JULIA: A Generic Static Analyser for the Java Bytecode

Fausto Spoto

Dipartimento di Informatica, Verona, Italy
fausto.spoto@univr.it

Abstract. We describe our software tool JULIA for the static analysis of full Java bytecode, for optimisation as well as verification. This tool is *generic* since abstract domains (analyses) are not part of JULIA but rather external classes that specialise its behaviour. Static analysis is performed through a denotational or constraint-based fixpoint calculation, focused on some program points called *watchpoints*. These points specify where the result of the analysis is useful, and can be automatically placed by the abstract domain or manually provided by the user. JULIA can be instructed to include a given set of library Java classes in the analysis, in order to improve its precision. Moreover, it gives abstract domains the opportunity to approximate control and data-flow arising from exceptions and subroutines.

1 Introduction

This paper describes the JULIA software tool that we have developed in order to apply the abstract interpretation technique [10] to the static analysis of Java bytecode [16]. The motivation underlying our effort is to provide a software support for optimising, verifying and reasoning upon Java bytecode applications before they are run, and when their source code is not available or does not even exist. Forseeing the behaviour of programs, before their actual execution, becomes more and more relevant as such programs increase in complexity and get used in critical situations such as medical operations, flight control or banking cards. Being able to *prove*, in an automatic way, that programs do adhere to their functional specifications is a basic factor to their success. This is particularly true for applications written in Java bytecode, distributed on the Internet or used inside a smart card, and hence potentially harmful to the client. In this perspective, analyses for security are attracting more and more interest [20]. But the information inferred by a static analysis can also be used for optimisation, documentation and debugging.

Abstract interpretation [10] has served as a primary framework for the formal derivation of static analyses from the property of interest. It features the ability to express correctness as well as optimality of a static analysis. It consists in executing the program over a *description* (the *abstract domain*) of the actual run-time data. By saturating all possible program execution paths, we get a

domain	description	size
`rt`	rapid type analysis (a kind of class analysis) [5]	743
`ps`	denotational set-based class analysis [18, 24]	841
`cps`	constraint and set-based class analysis [18, 24]	775
`er`	denotational escape analysis [7, 15]	1016
`cer`	constraint-based escape analysis [7, 15]	1030
`bni`	information-flow analysis with Boolean formulas [13, 20]	2659
`static`	static initialisation analysis	539

Fig. 1. The abstract domains currently implemented inside JULIA. Their size is given in number of Java source code lines, comments included.

provably correct picture of its run-time behaviour, which is more or less precise, depending on how much the chosen *description* approximates the actual data.

The goal of JULIA was to fulfill the following criteria:

- the analyser is *generic i.e.,* it does not embed any specific abstract domain but allows instead the addition of new abstract domains as external classes;
- the analyser allows one to specify the set of classes which must be analysed, called the *application classes*. They must not change from analysis-time to run-time (through dynamic loading); this would otherwise break the correctness of the analysis;
- the abstract domain developer has his work simplified as much as possible. Namely, he must be able to apply the formal framework of abstract interpretation to define its abstract domain, even for the most complex bytecodes and in the presence of all the intricacies of the Java bytecode. All he needs to do is to provide implementations of the abstract operations corresponding to the concrete bytecodes, together with a bottom element and a least upper bound operator;
- the analysis is *localised i.e.,* its cost is proportional to the number of program points where the abstract information must be computed (the *watchpoints*);
- the analyser does not impose any constraint on the precision of the abstract domain. Namely, it allows a given abstract domain to exploit the flow of control due to exceptions and subroutines to get a more precise analysis, yet allowing another domain to disregard the same flows and get a less precise analysis. Precision remains a domain-related issue [10];
- the analyser uses efficient techniques for computing the fixpoint needed for the static analysis [10]. These techniques are domain-independent, so that the abstract domain developer does not need to care about how the fixpoint is computed for its abstract domain.

JULIA is free software [23]. It currently includes seven abstract domains, which are described in Figure 1. *Class analysis* is used to transform some virtual calls into static calls, whenever their target is statically determined [25]. *Escape analysis* determines which creation instructions can safely allocate objects in the activation stack instead of the heap, since those objects will never outlive the

method which creates them [7, 15]. *Information-flow analysis* approximates the flow of data in a program, permitting one to spot violations of non-interference conditions in the analysis of security [20]. *Static initialisation analysis* determines the set of classes which are definitely initialised in a given program point, so that references to such classes do not induce a call to their static initialiser. We discuss it in Section 9 as a simple example of abstract domain.

The paper is organised as follows. Section 2 describes related work. Sections 3 to 8 show how each of the previous criteria have been attained with JULIA. Section 9 shows an example of abstract domain which can be plugged inside JULIA. Section 10 discusses the application of JULIA to multi-threaded programs. Section 11 presents the cost in time for the analysis of some non-trivial Java bytecode applications. Section 12 concludes.

2 Related Work

Because of the actual complexity of the Java bytecode, static analysers for full Java bytecode have not been developed intensively yet.

A decompilation tool, such as SOOT [26], is often used as the front-end of a static analyser for Java bytecode. Currently, class analyses similar to rapid type analysis are implemented inside SOOT. Decompilation is problematic when the bytecode is not the result of the compilation of Java, and maybe contains some *exotic* features of the Java bytecode that have no direct counterpart in Java, such as overlapping or recursive exception handlers (*i.e.,* catching exceptions thrown by themselves), or recursive Java bytecode subroutines (which cannot be decompiled into `finally` clauses). The INDUS tool [1] is an analyser based on SOOT. It currently includes some flavour of class analysis, escape analysis and analyses targetted for concurrent programs, to be coupled with a model-checker.

In [17], a set of tools and components for building language runtimes is shown. The bytecode they consider includes the Java bytecode as a special case. Such tools have been used to implement some static analyses as well. Their code preprocessing is much lighter than ours and consequently much faster.

A generic analyser for a subset of Java (rather than Java bytecode) is described in [19]. It has only be applied to small programs. It allows completely relational, flow and context-sensitive static analyses. In this sense it is quite close to our work.

Various flavours of rapid class analyses have been implemented in [25]. The tool is not available and genericity is not mentioned. Some benchmarks are similar to ours and their analysis seems faster than ours.

Escape analysis has been implemented through specialised analysers for Java source code only, rather than Java bytecode. For instance, the analysis in [7] works inside a commercial Java compiler. The construction is specific to escape analysis, and it cannot be immediately applied to other analyses. A more complex escape analysis has been implemented in [9] and it seems to perform like ours. However, it is not a generic nor localised tool.

A generic analyser for the Java Card bytecode has been defined in [8]. The approach is fascinating, since it is based on the *automatic* derivation of a correct static analyser from its same proof of correctness. However, the Java Card bytecode is simpler than the Java bytecode. Moreover, exceptions are not considered. No examples of analysis are shown. Hence, actual analysis times are unknown.

JDFA [2] performs constant propagation and liveness analysis for variables, but none of the analyses we show in Figure 9. JNUKE [4] performs *dynamic* rather than *static* analysis. No sensible comparison is hence possible with JULIA

3 A Generic Analyser

Being generic is a useful feature of a modern static analyser. Current programming languages, such as the Java bytecode, are so complex that the development of a new static analysis is hard and error-prone. However, different static analyses do share a lot. The preprocessing phase (Section 5) and the fixpoint computation (Section 8) are the same for every abstract domain. And they represent by themselves most of the development effort of a static analysis. It is hence convenient to develop and debug them once and for all, and to see new abstract domains as plug-in's which are added to the static analyser in order to specialise its behaviour.

Genericity requires however to provide an interface between the code preprocessor and the analyser. We solved this problem by specifying all bytecodes as state transformers. For each state transformer the abstract domain provides an approximation (see for instance Section 9).

To fulfill this requirement, we structured JULIA as in Figure 2. A code preprocessor, called ROMEO, feeds the preprocessed code into a generic fixpoint engine called JULIET. The latter uses an external module, the abstract domain, to abstract every single bytecode, but uses its own fixpoint strategies, independent from the abstract domain. The BCEL library [12] is a low-level interface to `.class` files.

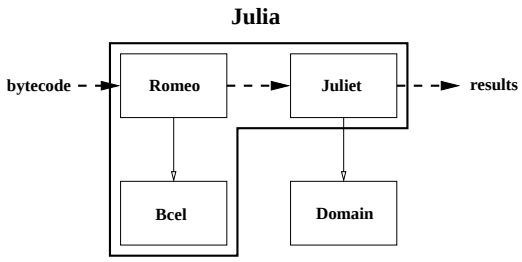

Fig. 2. The structure of JULIA.

Figure 1 shows that genericity leads to small abstract domain implementations, and hence faster and simpler development.

4 Application Classes

The Java Virtual Machine loads classes dynamically as they are needed during the execution of a program. Hence, we have no guarantee that the classes that will be loaded at run-time will correspond to those that were present in the system

during the static analysis. We might think to analyse a class without assuming anything about its surrounding environment. Any reference to an external class is treated through a *worst-case assumption* [11] claiming that nothing is known about its outcome. This is definitely correct, but often useless in an object-oriented language, where classes are tightly coupled through virtual method invocations, field accesses and constructor chaining. This approach results in static analyses of very little precision.

Instead, we follow here the solution to this problem used in the decompilation tool Soot [26], which allows one to make explicit assumptions about which classes (called *application classes*) are not allowed to change from analysis-time to run-time. As a consequence, we can inspect them during the analysis and gather abstract information which improves the precision of the analysis.

Application classes are typically those of the application we are analysing. Libraries are not considered application classes, usually. Hence, any reference to a library class is resolved through the worst-case assumption. However, stronger hypotheses than the worst-case assumption can be made. For instance, in [25], the set of application classes is assumed to be closed *wrt.* subclassing. This improves the precision of the analysis.

We assume that every abstract domain plugged inside Julia decides how to deal with references to non-application classes. It can use a worst-case assumption or other, stronger hypotheses. This must be clearly stated in its definition, so that the user of Julia can judge whether such hypotheses are realistic or not for his own analyses. For instance, our abstract domain `rt` for rapid type analysis assumes that application classes are downward closed, as in [25], while our domain `er` for escape analysis assumes that non-application classes have the same method and field signatures as in the system used for the analysis; their implementation can however change.

5 Bytecode Simplification (Preprocessing)

The application of abstract interpretation to a complex language such as the Java bytecode is a real challenge. This is because abstract interpretation allows us to derive a static analysis from a specification of the concrete semantics of a program given as an (operational or denotational) input/ouput map. But some Java bytecodes cannot be immediately seen as input/output maps. Examples are the control-related bytecodes such as `goto` or `lookupswitch`. Other bytecodes are input/ouput maps, but they are so complex that the application of abstract interpretation is very hard and error-prone. Examples are the four `invoke` bytecodes. Moreover, exceptions break the input/output behaviour of a bytecode, since for some input state there is no output state, but rather an exceptional state. We want to spare as much as possible the abstract domain developer from knowing the intricacies of the bytecode, and allow him to define correct (and potentially optimal) operations on the abstract domain corresponding to the concrete bytecodes.

To this goal, we apply a light *preprocessing* to the Java bytecode, in the sense that most of the bytecodes, those which are already input/output maps, are not transformed. The result is a graph of basic blocks [3] of a simplified Java bytecode, which we call JULIET bytecode. Edges between basic blocks model control. Conditional jumps use new *filter* bytecodes, which play exactly the same role as the `assume` statements used in [6]. These filter bytecodes can be used to improve the precision of a static analysis, as [6] shows. An example is in Section 9. They can also be conservatively abstracted as no-ops. Figure 3 shows a bytecode and its translation into a graph of basic blocks, where the `goon` new filter bytecodes select the execution path on the basis of the outcome of the `if_icmplt` test.

For instance, static initialisation of a class κ (and of all its superclasses) occurs the first time that a `new` κ, `getstatic` κ, `pustatic` κ or `invokestatic` κ bytecode is executed [16]. If this is the case, κ's static initialiser and those of κ's superclasses are run. This behaviour must be considered by a static analyser that faithfully respects the semantics of Java. Hence we compile a `new` κ bytecode into the JULIET code in Figure 4. The new `initialise` κ bytecode marks a given class (and all its superclasses) as already initialised. The new `call` $\kappa.\langle$`clinit`$\rangle$ bytecode stands for a call to the static initialiser of κ and to those of all its superclasses. The new `resolved_new` κ bytecode behaves like the old `new` κ bytecode, but it does not check for initialisation. All bytecodes in Figure 4 are now input/output maps. This *compilation* of the original `new` bytecode simplifies the subsequent static analysis. An example is in Section 9. In a similar way, an `invoke` instruction is *compiled* into a JULIET code which *explicitly* resolves the class, then resolves the method, then looks for the target method of the call (through a compiled lookup procedure), then creates the activation frame for the method, then calls the selected method and finally moves the return value of the called method into the operand stack of the caller. The domain developer does not need to know how a method is resolved and looked up by the Java Virtual Machine [16]. He does not need to know about visibility modifiers, nor about the exceptions which might be thrown during the method call. Everything has been compiled, he just has to abstract the resulting code. Also exception handlers are *compiled* into the code.

```
0 iconst_0     8 iload_1
1 istore_1     9 bipush 100
2 goto 8       11 if_icmplt 5
5 iinc 1 1     14 return
```

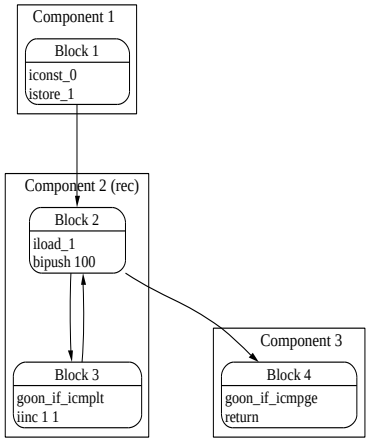

Fig. 3. A piece of bytecode and its graph of basic blocks.

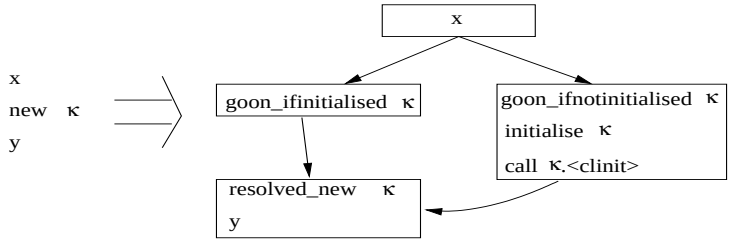

Fig. 4. The compilation of a `new` bytecode.

Since JULIET bytecode is derived by splitting complex Java bytecodes, it is more fine-grained than Java bytecode. Hence all properties of the Java bytecode can be expressed as properties of JULIET bytecode. In particular, we claim that the resulting JULIET bytecode has the same concrete semantics as the original Java bytecode. We are confident in this result since most of the Java bytecodes are not changed during the translation. The most complex bytecodes are translated by following their operational semantics in the Java Virtual Machine official documentation [16].

Since a graph of basic blocks of bytecode is used, we can fit all the complex features of the Java bytecode into that formalism (see Section 2). Namely, edges connecting the blocks of code let us represent exception handlers of any shape and recursive subroutines.

6 Localisation

The information computed by a static analysis is typically useful in some special program points only, called *watchpoints*. The number and position of the watchpoints depends on the way the abstract information is used to reason about the program. For instance, in the case of class analysis we want to know which virtual calls are actually *determined i.e.,* always lead to the same target method [25]. Hence a watchpoint must be put before the virtual calls of the program, so that we can use the abstract information collected there to spot determinism. In the case of escape analysis, we *bracket*, between an entry and an exit watchpoint, the methods containing a `new` bytecode. This allows us to spot the `new` bytecodes creating objects that never *escape* their creating method. Those objects can hence be allocated in the activation stack instead of the heap [7, 15].

Since, in general, watchpoints are *internal* program points, the denotation computed by a static analyser cannot be just an input/ouput map. A richer structure is needed. Moreover, it is desirable that the cost of the analysis scale with the number of watchpoints, in which case we say that the static analysis is *focused* or *localised*. This is important because it allows us to concentrate the typically little computational resources of time and memory on the watchpoints only, instead of the whole program. Hence larger programs can be analysed.

A general framework for focused static analyses was developed in [22] for a simple high-level language. In [21] we show how it can be applied to the Java bytecode, by exploiting the same simplification of the bytecode highlighted in Section 5. We have then implemented this localised analysis inside JULIET, the fixpoint engine of JULIA. Our experiments confirm that the resulting static analyses outperform their unfocused versions [21].

A positive property of our focused analyses is that the abstract domain developer is not aware of how the focusing technique works [22, 21]. He develops his abstract domain as for a simple input/ouput analysis.

Abstract domains for JULIA put watchpoints automatically, since they are aware of the goal of the analysis they implement. We report an example in Section 9. But the user can put watchpoints explicitly if he wants.

JULIA runs also unfocused, constraint-based analyses. Although they often perform worse than their focused versions, such analyses are often simpler to develop. For instance, we have developed both class and escape analysis in focused (denotational) and unfocused (constraint-based) way, which resulted in the focused `ps` and `er` and in the unfocused `cps` and `cer` domains (Figure 1). This does not mean that `cps` and `cer` are useless. Constraint-based static analyses can be made (completely or partially) flow-insensitive by merging (all or some) variable approximations. The same is much harder to achieve with denotational abstract domains. Hence, if flow-sensitivity is not important, as experiments show for class and escape analysis, then constraint-based analyses provide fast static analyses. In other cases, such as static initialisation analysis and control-flow analysis, flow-sensitivity is very important, so denotational, localised static analyses should be preferred.

7 No Constraints on Precision

Abstract interpretation [10] entails that the precision of a static analysis is domain-related. The precision of different domains can be formally compared without considering their implementation in the analysis. It is desirable that this situation be maintained in practice. Hence the analyser should not limit the precision of the analysis because of spurious constraints due to the way the analysis is implemented.

Many static analyses compile the source program into a constraint whose solution is an approximation of the abstract behaviour of the program. This has the drawback that a given variable of the constraint is used to represent the approximation of a program variable *throughout its whole existence*. But a program variable can hold different values in different program points (*flow sensitiveness*). Hence, this technique merges all those approximations in the same variable, thus imposing a limit to the precision of the analysis.

This situation improves by using *variable splitting* or *variable indexing* in order to multiply the variables used in the constraint to represent a given program variable. This means that the domain developer (who writes the compilation of the source program into a constraint) must be aware of the problem. Moreover,

if a given method is called from different contexts, the same approximation is still used for all such calls. *Method cloning* [27] can be used here, which further complicates the analysis. Consequently, to the best of our knowledge, it has never been implemented for the Java *bytecode*.

We prefer instead to stick to the traditional definition of abstract interpretation [10], so that static analysis works by computing a denotational fixpoint over data-flow equations derived from the structure of the program. This results in a flow and context sensitive analysis. The abstract domain might decide not to exploit this opportunity of precision, but no constraint is imposed by the analyser itself.

Similarly, the preprocessing of the Java bytecode performed by JULIA (Section 5) exposes the flows of control arising from the lookup procedures for virtual method invocations, from exceptions and from subroutines. Again, it is an abstract domain matter to decide whether those flows of control must be selectively chosen, in order to get a more precise analysis, or rather they must be considered as non-deterministic choices without any preference, thus getting a less precise analysis. Often, it is just a matter of trade-off between precision and cost of the analysis. For instance, the `rt` domain for rapid type analysis chooses between those flows non-deterministically, while the abstract domain `ps` for class analysis selects them in order to drive the analysis and collect more precise information.

8 Fixpoint Engine

Computing a *global* fixpoint over data-flow equations can be computationally expensive or even prohibitive. We have used some techniques to tame this complexity issue.

The first consists in building the maximal strongly connected components of the call graph of basic blocks and methods. These components are then sorted topologically and used to build the analysis of the whole program through *local* fixpoints. There is a local fixpoint for each recursive component. For instance, Figure 3 contains three components. The static analyser works by first computing the analysis for component 3 (which does not require any fixpoint) then the analysis for component 2 (which does require a local fixpoint) and, finally, the analysis for component 1 (without any fixpoint).

A second technique was originally developed for the static analysis of logic programs and is known as *abstract compilation* [14]. During the abstract interpretation process, a given bytecode is repeatedly abstracted because of loops and recursion. It becomes hence convenient to abstract it once and for all, and compute the fixpoint over an *abstract program i.e.,* a program where each bytecode has been substituted with its abstraction into the abstract domain. For instance, the code shown in Figure 3 first gets abstracted into the chosen abstract domain, as Figure 5 shows. Then, the fixpoint mechanism is applied as before.

Abstract compilation can be applied repeatedly. If the analysis a of a piece of code c is *stable i.e.,* it will not change anymore during the analysis, then c can be substituted with a. For instance, during the computation of the local fixpoint

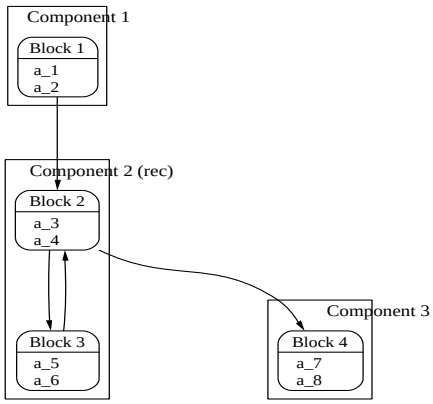

Fig. 5. The abstract compilation of the program in Figure 3.

for component 2 in Figure 5 we compute repeatedly the sequential composition of a_3 and a_4 and of a_5 and a_6. It is hence convenient to compute these compositions once and for all, before the fixpoint mechanism starts.

It must be noted that the abstract domain designer is not aware of the use of strongly connected components and abstract compilation, which are domain-independent techniques. Domain-specific fixpoint acceleration techniques will be added in the future to JULIA, through *widening* operators [10]. They are essential for using abstract domains with infinite ascending chains, such as polyhedra.

9 Writing Abstract Domains for Julia

New abstract domains can be developed and plugged inside JULIA. The domain developer must define the abstract counterparts of the concrete bytecodes, a bottom element, a least upper bound operator and how the watchpoints are put in the source code to perform the analysis implemented by the domain. Abstract domains for JULIA are Java classes that extend `juliet.BottomUpDomain`, if localisation is used (Section 6), or `juliet.ConstraintDomain`, otherwise. We describe here an abstract domain of the first kind, which is used for *static initialisation analysis*. This analysis collects the set of classes which are definitely initialised in each given program point. This information is useful before a `goon_ifinitialised` or `goon_ifnotinitialised` bytecode (Figure 4), which might be found to be redundant. If that is the case, they can be safely removed from the code, so that subsequent static analyses (class, escape analysis, *etc.*) will run on a simplified program, and be potentially faster and more precise. The set of initialised classes enlarges only when an `initialise` κ or a `goon_ifinitialised` κ bytecode is executed (Figure 4), and never shrinks. Indeed, we know that, after both bytecodes, class κ and all its superclasses are

initialised. From this idea, we implemented the abstract domain for static initialisation shown in Figures 6, 7 and 8.

In Figure 6, we see that an abstract element contains the set `initialised` of definitely initialised classes. The `init` method prepares the domain for the static analysis. In our case, it puts a watchpoint in front of each `goon_ifinitialised` bytecode. We do not put any watchpoint in front of the `goon_ifnotinitialised` bytecodes, since they are always coupled with a `goon_ifinitialised` (Figure 4), so one watchpoint is enough for both. The method `equals` checks when two abstract domain elements are the same. In our case, they must contain the same set of initialised classes.

Our `static` domain continues in Figure 7. The `toString` method is used for printing an abstract domain element, while the `output` method provides statistical information at the end of a static analysis, by reporting the number of redundant `goon_ifinitialised` tests. There is also a method that computes the `bottom` element of the abstract domain, another that `clones` an abstract domain element and another that computes the least upper bound (`lub`) of two abstract domain elements. This last operation consists in the intersection of the two sets of initialised classes, since a class is definitely initialised after a conditional if it is definitely initialised at the end of *both* its branches. The `compose` method computes the sequential composition of two abstract domain elements. It computes the union of the classes which are definitely initialised in both. The `analyse` method computes the abstraction of a bytecode `in` occurring in a basic block `cb` (Section 5). Normally, it returns an abstract domain element whose set of initialised classes is empty, except for the `initialise` κ and `goon_ifinitialised` κ bytecodes, whose execution initialises κ and all its superclasses. To this purpose, the private method `add` scans the class hierarchy from κ upwards. It stops as soon as a non-application class is found (Section 4), since we do not want to assume anything about non-application classes, not even their hierarchy. The last method, `applyAnalysis` in Figure 8, is called at the end of the analysis. It scans the watchpoints that we have put before a `goon_ifinitialised` κ bytecode, and checks whether the set of initialised classes there includes κ. If this is the case, the corresponding test for initialisation is useless since it will always succeed.

Appropriate `import` statements must be put at the beginning of Figure 6. The actual code is in the file `_static/Static.java` of JULIA, optimised by using bitmaps instead of `HashSet`s (the underscore in `_static` is needed since a keyword cannot be a package name in Java). However, the code in Figure 6, 7 and 8 is perfectly working, and able to analyse all valid Java bytecode. For instance, you can apply it to itself (*i.e.,* its compiled code) in *library mode i.e.,* by assuming that all its *public* methods may be called from outside. The result is that 11 out of a total of 36 static initialisation tests are found to be redundant. In Figures 6, 7 and 8, they are underlined. Namely, the recursive static calls inside `putWatchpoints` do not need any initialisation of the `Static` class, since it has already been initialised by the first static call. The `StringBuffer` creation inside `toString` does not need any initialisation of `StringBuffer` since it has already been initialised by the creation of a `StringBuffer` object in the previous line.

```
public class STATIC extends BOTTOMUPDOMAIN {

 private HASHSET initialised = new HASHSET(); // the classes initialised
 private static HASHSET all; // all classes checked for initialisation
 private static int useless, total; // useless/total initialisation tests
 private Static(HASHSET initialised) { this.initialised = initialised; }

 public void init(LOADER loader) {
  // we put the watchpoints in each method
  for (int i = 0; i < loader.program.methods.length; i++)
   putWatchpoints(loader.program.methods[i]);
 }

 private static void putWatchpoints(METHODCODE mc) {
  // we add the watchpoints in the instructions
  for (int i = 0; i < mc.blocks.length; i++)
   mc.blocks[i].ins = putWatchpoints(mc,mc.blocks[i].ins);
 }

 private static Instruction putWatchpoints(METHODCODE mc, INSTRUCTION ins) {
  if (ins instanceof SEQ) { // a sequence
   ((SEQ)ins).left = putWatchpoints(mc,((SEQ)ins).left);
   ((SEQ)ins).right = putWatchpoints(mc,((SEQ)ins).right);
   return ins;
  } // a watchpoint is put in front of every GoOnIFINITIALISED
  else if (ins instanceof GOONIFINITIALISED) {
   all.add(mc.ot); // we add this class to the set of all classes
   return new SEQ(new WATCHPOINT(ins,mc),ins);
  }
  else return ins;
 }

 public boolean equals(BOTTOMUPDOMAIN d) {
  return initialised.equals(((STATIC)d).initialised);
 }
```

Fig. 6. An abstract domain for static initialisation analysis.

```
public String toString() { // prints the set of initialised classes
  String res = "{"; Iterator it = initialised.iterator();
  while (it.hasNext()) {  res += it.next().toString();
    if (it.hasNext()) res += ",";  }
  return res + "}";
}

public String output() { // final statistics
  return super.output() +
    " Number of redundant initialisation tests/total . . . . . " +
    useless + "/" + total + "\n";
}

public BottomUpDomain bottom() { // all classes are initialised
  return new Static(new HashSet(all));
}

public BottomUpDomain id() { // clones this object
  return new Static(new HashSet(initialised));
}

public BottomUpDomain lub(BottomUpDomain d) { // result = lub(this,d)
  // we let initialised := intersect(initialised,other)
  HashSet temp = new HashSet(initialised);
  temp.removeAll(((Static)d).initialised);
  initialised.removeAll(temp);
  return this;
}

public BottomUpDomain compose(BottomUpDomain d) { // d, then this
  // we let initialised := union(initialised,other)
  initialised.addAll(((Static)d).initialised);
  return this;
}

public BottomUpDomain analyse(Instruction in, CodeBlock cb) {
  Static res = new Static();
  // only the following bytecodes affect the set of classes
  if (in instanceof INITIALISE)
    res.add(((INITIALISE)in).ot);
  else if (in instanceof GoOnIFINITIALISED)
    res.add(((GoOnIFINITIALISED)in).ot);
  return res;
}

private void add(ObjectType ot) {
  // adds, to initialised, ot and all its superclasses...
}
```

Fig. 7. An abstract domain for static initialisation analysis (cont'd).

```
  public void applyAnalysis(ProgramCode program) {
    super.applyAnalysis(program); // we compute "denotation"
    if (denotation == null) return; // no main or callee methods: we have finished
    HashMap wps = denotation.getWatchpoints();
    Iterator it = wps.keySet().iterator();
    Watchpoint wp;
    Static st;
    useless = total = 0;
    while (it.hasNext()) { // for each watchpoint, we get its abstract information
      wp = (Watchpoint)it.next(); st = (Static)wps.get(wp);
      if (wp.label instanceof GoOnIFINITIALISED) {
        total++;
        // we check whether this test is redundant
        if (st == null ||
            st.initialised.contains(((GoOnIFINITIALISED)wp.label).ot))
          useless++;
      }
    }
  }
}
```

Fig. 8. An abstract domain for static initialisation analysis (cont'd).

The access to the static `total` field of `Static` inside `output` does not require initialisation of `Static` since it has already been initialised by the previous access to `useless`. Similarly in `applyAnalysis` for the static fields `denotation`, `total` and `useless` (the ++ operator performs two accesses).

Our `static` domain can be made more precise by observing that if an instance method of a class κ is called, then κ and all its superclasses must have been initialised. Nevertheless, Figures 6, 7 and 8 show that writing an abstract domain for Julia is very simple. You do not see there any of the problems related with parsing the set of closed classes (Section 4), giving structure to the code (Section 5), focusing the analysis on the watchpoints (Section 6), making up a flow and context sensitive analysis (Section 7) or computing a fixpoint (Section 8). They have all been solved by Julia.

10 A Note on Multithreading

The abstract domain of Section 9 yields correct results for the analysis of both single and multi-threaded programs. Namely, during the analysis of a method, it assumes that only the current method can initialise classes. This hypothesis leads to correct (but imprecise) results if, instead, a concurrent thread autonomously initialises classes.

This is not a general result. Other abstract domains might yield *incorrect* results if used to analyse multi-threaded programs. However, this is not the case for any of the abstract domains in Figure 1 which are *object-insensitive i.e.,* they

compute flow and context insensitive approximations for the fields of the objects (this can be controlled in JULIA through a command-line option).

It must be noted, however, that there is currently a relative lack of theoretical results about abstract interpretation of imperative, object-oriented concurrent programs. The important point, here, is that the correctness of a static analysis implemented inside JULIA and applied to a concurrent program does not depend on the analyser itself, but on the abstract domain which is used for the analysis and the level of flow sensitivity that it features.

11 Benchmarks

We report the time of some static analyses of a set of benchmark programs. Some of them are analysed in *application mode* (A), which means that we assume that only the `main` method is called from outside. Some of them, such as multithreaded applets, are analysed in *library mode* (L), which means that we assume that all public methods can be called from outside. The experiments have been performed on a Pentium 2.4 Ghz machine with 1 gigabyte of RAM, running Linux 2.6, Sun Java Development Kit version 1.5 with HotSpot just-in-time compiler, and JULIA version 0.40.

Figure 9 reports the time of the preprocessing performed by ROMEO for loading the classes, parsing the bytecode and building the graph of basic blocks (Section 5). Note that if many static analyses are performed, one after another, then preprocessing time is spent only once. The same figure shows the results for some static analyses. These are the times of the fixpoint computation performed by JULIET.

We can hence affirm that JULIA features reasonable efficiency and precision at least for some middle-size applications.

12 Conclusion

Our static analyser JULIA is a generic static analyser for full Java bytecode. We have described its structure, the techniques involved in its implementation, an example of abstract domain and some examples of analysis. The latter show that the current version of JULIA is already able to analyse non-trivial applications.

References

1. Indus Project. Available at `http://indus.projects.cis.ksu.edu/`.
2. jDFA - The Data-Flow Analysis Framework for Java. Available at `http://jdfa.sourceforge.net/`.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.
4. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. of Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 462–465, Boston, MA, USA, 2004.

program	size	prepr.	`rt`		`cps`		`cer`		`static`	
		time	time	P	time	P	time	P	time	P
`Dhrystone`(A)	6/21/607	1254	97	40%	257	40%	373	75%	113	71%
`ImageView`(L)	2/20/1235	1120	146	10%	485	10%	670	8%	158	29%
`Morph`(L)	1/14/1367	1415	101	34%	410	34%	674	5%	129	11%
`JLex`(A)	25/131/12472	3408	312	12%	2799	14%	1972	20%	2137	62%
`JavaCup`(A)	45/315/14475	4825	485	24%	11772	26%	6239	38%	2045	67%
`Jess`(A)	264/1553/44001	11708	2755	41%	42725	46%	30011	21%	7986	56%
`jEdit`(A)	492/2643/98964	29401	5111	35%	177416	39%	146281	3%	18991	63%
`Julia`(A)	841/5040/142266	42055	10288	37%	413283	42%	470141	3%	33045	55%

Fig. 9. Size (in number of classes/methods/bytecodes), time (in milliseconds, for preprocessing and analysis) and precision P of the application of JULIA to some benchmarks. Precision P is expressed in number of dynamic calls which are found to be static for `rt` and `cps`; in number of creation points which are stack-allocated for `cer`; in number of class initialisation checks which are found to be useless for `static`. `Dhrystone` is a testbench for numerical computations; `ImageView` is an image visualisation applet; `Morph` is an image morphing program; `JLex` is a lexical analysers generator; `JavaCup` is a compilers' compiler; `Jess` is a rule-based language; `jEdit` is a text editor; `Julia` is our JULIA analyser itself.

5. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
6. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'04*, volume 36(5) of *SIGPLAN Notices*, pages 203–213, Snowbird, Utah, May 2001. ACM Press.
7. B. Blanchet. Escape Analysis for Java^{TM}: Theory and Practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):713–775, November 2003.
8. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. In D. A. Schmidt, editor, *Proc. of the European Symposium on Programming, ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 385–400, Barcelona, Spain, March-April 2004. Springer-Verlag.
9. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM TOPLAS*, 25(6):876–910, November 2003.
10. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
11. P. Cousot and R. Cousot. Modular Static Program Analysis. In R. N. Horspool, editor, *Proceedings of Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178, Grenoble, France, April 2002. Springer-Verlag.
12. Apache Software Foundation. BCEL - The Bytecode Engineering Library. `jakarta.apache.org/bcel/`, 2002.

13. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, January 2005. Springer-Verlag.
14. M. Hermenegildo, W. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programing*, 13(2 & 3):349–366, 1992.
15. P. M. Hill and F. Spoto. A Refinement of the Escape Property. In A. Cortesi, editor, *Proc. of the VMCAI'02 workshop on Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 154–166, Venice, Italy, January 2002.
16. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
17. K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a Customizable Intermediate Representation. In *Proc. of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators (IVME)*, San Diego, California, June 2003. ACM Press.
18. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.
19. I. Pollet. *Towards a Generic Framework for the Abstract Interpretation of Java*. PhD thesis, Department of Computing Science and Engineering, Catholic University of Louvain, 2004.
20. A. Sabelfeld and A. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
21. F. Spoto. Focused Static Analyses for the Java Bytecode. Available at `www.sci.univr.it/~spoto/papers.html`.
22. F. Spoto. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In P. Cousot, editor, *Proc. of the Static Analysis Symposium, SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 127–145, Paris, France, July 2001. Springer-Verlag.
23. F. Spoto. The JULIA Generic Static Analyser. Available at `www.sci.univr.it/~spoto/julia`, 2005.
24. F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):578–630, September 2003.
25. F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proc. of OOPSLA'00*, volume 35(10) of *SIGPLAN Notices*, pages 281–293, Minneapolis, Minnesota, USA, October 2000. ACM Press.
26. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode using the SOOT Framework: Is It Feasible? In D. A. Watt, editor, *Proc. of Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, April 2000.
27. J. Whaley and M. Lam. Cloning-based Context-Sensitive Pointer Alias Analysis using Binary Decision Diagrams. In W. Pugh and C. Chambers, editors, *Proc. of Programming Language Design and Implementation (PLDI)*, pages 131–144, Washington, DC, USA, June 2004.