Evaluating Call Graph Construction for JVM-hosted Language Implementations

David R. Cheriton School of Computer Science Technical Report CS-2015-03

Xiaoni Lai
University of Waterloo
xiaoni.lai@uwaterloo.ca

Zhaoyi Luo
University of Waterloo
zhaoyi.luo@uwaterloo.ca

Karim Ali
Technische Universität Darmstadt
karim.ali@cased.de

Ondřej Lhoták
University of Waterloo
olhotak@uwaterloo.ca

Julian Dolby
IBM T.J. Watson Research Center
dolby@us.ibm.com

Frank Tip
Samsung Research America
ftip@samsung.com

ABSTRACT

An increasing number of programming languages compile to the Java Virtual Machine (JVM), and program analysis frameworks such as WALA and SOOT support a broad range of program analysis algorithms by analyzing bytecode. While this approach works well when applied to bytecode produced from Java code, its efficacy when applied to other bytecode has not been studied until now.

We present qualitative and quantitative analysis of the soundness and precision of call graphs constructed from JVM bytecodes produced for Python, Ruby, Clojure, Groovy, Scala, and OCaml applications. We show that, for Python, Ruby, Clojure, and Groovy, the call graphs are unsound due to use of reflection, invokedynamic instructions, and run-time code generation, and imprecise due to how function calls are translated. For Scala and OCaml, all unsoundness comes from rare, complex uses of reflection and proxies, and the translation of first-class features in Scala incurs a significant loss of precision.

1. INTRODUCTION

The Java Virtual Machine (JVM) has been used to implement programming languages such as Python [21], Ruby [36], Clojure [15], Groovy [3], Scala [25], and OCaml [23]. By compiling these languages to JVM bytecode, language implementors significantly reduce the work to implement their languages. Moreover, JVMs are available for a wide range of platforms, making portability easier.

The Java Virtual Machine was designed for portable and efficient implementation of Java. By defining a relatively small set of bytecode instructions with clear semantics, the task of creating an interpreter or just-in-time compiler for Java is simplified significantly. The virtual machine abstracts away from the complexity and idiosyncrasies of Java.

The JVM has become a popular platform for developing static program analysis frameworks for Java, such as WALA [17] and SOOT [37]. These frameworks support a broad range of algorithms for static pointer analysis, call graph construction, data-flow analysis, and others. A JVM-based approach works well for Java because JVM bytecode is fairly close to Java. As a result, bytecode-based analysis frameworks are widely used in academia and industry.

We investigate how well this JVM-bytecode-based approach works when applied to bytecode produced from other languages. This is unknown as there is generally a much larger "gap" between source code and the JVM bytecodes to which they are translated. For example, we found that the Jython compiler translates a single function call in a Python program into a sequence of 5 method calls in bytecode, as will be discussed in Section 3.

We focus on call graph construction because call graphs are a prerequisite for most other program analysis tasks. We will examine the bytecodes for programs in the Python, Ruby, Clojure, Groovy, Scala, and OCaml languages. We study two issues: (i) the soundness of static call graphs computed from JVM bytecode for each language (i.e., whether they contain all methods and call edges that can arise in execution), and (ii) the precision of the static call graphs are (i.e., how many nodes and edges they contain that cannot arise in any program execution).

We conduct qualitative and quantitative experiments for each language. In the qualitative experiments, we inspect call graphs constructed by compiling a small "standard" example to bytecode, where we focus on the translation of function and method calls. We look for use of reflection, dynamic code generation, and invokedynamic instructions that challenge static analysis.

For the quantitative experiments, we use 11 programs from the Computer Language Benchmark Game suite [12] (hereinafter CLBG) with versions available in each language[1]. After compiling these programs to JVM bytecode, we construct static call graphs using a standard 0-CFA analysis [35] that is part of the WALA program analysis framework. We construct dynamic call graphs using an instrumentation-based dynamic call graph builder also part of WALA. Because the JVM cannot run with an instrumented version of

[1]FASTAREDUX does not have an implementation in Python, Groovy, and OCaml. Also, KNUCLEOTIDE and FANNKUCHREDUX do not have implementations in Groovy. For missing implementations, we ported existing implementations in CLBG to the target language. We verified correctness by comparing their output against expected output detailed in CLBG.

the Java standard library, we abstract the Java standard library with a single node in both the dynamic and static call graphs. Using ProBe [19], a call graph comparison tool, we measure unsoundness by identifying nodes and edges in the dynamic call graph but not the static one. Similarly, potential imprecision is identified when static call graphs are much larger than dynamic ones. We manually examine a number of such cases to identify if the static analysis is imprecise, or if the discrepancy occurs because code coverage is low. We conducted all of our experiments using Oracle's Java 8u25 JVM running on a machine with eight dual-core AMD Opteron 1.4 GHz CPUs (running in 64-bit mode) and capped the available RAM at 16 GB.

We conclude that call graphs constructed for the dynamically typed languages Python, Ruby, Clojure, and Groovy using bytecode-based static analysis are unsound, because of pervasive use of reflection, dynamic code generation, and invokedynamic instructions. Even if these challenges were overcome, the call graphs constructed for several of these languages (Python and Ruby) would remain highly imprecise because of the way in which function calls are translated.

For the statically typed languages OCaml and Scala, the results are better. All unsoundness comes from uses of reflection and proxies, which occur rarely in practice. In the case of Scala, precision is degraded because type information is lost when compiling features such as closures. The OCaml compiler uses MethodHandles to implement closures, but in a way that can be analyzed soundly by a static analysis.

The rest of this paper is organized as follows. Section 2 briefly reviews MethodHandles and invokedynamic, two recently introduced JVM features already used by several of the languages. Then, each of Sections 3–8 presents the experiments for one of the languages. Section 9 discusses threats to validity. Related work is discussed in Section 10. We conclude in Section 11.

2. BACKGROUND

In Java 7, the Java Virtual Machine was extended with MethodHandles and invokedynamic instructions, two features that facilitate the implementation of dynamic languages by deferring until run time the association between call sites and the methods that they invoke. These features are being adopted rapidly by language implementors, and several of the language implementations being studied in this paper already make use of them[2]. These dynamic features pose new challenges for static analysis, but the static analysis community has not paid significant attention to them until now. Therefore, we provide a brief review of the new JVM features, and the challenges they pose for static analysis.

MethodHandles.

According to the Java documentation, a method handle is "a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values"[3]. Informally, a method handle is a constant value that uniquely identifies a method and how it should be invoked (e.g., as a static call, or a virtual call). Furthermore, method handles can apply transformations to the sequence of arguments passed to the encapsulated method (e.g., unpacking an array into a sequence containing its values).

Method handles can be embedded in a class file's constant pool as constants to be loaded using ldc instructions. A new type of constant pool entry, CONSTANT_MethodHandle, refers directly to an associated CONSTANT_Methodref, CONSTANT_InterfaceMethodref, or CONSTANT_Fieldref constant pool entry. Alternatively, method handles can be created at run time by calling one of the factory methods in class java.lang.invoke.MethodHandles.Lookup (e.g., MethodHandles. Lookup.findVirtual), with arguments that specify the encapsulated method's parameter types and return type.

The method encapsulated by a method handle can be invoked by calling the MethodHandle.invoke() method, with arguments that should be bound to the method's receiver (in the case of virtual methods) and formal parameters.

In effect, the functionality provided by method handles is similar to that provided by the Java reflection API, but access checking is performed only once, upon creation of the handle, whereas java.lang.reflect.Method.invoke() performs an access check for each reflective call.

The invokedynamic *instruction.*

The invokedynamic instruction provides a mechanism for dynamically binding a method call to a target method at run time. It works as follows:

- When an invokedynamic instruction executes for the first time, its associated *bootstrap method* is executed. The association between invokedynamic instructions and their associated bootstrap methods is recorded in the *bootstrap table*, a new component of JVM .class files.
- A bootstrap method returns a java.lang.invoke.CallSite object that encapsulates a MethodHandle that identifies the method to be invoked. This method can be retrieved using the CallSite.getTarget() method, which is automatically invoked by the JVM at run time.
- The CallSite object returned by a bootstrap method is cached, so that for subsequent executions of an invokedynamic instruction, the JVM only needs to retrieve the method handle by executing CallSite.getTarget().

This method call resolution mechanism is considerably more flexible than the one that is used for the other JVM instructions for calling methods. In particular, invokevirtual and invokeinterface instructions specify a target method, and a call made through one of these instructions dispatches to a method that transitively overrides this target method. In other words, for invokevirtual and invokeinterface instructions, the name and parameter types of the method to be invoked are known at compile time, and a static analysis can analyze the inheritance hierarchy to conservatively approximate the set of methods that may be invoked by the call.

In the case of invokedynamic, there is no obvious way for a static analysis to approximate the set of possible call targets. The code in bootstrap methods can be arbitrarily complex, and there are no compile-time constraints on the name and parameter types of the method that is invoked subsequently. Further complicating matters, CallSite objects returned by bootstrap methods may be *mutable* call sites, for which the encapsulated method handle may be updated at run time.

[2]Note that, starting with Java 8, the bytecodes generated from Java programs also make use of invokedynamic when lambda-expressions (closures) are being compiled. Thus, the analysis challenges noted here are broadly applicable to statically typed and dynamically typed languages.

[3]See `http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html`.

```
1 ##hello.py
2 def foo():
3     bar()
4 def bar():
5     print "hello world"
6 foo()
```

Figure 1: A simple Python program.

3. PYTHON

Python [21] is a popular dynamically-typed object-oriented programming language. In addition to classes and objects, it supports lists, sets, and dictionaries as built-in data structures. Other key Python features include lambda expressions, comprehensions, and generators. Jython [18] is an implementation of Python that runs on the JVM. In our experiments, we used Jython version 2.7-b3, which is compatible with Python 2.7. Jython is closely integrated with the JVM platform, and allows users to import Java classes and export projects as standard .jar files.

3.1 Translation to JVM bytecode

We will use the small Python program of Figure 1 to illustrate how Jython translates Python source code to JVM .class files. This program declares two functions, foo and bar. The program calls foo on line 6, foo calls bar on line 3, which in turns prints "hello world" on line 5.

For the program of Figure 1, the Jython compiler generates a class hello$py that contains the main application logic. In general, the Jython compiler maps each function call in the Python source code to a sequence of method calls in the generated bytecode. As an example, we consider the translation of the call from foo to bar on line 3 in Figure 1. For this call, the following sequence of calls is generated:

1. hello$py.foo$1() calls a method PyObject.__call__() in the Jython runtime libraries,
2. PyObject.__call__() invokes another library method, PyCode.call(). This call is dynamically dispatched to PyBaseCode.call(),
3. PyBaseCode.call() invokes another call() method in the same class that dispatches to an overriding definition in PyTableCode.
4. PyTableCode.call() invokes hello$py.call_function() in the class containing the translated application functions.
5. hello$py.call_function() contains a switch statement in which each branch calls one of the application functions depending on the value of its first parameter, fid. The value of fid originates from an instance field PyTableCode.func_id that is read in method PyTableCode.call(). In the case where call_function() was indirectly invoked by foo$1, this field will be set in the constructor of PyTableCode to a value that will result in the selection of the branch of the switch that invokes hello$py.bar$2().

3.2 Qualitative Analysis

Suppose we want to construct a call graph for the program of Figure 1 by analyzing the bytecodes generated for it by the Jython compiler. As mentioned, method hello$py.call_function() calls each of f$0(), foo$1(), and bar$2(), which correspond to the top-level code and the functions foo and bar in the Python source code. For any other method call in the program (e.g., the call to foo from top-level code), a similar chain of call edges exists that includes method hello$py.call_function().

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	10,446	1,791	90	8,745	110,533	3,151	229	107,611
FK	10,400	1,784	80	8,696	100,214	3,105	203	97,312
FA	10,478	3,354	161	7,285	110,723	6,693	406	104,436
FR	10,479	3,361	150	7,268	110,605	6,712	383	104,276
KN	10,462	4,950	779	6,291	106,933	10,729	1,618	97,822
MB	10,441	1,801	94	8,734	107,835	3,158	233	104,910
NB	10,462	1,826	89	8,725	109,548	3,246	224	106,526
PD	10,289	1,768	82	8,603	104,166	3,114	213	101,265
RD	10,332	4,737	794	6,389	92,420	9,954	1,542	84,008
RC	10,374	4,525	418	6,267	95,154	9,847	1,039	86,346
SN	10,415	1,807	114	8,722	102,930	3,204	278	100,004

Table 1: Count of nodes and edges in the static and dynamic call graphs of the Jython CLBG programs.

Consequently, *every call site* in a Python source file is translated into a chain of method calls that involves hello$py.call_function(), which calls *every method* corresponding to a function in the same Python file[4].

Based on this observation, it is difficult to see how a bytecode-based analysis of the JVM bytecodes produced by Jython could compute a useful call graph. Projecting chains of call edges involving call_function() to the corresponding functions in the Python source code would result in a complete graph where every function is reachable from every call site. A standard context-sensitive analysis would need to account for at least 4 levels of calling context (i.e., 4-CFA in the terminology of [14]) to be able to resolve method calls inside functions such as hello$py.call_function() precisely. Furthermore, resolving the calls inside the switch statement inside call_function() requires precise tracking of integer numbers stored in heap locations that are used to identify the different functions. This is beyond the current state-of-the-art in call graph construction. Therefore, we conclude that producing precise call graphs from JVM bytecodes produced by Jython is infeasible.

So far, we have only discussed the precision of the constructed call graphs. However, there are significant challenges related to soundness as well. In further experiments, we observed that the constructed call graphs are unsound because all methods in imported modules are missing. Investigation revealed that this is because Jython generates code that relies on reflection to implement module import, and static analysis frameworks such as WALA typically are unable to reason about reflective code.

3.3 Quantitative Analysis

Table 1 shows the number of nodes and edges in the static and dynamic call graphs for the Jython programs in our benchmark suite[5]. Also shown are the number of

[4]Jython generates a separate class for each Python source file, each with its own call_function() method.

[5]Due to space constraints, we encode the names of the CLBG benchmarks in the tables with results as follows: BT=binarytrees, FK=fannkuchredux, FA=fasta, FR=fastaredux, KN=knucleotide, MB=mandelbrot, NB=nbody, PD=pidigits, RD=regexdna, RC=revcomp, SN=spectralnorm.

```
## hello.rb
def bar
  print "Hello, World!\n"
end

def foo
  bar
end

foo
```

Figure 2: A simple Ruby program.

nodes/edges that are in the dynamic call graphs but not in the static call graphs (columns D\S), and that are in the static call graphs but not in the dynamic call graphs (columns S\D) [6]. On average, about 75% of the methods and about 95% of the edges in the static call graphs do not occur in the dynamic call graphs. This significant imprecision mainly arises from the call chains involving call_function() discussed above. Moreover, features such as module imports that are implemented using reflection cause about 7% of the methods and 9% of the edges in the dynamic call graphs to be absent from the static call graphs.

4. RUBY

Ruby is a popular object-oriented programming language that supports *duck typing*, which means that the type of an object is defined by the operations that it supports. One of Ruby's key features is the *code block*, which allows programmers to pass a block of code as a parameter to a function, which can execute it using a yield statement.

JRuby[7] is a Java implementation of Ruby. In our experiments, we used JRuby version 1.7.13. Programmers can invoke "jrubyc" to compile their Ruby application into JVM bytecode. Interestingly, the JRuby compiler can optionally generate code that makes use of the invokedynamic instruction. In order to understand the impact of this feature, we conducted two sets of experiments, one with the use of invokedynamic enabled, and one where it was disabled.

4.1 Translation to JVM bytecode

Figure 2 shows a simple Ruby program similar to the Python program discussed previously. This program defines functions foo and bar, and top-level code that invokes foo. Furthermore, foo calls bar, and bar prints "hello world".

The JRuby compiler translates each Ruby source file into a separate class that defines methods main(), load(), and __file__(). The generated classes contain an additional method for each function in the Ruby source code[8]. For the program of Figure 2, a class hello with methods method__0$RUBY$bar and method__1$RUBY$foo is generated. Each function call in the Ruby source code is translated to a sequence of method calls in the generated bytecode. For the call from foo to bar, the following sequence is generated:

1. Method method__1$RUBY$foo invokes org.jruby.runtime.CallSite.call(...), which dynamically dispatches to org.jruby.runtime.callsite.CachingCallSite.call(...),
2. CachingCallSite.call(...) invokes a method CachingCallSite.cacheAndCall(...) in the same class,
3. Next, CachingCallSite.cacheAndCall(...) retrieves a DynamicMethod from a cache and invokes a method org.jruby.internal.runtime.methods.DynamicMethod.call(...) on that object. This triggers a sequence of calls to methods in the JRuby runtime and Java standard libraries that ultimately invokes a method call(...) in a class hello$method__0$RUBY$bar that is *generated at run time*.
4. Finally, hello$method__0$RUBY$bar.call(...) invokes the method hello.method__0$RUBY$bar.

When the use of invokedynamic instructions is enabled, the bootstrap method used by JRuby returns a MutableCallSite that is initialized with a MethodHandle pointing to the same dynamic dispatch procedure that is used if the use of invokedynamic is disabled. Therefore, the code that actually determines which method will be called is the same in each case. Once the specific target method of a call site has been determined (the first time that the call site is executed), the MethodHandle in the MutableCallSite is replaced by a handle for the actual method, so that the dispatch procedure need not be repeated on subsequent executions of the call site. However, from the point of view of a static analysis, the procedure used to determine the target of a call is equally complicated in either case, because the first execution of the call site uses the same dynamic dispatch procedure.

4.2 Qualitative Analysis

The code generated by the JRuby compiler poses serious challenges to static analysis. The main obstacle is the generation of classes such as hello$method__0$RUBY$bar at run time. A static analysis is unable to reason about the behavior of classes that are not available at analysis time, and will miss calls such as the one discussed above, causing methods such as hello.method__0$RUBY$bar to be absent in the call graph. In other words, the use of run-time code generation causes unsoundness in the static analysis.

In addition, consider that the methods CachingCallSite.call(...) and DynamicMethod.call(...) are invoked for every call that was present in the original application. From these methods, all methods such as hello.method__0$RUBY$bar and hello.method__1$RUBY$foo corresponding to functions in the original application are invoked. Hence, even if a static analysis were somehow able to account for run-time code generation, a straightforward mapping from edges in call graphs generated from JVM bytecodes to source functions would conclude that every call can invoke every function, causing the call graph to become a complete graph. To avoid such imprecision, a static analysis would need to employ at least 3 levels of call-string context-sensitivity, and it would need reason about heap-allocated cached objects as well, which is beyond the current state-of-the-art. Therefore, we conclude that generating precise call graphs from JVM bytecodes produced by JRuby is infeasible.

In Ruby, the require construct is used to import code from another file. Upon examination of code using this construct, we found that this is implemented in the JRuby runtime libraries by dynamically loading a script from a file using a ClassLoader, and then relying on the mechanisms described above to interpret these scripts. In general, static analysis

[6]The tables presented in Section 4–8 for the other languages under consideration have exactly the same structure.
[7]https://github.com/jruby/jruby/wiki/AboutJRuby
[8]In fact, JRuby generates two overloaded methods for each function in the Ruby source code, where one performs some additional checks before invoking the other. In the example under consideration, only the method that does not perform argument-checking is used.

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	14,406	7,202	2,831	10,035	72,193	15,321	8,082	64,954
FK	14,337	7,154	2,816	9,999	71,882	15,181	8,040	64,741
FA	14,385	7,407	2,946	9,924	71,980	15,852	8,511	64,639
FR	14,384	7,420	2,909	9,873	71,980	15,862	8,371	64,489
KN	14,410	7,433	2,948	9,925	72,128	15,900	8,446	64,674
MB	14,324	7,762	3,246	9,808	71,909	16,790	9,242	64,361
NB	14,408	7,325	2,905	9,988	72,121	15,659	8,317	64,779
PD	14,289	6,996	2,804	10,097	71,682	14,876	8,004	64,810
RD	14,404	7,312	2,918	10,010	71,986	15,610	8,342	64,718
RC	14,360	7,232	2,881	10,009	71,980	15,413	8,204	64,771
SN	14,333	7,471	3,058	9,920	71,939	16,101	8,790	64,628

Table 2: Count of nodes and edges in the static and dynamic call graphs of the JRuby CLBG programs.

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	14,326	7,349	2,954	9,931	71,817	15,662	8,470	64,625
FK	14,314	7,319	2,918	9,913	71,774	15,536	8,301	64,539
FA	14,327	7,525	3,033	9,835	71,789	16,218	8,839	64,410
FR	14,326	7,545	2,999	9,780	71,788	16,167	8,633	64,254
KN	14,327	7,557	3,047	9,817	71,812	16,266	8,843	64,389
MB	14,293	7,873	3,303	9,723	71,774	17,289	9,678	64,163
NB	14,315	7,438	3,002	9,879	71,795	16,040	8,735	64,490
PD	14,254	7,136	2,899	10,017	71,556	15,121	8,227	64,662
RD	14,288	7,414	3,026	9,900	71,622	15,843	8,673	64,452
RC	14,297	7,371	2,981	9,907	71,744	15,750	8,527	64,521
SN	14,291	7,571	3,112	9,832	71,766	16,603	9,240	64,403

Table 3: Count of nodes and edges in the static and dynamic call graphs of the JRuby (with invokedynamic support) CLBG programs.

is incapable of precisely accounting for code interpreted at run-time in this way, resulting in additional unsoundness.

4.3 Quantitative Analysis

The sizes of dynamic and static call graphs that we observed for the JRuby benchmarks are shown in Tables 2 (for code generated without invokedynamic) and 3 (for code generated with invokedynamic). The numbers are very similar across all benchmarks because the large JRuby library contains overwhelmingly more methods than the benchmark programs themselves. For reasons described above, the static call graphs miss large numbers of nodes and edges from the dynamic call graphs, primarily because of methods in classes that are generated at run time and therefore not known to the static analysis.

The computed call graphs also exhibit significant imprecision: in each benchmark, over 75% of the methods in the static call graph are absent from the dynamic call graph. This imprecision occurs overwhelmingly in the very large JRuby standard library. Due to its inability to model calls precisely, our analysis finds almost all of the standard library to be reachable, although only a relatively small fraction is actually used at run time by our subject programs.

As expected, the results for JRuby are very similar whether the use of invokedynamic is enabled or disabled.

```
(ns hello.core
  (:gen-class))

(defn bar [& args]
  (println "Hello, World!!"))

(defn foo [& args]
  (bar args))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (foo args))
```

Figure 3: A simple Clojure program.

5. CLOJURE

Clojure[9] [15] is a dialect of Lisp; key language features include higher-order functions, a powerful macro system, and concurrency control based on Software Transactional Memory. Strong support for Java interoperability is provided, by way of lightweight mechanisms for creating Java objects and calling Java methods. In the experiments discussed below, we have used Clojure version 1.5.1.

5.1 Translation to JVM bytecode

Figure 3 shows a simple Clojure program similar to the examples for Python and Ruby: -main calls foo, foo calls bar, and bar prints "Hello, world".

The Clojure compiler translates each Clojure function into a class (for convenience, we will refer to such classes as "function classes" in the discussion below). For the functions foo and bar in our example, function classes hello.core$foo and hello.core$bar are generated, respectively. Each such class defines a method doInvoke() that contains code corresponding to the original function in the Clojure source code, and a method getRequiredArity() that returns its number of required arguments. A typical function call such as the one from foo to bar is translated as follows:

1. Method hello.core$foo.doInvoke() calls IFn.invoke(); this call dynamically dispatches to a method RestFn.invoke() (the interface IFn and the class RestFn are both part of the Clojure run-time library).
2. Then, RestFn.invoke() performs some bookkeeping, including a call to getRequiredArity() on the object representing the target function.
3. Lastly, RestFn.invoke() calls doInvoke() on the object representing the target function, which represents the actual method body of the callee bar.

In the static initializer of class hello.core, which contains the main() method for the compiled program, code is dynamically loaded by calling RT.var("clojure.core","load").invoke("hello.core"). The "hello.core" argument is ultimately used as a classname by the Clojure runtime in a call to the Java Reflection API. Then, in hello.core.main(), a call ((IFn)main__var.get()).applyTo() is executed to launch

[9] http://clojure.org

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	1,687	3,002	2,543	1,228	10,666	5,675	4,943	9,934
FK	1,687	3,028	2,564	1,223	10,666	5,762	5,021	9,925
FA	1,687	2,994	2,532	1,225	10,666	5,640	4,915	9,941
FR	1,687	3,002	2,530	1,215	10,666	5,671	4,922	9,917
KN	1,687	3,269	2,785	1,203	10,666	6,326	5,555	9,895
MB	1,687	3,001	2,557	1,243	10,666	5,645	4,944	9,965
NB	1,687	3,071	2,567	1,183	10,666	5,815	5,007	9,858
PD	1,687	3,861	3,370	1,196	10,666	7,914	7,125	9,877
RD	1,687	3,002	2,543	1,228	10,666	5,677	4,947	9,936
RC	1,687	2,934	2,494	1,247	10,666	5,492	4,797	9,971
SN	1,687	2,947	2,500	1,240	10,666	5,511	4,807	9,962

Table 4: Count of nodes and edges in the static and dynamic call graphs of the Clojure CLBG programs.

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	6,316	953	145	5,508	30,697	1,665	294	29,326
FK	6,316	929	112	5,499	30,697	1,598	210	29,309
FA	6,475	983	127	5,619	31,441	1,750	293	29,984
FR	6,316	1,000	168	5,484	30,698	1,777	374	29,295
KN	6,316	1,064	201	5,453	30,698	1,879	416	29,235
MB	6,316	932	125	5,509	30,698	1,575	241	29,364
NB	6,316	1,058	185	5,443	30,698	1,889	398	29,207
PD	6,316	972	155	5,499	30,697	1,724	338	29,311
RD	6,316	928	120	5,508	30,697	1,602	233	29,328
RC	6,316	884	108	5,540	30,697	1,529	227	29,395
SN	6,316	968	139	5,487	30,697	1,702	301	29,296

Table 5: Count of nodes and edges in the static and dynamic call graphs of the Groovy CLBG programs.

the actual program, which ultimately calls hello.core$_main.doInvoke() using the calling mechanism illustrated above.

5.2 Qualitative Analysis

Clojure presents similar challenges for static analysis as we have seen for Python and Ruby. The use of reflection in compiled code will cause most static analyses to miss some code entirely. Specifically, in our example, since class hello.core__init (where function-classes such as hello.core$foo and hello.core$bar are instantiated) is loaded by reflection, any static analysis that resolves method calls by keeping track of sets of instantiated classes would omit methods such as hello.core$foo.doInvoke() from the static call graph. Also, in main(), the call to applyTo() should resolve to RestFn.applyTo(ISeq), which is inherited by hello.core$foo and will ultimately call hello.core$foo.doInvoke(). However, since all function-classes are deemed not instantiated, no implementor of RestFn is deemed instantiated. As a result, ((IFn)man_var.get()).applyTo() is resolved to call just a few trivial classes rather than the actual bodies of the user-defined functions. Similarly, we found that the translation of module imports by the Clojure compiler also involves the generation of reflective code. In conclusion, the use of reflection in code generated by the Clojure compiler is pervasive and causes great unsoundness for static analysis.

Even if reflection could be avoided somehow, a static analysis would still face significant challenges to achieve reasonable precision. The translation of function calls relies on indirections through functions such as invoke() and doInvoke(), and a static analysis would need several levels of call-string context-sensitivity to track function calls precisely.

5.3 Quantitative Analysis

Table 4 shows the number of nodes and edges in static and dynamic call graphs that we constructed for Clojure versions of the programs from the CLBG suite. The static call graphs for all programs have the same number of nodes and edges. Further investigation revealed that each static call graph consists of only a main method and parts of the Clojure runtime libraries. The application logic is completely missing in each case because WALA is unable to reason about the reflective code in the libraries, as we discussed above.

6. GROOVY

Groovy is a dynamically-typed object-oriented scripting language, with close integration with Java: most valid Java programs are also valid Groovy programs, although Groovy programs can take advantage of additional features. Groovy programs compile to Java bytecode to run on the JVM.

6.1 Translation to JVM bytecode

For calls between Groovy methods, every class contains several static methods that construct an array of CallSite objects, implemented in the standard library. This array is indexed by numbers that are assigned to each call site in the class. Each CallSite object is initialized with the name of the method to be called. At a call site, the generated Groovy code retrieves the corresponding CallSite object from the array and calls a method named call on it, passing any parameters. The call method calls many other methods in multiple classes within the Groovy standard library, and ultimately looks up an object of class GroovyObject and calls invokeMethod on it. The invokeMethod method looks up the name of the method to be called using a dynamic representation of the class hierarchy. Finally, invokeMethod uses Java reflection to invoke the desired method from the desired class.

The Groovy compiler can optionally generate bytecode containing invokedynamic instructions. In that case, the first time a call site is invoked, the bootstrap method returns a MutableCallSite. The MutableCallSite initially points to the same general lookup code that is used when compiling without invokedynamic. The first time the call site is executed and the desired target method is looked up, the MutableCallSite is updated with the MethodHandle of the target method, so that subsequent calls can call the target method directly.

6.2 Qualitative Analysis

The many levels of call indirection, object creation, dynamic data structure lookup, and reflection are too complicated for WALA to model. WALA cannot track the flow of the method name string constants all the way to the use of reflection to invoke the particular methods. Therefore, WALA does not generate any call edges for any call site in a Groovy program.

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	615	727	251	139	1,444	1,216	412	640
FK	615	701	227	141	1,444	1,149	348	643
FA	620	750	272	142	1,452	1,272	473	653
FR	615	780	306	141	1,445	1,329	528	644
KN	615	829	336	122	1,445	1,402	572	615
MB	615	750	276	141	1,445	1,238	444	651
NB	615	817	328	126	1,445	1,406	578	617
PD	615	751	276	140	1,444	1,283	482	643
RD	615	719	242	138	1,444	1,182	394	656
RC	615	672	202	145	1,444	1,105	322	661
SN	615	733	256	138	1,444	1,246	444	642

Table 6: Count of nodes and edges in the static and dynamic call graphs of the Groovy (with invokedynamic support) CLBG programs.

```
31 object hello {
32   def main(args: Array[String]) = {
33     (new T with A).bar
34     O.bar
35     (new C).bar
36   }
37 }
38 trait A
39 trait T {
40   def foo = bar
41   def bar = println("T.bar")
42 }
43 class C {
44   def foo = bar
45   def bar = println("C.bar")
46 }
47 object O {
48   def foo = bar
49   def bar = println("O.bar")
50 }
```

Figure 4: A simple Scala program.

6.3 Quantitative Analysis

This sizes of dynamic and static call graphs for the Groovy benchmarks are shown in Tables 5 (for code generated without invokedynamic) and 6 (for code with invokedynamic).

For each Groovy program, only the main method appears in the static call graph, because our static analysis is unable to generate any call edges for any call sites in the Groovy code. This explains why the static call graphs have the same size. The static call graphs do contain many methods of the Groovy standard library that are transitively called from boilerplate code in the generated main class.

Quantitatively, the unsoundness is minor. This is because the Groovy library is much larger than the application programs (for which the call graphs are completely unsound), and the library is written in a normal Java style that WALA can analyze soundly (reflection is used sparingly).

Without invokedynamic, the static graphs are much larger than the dynamic graphs because the static analysis assumes that most of the Groovy library could be called, but the benchmark programs use only a small fraction. On the other hand, for the code with invokedynamic, the static call graphs are smaller than the dynamic ones. The static analyzer considers most of the library unreachable for two reasons. First, it misses calls due to invokedynamic instructions. Second, generated classes that implement calls using invokedynamic require much less boilerplate initialization code. The spurious control flow from initialization code into the library that is inferred for the non-invokedynamic classes is not inferred for the generated classes that use invokedynamic. The library code missing from the static call graph is also reflected in a small increase in unsoundness in the invokedynamic call graphs compared to the non-invokedynamic ones.

7. SCALA

Scala [25] is a statically-typed, object-oriented, functional programming language. Scala supports functional programming idioms such as pattern matching, lazy evaluation, and closures. In addition to classes and objects, Scala supports *traits*, which encapsulate a group of method and field definitions so that they can be mixed into classes. Scala is fully interoperable with Java: Scala code can import and use Java classes, and vice versa. The Scala compiler compiles to Java bytecode. Our experiments used Scala version 2.10.2.

7.1 Translation to JVM bytecode

In this section, we illustrate how scalac translates different kinds of method calls. Figure 4 shows a Scala program that defines traits A and T, class C, and objects hello and O. The main method calls method bar in objects with three different types: the trait composition (A with T) on line 33, the object O on line 34, and the class C on line 35.

A Scala class is translated into one JVM class file that contains the bytecode translation of all its methods. A Scala object is translated into two JVM class files. For the object O in Figure 4, two classes named O$ and O are generated. O$ contains the methods, foo and bar from the original Scala object O. It also defines a static field MODULE$ to enforce the singleton pattern. The other class, O, defines static methods that act as hooks to the methods defined in O$.

A Scala trait, such as T in our example, is translated into two JVM class files: T and T$class. Interface T contains declarations for the methods, foo and bar, of the Scala trait T. T$class is an abstract class that defines static methods that contain the bytecode translation of the concrete methods, foo and bar, of the Scala trait T.

Finally, a Scala trait composition, such as (A with T) in the example, is translated into an anonymous class, hello$$anon$1, that implements all its traits.

Consider the calls to bar on lines 33–35 in Figure 4. The first call on line 33 corresponds to two method calls in the generated bytecode. The second call on line 34 corresponds to exactly one method call in the generated bytecode. Similarly, the third call on line 35 corresponds to one method call in the generated bytecode. Interestingly, the call from foo to bar on line 40 is translated into an invokeinterface bytecode instruction in the generated class T$class. This is because the target of that call depends on the type of the trait composition that is used as the receiver of the call.

7.2 Qualitative Analysis

The JVM bytecode generated by the Scala compiler for

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	788	521	0	267	1,256	648	1	609
FK	1,315	805	0	510	2,436	1,108	1	1,329
FA	1,011	603	0	408	1,852	808	1	1,045
FR	1,342	776	2	568	2,602	1,060	3	1,545
KN	2,604	1,558	0	1,046	5,779	2,424	3	3,358
MB	993	583	0	410	1,804	772	1	1,033
NB	966	645	0	321	1,756	876	1	881
PD	1,243	696	0	547	2,634	1,009	1	1,626
RD	983	596	0	387	1,627	772	1	856
RC	849	538	0	311	1,411	673	1	739
SN	1,064	622	0	442	1,904	810	1	1,095

Table 7: Count of nodes and edges in the static and dynamic call graphs of the Scala CLBG programs.

the example in Figure 4 reflects the source code more directly than its counterparts for the dynamically-typed languages discussed in the previous three sections. This allows a static analysis to generate call graphs for Scala-generated bytecodes that are more sound. For the example of Figure 4, no significant challenges for static analysis are evident.

Nevertheless, as reported by Ali et al. [2], analyzing the JVM bytecodes generated by the Scala compiler can result in call graphs that are much less precise than those that can be constructed from the original Scala source code. This loss of precision occurs because significant type information is lost in the process of translating certain Scala features (e.g., closures) to JVM bytecode.

The Scala compiler translates closures into anonymous function classes ($anonfun), where each class extends one of the scala.FunctionN<T> where N is the arity and T is the return type of the closure. Each $anonfun class overrides the apply method inherited from its superclass and instantiates the type parameter T with a different type. However, at the bytecode level, all type parameters are erased. Therefore, a bytecode-based static analysis algorithm will create edges to all the apply methods of subclasses of scala.FunctionN<T> *from each of the call sites* to scala.FunctionN.apply(), thus rendering the produced static call graphs extremely imprecise. Ali et al. [2] present a family of algorithms for constructing call graphs of Scala programs from source code that avoids this loss of precision.

For certain Scala features (e.g., mutable fields in anonymous classes), the Scala compiler generates JVM bytecodes containing reflective method calls, which challenges sound static analysis. A sound static analysis would have to make conservative approximations that cause the static call graph to become extremely large and imprecise.

7.3 Quantitative Analysis

For each of the benchmark programs in Table 7, except FASTAREDUX, the methods and edges in the dynamic call graph are subsets of those in the corresponding static call graphs, so they are sound for this execution. However, in FASTAREDUX, there are 2 methods and 2 call edges that are missing in the static call graph compared to the dynamic call graphs. Further investigation revealed that this unsoundness arises from the use of reflection in the bytecodes generated by the Scala compiler for converting collections into arrays, similar to what the Java method java.util.ArrayList.toArray(T[]) does. When we examined the precision of the static call graph, we found that on average, about 16% of the edges that are in the static call graphs but not in the dynamic call graphs involve calls to/from apply() methods.

```
let bar x y =
   print_string "before calling print_hello in bar\n
       ";
   print_string x;
   print_string y;
   print_string "after calling print_hello in bar\n"
       ;;

let foo x y =
   print_string "before calling bar in foo\n";
  x y;
   print_string "after calling bar in foo\n";;

foo (bar "Hello, World\n") "Hello Again\n";;
```

Figure 5: A simple OCaml program.

8. OCAML

OCaml is a general purpose programming language supporting functional, imperative and object-oriented styles[10]. Types are strong and static, and inferred by the compiler—that frees programmers from stating them.

OCaml-Java[11] is a compiler that directly compiles OCaml source code to Java bytecode and provides mechanisms for interoperability with Java. We used OCaml-Java version 2.0-alpha2, based on OCaml version 4.01.0, for the experiments in this paper. This version of OCaml-Java requires at least the Java 7 virtual machine to run compiled programs. The OCaml-Java compiler directly generates the .jar file for an OCaml program. The OCaml-Java standard library is included in the .jar file. The .jar file contains three folders—ocaml, org, and pack. The ocaml and org folders contain the standard library, which is the same for every OCaml program. The pack folder contains class files generated from the OCaml input program and a class called ocamljavaMain. This is the main class identified in the manifest of the .jar file, and serves as the driver of the whole OCaml program.

8.1 Translation to JVM bytecode

We will use the example program of Figure 5 to illustrate how OCaml-Java translates OCaml source code to JVM class files. Figure 5 shows an OCaml program that declares functions foo and bar. This program illustrates currying, in the partial call to bar with one argument "Hello, World". This closure is passed to bar along with the argument "Hello Again". Function foo prints messages and calls its argument x (which is bound to bar) with y (bound to "Hello Again") as a parameter. Function bar prints its arguments.

The translation of this example program is mostly straightforward, compared to some other systems in this study. The OCaml compiler translates syntactic functions to corresponding bytecode methods, and direct function calls, such as the call to foo, as invokestatic bytecodes. Currying gets

[10]http://ocaml.org/learn/description.html
[11]http://ocamljava.x9c.fr/preview/

	nodes				edges			
	static	dyn.	D\S	S\D	static	dyn.	D\S	S\D
BT	5,963	193	0	5,770	74,952	1,494	60	73,518
FK	5,973	193	0	5,780	74,779	1,516	65	73,328
FA	5,966	187	0	5,779	75,259	1,193	15	74,081
FR	5,973	194	0	5,779	75,310	1,472	62	73,900
KN	5,981	193	0	5,788	77,845	1,824	76	76,097
MB	5,958	192	0	5,766	74,728	1,446	58	73,340
NB	6,977	195	0	6,782	88,488	1,498	58	87,048
PD	5,986	186	0	5,800	76,126	1,444	41	74,723
RD	5,962	203	0	5,759	74,952	2,106	88	72,934
RC	5,960	183	0	5,777	75,029	1,194	15	73,850
SN	5,963	197	0	5,766	74,924	1,484	58	73,498

Table 8: Count of nodes and edges in the static and dynamic call graphs of the OCaml CLBG programs.

translated as constructing a closure object using org.ocamljava.runtime.values.Value.createClosure, and first-class functions generally are represented by java.lang.invoke. MethodHandle objects. The actual code to implement the currying gets generated as an extra function object. These bytecode constants represent specific methods directly, and obviate complex uses of string-based reflection. The call to the function x in foo gets translated as an invokevirtual on the closure object. The method on the closure calls the closed function, in this case bar, using the MethodHandle method invokeExact, avoiding reflection using strings.

In a bit more detail, the partial call to bar is translated by calling createClosure, then calling setClosure on it with a MethodHandle representing bar, and then recording the closure parameter "Hello, World" with set2 on the closure. The key is that the first-class functions are named explicitly with MethodHandle objects, which greatly eases their analysis.

8.2 Qualitative Analysis

Functional languages like OCaml have first-class functions, currying and closures; as such, one might expect OCaml to have the same problem as the scripting languages, i.e. pervasive reflection leading to unsoundness and imprecision. However, the OCamljava implementation exploits the MethodHandle mechanism to great effect, making heavy use of constant method handles embedded in the bytecode. Thus, first-class functions manifest as explicit method constants; WALA models these constants and invocations on them. Hence, functions such as bar appear in the call graph.

This does not, in itself, make analysis precise, since functions passed as arguments may cause imprecision in context-insensitive analysis, just as dynamic dispatch on parameters can in object-oriented languages. However, this is the same well-studied problem of context sensitivity which has inspired so many techniques for object-oriented languages.

8.3 Quantitative Analysis

Table 8 presents the call graph construction results on the OCaml versions of the benchmarks. The impact of using MethodHandles in a way that is amenable to static analysis is apparent from the absence of unsoundness in the methods in all of the static call graphs. There is some unsoundness in the edges—always less than 5%—due mostly to idioms in the OCaml runtime involving the use of java.lang.reflect.Proxy for method calls. WALA does not understand this reflective idiom, so edges are missing from the static graph. These missing edges sometimes cause further missing edges as needed code is deemed unreachable by the static analysis. Also, proxies result in runtime-generated code appearing on the stack, so the dynamic call graphs contain edges that do not correspond to any source code and hence will be missing from the static graph.

Precision is low across all programs, with the vast bulk of both nodes and edges in the static call graph not being in the dynamic one. There are three major causes of this:

1. Imprecision in what runtime primitives are used to access values; values are sometimes stored in a boxed form (org.ocamljava.runtime.Value) and indirections used to access and convert them make our context-insensitive analysis imprecise.
2. MethodHandle objects are passed to runtime primitives to handle calls, and context-insensitive analysis of these primitives causes significant imprecision.
3. These issues cause more of the standard library to be reachable, which adds further imprecision as edges and nodes from those functions get added.

We observe that the soundness of our analysis of OCaml programs raises hope that pervasive and careful use of features such as method handles will ease analysis of a broad class of languages. Also, the imprecision in our OCaml analysis could be addressed to a great extent by techniques already used for similar concerns in prior analyses, e.g., CPA [1].

9. THREATS TO VALIDITY

A critical reader might argue that the programs studied in this paper are small, that they do not cover the full range of each language's features, and that they are perhaps not representative of real-world programs.

We do not consider the above considerations to be serious reasons for concern because the primary conclusions of our study (i.e., whether soundness or imprecision occurs for each language under consideration) are evident from the manual analysis of small example programs, and supported by our quantitative experiments with the CLBG programs. Analyzing larger programs that make use of additional language features would yield the same conclusions because such programs would make use of language features such as function calls that already give rise to unsoundness or imprecision. Furthermore, the use of an existing benchmark suite such as CLBG, with variants of the same programs for each language, enables us to study the different programming languages in a way that is uniform and consistent.

In general, computing precise static call graphs is undecidable, and in this paper, we have used dynamic call graphs to estimate the precision of static call graphs. However, a dynamic call graph provides an under-approximation of a precise static call graph, and the reader may wonder if code coverage is reasonable. To address this concern, we measured basic block coverage. On average, across all the languages under consideration, the program input used to run the CLBG benchmark suite provides a basic block coverage of 87%, which is quite high. We are unable to present the coverage results in detail due to space limitations.

10. RELATED WORK

Dynamic Studies.

Most of the studies of non-Java languages that compile to JVM bytecode have evaluated the dynamic behavior of the bytecode, particularly from the point of view of a virtual machine that executes it. In contrast, our study evaluated the generated bytecode from the point of view of a static analyzer. Li et al. [20] have studied the JVM bytecode generated by JRuby, Jython, Scala, and Clojure, for the Computer Language Benchmarks Game (CLBG) programs. They focused on dynamic behavior: they measured the diversity of bytecode instruction sequences executed, the sizes of methods, the depths of the runtime stack, the hotness distribution of methods and basic blocks, the sizes and lifetimes of objects, and the amount of boxing of primitive types. Sarimbekov et al. [30] also studied JRuby, Jython, and Clojure (but not Scala), on the CLBG programs. They measured runtime behavior: polymorphic calls, immutability of fields, objects, and classes, lifetimes of objects, amount of memory zeroing, and the number of evaluations of identity hash codes. Sewe et al. [34] introduced a benchmark suite of Scala programs similar to the DaCapo suite [5] of Java programs. They compared the dynamic behavior of these programs to that of the DaCapo Java programs.

Multilingual Virtual Machines.

The Microsoft Common Language Runtime (CLR) was designed from the outset to support multiple source languages, including C#, C++, and Visual Basic, and has since been used as the target of many others. Gordon et al. [13] presented a type system for the CLR Intermediate Language (CIL). Bebenita et al. [4] used CIL as the bytecode language for a tracing JIT specifically designed for dynamic scripting languages like JavaScript.

Beyond the approach of compiling multiple source languages to a common bytecode intermediate language, recent work has adapted the virtual machine more deeply to support new languages. Castanos et al. [7] modified an existing JIT compiler to exploit dynamic characteristics of Python, yielding performance improvements. Würthinger et al. [38] built a virtual machine that allows custom source front-ends for a variety of languages. These front-ends interpret, profile, and optionally transform the source programs. The system later partially evaluates these interpreters to generate machine code. The resulting system performs better than compiling to Java bytecode first because the front-end interpreters that are partially evaluated can perform profile-directed optimizations that are specific to a given source language. Savrun-Yeniceri et al. [31] present techniques for using forms of threaded code generation to improve JVM-hosted interpreters. Their goal is efficiency; the problem they address is that simple interpreters make heavy use of indirect jumps, which harms branch prediction, and threaded code minimizes that. In our case, static analysis of such simpler control flow would likely be easier, for the same reasons that branch prediction benefits.

JSR 292 introduced invokedynamic bytecode, which enables Java bytecode to call dynamically specified methods. This can greatly simplify the method invocation sequences that we have identified in bytecode generated from dynamic languages. However, because the target methods are specified dynamically, invokedynamic also makes it more difficult for static analysis tools to construct call graphs.

Other Languages.

The translation of various programming languages to bytecode-based platforms has received considerable attention in the literature. Clerc et al. [8] explain and motivate how OCaml is translated to Java bytecode. Several papers and theses discuss how Scala is compiled to Java bytecode [32, 9, 11, 10]. In addition to the language implementations that we reported on in Sections 3–8, many other languages have been compiled to bytecode, e.g., Scheme [33, 6], Star [22], Pizza [26, 24]. Yermolovich et al. [41] even translate machine language code to Java bytecode.

The use of dynamic features has been studied for languages that are not normally compiled to Java bytecode. Richards et al. [29, 28] studied the use of dynamic features in JavaScript, especially the eval construct. Hills et al. [16] studied the use of various features in PHP programs, including dynamic features such as eval.

Xu and Rountev evaluated a regression test selection analysis for AspectJ [39]. They found the analysis to be extremely imprecise when based on call graphs constructed from the Java bytecode generated by compiling AspectJ programs. To improve precision, they introduced the *interaction graph*, a structure similar to a call graph, but which explicitly models AspectJ features, and evaluated an analysis for constructing such graphs from AspectJ source code [40].

11. CONCLUSIONS

We have investigated whether a JVM bytecode-based static call graph construction works well for bytecode produced from other languages. We conducted experiments to explore the soundness and precision of such static call graphs produced for Python, Ruby, Clojure, Groovy, Scala, and OCaml. In a set of qualitative experiments, we manually examined the translation of small example programs to observe potential challenges to call graph construction. In a set of quantitative experiments, we compare static and dynamic call graphs for 11 programs from the CLBG suite to assess unsoundness and imprecision in practice.

Our results show that, for the dynamically-typed languages, Python, Ruby, Clojure, and Groovy, call graph construction is dramatically unsound; heavy use of reflection, run-time code generation, and invokedynamic instructions pose significant challenges to static analysis. Furthermore, these call graphs are also highly imprecise due to how function calls are translated into JVM bytecode. For the statically-typed languages, Scala and OCaml, the situation is more promising. For these languages, all unsoundness comes from rare, complex uses of reflection and proxies, which does not seem to arise frequently in practice. In the case of Scala, the translation of programming idioms related to first-class features causes significant loss of precision, and for OCaml loss of precision arises due to control flow similar to what has been observed in Java.

While the experiments reported on in this paper are concerned with call graph construction, we consider our conclusions to be broadly applicable to bytecode-based interprocedural static analyses, because call graphs are a prerequisite for most static analyses. We conclude that language implementors should consider carefully if they want to rely on dynamic mechanisms such as reflection and invokedynamic.

While these mechanisms may ease the task of compilation, they render static analysis effectively useless, thus impeding the development of IDE-based tools (e.g., for refactoring) that are especially useful for dynamically typed languages.

12. REFERENCES

[1] O. Agesen. The cartesian product algorithm. In M. Tokoro and R. Pareschi, editors, *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Aarhus, Denmark, August 7-11, 1995*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer Berlin Heidelberg, 1995.

[2] K. Ali, M. Rapoport, O. Lhoták, J. Dolby, and F. Tip. Constructing call graphs of scala programs. In R. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 54–79. Springer, 2014.

[3] K. Barclay and J. Savage. *Groovy programming: an introduction for Java developers*. Morgan Kaufmann, 2010.

[4] M. Bebenita, F. Brandner, M. Fähndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: a trace-based JIT compiler for CIL. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 708–725. ACM, 2010.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In P. L. Tarr and W. R. Cook, editors, *OOPSLA*, pages 169–190. ACM, 2006.

[6] Y. Bres, B. P. Serpette, and M. Serrano. Bigloo.NET: compiling Scheme to .NET CLR. *Journal of Object Technology*, 3(9):71–94, 2004.

[7] J. G. Castaños, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 195–212. ACM, 2012.

[8] X. Clerc. OCaml-Java: an ML implementation for the Java ecosystem. In Plümicke and Binder [27], pages 45–56.

[9] I. Dragos. *Compiling Scala for Performance*. PhD thesis, IC, Lausanne, 2010.

[10] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47. ACM, 2009.

[11] G. Dubochet and M. Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In I. Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41, 2009.

[12] B. Fulgham. The Computer Language Benchmarks Game. `http://benchmarksgame.alioth.debian.org`, March 2014.

[13] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 248–260, New York, NY, USA, 2001. ACM.

[14] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[15] S. Halloway and A. Bedra. *Programming Clojure*. Pragmatic Bookshelf, 2 edition, 2012.

[16] M. Hills, P. Klint, and J. J. Vinju. An empirical study of PHP feature usage: a static analysis perspective. In M. Pezzè and M. Harman, editors, *ISSTA*, pages 325–335. ACM, 2013.

[17] IBM. T.J. Watson Libraries for Analysis WALA. `http://wala.sourceforge.net/`, April 2013.

[18] J. Juneau. Polyglot Programmer: Jython 101 – A Refreshing Look at a Mature Alternative. *Oracle Java Magazine*, 2013. Available from `http://www.oraclejavamagazine-digital.com`.

[19] O. Lhoták. Comparing call graphs. In M. Das and D. Grossman, editors, *PASTE*, pages 37–42. ACM, 2007.

[20] W. H. Li, D. R. White, and J. Singer. JVM-hosted languages: they talk the talk, but do they walk the walk? In Plümicke and Binder [27], pages 101–112.

[21] M. Lutz. *Learning Python*. O'Reilly, 5 edition, 2013.

[22] F. McCabe and M. Sperber. Feel different on the Java platform: the star programming language. In Plümicke and Binder [27], pages 89–100.

[23] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real-World OCaml: Functional Programming for the Masses*. O'Reilly, 2013.

[24] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your Pizza - translating parameterised types into Java. In M. Jazayeri, R. Loos, and D. R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 114–132. Springer, 1998.

[25] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, 2nd edition, 2012.

[26] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 146–159. ACM Press, 1997.

[27] M. Plümicke and W. Binder, editors. *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. ACM, 2013.

[28] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In M. Mezini, editor, *ECOOP*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[29] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In B. G. Zorn and A. Aiken, editors, *PLDI*,

pages 1–12. ACM, 2010.

[30] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder. Characteristics of dynamic JVM languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 11–20, New York, NY, USA, 2013. ACM.

[31] G. Savrun-Yeniçeri, W. Zhang, H. Zhang, E. Seckler, C. Li, S. Brunthaler, P. Larsen, and M. Franz. Efficient hosted interpreters on the JVM. *TACO*, 11(1):9, 2014.

[32] M. Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, EPFL, 2005.

[33] B. P. Serpette and M. Serrano. Compiling Scheme to JVM bytecode: : a performance study. In M. Wand and S. L. P. Jones, editors, *ICFP*, pages 259–270. ACM, 2002.

[34] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con Scala: design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *OOPSLA*, pages 657–676, 2011.

[35] O. Shivers. Control-flow analysis in scheme. In R. L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 164–174. ACM, 1988.

[36] D. Thomas, A. Hunt, and C. Fowler. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, 4 edition, 2013.

[37] 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 *CC*, pages 18–34, 2000.

[38] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In A. L. Hosking, P. T. Eugster, and R. Hirschfeld, editors, *Onward!*, pages 187–204. ACM, 2013.

[39] G. H. Xu and A. Rountev. Regression test selection for AspectJ software. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 65–74. IEEE Computer Society, 2007.

[40] G. H. Xu and A. Rountev. AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software. In T. D'Hondt, editor, *Proceedings of the 7th International Conference on Aspect-Oriented Software Development, AOSD 2008, Brussels, Belgium, March 31 - April 4, 2008*, pages 36–47. ACM, 2008.

[41] A. Yermolovich, A. Gal, and M. Franz. Portable execution of legacy binaries on the Java Virtual Machine. In L. Veiga, V. Amaral, R. N. Horspool, and G. Cabri, editors, *PPPJ*, volume 347 of *ACM International Conference Proceeding Series*, pages 63–72. ACM, 2008.