uses the metainformation described in Section 2. Since the `extend` operation is usually computationally expensive and may introduce further imprecision, it is desirable to avoid it whenever possible. For that reason, the analysis can take advantage of some compiler invariants, such as the equal signature shared by all the internal methods contained in the same Java method. Because of having the same number and naming of formal parameters, the `extend` operation turns out to be unnecessary when the call is invoked from an internal method –the categorization is contained in the metainformation– and targets an internal method.

Example 1. We show how an example of mutual recursion (`Vector$append`) described in Fig. 2 is handled by the fixpoint algorithm defined in Figs. 3 and 4. 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 Fig. 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.

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, 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

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	$\{app_{12}\}$

step	method	λ	λ'	st	dep
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**	$\emptyset$

Fig. 5. Fixpoint calculation for `Vector$append`

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*.

4 Initial Experimental Results

We have completed a preliminary implementation of our framework and tested it by using a nullity domain. Our experimental results are summarized in Fig. 6; the benchmarks belong to the JOlden suite [6]. The first three columns 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 a little big 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 Section 2. The next two columns strictly correspond 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 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. 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 [23], we do not support incrementality at that level in

name	k	m	b	pp	st	ast	pt	at
jolden.health.Health	8	30	620	833	3.5	2964	1.1	4.7
jolden.bh.BH	9	70	1198	1473	4.2	6266	3.2	24.3
jolden.voronoi.Voronoi	6	73	988	1108	2.6	2850	2.2	5.9
jolden.mst.MST	6	36	443	304	2.7	844	0.1	0.7
jolden.power.Power	6	32	997	1143	4.2	4743	2.1	12.1
jolden.treeadd.TreeAdd	2	12	193	217	2.2	468	2.0	0.3
jolden.em3d.Em3d	4	22	444	566	5.5	3100	0.1	4.8
jolden.perimeter.Perimeter	10	45	543	770	3.1	2366	0.1	1.8
jolden.bisort.BiSort	2	15	323	467	4.2	1934	0.1	2.9
jolden.all.All	42	287	5168	6432	6.1	39159	10.5	163.7

Fig. 6. Analysis times, number of program points, and number of abstract states on a Pentium M 1.73Ghz with 1Gb of RAM.

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 or abstract domains with non-linear worst-case complexity in their operations.

5 Related work

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., [4, 5]) focuses on particular properties and therefore their solutions (abstract domains) are tied to them, even when they are explicitly multipurpose, like TVLA [16]. In [22] 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 [26] 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 [14]. 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 [24], the resulting analyzers typically lack some of the characteristics that are the objective of our proposal, and, specially, context sensitive results. Finally, Cibai [19] 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.

6 Conclusions

We have presented a novel abstract interpretation framework, which is generic in terms of the source language and abstract domain in use. The framework is built upon a decompilation phase that results in a control flow graph (CFG) where

the operational semantics is made explicit, and an analysis phase based upon an efficient, precise fixpoint algorithm which is concisely described in this paper and considered itself an important contribution of our work. This 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.

Acknowledgments

The authors are supported by the Prince of Asturias Chair in Information Science and Technology at UNM. This work was also funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the *PROMESAS* project.

References

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

[2] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ml to java bytecodes. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 129–140, New York, NY, USA, 1998. ACM Press.

[3] 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.

[4] 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.

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

[7] 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.

[8] 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.

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

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

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

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

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

[14] 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.

[15] Xavier Leroy. Java bytecode verification: An overview. In *CAV*, pages 265–285, 2001.

[16] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.

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

[18] 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.

[19] Francesco Logozzo. Cibai: An abstract interpreation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*. To appear, Jan 2007.

[20] 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.

[21] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127, London, UK, 2002. Springer-Verlag.

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

[23] 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.

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

[25] Erik Ruf. Effective synchronization removal for java. In *PLDI*, pages 208–218, 2000.

[26] 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.

[27] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.