A Flexible, (C)LP-based Approach to the Analysis of Object-Oriented Programs

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

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

Abstract. Static analyses of object-oriented programs usually rely on intermediate representations that respect the original semantics while having a more uniform and basic syntax. Most of the work involving object-oriented languages and abstract interpretation usually omits the description of that language or just refers to the Control Flow Graph (CFG) it represents. However, this lack of formalization on one hand results in an absence of assurances regarding the correctness of the transformation and on the other it typically strongly couples the analysis to the source language. In this work we present a framework for analysis of object-oriented languages in which in a first phase we transform the input program into a representation based on Horn clauses. This allows on one hand proving the transformation correct attending to a simple condition and on the other being able to apply an existing analyzer for (constraint) logic programming to automatically derive a safe approximation of the semantics of the original program. The approach is flexible in the sense that the first phase decouples the analyzer from most language-dependent features, and correct because the set of Horn clauses returned by the transformation phase safely approximates the standard semantics of the input program. The resulting analysis is also reasonably scalable due to the use of mature, modular (C)LP-based analyzers. The overall approach allows us to report results for medium-sized programs.

1 Introduction

Analysis of object-oriented languages using abstract interpretation [10] is currently the subject of significant research (see, e.g., [21] and its references). The abstract interpretation approach brings an interesting and 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. 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 [31] for Java or BoogiePL [11] for .NET.

In this paper we propose the use of a Horn clause-based representation as an intermediate language. Our objective is twofold. On one hand we would like to take advantage of existing analyzers for (constraint) logic programs. On the other, we want to be able to offer assurances that the output of the process of transformation into the intermediate representation safely approximates the standard semantics of the input program. Performing the analysis using logic programming tools offers a number of advantages, such as the relative maturity and sophistication of the solutions available, like abstract interpreters [13, 16] (which offer parametric, efficient, and modular fixpoint algorithms) and verifiers. A second strength of our transformational approach is that the framework can be easily adapted to the analysis of other languages without having to redefine the fixpoint algorithm [23]. In fact, using the 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 by describing our methodology (Section 2) and our approach to ensuring correctness using some fundamental parts of the transformation of Java programs into our representation as examples (Section 3). Section 4 shows how analysis of specific aspects of Java can be optimized using metainformation. We then illustrate the application of our approach to other languages, such as C# (Section 5). We also report on an implementation of the ideas presented in this paper using the abstract interpretation-based CiaoPP [16] framework. It can be configured for many different analyses by simply plugging the corresponding abstract domain. The examples try to detect null pointer dereferences (nullity analysis) and eliminate dynamic dispatch (class analysis) in Java programs. The experiments in Section 6 show that the technique scales well in non trivial scenarios, and results in smaller analysis times than similar previous work. Related abstract interpretation-based frameworks, and how they differ from ours, are discussed in Section 7, and Section 8 presents our conclusions.

2 Methodology

Our framework is composed of a front-end preprocessor and a back-end analyzer, as shown in Figure 1. The preprocessor transforms an input in Java source format into a set of Horn clauses that represent a safe approximation of its standard semantics (Sect 3). Sometimes the source code is not available, so we also accept Java bytecode as a valid input format. In this case the (de)compilation from bytecode to Horn clauses is based on a postprocessing of the Jimple representation returned by the Soot [31] tool. It is beyond the scope of this paper to provide a detailed description of this particular transformation; the reader is referred to [23] (which presents the transformation and a specific fixpoint algorithm) for details. In both cases the same subset of the language is covered by the framework. Our ultimate objective is to support the full Java language

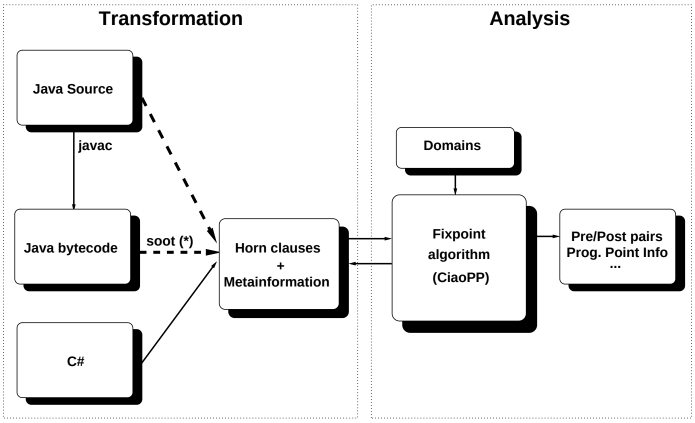


Fig. 1. Pipeline of transformation and analysis

but the current implementation has some limitations: it does not support dynamic loading of classes, threads, or runtime exceptions. Also, analysis of the JDK libraries is done under a worst-case assumption. We have distinguished in the figure the two implemented transformations with dotted arrows to clearly separate them from already existing phases.

Different languages can be incorporated into the framework (i.e., analyzed) by providing a *correct* transformation for them. For example, programs written in Ciao are obviously also accepted by the system as input. Compatibility with other object-oriented languages written in languages like C#, that share many syntactic and semantics features with Java, is easily achievable as illustrated in Section 5.

The resulting Horn-clause intermediate representation is then analyzed using the CiaoPP framework [16] and benefits from its advanced features: efficient computation of fixpoints using memoization, context-sensitivity, modularity, etc. The programmer needs only to implement (in Ciao) 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. 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 given 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 runtime. Metainformation computed 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 [18]), pinpoint errors in the original program, or implement compiler optimizations.

3 Correctness of the transformation phase

Our Horn clause representation of a Java program is basically an unfolded, three-address version of the source where the operational semantics of some instructions is made explicit. The transformed code is denoted by the c subindex: for example, the result of transforming a virtual invocation $v\,.\,\mathtt{m}(v_1, \ldots, v_n)$ is $v_c\,.\,\mathtt{m}_c(v_{1_c}, \ldots, v_{n_c}) = v\,.\,\mathtt{m}_c(v_1, \ldots, v_n)$, since variable expressions are not transformed ($v_c = v$).

Correctness of the transformation requires that the original program *prog* be emulated by $prog_c$ thus $\mathcal{C}[\![prog]\!] = \mathcal{C}[\![prog_c]\!]$, where the semantics operator $\mathcal{C}[\![]\!] : com \mapsto (\mathcal{D} \mapsto \mathcal{D})$ takes as input a command *com* and a concrete state, and returns the output state. The operator has been defined in [15] and (from a denotational point of view) in [2, 29]. Correctness of the preprocessing and analysis requires that if the set of Horn clauses program is safely approximated (using a given abstract domain) by the analysis, so is the original: $\mathcal{C}^*[\![prog]\!] = \mathcal{C}^*[\![prog_c]\!]$. The operator $\mathcal{C}^*[\![]\!] : com \mapsto (\mathcal{D}^* \mapsto \mathcal{D}^*)$ is the abstract counterpart of $\mathcal{C}[\![]\!]$.

We will take a slightly different approach by interpreting Java semantics as a particular case of SLD [17] resolution, in which the *computation* rule in use is left-to-right (commands are executed in the order they appear in the program) and the *search* rule used to determine the target method in an invocation does not really matter, since execution of the Java program is deterministic and therefore for any literal there is exactly one clause that unifies with it at runtime. Therefore, if $\mathcal{S}[\![]\!] : com \mapsto (\mathcal{D} \mapsto \mathcal{P}(\mathcal{D}))$ is the SLD semantics operator, the condition $\mathcal{S}[\![prog]\!] = \{\mathcal{C}[\![prog]\!]\}$ ensures $\mathcal{S}^*[\![prog]\!] = \mathcal{C}^*[\![prog]\!]$. Again, $\mathcal{S}^*[\![]\!] : com \mapsto (\mathcal{D}^* \mapsto \mathcal{D}^*)$ is the (collecting) abstract version of $\mathcal{S}[\![]\!]$.

This formalization is useful since it helps in understanding the Java source as a set of Horn clauses (methods) composed by zero or more goals, the commands. It is also helpful because our transformation introduces new clauses such that now more than one clause might unify with a given literal. This is equivalent to saying that the execution of the transformed program on some input state might result in multiple output states, of which one is the unique state that the original program would return: $\mathcal{S}[\![prog]\!] \subseteq \mathcal{S}[\![prog_c]\!]$. An interesting property of that transformed program is that its abstract semantics $\mathcal{S}^*[\![prog_c]\!]$ still correctly approximates that of the original, i.e., $\mathcal{S}^*[\![prog]\!] \leq \mathcal{S}^*[\![prog_c]\!]$. Therefore, all we have to prove in order to show that the results of the analysis are correct is that $\mathcal{S}[\![prog]\!] \subseteq \mathcal{S}[\![prog_c]\!]$ (or $\mathcal{C}[\![prog]\!] \in \mathcal{S}[\![prog_c]\!]$) holds. Space limitations prevent us from including the proofs for the whole transformation algorithm. Instead, we provide the proof for the case of the virtual invocations expression, which is one of the most complex operations supported.

3.1 Correctness of a virtual invocation

The description of the standard semantics in this section is a slightly simplified version of the more formal specification described in [29]. We distinguish between two different kinds of invocations: virtual and static. Assume that calls

```
staticCallSemantics(k$m($v, v_1, \ldots, v_n$), $\sigma$)
    $s$ = signature($call$)
    $body$ = getBody($k\$m$, $s$)
    return(bodySemantics($body$, $\sigma$))

virtualCallSemantics(k?m($v, v_1, \ldots, v_n$), $\sigma$)
    $s$ = signature($call$)
    $c$ = lookup(runtime_class($v$), $s$)
    return staticSemantics(c$m($v, v_1, \ldots, v_n$), $\sigma$)

lookup($k$, $s$)
    $a = k$
    do
      if declares($a$, $s$)
        return($a$)
      $a$ = ancestor($a$)
    while ($true$)
```

```
compileStaticCall(k$m($v, v_1, \ldots, v_n$), $prog_c$)
    return k$m($v, v_1, \ldots, v_n$)

compileVirtualCall(k?m($v, v_1, \ldots, v_n$), $prog_c$)
    $s$ = signature($call$)
    $C$ = resolve($k$, $s$)
    forall $c \in C$ add to $prog_c$ the clause
        k$dyn*m($v, v_1, \ldots, v_n$) : −
           c$m($v, v_1, \ldots, v_n$)
    return k$dyn*m($v, v_1, \ldots, v_n$)

resolve($k$, $s$)
    $result = \emptyset$
    $Sub$ = subclasses($k$) $\cup \{k\}$
    forall $sub \in Sub$
        $sk$ = lookup($sub$, $s$)
        $result = result \cup sk$
    return $result$
```

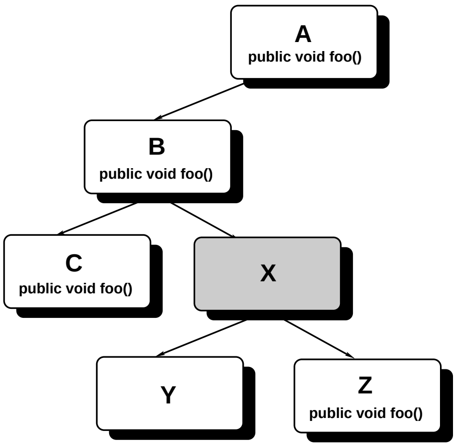
Fig. 2. Standard semantics (left) and transformation (right) of method calls

of the first type have been rewritten as `k?m`$(v, v_1, \ldots, v_n)$ and the static ones as `k$m`$(v, v_1, \ldots, v_n)$, where k is the declared type of v. Note that we rewrote the call syntax so the invoked object v is now the first actual parameter. The main difference between the two is that while in virtual invocations we need to figure out the particular class of v through a *lookup* in the class hierarchy, that operation is unnecessary in static calls since there is only one possible receiver.

In the left column of Figure 2 we present the pseudocode for the semantics of a static call (here denoted by `staticCallSemantics`) and a virtual call (here denoted by `virtualCallSemantics`). The particular signature of the invocation has to be calculated in order to distinguish which implementation to choose, since in Java (as in the Horn clauses) there can be many methods with the same name and arity, but here they will differ in the type of at least one of the formal parameters. Also, we will assume that there exists a function `runtime_class` that returns the runtime type of the object passed as parameter.

We refer to the tuple $(v, v_1, \ldots, v_n)$ as *pars*. The standard semantics of the call in the original program is $\mathcal{C}[\![k\texttt{?m}(pars)]\!]\sigma = \mathcal{C}[\![c\texttt{\$m}(pars)]\!]\sigma$, where c is the value returned by `lookup`(`runtime_class`$(v), s)$. The SLD semantics of the transformed version is $\mathcal{S}[\![k\texttt{?m}_c(pars)]\!]\sigma$, which the transformation ensures to be $\mathcal{S}[\![k\texttt{\$dyn*m}(pars)]\!]\sigma = \bigcup_i \mathcal{S}[\![c_i\texttt{\$m}(pars)]\!]\sigma$, where $c_i \in$ `resolve`(k, s). The correctness condition is now reduced to proving that c is equal to some c_i. This is equivalent to showing that `lookup`(runtime_class$(v), s) \in$ `resolve`(k, s), which can be further rewritten as `lookup`(`runtime_class`$(v), s) \in \{$`lookup`$(sub, s) \mid sub \in$ `subclasses`$(k) \cup \{k\}\}$. But the runtime type of v can only be k or a subclass of it in a type safe language as Java, and therefore the condition always holds.

Example 1. Assume a hierarchy of classes like in Figure 3. The root class `A` declares a method `foo` which is further redefined (overwritten) in subclasses `B`, `C`, and `Z`. If the original program in Figure 3a) contains a virtual invocation to `foo` in an instance declared as being of class `X`, our compiler automatically



```
someMethod(){
  X v;
  ...
  v.foo();
}
```

(a) snippet of the original program

```
SomeClass$someMethod():-
  ...
  X$dyn*foo(v).

X$dyn*foo(w):-
   B$foo(w)
X$dyn*foo(w):-
   Z$foo(w).
```

(b) snippet of the transformed program

Fig. 3. Transformation of a virtual invocation

transforms it into a call to a new method with two new clauses (methods) that represent all the possible receiver implementations for the call. Because `X` is a direct subclass of `B`, it can never inherit the original `A` implementation but only the `B` one, represented by the first clause of `x$dyn$foo`. Alternatively, any object of type `Y` and `Z` is also of type `X` and therefore we include a call to the `Z` version of `foo` in the second clause. The `C` implementation is discarded because of type incompatibility.

The process described has many interesting properties. First, it is based on assuming SLD resolution semantics for the transformed Horn clause program. This allows reusing existing analyzers without having to redefine the abstract unification operator in order to deal with language-dependent features, as in the case of virtual invocation. We implemented our Java analyses on top of the CiaoPP Prolog analyzer [16] without modifying its code, even when specific abstract domains and "builtin" definitions for Java language constructs had to be provided. A second strength is that correctness of the transformation depends only on showing that $\mathcal{C}[\![comm]\!] \in \mathcal{S}[\![comm_c]\!]$ holds for every command (and expression) in the source language. Although not trivial, the proof can be slightly modified for similar languages to Java, so neither the compiler nor the abstract domains need to be completely rewritten. In the case of Ciao, the proof is trivial since $prog_c = prog$.

4 Metainformation

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 original source are known. In other occasions the abstract domain needs information about the program that cannot be found in

```
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. 4. Vector example: source code and corresponding metainformation

the intermediate representation. Both demands are solved via *metainformation* files. We illustrate this point with the example in Figure 4, which shows an alternative version of the JDK `Vector` class. The descendant `SubVector` contains an alternative version of the `append` method. The corresponding (Ciao) code (represented as a Control Flow Graph) is shown in Figure 5; we omitted the constructor (`init`) clauses for simplicity.

Space reasons prevent us from listing a complete description of the metainformation; only hierarchy and method type tables are shown in Figure 4. In the case of the parent-child relations, the purpose is to provide access for the abstract domain to the class tree, the more obvious application being class analysis [3]. 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, avoiding *projection* and *extension* operations [5] (e.g., for blocks that share variable scope with the calling context, such as conditionals).

An *entry* method corresponds in the original program to the first clause [15] of the Java method of the same name and shares its signature, except for an extra parameter that represents the value returned. The other clauses 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 clauses are `if`, `while` or `for` loops. In the example, we can see how the `if (e==null)...else` conditional in the `Vector` implementation of `append` is converted into two different clauses, one for each branch, which actually share the same name `Vector$append#1#2` (Figure 5). In this case, the internal method is composed of two clauses which

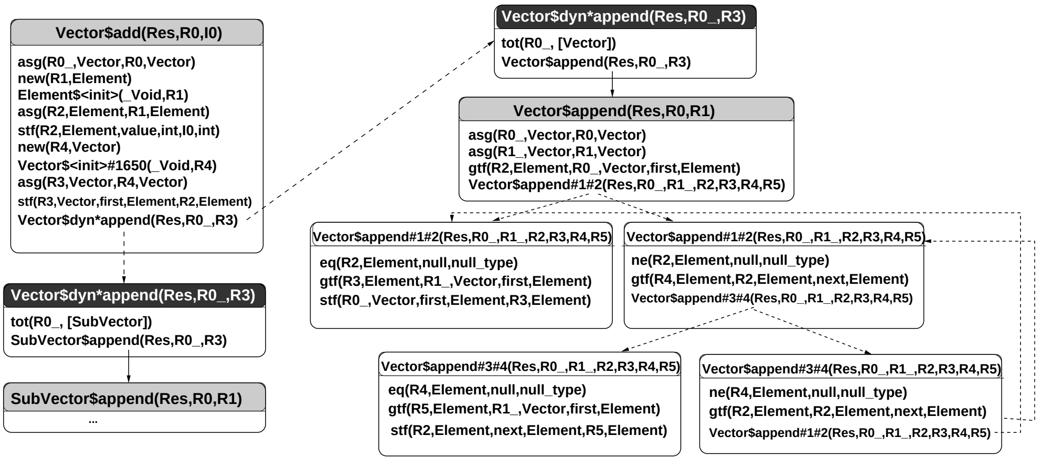

Vector$add(Res,R0,I0)
asg(R0_,Vector,R0,Vector)
new(R1,Element)
Element$<init>(_Void,R1)
asg(R2,Element,R1,Element)
stf(R2,Element,value,int,I0,int)
new(R4,Vector)
Vector$<init>#1650(_Void,R4)
asg(R3,Vector,R4,Vector)
stf(R3,Vector,first,Element,R2,Element)
Vector$dyn*append(Res,R0_,R3)
Vector$dyn*append(Res,R0_,R3)
tot(R0_, [SubVector])
SubVector$append(Res,R0_,R3)
SubVector$append(Res,R0,R1)
...
Vector$dyn*append(Res,R0_,R3)
tot(R0_, [Vector])
Vector$append(Res,R0_,R3)
Vector$append(Res,R0,R1)
asg(R0_,Vector,R0,Vector)
asg(R1_,Vector,R1,Vector)
gtf(R2,Element,R0_,Vector,first,Element)
Vector$append#1#2(Res,R0_,R1_,R2,R3,R4,R5)
Vector$append#1#2(Res,R0_,R1_,R2,R3,R4,R5)
eq(R2,Element,null,null_type)
gtf(R3,Element,R1_,Vector,first,Element)
stf(R0_,Vector,first,Element,R3,Element)
Vector$append#1#2(Res,R0_,R1_,R2,R3,R4,R5)
ne(R2,Element,null,null_type)
gtf(R4,Element,R2,Element,next,Element)
Vector$append#3#4(Res,R0_,R1_,R2,R3,R4,R5)
Vector$append#3#4(Res,R0_,R1_,R2,R3,R4,R5)
eq(R4,Element,null,null_type)
gtf(R5,Element,R1_,Vector,first,Element)
stf(R2,Element,next,Element,R5,Element)
Vector$append#3#4(Res,R0_,R1_,R2,R3,R4,R5)
ne(R4,Element,null,null_type)
gtf(R2,Element,R2,Element,next,Element)
Vector$append#1#2(Res,R0_,R1_,R2,R3,R4,R5)

Fig. 5. Call Graph for the example in Figure 4

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 clause or another). Entry clauses are marked in grey, internal ones in white; dotted arrows denote non-deterministic flows while the continuous ones symbolize deterministic calls.

Another flow transformation (*extra* clauses) tries to expose the internal structure of some complex Java features, which sometimes encode sophisticated operations. That is the case of the virtual invocations studied in Section 3. Coming back to the example in Figure 4, 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* clause; in the example we have two, `Vector$dyn*append`, marked in black. They behave in a very similar way to the conditional discussed previously, since the program flow might go through two alternative paths (clauses), one for each implementation of `append`. Each branch contains a guard (`tot`, see the first statement in each of the `Vector$dyn*append` clauses) listing the acceptable types for the callee.

It is interesting how, in an analogous way to the clause case, we introduced *extra* statements to further simplify analysis. For example, the mentioned `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 clauses that come from conditional constructions. In Figure 5 the `eq` call at the beginning of the leftmost `Vector$append#1#2` clause 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 [23].

```
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.lang.String,"English",java.lang.String),
    ret.

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

Fig. 6. Transformation for dynamic dispatch in Java

5 Explicit semantics in other OO languages

Our framework can be adapted to other languages apart from Java (and Ciao), especially for those like C# that share similar syntax and statement semantics to Java. The examples in Figures 6 and 7 illustrate this point. In Figure 6, the value returned by the `getDefaultLanguage` invocation in the `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 [15]. The C# language is quite similar in most aspects, but polymorphic invocations have been further refined (and complicated). In Figure 7 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 6 and 7 how the Horn clause representation is almost identical in both cases, except for the bodies of the two `Location$dyn*getDefaultLanguage` 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

```
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_).
```

Fig. 7. Transformation for dynamic dispatch in C#

`Location$getDefaultLocation` while invocations on objects of (exactly) class `China` are redirected to `China$getDefaultLocation`.

The abstract domain is not required to know anything about 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.

6 Experimental results

We have completed a preliminary implementation of our framework within the CiaoPP preprocessor [16]. CiaoPP offers a parametric and efficient top-down analysis engine with a good number of abstract domains, including the ones illustrated in this section. The efficiency of the algorithm relies on keeping dependencies between different predicates during analysis so that only the really affected parts need to be revisited after a change during the fixpoint process. In addition, recomputation is avoided using *memoization*[12]. Another characteristic is that it is *multivariant* (i.e., abstract calls to a given predicate 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.

We have performed two experiments with our framework using the benchmarks corresponding to the JOlden suite [9]. The first experiment is summarized in Figure 8 and shows the scalability of the transformation phase. The first three columns contain basic metrics about the application: number of classes (k), methods (m) and instructions (i). Since the latter corresponds to the bytecode representation of the source, we also list how many program points (pp)

name	k	m	i	pp	ct
`jolden.health.Health`	8	30	637	933	1.1
`jolden.bh.BH`	9	70	1208	1739	3.2
`jolden.voronoi.Voronoi`	6	73	988	1340	2.2
`jolden.mst.MST`	6	36	445	665	0.1
`jolden.power.Power`	6	32	1017	1270	2.1
`jolden.treeadd.TreeAdd`	2	12	193	274	2.0
`jolden.em3d.Em3d`	4	22	447	669	0.1
`jolden.perimeter.Perimeter`	10	45	543	814	0.1
`jolden.bisort.BiSort`	2	15	323	476	0.1
`jolden.all.All`	50	317	5839	7251	11.0

Fig. 8. Statistics of the transformation phase.

are present in the Horn clause program analyzed. This metric slightly differs from the number of instructions in the sense that extra clauses and builtins make it somewhat larger; *pp* also provides a better approximation of the size and complexity of the program analyzed because the semantics of the object-oriented program is made explicit, as seen in Section 2. The fifth column (*ct*) shows the time invested (given in seconds) in transforming the input program and producing the Horn clause version and the metainformation.

The second experiment shown in Figure 9 illustrates the scalability, efficiency, and precision of the analysis component of our framework. We first use a simple abstract domain, Nullity, capable of approximating which variables are definitely null and which ones definitely point to a non-null location. The second abstract domain is a Class Hierarchy Analysis [3], 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 Class Hierarchy Analysis shows the scalability of our framework for a domain with non-linear worst-case complexity in its operations. Additionally, it also reflects the usefulness of *metainformation* files since they are required by the CHA domain in order to access the hiercharchy tree. The columns labeled pp' show the number of program points reachable by the analyses. Therefore, pp' may differ from *pp* because the number of analyzed program points is not always the total number of program points in the program: some commands are found to be 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. Columns *ast* provide the total number of these abstract states inferred by analyses. The level of multivariance is the ratio ast/pp', presented in columns *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 which includes the extraction of metainformation for each method in the Horn clause program and the construction of the class hierarchy) and *at* (analysis time); both are also given in seconds.

The benchmarks have been tested in both experiments on a Pentium M 1.73Ghz with 1Gb of RAM , and averaging several runs after eliminating the best and worst values. We chose to show separately the total times of the two phases

		Nullity				CHA			
	pt	pp'	ast	st	at	pp'	ast	st	at
`jolden.health.Health`	2.1	921	5836	6.3	9.6	933	3542	3.8	52.1
`jolden.bh.BH`	2.2	1739	12384	7.1	50.1	1739	4757	2.7	59.4
`jolden.voronoi.Voronoi`	2.2	1277	5492	4.3	11.5	1340	5147	3.8	81.3
`jolden.mst.MST`	2.1	496	1503	3.0	1.1	665	1609	2.4	11.6
`jolden.power.Power`	2.1	1270	10560	8.3	29.9	1270	2908	2.3	32.7
`jolden.treeadd.TreeAdd`	2.0	274	880	3.2	0.6	274	729	2.6	6.1
`jolden.em3d.Em3d`	2.0	669	5565	8.3	0.9	669	3320	4.9	49.5
`jolden.perimeter.Perimeter`	2.1	814	2653	3.2	1.7	814	3731	4.5	25.0
`jolden.bisort.BiSort`	2.1	476	3353	7.0	5.8	476	1614	3.4	15.6
`jolden.all.All`	2.6	7188	48476	6.7	145.9	7251	29586	4.1	391.2

Fig. 9. Statistics for the Nullity and Class Hierarchy (CHA) domains.

(transformation and analysis) because we expect the transformation process to be fully run only once. Later executions can use incremental compilation for those files that changed, so that the overhead of the preprocessing phase should be almost negligible in medium and large programs. Although the same approach can be taken for the analysis [27], the current implementation is not incremental.

7 Related work

Most of the previous research in analysis of object-oriented programs 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, 28]) or statically verify certain properties about the runtime behavior of the code (e.g., [14, 19]). In contrast to this concentration and progress on the development of new, refined domains there has been comparatively little work on the formal specification of the *intermediate language* to which the analyzed program is transformed or in the application of existing logic programming techniques. In [24] 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. Also, the technique is presented from a more informal perspective and no analysis is attempted over the transformed logic programs.

More closely related to ours is the work presented in [1], which draws in part on the ideas of [25]. The authors also 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 [6], 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. As in our case, the analysis and verification experiments are performed using the CiaoPP [16] tool.

While the approach of [1] 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 clearly varies significantly depending on the input program. Our approach 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 our 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, our 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 our approach also allows us to support a much larger subset of the language than in [1], including many features such as exceptions, inheritance, interfaces, etc. Furthermore, since the fixpoint algorithm is sensitive to the size and characteristics of the intermediate representation, the fact that our direct translation guarantees a compact intermediate representation can arguably result in a more scalable solution. For example, in this work (see Section 6) we have been able to report on examples that are about twenty times larger and can be analyzed in less time.

In most of the (non CLP-based) abstract interpretation framework for analysis of Java (e.g., [4, 7]) 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 [20]. In [26] the authors use a framework that is closely related to Gaia [8]. However, the intermediate representation is not described and the semantics of the interprocedural operations is again tied to the Java language. Also, the benchmarks used are smaller than those that we report on. The more recent Julia framework [30] is intended to be generic from the point of view of domains but once more also targets Java as unique source language. This framework is capable of analyzing large programs in a top-down way, as in our approach, the main other difference being that we support multivariance, inherited from the CiaoPP analyzer. Finally, in [22] another interesting generic static analyzer for the modular analysis and verification of Java classes is presented. The algorithm presented is also top down but is again tailored specifically to Java source.

8 Conclusions and future work

We have presented a transformation-based framework for analysis of object-oriented programs, which is generic in terms of the source language and abstract domain in use. The framework consists of a two-step process: a transformation of the program into a set of Horn clauses that represents a correct approximation of its standard semantics, and a mature and sophisticated fixpoint algorithm. We claim that our approach is flexible in the sense that the first phase decouples the fixpoint algorithm from any language-dependent feature. Furthermore, our experimental evaluations support the scalability of our framework showing results for medium-sized programs as well as its efficiency analyzing them in a reasonable amount of time, and precision showing high rates of multivariance.

We have performed some promising experiments on an ample subset of Java, as shown in this paper, but our aim is to support the full Java language. Also, we are currently incorporating more sophisticated abstract domains (e.g., points-to/sharing analysis). Moreover, we expect to increase the scalability of our approach, analyzing larger programs than shown in this paper. To this end, we are studying the inclusion of modular and incremental features in our fixpoint algorithm.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 124–139. Springer-Verlag, January 2007.
2. Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.
3. David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA*, pages 324–341, 1996.
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. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
6. 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`.
7. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.
8. 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.
9. JOlden Suite Collection. `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.
10. 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.
11. 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.
12. S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.
13. Christian Fecht. Gena - a tool for generating prolog analyzers from specifications. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 418–419, London, UK, 1995. Springer-Verlag.

14. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of VMCAI*, LNCS. Springer-Verlag, 2005.
15. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.
16. 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.
17. R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.
18. 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.
19. Xavier Leroy. Java bytecode verification: An overview. In *CAV*, pages 265–285, 2001.
20. Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
21. 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.
22. Francesco Logozzo. Cibai: An abstract interpreation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*. To appear, Jan 2007.
23. 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. To appear.
24. 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.
25. 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.
26. Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java*. PhD thesis, Catholic University of Louvain, 2004. Dept. of Computer Science.
27. 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.
28. Erik Ruf. Effective synchronization removal for java. In *PLDI*, pages 208–218, 2000.
29. Stefano Secci and Fausto Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, pages 320–335, 2005.
30. 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.
31. 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.