JULIA: A Generic Static Analyser for the Java Bytecode

Fausto Spoto
Dipartimento di Informatica, Università di Verona, Italy
e-mail: fausto.spoto@univr.it

***Abstract*— We describe our software tool JULIA for the static analysis of full sequential Java bytecode. This tool is *generic* in the sense that no specific abstract domain (analysis) is embedded in JULIA. Instead, abstract domains are provided as external classes that specialise the behaviour of JULIA. Static analysis is performed through a denotational 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.**

I. INTRODUCTION

This paper describes a software tool, called JULIA, that we have developed in order to apply the abstract interpretation technique [7] to the static analysis of Java bytecode [14]. The motivation underlying our effort is that of providing a software support to reason on Java bytecode applications and prove their abstract properties before the same application is run, and when the source code of the application is not available or does not even exist. The possibility of 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 for the success of such programs. This is particularly true for applications written in the Java bytecode, that are distributed in the Internet or used inside a smartcard, and might be potentially harmful to the client. Hence, analyses for security are attracting more and more interest [15]. But the information inferred by a static analysis can also be used to optimise a program, as well as for documentation and debugging.

Abstract interpretation [7] 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* of the real data it will be fed with at run-time. This description is the *abstract domain*. By saturating all possible execution paths in the program, we get a 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.

Our goal has been the development of a new static analyser JULIA for full sequential Java bytecode which fulfills 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 a set of *application classes*, which will be the same whenever and wherever they are loaded, so that analysis can be applied to them;
- 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;
- 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 through 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 must remain a domain-related issue [7];
- the analyser implements efficient techniques for computing the fixpoint needed for the static analysis [7]. These techniques must be domain-independent, so that the abstract domain developer need not care about how the fixpoint is computed for its abstract domain.

JULIA is free software [18]. We have currently implemented two abstract domains for JULIA. The first performs *class analysis* [20] through our formalisation [19] inside abstract interpretation of the original idea in [2]. The second is an implementation of our abstract domain for *escape analysis* [4] defined in [13].

The paper is organised as follows. Section II describes related work. Sections III to VIII show how each of the previous criteria have been attained with JULIA. Since this is a description of the analyser, we do not provide a thorough discussion of the techniques involved, but refer instead to other papers where they have been defined and discussed. Section IX describes the interface of an abstract domain *i.e.,* how new abstract domains can be plugged inside JULIA. Section X gives

the user interface of JULIA. Section XI shows some examples of the use of the analyser. Section XII shows how watchpoints can be explicitly placed in the code to analyse. Section XIII presents the cost in time of the class and escape analysis of some non-trivial Java bytecode applications. Section XIV concludes.

II. Related Work

Because of the actual complexity of the Java bytecode, static analyser for full (sequential) Java bytecode have not been developed intensively yet.

The SOOT bytecode decompiler [21] can be used as the front-end of a generic static analyser for Java bytecode. Currently, class analyses similar to ours are implemented inside SOOT. No more complex analyses, such as our escape analysis, have been implemented this way yet. Moreover, decompilation poses some problems when the bytecode is not the result of the compilation of Java, and maybe contains some *exotic* features of the Java bytecode that cannot be modeled in Java, such as overlapping or even recursive exception handlers (*i.e.*, catching exceptions thrown by themselves), or recursive Java bytecode subroutines (which cannot be decompiled into `finally` clauses).

Escape analysis has been implemented through specialised analysers for Java source code only, rather than Java bytecode. For instance, the analysis in [4] works inside a commercial Java compiler. The construction is specific to escape analysis, and it cannot be immediately applied to other analyses.

A generic analyser for the Java Card bytecode has been defined and developed in [6]. 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.

Finally, ours is the only implementation of a *focused* analysis for the Java bytecode.

III. A Generic Analyser

Being generic is an important 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 V) and the fixpoint computation (Section VIII) 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.

A pictorial representation of this situation is shown in Figure 1, which presents the structure of our JULIA analyser. 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 has its own fixpoint strategies, independent from the abstract domain. The BCEL library [11] is a low-level interface to `.class` files.

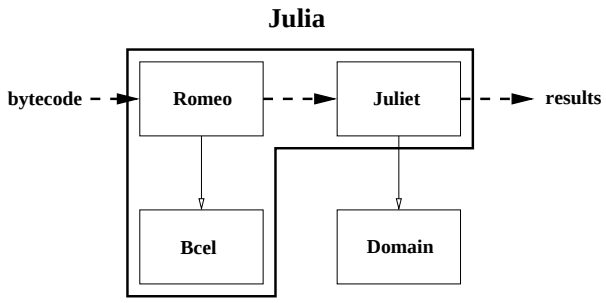

Fig. 1. The structure of our JULIA analyser.

The following table shows the difference in size of the analyser and of two abstract domains. You can see that it is very convenient to write a generic analyser, since a very large part of the code is shared between different abstract analyses. *rt* is our abstract domain for class analysis. $\mathcal{ER}$ is our abstract domain for escape analysis.

module	number of classes	size in bytes
ROMEO	93	170724
JULIET	18	26586
rt	1	12966
$\mathcal{ER}$	11	45519

IV. Application Classes

The Java bytecode loads classes dynamically as they are needed during the execution of a program. As a consequence, 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* [8] claiming that nothing is known about its outcome. This is definitely correct, but practically useless in an object-oriented language, where classes are tightly coupled through virtual method invocations, field accesses and constructor chaining. Following 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 [21]. It consists in assuming that a set of classes, called *application classes*, will be the same between analysis and run time. As a consequence, we can inspect them during the analysis and gather abstract information which will improve the precision of the analysis.

Application classes are typically those of the application we are analysing. Libraries are not considered application classes, usually. Any reference to them is resolved through a worst-case assumption. However, stronger hypotheses than the worst-case assumption can be made. For instance, in [20], application classes are assumed to be downward closed *wrt.*

subclassing: the subclasses of an application class are still application classes. This improves the precision of the analysis.

In general, 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, in order to improve the precision of the analysis. Of course, this must be clearly stated in the definition of the abstract domain, 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 class analysis assumes that application classes are downward closed, as in [20], while our domain $\mathcal{ER}$ 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.

We refer to reader to Section X for the description of how the set of application classes is specified by the user of JULIA.

V. 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 (operation or denotational) input/ouput map. But some Java bytecode 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 an exceptional output must be taken into account. 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, which results in its transformation into a graph of *basic blocks* [1], more suitable to abstract intepretation. Namely, only bytecodes which can be seen as input/ouput maps are left in the code. Control is modeled through edges between basic blocks. Conditional jumps use new *filter* bytecodes, which plays exactly the same role as the `assume` statements used in [3]. For instance, the following bytecode method:

```
Method void spin()
   0 iconst_0
   1 istore_1
   2 goto 8
   5 iinc 1 1
   8 iload_1
   9 bipush 100
  11 if_icmplt 5
  14 return
```

is translated into the graph of basic blocks shown in Figure 2, where the `goon` new *filter* bytecodes select the cases when the guard of the condition `if_icmplt` bytecode is true or false, respectively.

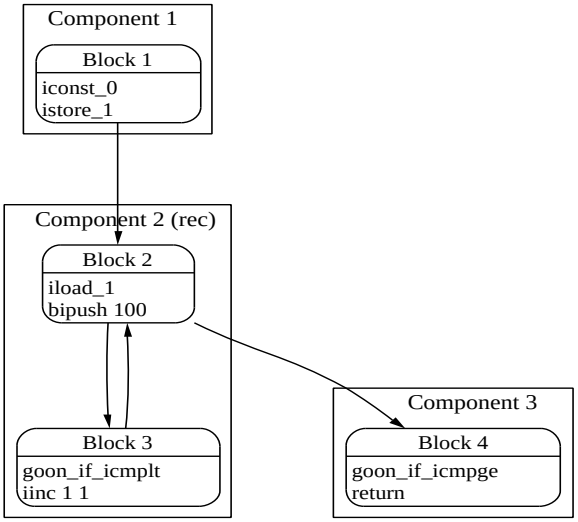

Fig. 2. A graph of basic blocks of code.

The most complex bytecodes are split into many smaller bytecodes, to which abstract interpretation can be easily applied. Exception handlers are *compiled* into the code, as well as the lookup procedure for methods. The result is a graph of basic blocks of a simplified Java bytecode, which we call JULIET bytecode, like that in Figure 2. For instance, 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 need not know how a method is resolved and looked up by the Java Virtual Machine [14]. He need not 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.

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

For a detailed description of this preprocessing of the Java bytecode, see [16].

VI. LOCALISATION

The information computed by a static analysis is typically useful in some special program points only, called *watch-points*. 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 [20]. 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* from the method where the bytecode occurs. Those objects can hence be allocated in the call stack instead of the heap [4], [13].

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 would be desirable that the cost of the analysis scale with the number of watchpoints. If this is the 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 [17] for a simple high-level language. In [16] we show how it can be applied to the Java bytecode, by exploiting the same simplification of the bytecode highlighted in Section V. We have then implemented this localised analysis inside JULIET, the fixpoint engine of JULIA. Our experiments confirm the focused nature of the resulting static analyses for the Java bytecode [16].

Note that a good property of our focused analyses is that the abstract domain developer need not be aware of how the focusing technique works [17], [16]. He develops his abstract domain as for a simple input/ouput analysis.

In JULIA, watchpoints can be put automatically by the abstract domain, which is aware of the goal for which its analysis has been developed. Or they can be explicitly put by the user. Section XII shows an example.

VII. NO CONSTRAINTS ON PRECISION

The theory of abstract interpretation [7] entails that the precision of the static analysis is a domain-related issue. Abstract domains of different precision can be formally compared without considering the way the static analysis is implemented.

This situation must be maintained in practice. Hence the static analyser must not limit the precision of the analysis because of some spurious constraint consequent from the way the analysis is implemented.

Many static analyses work by compiling 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 in the constraint is used to represent the approximation of a program variable *throughout its all 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 can be improved by using *variable splitting* in order to multiply the variables used in the constraint to represent the same variable of the program. But this means that the abstract domain developer (who writes the compilation of the source program into a constraint) must be aware of the problem. Moreover, this does not solve the problem of the same method called from different contexts since the same approximations are still used for all those calls. Again, *method replication* can be used here, but this further complicates the analysis. In particular, none of these two techniques has ever been implemented for the Java bytecode.

We prefer instead to stick to the traditional definition of abstract interpretation [7], so that the static analysis works by computing a fixpoint over data-flow equations derived from the structure of the program. This results in a data and control-flow sensitive analysis. The abstract domain might be so imprecise that it does not 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 V) exposes the flows of control arising from the lookup procedures for virtual methods invocations, from the exceptions and from the subroutine handling mechanism. 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 they must be rather 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 class analysis chooses between those flows non-deterministically, while the abstract domain $\mathcal{ER}$ for escape analysis selects them in order to drive the analysis and collect more precise information.

VIII. FIXPOINT ENGINE

Computing a *global* fixpoint over data-flow equations can be computationally expensive or even prohibitive. There are, however, some traditional or new techniques that we have used to speed up the computation of the fixpoint.

The first technique consists in building the maximal strongly connected components of the calling 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, one at most for each component. For instance, Figure 2 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).

Another technique was originally developed for the static analysis of logic programs and is known as *abstract compilation* [12]. The idea is that, during the abstract interpretation process, the same 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 2 gets first abstracted into the chosen abstract domain, as Figure 3 shows. Then,

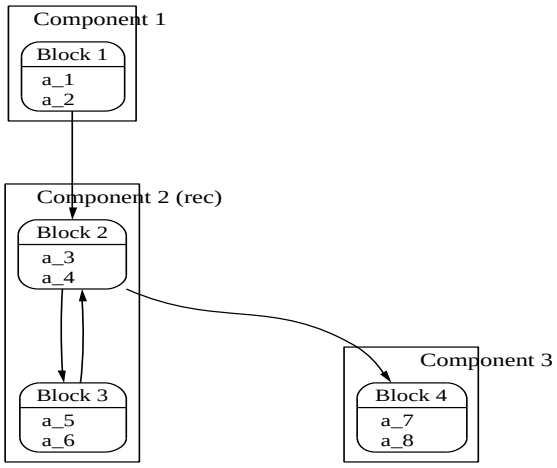

Fig. 3. The abstract compilation of the program in Figure 2.

the fixpoint mechanism is applied as before, by following the strongly connected components.

The abstract compilation process can be moved forward. If the analysis of a part of the program is *stable i.e.,* it will not change anymore during the analysis, then it can be substituted with its abstract analysis. For instance, during the computation of the local fixpoint for component 2 in Figure 3 we compute repeatedly the sequential composition of a_3 with a_4 and of a_5 with a_6. It is hence convenient to compute these compositions once and for all, before the fixpoint mechanism starts.

Abstract compilation can hence be seen as an iterated compilation of a program (*i.e.,* a graph of basic blocks of code) into an abstract domain element, which is the analysis of the whole program. By avoiding repeated computations, it reduces significantly the cost of the fixpoint calculation. In our experiments, it manages sometimes to double the speed of the analyser, but its actual benefit depends on how much loops and recursion are used in the analysed program.

We use a new technique to improve the efficiency of the fixpoint calculation for the focused analyses [17]. Because of the watchpoints, many composition operations on the abstract domain are actually repeated for different watchpoints, if these are syntactically close. It is hence convenient to delay such conjuntions as much as possible and apply them once for many watchpoints. For reasons of space, we cannot detail this technique further. We just say that it has been implemented by using special data structures, called *watchpoint trees* or *wrees*, that express the relationships between the compositions needed for each watchpoint. Tree operations allow then to group the compositions in order to better schedule their execution.

The final technique consists in using destructive operations for composing abstract domain elements. For instance, in Figure 3, once we compute the analysis of component 2, we compose it sequentially with that of component 1. Since the former is not needed anymore, destructive composition can be used, which results in faster and less memory-hungry

```
domain();
void introduceYourself();
boolean processArg(String arg);
void init(Loader loader);
BottomUpDomain id();
BottomUpDomain bottom();
BottomUpDomain analyse(Instruction in);
BottomUpDomain lub(BottomUpDomain d);
BottomUpDomain compose(BottomUpDomain d);
boolean equals(BottomUpDomain d);
String toString();
void applyAnalysis(GraphAnalysis analysis);
String output();
```

Fig. 4. The interface between JULIA and an abstract domain.

operations. This technique, which is an instance of *dynamic programming*, cannot be applied to those abstract domain elements that, for some reason, cannot be destroyed during the analysis. Examples are those that are explicitly required by the user of JULIA, through the user interface described in Section X.

It must be noted that the abstract domain designer need not be aware of the use of strongly connected components, abstract compilation and watchpoint trees, which are domain-independent techniques. Dynamic programming, instead, requires him to write destructive abstract operations on the domain. Otherwise, the analyser will still work, although slightly slower.

Domain-specific fixpoint acceleration techniques will be added in the future to JULIA, through *widening* operators [7]. However, we are already able to analyse middle-size Java bytecode programs, as Section XIII shows.

IX. WRITING ABSTRACT DOMAINS FOR JULIA

New abstract domains can be developed and plugged inside JULIA. The abstract domain developer must define the abstract counterpart of each concrete bytecode, as well as define how the watchpoints must be put in the source code in order to perform the analysis implemented by the abstract domain.

An abstract domain for JULIA is a Java class extending the `juliet.BottomUpDomain` class. Figure 4 shows the most important methods it must define.

An abstract domain must define an empty constructor, whose name is the same as that of the domain. It must define the `introduceYourself` method, which prints a description of the abstract domain. The `processArgs` method allows an abstract domain to add new switches to the command-line interface of JULIA (Section X), which select domain-specific options. The `init` method is called before the abstract domain is used for the analysis. It receives as an argument the `Loader` class which has been used to preprocess the program which is going to be analysed (Section V). Here is where the abstract domain can decide to put some watchpoint in the code, depending on the analysis it implements. The JULIET code of the program is indeed accessible from the loader. The abstract domain can also use this call for book-keeping. For instance, to collect the overall

set of classes in the case of class analysis, or the overall set of `new` bytecodes in the case of escape analysis. The `id` method clones an element of the abstract domain, while the `bottom` method returns its least element. The `analyse` method returns the abstract element corresponding to each concrete bytecode (the `Instruction` argument). The `lub` and `compose` methods compute the least upper bound of two abstract domain elements (needed to merge the results of conditionals, virtual calls and, in general, branches in the flow of control) and the sequential composition of the abstract domain elements (used to model the sequential composition of bytecodes), respectively. The `equals` method checks if two abstract domain elements are the same. This is needed to stop the fixpoint computation of Section VIII. The `toString` method returns a human-readable representation of an abstract domain element. The `applyAnalysis` method applies the result of the analysis (*i.e.,* the analyses of the call-graph of methods) to the program. For instance, in the case of class analysis, `applyAnalysis` determines which virtual methods have actually one possible target only. In the case of escape analysis, it determines which `new` bytecodes create objects that do not escape from the method where they occur. Finally, the `output` method is used to add domain-specific statistical information to those presented by JULIA at the end of the analysis (see Figure 7).

Abstract domains can be implemented with every technique. For instance, we used binary decision diagrams [5] to implement rt, while we used set-constraints [10] to implement $\mathcal{ER}$.

X. COMMAND-LINE ARGUMENTS

In the following, we assume that the classes of JULIA and those of the application we analyse are in the Java class path. Then JULIA can be invoked by typing

`java julia.Main` *main_class_name*

where *main_class_name* is the fully qualified name of the Java class we want to analyse, without the `.class` extension.

JULIA accepts a set of command-line options. Some are related to the preprocessing of the bytecode, others to the fixpoint computation, others to the output of the analysis. Moreover, there are options specific to each abstract domain.

General Options

`-h -help`
Displays the list of command-line options and exits.

`-version`
Prints the version of JULIA which is being used.

`-gpl`
Prints the GNU General Public Licence terms and exits.

`-redistribute`
Prints the redistribution policy and exits.

`-v -verbose`
Runs in verbose mode. This can significantly slow down the analysis.

`-app`
Runs in application mode. This means that all user classes referenced from *main_class_name* are recursively loaded, resolved and analysed *i.e.,* they are considered *application classes* (Section IV). Library classes are *not* application classes even if they are referenced from an application class. Instead, they must be included explicitly through the `-i` or `-wi` options.

`-i` *pkg* `-include` *pkg*
All classes whose name starts with *pkg* are loaded, resolved and analysed *i.e.,* they are considered *application classes* (Section IV). The result of the analysis is then applied to those classes. This is the way you let JULIA include a library in the analysis *and optimise or verify it*. This can significantly slow down the analysis or even make it computationally prohibitive.

`-wi` *pkg* `-weak-include` *pkg*
All classes whose name starts with *pkg* are loaded, resolved and analysed *i.e.,* they are considered *application classes* (Section IV). However, the result of the analysis is *not* applied to those classes. This is the way you let JULIA include a library in the analysis. The same cost issues stated for `-i` above apply here.

`-library`
Assumes that all `public` methods of *main_class_name* can be called by the user. If this option is not specified, only the `main` method of *main_class_name* (if any) is assumed to be callable by the user. This option reduces the precision of the analysis, but allows one to analyse a library.

Analysis Options

`-a` *dom* `-abstract` *dom*
Performs static analysis by using the specified abstract domain *dom*. Current alternatives for *dom* are

- `n none`
  Does not perform any static analysis. This is the default. Hence, by default only preprocessing takes place.
- `er`
  Performs escape analysis through the $\mathcal{ER}$ domain [13].
- `rto`
  Performs rapid type analysis through the rt domain formalised in [19] from the original idea in [2].
- `t test`
  Uses a test abstract domain of one abstract element only.

`-nowrees`
Disables the use of the *watchpoint trees* (Section VIII). This can only make the efficiency of the analyser poorer. It is only useful for benchmarking the analyser.

`-noabs`
Disables the use of abstract compilation [12] during the fixpoint computation (Section VIII). This reduces the efficiency of the analyser. It is only useful for benchmarking the analyser.

`-nodyn`
Disables the use of dynamic programming during the fixpoint computation (Section VIII). This results in slightly poorer performance. It is only useful for benchmarking the analyser.

`-all`
Computes (and, if required, prints out) the denotation of all

methods. Normally, only the denotations of the methods in *main_class_name* are computed. This option can lead to a slower and more memory-hungry analysis, because it reduces the possibility of dynamic programming through destructive abstract operations (Section VIII).

`-random k`
Puts approximately k random watchpoints per 1000 bytecodes. Useful for benchmarking the focused nature of JULIA.

Output Options

`-d` *dir* `-output-dir` *dir*
Specifies the *output directory*, which is where the results of the analysis must be saved. The default is `juliaOutputs`.

`-f` *format* `-output-format` *format*
Specifies the output format of the analyser. For every method whose denotation is computed (see `-all`), the information of type *format* is saved in the output directory (see `-d`). The following alternatives are currently available for *format*:

- `a abstract`
  Stores the abstract denotation of the methods (if an analysis is specified). This can significantly slow down the analysis or even make it unfeasible for large programs.
- `b bytecode`
  Stores the original bytecode into `.bc` files.
- `g graph`
  Stores the JULIET bytecode structure in `dot` graph notation. To get a postscript file from it (such that shown in Figure 2), type `dot -Tps file.dot -o file.ps`.
- `j juliet`
  Stores a textual print-out of the JULIET bytecode.
- `n none`
  Produces no output. This is the default.

`-cf`
Prints the method call graph of the program. Class hierarchy analysis [9] is used here. This output can be very large.

`-s -time`
Prints statistical information at the end of JULIA execution. It includes the size of intra and inner-methods strongly connected components, as well as running times for the preprocessing of the bytecode and for the actual analysis. Domain-specific information can be added here.

Domain-Specific Options

These options are only active when an abstract domain is selected through the `-a` switch. They are usually concerned with the way the abstract information must be used.

`-countdet`
Only active for rapid type analysis (`-a rto` option). It states that the abstract information must be used to count the number of determined virtual calls [20].

`-countstk`
Only active for escape analysis (`-a er`). It states that the abstract information must be used to count the number of `new` bytecodes that create objects that do not escape their defining method [13].

```
package testcases;

class Tax {
  public static void main(String argv[]) {
    Person a = new Person();
    a.payTax();

    a = new Professor();
    // julia.Watchpoint.watchThis("hello");
    a.payTax();
  }
}
```

Fig. 5. An example Java source code.

XI. EXAMPLES

We show here some examples of static analysis with JULIA.

The abstract analysis or *denotation* of a method is a map from the abstract properties of its input to the abstract properties of its output. Let us compute this input/ouput denotation for the compilation into Java bytecode of the methods of the `Tax` class in Figure 5, with rapid type analysis. We type:

```
java julia.Main -f a -a rto -app testcases.Tax
```
(1)

(we will drop `testcases` in the following). This makes JULIA write inside the `juliaOutput` directory the denotations of the methods of the class `Tax`. Namely, the following files are created there:

```
Tax.<init>{}V.rto
Tax.main{[Ljava.lang.String;}V.rto
```

If you are interested in the denotations of *all* the methods processed during the analysis, add `-all` to (1). The print-out of an input/output map looks differently for each abstract domain. For instance, that for the `rt` abstract domain is like that shown in the upper part of Figure 6 (the *behaviour* part). For rt, the map specifies how the set of instantiated classes increases during the execution of the method [19]. For instance, if we start the execution of `main` inside `Tax` with no class instantiated yet (*i.e.*, with the empty set), Figure 6 shows that at the end of the method it is possible to have instantiated classes from the set $\{\texttt{Person}, \texttt{Professor}, open\}$ *i.e.*, only the two application classes `Person` and `Professor` might have been instantiated, as well as all non-application classes (represented by $open$). In particular, `Tax` is not instantiated at the end of the execution of `main`.

Let us now actually *use* the class analysis to spot some possible optimisations in the code. Type:

```
java julia.Main -a rto -countdet -s -app Tax
```
(2)

You will get the statistical information shown in Figure 7 (actual times might change). Three classes have been parsed. Namely, `Tax` and the classes of the application whose methods are called or whose fields are accessed from `Tax`. The class

```
Behaviour:
{Tax,Person,Professor,open}  --> {Tax,Person,Professor,open}
{Tax,Person,Professor}  --> {Tax,Person,Professor,open}
{Tax,Person,open}  --> {Tax,Person,Professor,open}
{Tax,Person}  --> {Tax,Person,Professor,open}
{Tax,Professor,open}  --> {Tax,Person,Professor,open}
{Tax,Professor}  --> {Tax,Person,Professor,open}
{Tax,open} --> {Tax,Person,Professor,open}
{Tax} --> {Tax,Person,Professor,open}
{Person,Professor,open} --> {Person,Professor,open}
{Person,Professor} --> {Person,Professor,open}
{Person,open} --> {Person,Professor,open}
{Person} --> {Person,Professor,open}
{Professor,open} --> {Person,Professor,open}
{Professor} --> {Person,Professor,open}
{open} --> {Person,Professor,open}
{} --> {Person,Professor,open}

Watched instructions:
watchpoint: hello:
{Tax,Person,Professor,open}  --> {Tax,Person,Professor,open}
{Tax,Person,Professor}  --> {Tax,Person,Professor,open}
{Tax,Person,open}  --> {Tax,Person,Professor,open}
{Tax,Person}  --> {Tax,Person,Professor,open}
{Tax,Professor,open}  --> {Tax,Person,Professor,open}
{Tax,Professor}  --> {Tax,Person,Professor,open}
{Tax,open} --> {Tax,Person,Professor,open}
{Tax} --> {Tax,Person,Professor,open}
{Person,Professor,open} --> {Person,Professor,open}
{Person,Professor} --> {Person,Professor,open}
{Person,open} --> {Person,Professor,open}
{Person} --> {Person,Professor,open}
{Professor,open} --> {Person,Professor,open}
{Professor} --> {Person,Professor,open}
{open} --> {Person,Professor,open}
{} --> {Person,Professor,open}
```

Fig. 6. The denotation of `main` in `Tax` with a watchpoint.

`java.lang.Object` is *not* parsed, despite the fact that it is called from the constructor of `Tax` by constructor chaining. This is because it is not a class of the application, but it belongs to a library. You have to include it explicitly through the `-wi` option if you really want to include its code in the analysis.

In the previous example, JULIA told us, in particular:

```
Number of determined/undeterminable/total
  virtual calls  . . 2 (40%)/2/5
```

This means that two out of five virtual calls in the code can never be considered determined (because they can lead to a method outside the set of application classes, see [20]). Other two virtual calls are instead recognised as *determined*, hence implicitly static, by the class analysis. A virtual call, finally, although it always leads to a method of the application classes, seems inherently polymorphic to JULIA. If you want to know exactly which calls are recognised as *determined* by JULIA, add `-verbose` to (2). You will see, among other things:

```
invokevirtual cry()V is determined
invokevirtual payTax()V is determined
```

The former call is inside `Professor`, and is recognised as determined at preprocessing time by hierarchy class analysis [9]. The latter call is the first `payTax` call inside `Tax` (Figure 5) and is recognised as determined at rapid type analysis time.

If you want to see the transformation of the source Java bytecode into the JULIET bytecode, add the `-f j` switch to (2). Then JULIA writes the following files inside the `juliaOutput` directory:

```
Person.<init>{}V.juliet
Person.payTax{}V.juliet
Professor.cry{}V.juliet
Professor.<init>{}V.juliet
Professor.payTax{}V.juliet
Tax.<init>{}V.juliet
Tax.main{[Ljava.lang.String;}V.juliet
```

Instead of rapid type analysis, you may want to perform escape analysis of `Tax`. Then you change into `er` the abstract domain in (2) and specify that you want to try to stack-allocate the `new` bytecodes [13]:

```
java julia.Main -a er -countstk -s -app Tax
```

JULIA prints now:

```
Stack-allocatable creation points/total: 0/2 (0%)
```

which means that no `new` bytecode (*creation point*) is recognised as stack-allocatable, out of two occurring in the code.

```
Romeo's statistics:
  Number of classes parsed . . . . . 3
  Number of methods processed  . . . . . 7
  Number of strongly connected components of methods . . . . . 7
  Average dimension of a strongly connected component of methods . 1.0
  Average blocks per method  . . . . . 10.428572
  Average strongly connected components of blocks per method . . . 10.428572
  Average dimension of a method's strongly connected component . . 1.0
  Number of bytecodes before preprocessing . . . . . 33
  Number of bytecodes after preprocessing  . . . . . 137
  Time for preprocessing (ms)  . . . . . 889

Juliet's statistics:
  Number of abstract joins . . . . . 65
  Number of abstract compositions  . . . . . 147
  Number of abstract copies  . . . . . 164
  Number of watchpoints  . . . . . 2
  Time for the analysis (ms) . . . . . 96
  Time for applying the analysis (ms)  . . . . . 1

Domain specific statistics:
  Number of determined/undeterminable/total virtual calls  . . . . 2 (40%)/2/5

Global statistics:
  Average time per class (ms)  . . . . . 328
  Average time per method (ms) . . . . . 140
  Total time (ms)  . . . . . 986
```

Fig. 7. The statistics of the class analysis of `Tax`

We can improve the precision of this result by including the `java.lang.Object` class in the analysis:

```
java julia.Main -a er -countstk -s
  -app -wi java.lang.Object Tax
```

so that now we get:

```
Stack-allocatable creation points/total: 2/2 (100%)
```

XII. Working with the Watchpoints

Watchpoints are usually put automatically by the abstract domain in places where the abstract information can be used for some transformation or optimisation (see Section IX). This is what happens if you use the `-countdet` or `-countstk` options for class and escape analysis, respectively. However, it might be the case that you want to specify manually the program points where the analysis must be focused. For instance, suppose that you want to know which is the set of classes instantiated before the second call of `payTax` inside `Tax`. Then you can put a *watchpoint* there, by uncommenting the relative line in in Figure 5. You must recompile `Tax` now and apply rapid type analysis asking to store the denotation of the methods of `Tax` (as in (1)). A *watched instruction* is present now inside the denotation stored in the file for the `main` method, as shown in Figure 6. That denotation contains an input/output map related to watchpoint *hello*. The input is the set of classes that are instantiated before the call to `main`. The output is an overapproximation of the set of classes that are instantiated at program point *hello*. For instance, if the input is $\{\texttt{Tax}, \mathit{open}\}$ (*i.e.*, the only application class instantiated at the beginning of the execution of `main` is `Tax`; all other library classes might have been instantiated) then the classes $\{\texttt{Tax}, \texttt{Person}, \texttt{Professor}, \mathit{open}\}$ are instantiated *at most* at watchpoint *hello*.

program	class analysis		escape analysis	
	prepr.	analysis	prepr.	analysis
`Dhrystone`	0.509	0.149	0.721	0.825
`ImageViewer`	0.576	0.201	0.788	1.713
`Morph`	0.637	0.243	0.813	1.283
`JLex`	2.408	1.245	2.609	38.223
`JavaCup`	5.467	15.008	11.859	469.108
`Julia`	2.155	1.624	2.372	319.259

Fig. 8. The cost in time of our class and escape analyses.

XIII. Benchmarks

We have applied both class and escape analysis, through our two domains rt and $\mathcal{ER}$, to some middle-size applications (between 604 and 15854 bytecodes in size). `Dhrystone` is a testbench for numerical computations; `ImageViewer` is an image visualisation applet; `Morph` is an image morphing program; `JLex` is the Java version of the well-known `lex` lexical analysers generator; `JavaCup` is a compilers' compiler; `Julia` is our JULIA analyser itself, without the abstract domains and the classes representing the Java bytecodes. The experiments have been performed on a CENTRINO^{TM} 1.4 Ghz machine with 512 megabytes of RAM, running Linux^{TM} 2.4, Sun^{TM} Java Development Kit version 1.3.1 and JULIA version

program	class	escape
Dhrystone	18 (100%)	9 (60%)
ImageViewer	12 (100%)	0 (0%)
Morph	18 (100%)	0 (0%)
JLex	100 (97%)	27 (14%)
JavaCup	535 (87%)	137 (29%)
Julia	91 (91%)	82 (34%)

Fig. 9. The precision of our class and escape analyses.

0.31. You can run the same tests through the `class_analyses` and `escape_analyses` scripts provided with JULIA [18].

Figure 8 reports the times for the preprocessing and for the class and escape analyses of the applications, in seconds. To get the overall time for an analysis, add the preprocessing time to the given analysis time. In the case of escape analysis, we have included `java.lang.Object` and all `java.lang.String*` classes into the analysis, to get more precise results (see the `-wi` option in Section X). This is why preprocessing times are different for class and escape analysis.

The precision of the analyses is shown in Figure 9. In the case of class analysis, we report the number and percentage of the virtual calls that are found to have one possible target at run-time. Only virtual calls leading always inside application classes are considered, exactly as in [20]. For escape analysis, we provide instead the number and percentage of `new` bytecodes that are found to create objects that never escape the method where the bytecode occurs.

We can hence affirm that JULIA features reasonable efficiency and precision at least for some middle-size applications. In order to apply JULIA to larger benchmarks, some form of widening must be used [7]. In the case of class analysis, Figure 9 confirms the claim made in [20] that, if only virtual calls leading always inside application classes are considered, then rapid type analysis (which inspired our *rt* abstract domain) is very precise. Our high degree of precision for class analysis, shown in Figure 9, is probably also a consequence of the fact that our abstract domain *rt* implements rapid type analysis in a data and control-flow sensitive way.

XIV. CONCLUSION

Our static analyser JULIA is the first generic static analyser for full sequential Java bytecode. We have described its structure, the techniques involved in its implementation and shown some examples of analysis. Our benchmarks (Section XIII) show that the current implementation of JULIA is already able to analyse non-trivial applications.

Our ongoing work consists in improving the efficiency of the analyser and provide new abstract domains for more precise class and escape analysis and for security-related topics [15].

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.

[2] 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.

[3] 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.

[4] B. Blanchet. Escape Analysis for Java^{TM}: Theory and Practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):713–775, November 2003.

[5] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] 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.

[7] 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.

[8] 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.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *Proc. of ECOOP'95*, volume 952 of *LNCS*, pages 77–101, Århus, Denmark, August 1995. Springer-Verlag.

[10] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, 2000.

[11] Apache Software Foundation. BCEL - The Bytecode Engineering Library. `jakarta.apache.org/bcel/`, 2002.

[12] 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.

[13] 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.

[14] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[15] A. Sabelfeld and A. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[16] F. Spoto. Focused Static Analyses for the Java Bytecode. Available at `www.sci.univr.it/~spoto/papers.html`.

[17] 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.

[18] F. Spoto. The JULIA Generic Static Analyser. Available at `www.sci.univr.it/~spoto/julia`, 2004.

[19] 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.

[20] 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.

[21] 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.