Sawja: Static Analysis Workshop for Java

Laurent Hubert[1], Nicolas Barré[2], Frédéric Besson[2], Delphine Demange[3], Thomas Jensen[2], Vincent Monfort[2], David Pichardie[2], and Tiphaine Turpin[2]

[1] CNRS/IRISA, France
[2] INRIA Rennes - Bretagne Atlantique, France
[3] ENS Cachan - Antenne de Bretagne/IRISA, France

Abstract. Static analysis is a powerful technique for automatic verification of programs but raises major engineering challenges when developing a full-fledged analyzer for a realistic language such as JAVA. Efficiency and precision of such a tool rely partly on low level components which only depend on the syntactic structure of the language and therefore should not be redesigned for each implementation of a new static analysis. This paper describes the SAWJA library: a static analysis workshop fully compliant with JAVA 6 which provides OCAML modules for efficiently manipulating JAVA bytecode programs. We present the main features of the library, including i) efficient functional data-structures for representing a program with implicit sharing and lazy parsing, ii) an intermediate stack-less representation, and iii) fast computation and manipulation of complete programs. We provide experimental evaluations of the different features with respect to time, memory and precision.

Introduction

Static analysis is a powerful technique that enables automatic verification of programs with respect to various properties such as type safety or resource consumption. One particular well-known example of static analysis is given by the JAVA Bytecode Verifier (BCV), which verifies at loading time that a given JAVA class (in bytecode form) is type safe. Developing an analysis for a realistic language such as JAVA is a major engineering task, challenging both the companies that want to build robust commercial tools and the research scientists who want to quickly develop prototypes for demonstrating new ideas. The efficiency and the precision of any static analysis depend on the low-level components which manipulate the class hierarchy, the call graph, the intermediate representation (IR), etc. These components are not specific to one particular analysis, but they are far too often re-implemented in an *ad hoc* fashion, resulting in analyzers whose overall behaviour is sub-optimal (in terms of efficiency or precision). We argue that it is an integral part of automated software verification to address the issue of how to program a static analysis platform that is at the same time efficient, precise and generic, and that can facilitate the subsequent implementation of specific analyzers.

This paper describes the SAWJA library—and its sub-component JAVALIB—which provides OCAML modules for efficiently manipulating JAVA bytecode programs, and building bytecode static analyses. The library is developed under the GNU Lesser General Public License and is freely available at `http://sawja.inria.fr/`.

SAWJA is implemented in OCAML [17], a strongly typed functional language whose automatic memory management (garbage collector), strong typing and pattern-matching facilities make particularly well suited for implementing program processing tools. In particular, it has been successfully used for programming compilers (e.g., Esterel [24]) and static analyzers (e.g., Astrée [3]).

The main contribution of the SAWJA library is to provide, in a unified framework, several features that allow rapid prototyping of efficient static analyses while handling all the subtleties of the JAVA Virtual Machine (JVM) specification [20]. The main features of SAWJA are:

– parsing of **`.class`** files into OCAML structures and unparsing of those structures back into **`.class`** files;
– decompilation of the bytecode into a high-level stack-less IR;
– sharing of complex objects both for memory saving and efficiency purpose (structural equality becomes equivalent to pointer equality and indexation allows fast access to tables indexed by class, field or method signatures, etc.);
– the determination of the set of classes constituting a complete program (using several algorithms, including Rapid Type Analysis (RTA) [1]);
– a careful translation of many common definitions of the JVM specification, e.g., about the class hierarchy, field and method resolution and look-up, and intra- and inter-procedural control flow graphs.

This paper describes the main features of SAWJA and their experimental evaluation. Sect. 1 gives an overview of existing libraries for manipulating JAVA bytecode. Sect. 2 describes the representation of classes, Sect. 3 presents the intermediate representation of SAWJA and Sect. 4 presents the parsing of complete programs.

1 Existing Libraries for Manipulating Java Bytecode

Several similar libraries have already been developed so far and some of them provide features similar to some of SAWJA's. All of them, except BARISTA, are written in JAVA.

The Byte Code Engineering Library[4](BCEL) and ASM[5] are open source JAVA libraries for generating, transforming and analysing JAVA bytecode classes. These libraries can be used to manipulate classes at compile-time but also at runtime, e.g., for dynamic class generation and transformation. ASM is particularly optimised for this latter case: it provides a visitor pattern which makes possible local class transformations without even building an intermediate parse-tree. Those libraries are well adapted to instrument JAVA classes but lack important

4 `http://jakarta.apache.org/bcel/`
5 `http://asm.ow2.org/`

features essential for the design of static analyses. For instance, unlike Sawja, neither BCEL nor ASM propose a high-level intermediate representation (IR) of bytecode instructions. Moreover, there is no support for building the class hierarchy and analysing complete programs. The data structures of Javalib and Sawja are also optimized to manipulate large programs.

The Jalapeño Optimizing Compiler [6] which is now part of the Jikes RVM relies on two IR (low and high-level IR) in order to optimize bytecode. The high-level IR is a 3-address code. It is generated using a symbolic evaluation technique described in [30]. The algorithm we use to generate our IR is similar. Our algorithm works on a fixed number of passes on the bytecode while their algorithm is iterative. The Jalapeño high-level IR language provides explicit check instructions for common run-time exceptions (e.g., `null_check`, `bound_check`), so that they can be easily moved or eliminated by optimizations. We use similar explicit checks but to another end: static analyses definitely benefit from them as they ensure expressions are error-free.

Soot [29] is a Java bytecode optimization framework providing three IR: Baf, Jimple and Grimp. Optimizing Java bytecode consists in successively translating bytecode into Baf, Jimple, and Grimp, and then back to bytecode, while performing diverse optimizations on each IR. Baf is a fully typed, stack-based language. Jimple is a typed stack-less 3-address code and Grimp is a stack-less representation with tree expressions, obtained by collapsing Jimple instructions. The IR in Sawja and Soot are very similar but are obtained by different transformation techniques. They are experimentally compared in Sect. 3. Sawja only targets static analysis tools and does not propose inverse transformations from IR to bytecode. Several state-of-the-art control-flow analyses, based on points-to analyses, are available in Soot through Spark [18] and Paddle [19]. Such libraries represent a coding effort of several man-years. To this respect, Sawja is less mature and only proposes simple (but efficient) control-flow analyses.

Wala [15] is a Java library dedicated to static analysis of Java bytecode. The framework is very complete and provides several modules like control flow analyses, slicing analyses, an inter-procedural dataflow solver and a IR in SSA form. Wala also includes a front-end for other languages like Java source and JavaScript. Wala and its IBM predecessor DOMO have been widely used in research prototypes. It is the product of the long experience of IBM in the area. Compared to it, Sawja is a more recent library with less components, especially in terms of static analyses examples. Nevertheless, the results presented in Sect. 4 show that Sawja loads programs faster and uses less memory than Wala. For the moment, no SSA IR is available in Sawja but this is foreseen for the future releases.

Julia [26] is a generic static analysis tool for Java bytecode based on the theory of abstract interpretation. It favors a particular style of static analysis specified with respect to a denotational fixpoint semantics of Java bytecode. Initially free software, Julia is not available anymore.

Barista [7] is an OCaml library used in the OCaml-Java project. It is designed to load, construct, manipulate and save Java class files. Barista also features a Java API to access the library directly from Java. There are two representations: a low-level representation, structurally equivalent to the class file format as defined by Sun, and a higher level representation in which the constant pool indices are replaced by the actual data and the flags are replaced by enumerated types. Both representations are less factorized than in Javalib and, unlike Javalib, Barista does not encode the structural constraints into the OCaml structures. Moreover, it is mainly designed to manipulate single classes and does not offer the optimizations required to manipulate sets of classes (lazy parsing, hash-consing, etc).

2 High-level Representation of Classes

Sawja is built on top of Javalib, a Java bytecode parser providing basic services for manipulating class files, i.e., an optimised high-level representation of class files, pretty printing and unparsing of class files.[6] Javalib handles all aspects of class files, including stackmaps (J2ME and Java 6) and Java 5 annotation attributes. It is made of three modules: $\boxed{\text{Javalib}}$, $\boxed{\text{JBasics}}$, and $\boxed{\text{JCode}}$[7].

Representing class files constitutes the low-level part of a bytecode manipulation library. Our design choices are driven by a set of principles which are explained below.

Strong typing. We use the OCaml type system to make explicit as much as possible the structural constraints of the class file format. For example, interfaces are only signaled by a flag in the Java class file format and this requires to check several consistency constraints between this flag and the content of the class (interface methods must be abstract, the super-class must be `java.lang.Object`, etc.). Our representation distinguishes classes and interfaces and these constraints are therefore expressed and enforced at the type level. This has two advantages. First, this lets the user concentrate on admissible class files, by reducing the burden of handling illegal cases. Second, for the generation (or transformation) of class files, this provides good support for creating correct class files.

Factorization. Strong typing sometimes lacks flexibility and can lead to unwanted code duplication. An example is the use of several, distinct notions of types in class files at different places (JVM types, Java types, and JVM array types). We factorize common elements as much as possible, sometimes by a compromise on strong typing, and by relying on specific language features

[6] Javalib is a sub-component of Sawja, which, while being tightly integrated in Sawja, can also be used as an independent library. It was initiated by Nicolas Cannasse before 2004 but, since 2007, we have largely extended the library. We are the current maintainers of the library.

[7] In the following, we use boxes around Javalib and Sawja module names to make clickable links to the on-line API documentation

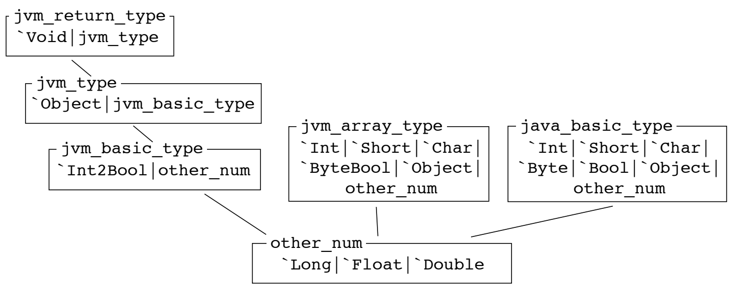


Fig. 1. Hierarchy of JAVA bytecode types. Links represent the subtyping relation enforced by polymorphic variants (for example, the type `jvm_type` is defined by **`type`** `jvm_type = [ |'Object |jvm_basic_type ]`).

such as polymorphic variants[8]. Fig. 1 describes the hierarchy formed by these types. This factorization principle applies in particular to the representation of op-codes: many instructions exist whose name only differ in the JVM type of their operand, and variants exist for particular immediate values (e.g., `iload`, `aload`, `aload_$n$`, etc.). In our representation they are grouped into families with the type given as a parameter (`OpLoad` **`of`** `jvm_type * int`).

Lazy Parsing. To minimise the memory footprint, method bodies are parsed on demand when their code is first accessed. This is almost transparent to the user thanks to the `Lazy` OCAML library but is important when dealing with very large programs. It follows that dead code (or method bodies not needed for a particular analysis) does not cause any time or space penalty.

Hash-consing of the Constant Pool. For a JAVA class file, the constant pool is a table which gathers all sorts of data elements appearing in the class, such as Unicode strings, field and method signatures, and primitive values. Using the constant pool indices instead of actual data reduces the class files size. This low-level aspect is abstracted away by the JAVALIB library, but the sharing is retained and actually strengthened by the use of *hash-consing*. Hash-consing [11] is a general technique for ensuring maximal sharing of data-structures by storing all data in a hash table. It ensures unicity in memory of each piece of data and allows to replace structural equality tests by tests on pointers. In JAVALIB, it is used for constant pool items that are likely to occur in several class files, i.e., class names, and field and method signatures. Hash-consing is global: a class name like `java.lang.Object` is therefore shared across all the parsed class files. For JAVALIB, our experience shows that hash-consing is always a winning strategy; it reduces the memory footprint and is almost unnoticeable in terms of running time[9]. We implement a variant which assigns hash-consed values a unique (integer) identifier. It enables optimised algorithms and data-structures.

[8] Polymorphic variants are a particular notion of enumeration which allows the sharing of constructors between types.
[9] The indexing time is compensated by a reduced stress on the garbage collector.

In particular, the JAVALIB API features sets and maps of hash-consed values based on Patricia trees [23], which are a type of prefix tree. Patricia trees are an efficient purely functional data-structure for representing sets and maps of integers, i.e., identifiers of hash-consed values. They exhibit good sharing properties that make them very space efficient. Patricia trees have been proved very efficient for implementing flow-sensitive static analyses where sharing between different maps at different program points is crucial. On a very small benchmark computing the transitive closure of a call graph, the indexing makes the computation time four times smaller. Similar data-structures have been used with success in the Astrée analyzer [3].

Visualization. SAWJA includes functions to print the content of a class into different formats. A first one is simply raw text, very close to the bytecode format as output by the `javap` command (provided with Sun's JDK).

A second format is compatible with Jasmin [22], a JAVA bytecode assembler. This format can be used to generate incorrect class files (e.g., during a JAVA virtual machine testing), which are difficult to generate with our framework. The idea is then, using a simple text editor, to manually modify the Jasmin files output by SAWJA and then to assemble them with Jasmin, which does not check classes for structural constraints.

Finally, SAWJA provides an HTML output. It allows displaying class files where the method code can be folded and unfolded simply by clicking next to the method name. It also makes it possible to open the declaration of a method by clicking on its signature in a method call, and to know which method a method overrides, or by which methods a method is overridden, etc. User information can also be displayed along with the code, such as the result of a static analysis. From our experience, it allows a faster debugging of static analyses.

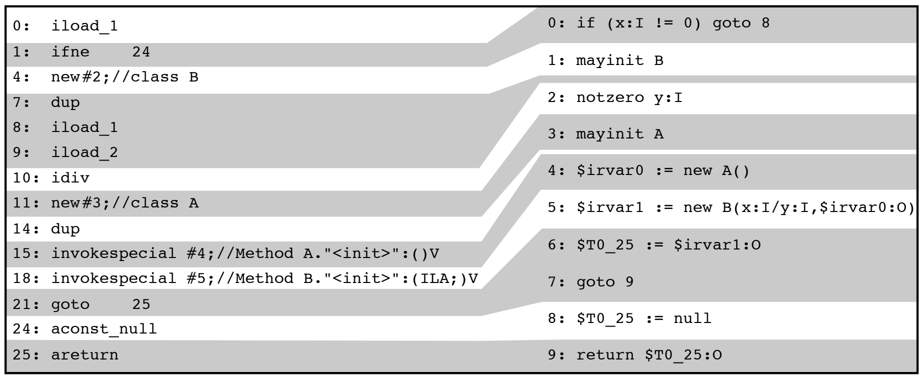
3 Intermediate Representation

The JVM is a stack-based virtual machine and the intensive use of the operand stack makes it difficult to adapt standard static analysis techniques that have been first designed for more classic variable-based codes. Hence, several bytecode optimization and analysis tools work on a bytecode *intermediate representation* (IR) that makes analyses simpler [6,29]. Surprisingly, the semantic foundations of these transformations have received little attention. The transformation that is informally presented here has been formally studied and proved semantics-preserving in [10].

3.1 Overview of the IR Language

Fig. 2 gives the bytecode and IR versions of the simple method

```
B f(int x, int y) { return (x==0)?(new B(x/y, new A())):null;}
```



```
0:  iload_1
1:  ifne    24
4:  new#2;//class B
7:  dup
8:  iload_1
9:  iload_2
10: idiv
11: new#3;//class A
14: dup
15: invokespecial #4;//Method A."<init>":()V
18: invokespecial #5;//Method B."<init>":(ILA;)V
21: goto    25
24: aconst_null
25: areturn
```

```
0: if (x:I != 0) goto 8
1: mayinit B
2: notzero y:I
3: mayinit A
4: $irvar0 := new A()
5: $irvar1 := new B(x:I/y:I,$irvar0:O)
6: $T0_25 := $irvar1:O
7: goto 9
8: $T0_25 := null
9: return $T0_25:O
```

Fig. 2. Example of bytecode (left) (obtained with `javap -c`) and its corresponding IR (right). Colors make explicit the boundaries of related code fragments.

The bytecode version reads as follows : the value of the first argument `x` is pushed on the stack at program point 0. At point 1, depending on whether `x` is zero or not, the control flow jumps to point 4 or 24 (in which case the value **`null`** is returned). At point 4, a new object of class `B` is allocated in the heap and its reference is pushed on top of the operand stack. Its address is then duplicated on the stack at point 7. Note the object is *not initialized* yet. Before the constructor of class `B` is called (at point 18), its arguments must be computed: lines 8 to 10 compute the division of `x` by `y`, lines 11 to 15 construct an object of class `A`. At point 18, the non-virtual method `B` is called, consuming the three top elements of the stack. The remaining reference of the `B` object is left on the top of the stack and represents from now on an *initialized* object.

The right side of Fig. 2 illustrates the main features of the IR language.[10] First, it is *stack-less* and manipulates *structured expressions*, where variables are annotated with *types*. For instance, at point 0, the branching instruction contains the expression `x:I`, where `I` denotes the type of Java integers. Another example of recovered structured expression is `x:I/y:I` (at point 5). Second, expressions are *error-free* thanks to explicit checks: the instruction `notzero y:I` at point 2 ensures that evaluating `x:I/y:I` will not raise any error. Explicit checks additionally guarantee that the order in which *exceptions* are raised in the bytecode is preserved in the IR. Next, the object creation process is syntactically simpler in the IR because the two distinct phases of (i) allocation and (ii) constructor call are merged by *folding* them into a single IR instruction (see point 4). In order to simplify the design of static analyses on the IR, we forbid side-effects in expressions. Hence, the nested object creation at source level is decomposed into two assignments (`$irvar0` and `$irvar1` are temporary variables introduced by the transformation). Notice that because of side-effect free expressions, the order

[10] For a complete description of the IR language syntax, please refer to the API documentation of the $\boxed{\text{JBir}}$ module. A 3-address representation called $\boxed{\text{A3Bir}}$ is also available where each expression is of height at most 1.

in which the `A` and `B` objects are allocated must be reversed. Still, the IR code is able to preserve the *class initialization order* using the dedicated instruction `mayinit` that calls the static class initializer whenever it is required.

3.2 IR Generation

The purpose of the SAWJA library is not only static analysis but also lightweight verification [25]: the verification of the result of a static analysis, i.e., checking that it is indeed a fixpoint, in a single pass over the method code. To this end, our transforming algorithm operates in a fixed number of passes on the bytecode, i.e., without performing fixpoint iteration.

JAVA subroutines (bytecodes `jsr/ret`) are inlined. Subroutines have been pointed out by the research community as raising major static analysis difficulties [27]. Our restricted inlining algorithm cannot handle nested subroutines but is sufficient to inline all subroutines from Sun's JAVA 7 JRE.

The IR generation is based on a symbolic execution of the bytecode: each bytecode modifies a stack of symbolic expressions, and potentially gives rise to the generation of IR instructions. For instance, bytecodes at lines 8 and 9 (left part of Fig. 2) respectively push the expressions `x` and `y` on the symbolic stack (and do not generate IR instructions). At point 10, both expressions are consumed to build both the IR explicit check instruction and the expression `x/y` which is then pushed, as a result, on the symbolic stack. The non-iterative nature of our algorithm makes the transformation of jumping instructions non-trivial. Indeed, during the transformation, the output symbolic stack of a given bytecode is used as the entry symbolic stack of all its successors. At a join point, we thus must ensure that the entry symbolic stack is the same regardless of its predecessors. The idea is here to empty the stack at branching points and restore it at join points, using dedicated temporary variables. More details can be found in [10]. IR expression types are computed using a standard type inference algorithm similar to what is done by the BCV. It only differs in the type domain we used, which is less precise, but does not require iterating. This additionally allows us interleaving expression typing with the IR generation, thus resulting in a gain in efficiency. This lack of precision could be easily filled in using the stackmaps proposed in the JAVA 6 specification.

3.3 Experiments

We validate the SAWJA IR with respect to two criteria. We first evaluate the time efficiency of the IR generation from JAVA bytecode. Then, we show that the generated code contains a reasonable number of local variables. We additionally compare our tool with the SOOT framework. Our benchmark libraries are real-size JAVA code available in `.jar` format. This includes Javacc 4.0 (JAVA Compiler Compiler), JScience 4.3 (a comprehensive JAVA library for the scientific community), the JAVA runtime library 1.5.0_12 and SOOT 2.2.3.

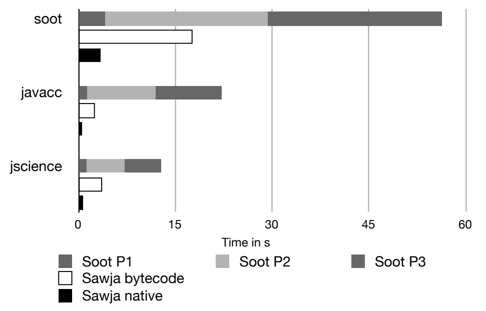


Fig. 3. SAWJA and SOOT IR generation times

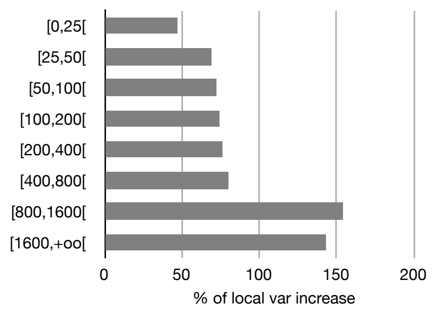


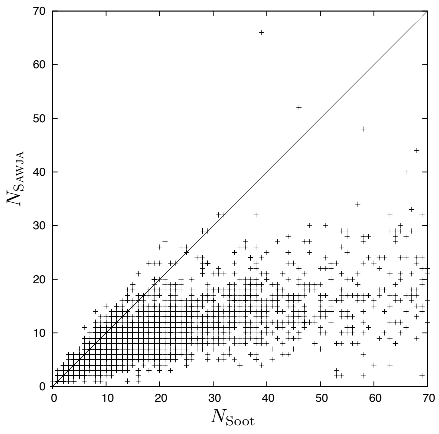
Fig. 4. SAWJA: local variable increase

IR Generation Time. In order to be usable for lightweight verification, the bytecode transformation must be efficient. This is mainly why we avoid iterative techniques in our algorithm. We compare the transformation time of our tool with the one of SOOT. The results are given in Fig. 3. For each benchmark library[11], we compare our running time for transforming all classes with the running time of SOOT. Here, we choose to generate with SOOT the Grimp representation of classes[12], the closest IR to ours that SOOT provides. Grimp allows expressions with side-effects, hence expressions are somewhat more aggregated than in our IR. However, this does not inverse the trend of results. We rely on the time measures provided by SOOT, from which we only keep three phases: generation of naive Jimple 3-address code (P1), local def/use analysis used to simplify this naive code (P2), and aggregation of expressions to build Grimp syntax (P3). (Other phases, like typing, are not directly relevant.) Unlike JAVA code, OCAML code is usually executed in native form. For the comparison not to be biaised, we compare execution times of both tools in bytecode form and also give the execution time of SAWJA in native form. These experiments show that SAWJA (both in bytecode and native mode) is very competitive with respect to SOOT, in terms of computation efficiency. This is mainly due to the fact that, contrary to SOOT, our algorithm is non-iterative.

Compactness of the Obtained Code. Intermediate representations rely on temporary variables in order to remove the use of operand stack and generate side-effect free expressions. The major risk here is an explosion in the number of new variables when transforming large programs.

In practice our tool stays below doubling the number of local variables, except for very large methods ($>$ 800 bytecodes). Fig. 4 presents the percentage of local variable increase induced by our transformation, for each method of our benchmarks, and sorting results according to the method size (indicated by numbers in brackets). The number of new variables stays manageable and we

[11] For scale reason, the JAVA runtime library measures are not shown here.
[12] The SOOT transformation is without any optimisation option.



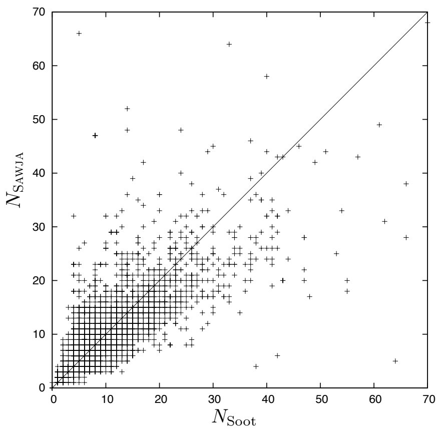


Fig. 5. Local variable increase ratio between Sawja and Soot.

believe it could be further reduced using standard optimization techniques, as those employed by Soot, but this would require to iterate on each method.

We have made a direct comparison with Soot in terms of the local variable increase. Fig. 5 presents two measures. For each method of our benchmarks we count the number N_{SAWJA} of local variables in our IR code and the number N_{Soot} of local variables in the code generated by Soot. A direct comparison of our IR against Grimp code is difficult because it allows expressions with side-effects, thus reducing the amount of required variables. Hence, in this experiment, the comparison is made between Soot's 3-address IR (Jimple) and our 3-address IR. For each method we draw a point of coordinate $(N_{Soot}, N_{\text{SAWJA}})$ and see how the points are spread out around the first bisector. For the left diagram, Soot has been launched with default options. For the right diagram, we added to the Soot transformation the local packer that reallocates local variables using use/def information (and hence increases the transformation time). Our transformation competes well, even when Soot uses this last optimization. We could probably improve this ratio using a similar packing, but this would require to iterate on the code.

4 Complete Programs

Whole program analyses require a model of the global control-flow graph of an entire Java program. For those, Sawja proposes the notion of *complete programs*. Complete programs are equipped with a high-level API for navigating the control-flow graph and are constructed by a preliminary control-flow analysis.

4.1 API of Complete Programs

Sawja represents a complete program by a record. The field `classes` maps a class name to a class node in the class hierarchy. The class hierarchy is such that any class referenced in the program is present. The field `parsed_methods` maps a fully qualified method name to the class node declaring the method and the implementation of the method. The field `static_lookup_method` returns the set of target methods of a given field. As it is computed statically, the target methods are an over-approximation.

The API allows navigating the intra-procedural graph of a method taking into account jumps, conditionals and exceptions. Although conceptually simple, field and method resolution and the different method look-up algorithms (corresponding to the instructions `invokespecial`, `invokestatic`, `invokevirtual`, `invokeinterface`) are critical for the soundness of inter-procedural static analyses. In SAWJA, great care has been taken to ensure an implementation fully compliant with the JVM specification.

4.2 Construction of Complete Programs

Computing the exact control-flow graph of a JAVA application is undecidable and computing a precise (over-)approximation of it is still computationally challenging. It is a field of active research (see for instance [19,4]). A complete program is computed by: (1) initializing the set of reachable code to the entry points of the program, (2) computing the new call graph, and (3) if a (new) edge of the call graph points to a new node, adding the node to the set of reachable code and repeating from step (2). The set of code obtained when this iteration stops is an over-approximation of the complete program.

Computing the call graph is done by resolving all reachable method calls. Here, we use the functions provided in the SAWJA API presented in Sect. 4.1. While `invokespecial` and `invokestatic` instructions do not depend on the data of the program, the function used to compute the result of `invokevirtual` and `invokeinterface` need to be given the set of object types on which the virtual method may be called. The analysis needs to have an over-approximation of the types (classes) of the objects that may be referenced by the variable on which the method is invoked.

There exists a rich hierarchy of control-flow analyses trading time for precision [28,12]. SAWJA implements the fastest and most cost-effective control-flow analyses, namely Rapid Type Analysis (RTA) [1], XTA [28] and Class Reachability Analysis (CRA), a variant of Class Hierarchy Analysis [9].

Soundness. Our implementation is subject to the usual caveats with respect to reflection and native methods. As these methods are not written in JAVA, their code is not available for analysis and their control-flow graph cannot be safely abstracted. Note that our analyses are always correct for programs that use neither native methods nor reflection. Moreover, to alleviate the problem, our RTA implementation can be parametrised by a user-provided abstraction of native methods specifying the classes it may instantiate and the methods it may call. A better account of reflection would require an inter-procedural string analysis [21] that is currently not implemented.

Implemented Class Analyses

RTA. An object is abstracted by its class and all program variables by the single set of the classes that may have been instantiated, i.e., this set abstracts all the objects accessible in the program. When a virtual call needs to be resolved, this set is taken as an approximation of the set of objects that may be referenced by the variable on which the method is called. This set grows as the set of reachable methods grows.

SAWJA's implementation of RTA is highly optimized. While static analyses are often implemented in two steps (a first step in which constraints are built, and a second step for computing a fixpoint), here, the program is unknown at the beginning and constraints are added on-the-fly. For a faster resolution, we cache all reachable virtual method calls, the result of their resolution and intermediate results. When needed, these caches are updated at every computation step. The cached results of method resolutions can then be reused afterwards, when analyzing the program.

XTA. As in RTA, an object is abstracted by its class and to every method and field is attached a set of classes representing the set of objects that may be accessible from the method or field. An object is accessible from a method if: (i) it is accessible from its caller and it is of a sub-type of a parameter, or (ii) it is accessible from a static field which is read by the method, (iii) it is accessible from an instance field which is read by the method and there an object of a sub-type of the class in which the instance fields is declared is already accessible, or (iv) it is returned by a method which may be called from the current method.

To facilitate the implementation, we built this analysis on top of another analysis to refine a previously computed complete program. This allows us using the aforementioned standard technique (build then solve constraints). For the implementation, we need to represent many class sets. As classes are indexed, these sets can be implemented as sets of integers. We need to compute fast union and intersection of sets and we rarely look for a class in a set. For those reasons, the implementation of sets available in the standard library in OCAML, based on balanced trees, was not well adapted. Instead we used a purely functional set representation based on Patricia trees [23], and another based on BDDs [5] (using the external library BuDDy available at `http://buddy.sourceforge.net`).

CRA. This algorithm computes the complete program without actually computing the call graph or resolving methods: it considers a class as *accessible* if it is referenced in another class of the program, and considers all methods in reachable classes as also reachable. When a class references another class, the first one contains in its constant pool the name of the later one. Combining the lazy parsing of our library with the use of the constant pool allows quickly returning a complete program without even parsing the content of the methods. When an actual method resolution, or a call graph, is needed, the Class Hierarchy Analysis (CHA) [9] is used. Although parts of the program returned by CRA will be parsed during the overlying analysis, dead code will never by parsed.

		Soot	Jess	Jml	VNC	ESC/Java	JDTCore	Javacc	JLex
C	CRA	5,198	5,576	2,943	5,192	2,656	2,455	2,172	2,131
	RTA	4,116	2,222	1,641	1,736	1,388	1,163	792	752
	CRA	49,810	47,122	26,906	44,678	23,229	23,579	19,389	18,485
	W-RTA	32,652	4,303	17,740	?	9,560	7,378	3,247	1,419
M	RTA	32,800	12,561	11,697	9,218	8,305	9,137	4,029	3,157
	XTA	14,251	10,043	9,408	6,534	7,039	8,186	3,250	2,392
	W-0CFA	37,768	9,927	15,414	?	9,088	6,830	3,009	1,186
	CRA	2,159,590	799,081	418,951	694,451	354,234	347,388	258,674	244,071
	W-RTA	2,788,533	78,444	614,216	?	279,232	146,119	34,192	13,256
E	RTA	1,400,958	141,910	149,209	79,029	101,257	114,454	35,727	23,209
	XTA	297,754	94,189	103,126	48,817	74,007	86,794	26,844	15,456
	W-0CFA	856,180	183,191	187,177	?	87,163	77,875	21,475	4,360
	CRA	8	8	4	7	4	5	4	4
	W-RTA	74	7	23	?	12	12	7	5
T	RTA	13	4	4	3	3	4	2	2
	XTA	187	18	16	11	10	14	5	4
	W-0CFA	2,303	209	40	?	27	26	16	7
	CRA	87	83	51	80	45	47	36	35
	W-RTA	248	44	128	?	84	101	42	8
S	RTA	132	60	54	51	43	52	26	20
	XTA	810	198	184	153	147	157	112	107
	W-0CFA	708	238	215	?	132	134	125	26

Table 1. Comparison of algorithms generating a program call graph (with SAWJA and WALA): the algorithms of SAWJA (CRA,RTA and XTA) are compared to WALA (W-RTA and W-0CFA) w.r.t the number of loaded classes (C), reachable methods (M) and number of edges (E) in the call graph, their execution time (T) in seconds and memory used (S) in megabytes. Question marks (?) indicate clearly invalid results.

Experimental Evaluation. We evaluate the precision and performances of the class analyses implemented in SAWJA on several pieces of JAVA software[13] and present our results in Table 1. We compared the precision of the 3 algorithms used to compute complete programs (CRA, RTA and XTA) with respect to the number of reachable methods in the call graph and its number of edges. We also give the number of classes loaded by CRA and RTA. We provide some results obtained with WALA (r3767). Although precision is hard to compare[14], it indicates that, on average, SAWJA uses half the memory and time used by WALA per reachable method with RTA.

Conclusion

We have presented the SAWJA library, the first OCAML library providing state-of-the-art components for writing JAVA static analyzers in OCAML.

[13] Soot (2.3.0), Jess (7.1p1), JML (5.5), TightVNC Java Viewer (1.3.9), ESC/Java (2.0b0), Eclipse JDT Core (3.3.0), Javacc (4.0) and JLex (1.2.6).

[14] Because both tools are unsound, a greater number of method in the call graph either mean there is a precision loss or that native methods are better handled.

The library represents an effort of 1.5 man-year and approximately 22000 lines of OCaml (including comments) of which 4500 are for the interfaces. Many design choices are based on our earlier work with the NIT analyzer [13]. It is a quite efficient tool, able to analyze a complete program of more than 3000 classes and 26000 methods to infer nullness annotations for fields, method signatures and local variables to prove the safety of 84% of dereferences in less than 2 minutes. Using our experience from the NIT development, we designed Sawja as a generic framework to allow every new static analysis prototype to share the same efficient components as NIT. Indeed, Sawja has already been used in two implementations for the ANSSI (The French Network and Information Security Agency) [16,14]; Nit has been ported to the current version of Sawja, improving its performances by 30% in our first tests; while being integrated in Sawja, the class analyses presented in Section 4.2 rely on the underlying features and can be seen as use cases of Sawja; and other small analyses (liveness, interval analyses, etc.) are also available on Sawja's web site.

Several extensions are planned for the library. Displaying static analysis results is a first challenge that we would like to tackle. We would like to facilitate the transfer of annotations from Java source to Java bytecode and then to IR, and the transfer of analysis results in the opposite direction. We already provide HTML outputs but ideally the result at source level would be integrated in an IDE such as Eclipse. This manipulation has been already experimented in one of our earlier work for the NIT static analyzer and we plan to integrate it as a new generic Sawja component. To ensure correctness, we would like to replace some components of Sawja by certified extracted code from Coq [8] formalizations. A challenging candidate would be the IR generation that relies on optimized algorithms to transform in at most three passes each bytecode method. We would build such a work on top of the Bicolano [2] JVM formalization that has been developed by some of the authors during the European Mobius project.

References

1. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of OOPSLA'96*, pages 324–341, 1996.
2. Bicolano - web home. `http://mobius.inria.fr/bicolano`.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of PLDI'03*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
4. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, 2009.
5. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.
6. M G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proc. of JAVA'99*, pages 129–141. ACM, 1999.
7. Xavier Clerc. Barista. `http://barista.x9c.fr/`.
8. The Coq Proof Assistant. `http://coq.inria.fr/`.

9. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of ECOOP'95*, volume 952 of *LNCS*, pages 77–101. Springer, August 1995.
10. D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for Java bytecode. Research Report 7021, INRIA, 2009. `http://www.irisa.fr/celtique/ext/bir/rr7021.pdf`.
11. A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.
12. D. Grove and C. Chambers. A framework for call graph construction algorithms. *Toplas*, 23(6):685–746, 2001.
13. L. Hubert. A Non-Null annotation inferencer for Java bytecode. In *Proc. of PASTE'08*, pages 36–42. ACM, November 2008.
14. L. Hubert, T. Jensen, V. Monfort, and D. Pichardie. Enforcing secure object initialization in Java. In *Proc. of ESORICS*, LNCS, 2010. To appear.
15. IBM. The T.J. Watson Libraries for Analysis (Wala). `http://wala.sourceforge.net`.
16. T. Jensen and D. Pichardie. Secure the clones: Static enforcement of policies for secure object copying. Technical report, INRIA, June 2010. Presented at OWASP 2010.
17. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system.* Inria, May 2007. `caml.inria.fr/ocaml/`.
18. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proc. of CC*, volume 2622 of *LNCS*, pages 153–169. Springer, 2003.
19. O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.
20. T. Lindholm and F. Yellin. *The Java$^{\text{TM}}$ Virtual Machine Specification, Second Edition.* Prentice Hall PTR, 1999.
21. V. Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *Proc. of APLAS*, pages 139–160. Springer, 2005.
22. J. Meyer and T. Downing. *Java Virtual Machine.* O'Reilly Associates, 1997. `http://jasmin.sourceforge.net`.
23. D. R. Morrison. PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4), 1968.
24. B. Pagano, O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury, and J.L. Colaço. Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework. In *Proc. of ICFP*, pages 215–220. ACM, 2009.
25. E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.
26. F. Spoto. JULIA: A generic static analyser for the Java bytecode. In *Proc. of the Workshop FTfJP*, 2005.
27. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc of POPL,98*, pages 149–160. ACM Press, 1998.
28. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. of OOPSLA'00*, pages 281–293. ACM Press, October 2000.
29. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java bytecode optimization framework. In *Proc. of CASCON*, 1999.
30. J. Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.