Java Code Analysis and Transformation into AWS Lambda Functions

Josef Spillner and Serhii Dorodko
Zurich University of Applied Sciences, School of Engineering
Service Prototyping Lab (blog.zhaw.ch/icclab/)
8401 Winterthur, Switzerland
{josef.spillner,dord}@zhaw.ch

February 17, 2017

Abstract

Software developers are faced with the issue of either adapting their programming model to the execution model (e.g. cloud platforms) or finding appropriate tools to adapt the model and code automatically. A recent execution model which would benefit from automated enablement is Function-as-a-Service. Automating this process requires a pipeline which includes steps for code analysis, transformation and deployment. In this paper, we outline the design and runtime characteristics of Podilizer, a tool which implements the pipeline specifically for Java source code as input and AWS Lambda as output. We contribute technical and economic metrics about this concrete 'FaaSification' process by observing the behaviour of Podilizer with two representative Java software projects.

1 Motivation

Cloud computing brings along new programming models. Ranging from monolithic virtual machines or containers over composite microservices to finer-grained functions and stream operators, an application developer is faced with a complex choice of the right model and technology for cloud-enabled, cloud-aware and cloud-native applications [1]. Once the model is chosen, the switch to another model becomes non-trivial, in particular for the direction from coarse- to fine-grained structures. As the Function-as-a-Service (FaaS) class is becoming more popular [2], its exploitation would benefit from an automated transformation of legacy code and of generic new code to the code conventions expected by this model. Program transformation and especially source code transformation are established software engineering techniques for automated profiling, security improvements, optimisation and refactoring [3, 4]. In the quickly evolving cloud environments, it becomes essential to add exposure to new target platforms

to this list considering that many applications are merely programmed against specific interfaces rather than generated through a top-down model-driven architecture [5].

While most FaaS providers support multiple programming languages, the scope of this study mandates to pick one for a detailed analysis. The Java programming language is widely taught and used to implement services. Typical programming models supported by the default development kit encompass Servlets, EJBs and diverse component models, and JAX-WS/JAX-RS for web services. Third-party Java frameworks offer even more choices, including Restlet and Apache CXF. Therefore, despite being only supported by few FaaS providers, we pick Java as input format for our research. The implications range beyond just programming. Most Java software projects are expected to be built either manually by opening the project in a suitable development environment (e.g. Eclipse) or in an automated way by executing a build script (e.g. Ant, Maven, Gradle). The build instructions can be exploited to automate the software transformation. On the provider side, we limit our study to Amazon Web Services (AWS) Lambda as it is one of the few services for hosting Java functions as a service.

Hence, we aim at contributing novel insight into the automated transformation of legacy applications into FaaS-hosted cloud applications which we name FaaSification. We claim that our tool, Podilizer[1], is the first one which performs such a transformation from monolithic Java code to AWS Lambda units, and it does so with sufficient quality to be considered in cloud application prototyping projects. Due to the restriction to Lambda, we refer to this process more specifically as Lambdafication.

Related work is available on the comparison of pricing effects for monolithic and microservice applications using AWS Lambda [6, 7]. Compared to these general analyses and formalisations, this work motivates finding an automated transformation to let developers make the policy choice at a late point in time while hiding the actual mechanism to enact the policy. Further related work covers FaaS providers and implementations, for instance OpenLambda [8], but does not contribute to their integration into the software and service engineering process.

The paper is structured as follows. First, we outline our research questions and approach and inform about how to map object-oriented programming to functional services. The mapping description is complemented by a an abstract pipeline architecture and a concrete architecture of our realisation thereof, the Podilizer tool. Afterwards, we explain the experimental evaluation and discuss the extracted findings. The paper concludes with an open discussion of how software should be written for the cloud.

[1] Podilizer is publicly available at `https://github.com/serviceprototypinglab/podilizer`

2 Approach

Our approach consists of three parts. First, we identify two research questions. Then, we explain general decisions which must be taken by any transformation tool related to the programming model, the handling of stateful objects and the design of a transformation process. The third part presents both the design and the implementation of our transformation tool Podilizer.

2.1 Research Questions

The planned code transformation process leads to two research questions (RQ_1 and RQ_2). Our approach is focused on generating empirical results and deriving the answers accordingly.

RQ_1: Is it economically viable to run a Java application entirely over FaaS? The comparison baseline would be conventional programmable platform (PaaS) models by deploying the application onto a suitable application server as well as programmable infrastructure (IaaS) by wrapping the Java application into a container or a virtual machine.

RQ_2: Is it technically feasible to automate this process? And if so, which percentage of code coverage can be expected, which performance can be achieved, and which code is easier, hard or impossible to convert?

2.2 Programming $\mapsto$ Execution Model Mapping

FaaS is inherently bound to the functional programming paradigm. Its characteristics under a pure interpretation are determined by stateless computations with strict use of invocation parameters and return values without global variables. In practice, just as functional programming languages have introduced techniques to cause side effects and manage state, for instance through monads, so do most FaaS interfaces, for instance through access to storage services.

The function orientation is in contrast to Java's model which is predominantly an object-oriented language. Even though few functional programming concepts are available through the Functional Java library and starting with Java 8 with native Lambda expressions [9], the dominant share of code is written in a pure object-oriented way. The same observation applies to similar programming language. Therefore, the code translation needs to take the paradigm shift into account. According to Plumbr, an application monitoring provider, the Java language versions in use in 2015 were Java 8 (20.84%), Java 7 (59.37%) and Java 6 (19.79%) [10]. Following industry relevance, our research focuses on Java 7/8.

The challenges are then related to the mapping of Java classes to appropriately packaged Java FaaS functions, called FaaS units or functional units. The mapping needs to account for empty methods, getters and setters, constructors and singletons. Beyond the code, typical Java project conventions such as the presence of a `src` folder, but also the absence thereof and exceptions from the conventions, need to be accounted for. Finally, the mapping needs to consider

the grouping of methods per functional unit to avoid excessive network calls and ensure that all dependency methods referenced from each method can be resolved.

2.3 State Handling

Java methods are often stateful through instance attributes. The state handling of the resulting decomposed functions can either extend the method signature to pass in and out all attributes which are accessed and modified, or use server-side state. AWS Lambda offers both an S3 blob storage interface and local environment variables. However, the variables are restricted to read-only access from the runtime [11]. Weighting the advantages of S3 (performance) against extended method signatures (price, functional purity), our approach uses the latter technique.

In Java, methods with parameters are integral parts of classes and are used to change the state of the corresponding instances: $Class.method(params)$. The instance is self-referenced implicitly with the keyword *this*. According to the Lambda programming model, every function unit assumes a specific stateless class with a method handler which is triggered when the function is invoked. The statelessness is due to not guaranteeing the same object of this class to be used for subsequent requests. Early works to compile Java code into a typed Lambda calculus have suggested making the self-references explicit by enhancing the method signatures with it [12] which is the approach chosen by us as well.

The translation process thus rewrites the method header with the Lambda-required signature and the method body with generated code. This code first initialises the invocation credentials, creates an input object to save the instance state as well as any method parameters, initialises the Lambda invoker with the created input object serialised to JSON, calls the method ($Class.handleRequest(input, output, context)$), fetches the result from the deserialised output object, and renews the instance state using the result object. The credentials are read from the environment and upon failure from a configuration file which permits the generated code to still run outside of the Lambda environment.

2.4 FaaSification Pipeline

FaaSification is the process of converting a code structure into a format which is executable on FaaS. In our approach, this process is represented by a superscalar pipeline which allows for parallel processing of each of its steps. The first step is the static code parsing and analysis (A). The second step is the decomposition into functional units and a remainder which includes the code identified as incompatible to the target FaaS platform (D). The third step is the source-to-source translation of the functional units into FaaS units, adhering to the calling conventions of the target platform (F). The fourth step is the compilation and dependency assembling of these units (C), and the fifth step is their upload, deployment and configuration to turn them into ready to use microservices (U).

An optional sixth step is the systematic test of all deployed functions and the verification of the successful transformation (U). Fig. 1 gives an example of a FaaSification pipeline whose parallel execution depends on resource consumption superpositioning and on dependencies between methods before the ability to run unit tests.

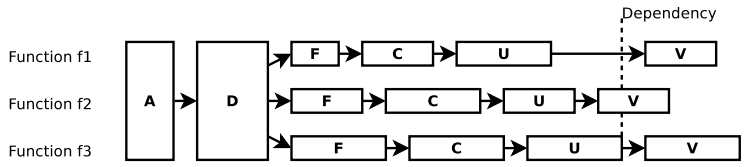

Figure 1: FaaSification pipeline

2.5 Podilizer Design

The pipeline thinking is reflected in the design for a local tool called Podilizer run by software engineers as part of their development environment. Podilizer implements the FaaSification pipeline in its Lambdafication flavour by recursively scanning directories for Java projects and processing each project and each Java source file until the code is available to be invoked as Lambda function. The pipeline steps are incremental and the tool allows for continuations starting from each preceding step. This design choice makes it fault-tolerant and debug-friendly in addition to an easy parallelisation. Table 1 informs about the steps ($A \rightarrow D \rightarrow F \rightarrow C \rightarrow U \rightarrow V$) and the associated continuation points and requirements. The source build files are not used because the project is immediately dissected into functions. The Maven build file per target function is generated by Podilizer but may need project-specific customisation to incorporate dependency libraries into the build process. The AWS credentials are assumed to be readily available on the local system, for instance by a prior installation of the AWS CLI. The remaining requirements are all extracted from the analysed projects.

Table 1: FaaSification pipeline steps and continuations.

Step	**Continuation Points**	**Requirements**
A: code analysis	– (internal AST)	source code directory
D: code decomposition	– (internal list of ASTs)	–
F: function translation	target source directory	–
C: compilation	target binary directory	Maven buildfile (cust.)
U: upload	deployed functions list	AWS credentials
V: verification	–	unit test definitions

2.6 Podilizer Implementation

Podilizer is itself implemented in Java, using the JavaParser framework for static code analysis and the Maven library to inject build instructions into the target software projects. It executes the AWS CLI tool as external process for all interaction with AWS Lambda and any supported unit test framework it finds for the verification step. Currently, JUnit is supported.

Fig. 2 gives an overview about the Podilizer implementation. Its main components are the translator which covers the first half of the pipeline ($A \rightarrow D \rightarrow F[\rightarrow C]$) in which the use of Maven on all generated build files (C) is optional, and the triggering of the upload (U) as well as the unit test execution (V).

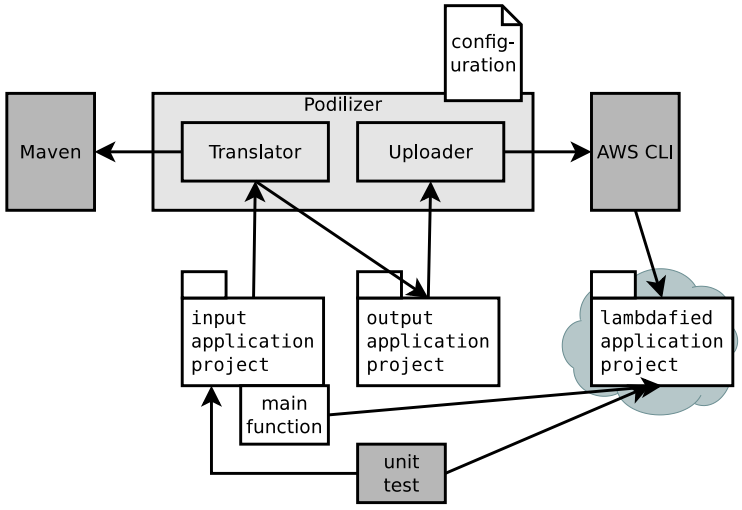

Figure 2: Implementation architecture of Podilizer

The implementation complexity of Podilizer is moderate with about 2100 lines of Python code. The invocation takes the most important parameters such as pipeline steps, source directory and target directory as command-line parameters. Further configuration details can be specified in a YAML file which is parsed on startup.

3 Trials and Findings

We evaluate Podilizer experimentally with a standard research testbed approach shown in Fig. 3. All input to the experiments is recorded in public versioned repositories, and all raw output is captured in another dedicated repository. The versioning allows for finding improvements and regressions over time as the software evolves. All experiments are tracked in a public Open Science Notebook and tools for reproducibility, repeatability and recomputation are

made available as well in a Podilizer Repeatability project[2]. The main results of the current implementation are reflected in this section.

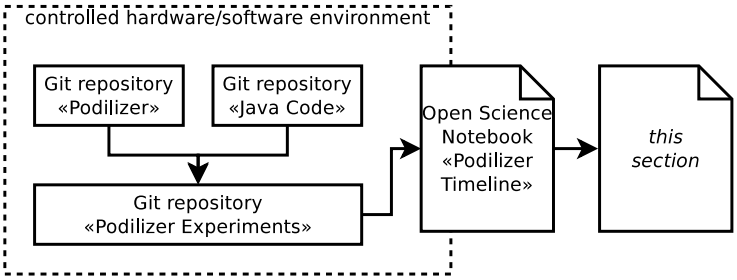

Figure 3: Testbed for performing experiments on Podilizer

3.1 Experiment Setup

Each step of the pipeline is associated to a unique check for success. The first three steps are performed internally by Podilizer within one procedure, whereas the three remaining ones are merely automated by running executables out of which one is provided by Podilizer, too. The most crucial check is the final one which is successful if all deployed functions are remotely invocable. According to DZone, about 30.7% of all Java projects hosted on Github depend on JUnit which calls for an integration to ensure systematic testing of the deployment [13]. Table 2 summarises all steps and checks.

Table 2: FaaSification pipeline steps and checks.

Step	**Check**
A: code analysis	JavaParser internal return value
D: code decomposition	Podilizer internal
F: function translation	Podilizer internal
C: compilation	compiler/build tool exit status
U: upload	Podilizer deployer exit status
V: verification	call test, unit test exit status

Podilizer is instrumented with millisecond-precision logging to reveal the duration of each pipeline step. In addition to the performance, the quality of the transformation can be measured by the ratio of successful checks against all which are performed in each step.

The economic aspect requires a comparison between the execution of the lambdafied application compared to a monolithic execution in a configuration which matches the performance. In the absence of a general performance estimation formula, a manual calibration specific to each software application under test is therefore needed.

[2]Podilizer Repeatability in the Open Science Framework: `https://osf.io/c886p/`

The reference input project set consists of six software applications which represent the large variety of Java software engineering, ranging from 28 to 771 significant lines of code (SLOC), diverse interaction forms (none, standard input and output, graphical, files, HTTP methods) and build tools (javac, make, maven, ant) and artefact type (applications, libraries, plugins, tests). The software projects are a graphical window with buttons (P1), mathematical functions (P2), calculation of shipping containers and boxes (P3), public transport information (P4), image processing (P5) and domain-specific language parsing and evaluation (P6).

3.2 Results

We have run the experiment on a Dell Latitude E7450 notebook with Intel Core i7-5600 quad-core processor clocked at 2.60 GHz. The notebook was connected to SWITCHlan, the Swiss university network, via 1000baseT Ethernet, and installed with Ubuntu Linux and OpenJDK 8. The results differ depending on the chosen software project to translate. The values are also influenced by the hardware, the used software tools (Podilizer, Maven, JUnit), the provider (AWS Lambda) and the network connection in between. A full specification and self-contained virtual machine is made available as part of the Podilizer Repeatability project.

Table 3 informs about the performance and quality of the FaaSification pipeline for project P1. The values for C and D represent the sum of individual measurements for five functional units with a relatively low deviation for local compilation but a higher one for network-dependent upload: $C_{min} = 3832ms$ and $C_{max} = 4038ms$, $D_{min} = 5743ms$ and $D_{max} = 10414ms$. The verification step is omitted due to the lack of unit tests in P1.

Table 3: FaaSification pipeline characteristics for P1.

Step	**Performance**	**Quality**
A: code analysis	0.055s	100%
D: code decomposition	0.002s	100%
F: function translation	0.122s	100%
C: compilation	10.173s	100%
U: upload	21.238s	100%
V: verification	–	–
TOTAL	31.590s	success

For comparison, Tables 4 and 5 contain the separate performance and quality values for P2–P6. Concerning the performance, the first two steps (A, D) almost always execute in less than a single Lambda billing period ($100ms$) whereas the functional decomposition takes up to one second depending on the code complexity. The compilation and upload take consistently much longer in comparison. A just-in-time transformation is precluded and optimisation techniques are needed to overcome this limitation. Nevertheless, the entire transformation

process including systematic unit testing performs in an acceptable timeframe and can be further optimised by stronger parallelisation depending on the build system and unit test framework.

Table 4: FaaSification pipeline characteristics (performance) for P2–P6.

Step	**P2:P**	**P3:P**	**P4:P**	**P5:P**	**P6:P**
A	0.054s	0.058s	0.074s	0.074s	0.105s
D	0.002s	0.005s	0.010s	0.011s	0.028s
F	0.096s	0.302s	0.867s	0.025s	0.701s
C	10.530s	17.777s	37.707s	–	22.901s
U	21.349s	31.141s	65.075s	–	44.858s
V	11.942s	–	13.927s	–	–
TOTAL	43.973s	49.283s	117.657s	–	68.593s

The achieved quality is binary as the only failed transformation process (P5) is due to a crash of the transformator itself. A more graceful partial transformation by tainting problematic methods would help to raise the total percentage above 0%.

Table 5: FaaSification pipeline characteristics (quality) for P2–P6.

Step	**P2:Q**	**P3:Q**	**P4:Q**	**P5:Q**	**P6:Q**
A	100%	100%	100%	100%	100%
D	100%	100%	100%	100%	100%
F	100%	100%	100%	0%	100%
C	100%	100%	100%	0%	100%
U	100%	100%	100%	0%	100%
V	100%	–	100%	–	–
TOTAL	success	success	success	fail	success

These results contain data points which lead to the answer of RQ_2. The automated translation is feasible, with high code coverage for simple but heterogeneous code projects. The failures in the experiment are due to dynamic classloading for plugins and the insufficient handling of such constructs by the transformer.

When the deployment process is finished, the interest shifts to the execution performance of each software application. Table 6 compares the monolithic execution locally and on an equivalent IaaS and PaaS setup (AWS EC2 and Elastic Beanstalk, respectively) against the one with decomposed functions for all six analysed software projects. The EC2 execution occurrs on a single-core Ubuntu node with 3.5 GB main memory and an SSD. It uses Xinetd to trigger the execution from an opened TCP connection to Xinetd's ports 10001 to 10006 for P1 to P6. The local and EC2 invocations use the startup sequence of the build system, for instance $mvn : exec$, which causes additional delays.

As expected, P1 fails on the server due to being a graphical application, and

Table 6: Application execution performance comparison.

Flavour	**P1:X**	**P2:X**	**P3:X**	**P4:X**	**P5:X**	**P6:X**
Notebook local	–	0.71s	1.87s	1.25s	0.08s	0.13s
AWS EC2 local	–	1.18s	2.99s	1.92s	0.09s	0.18s
AWS EC2 Xinetd	–	1.16s	2.86s	1.57s	0.12s	0.22s
AWS Beanstalk	–	0.36s	0.36s	1.79s	–	0.36s
AWS Lambda	–	8.77s	9.74s	12.20s	–	–

its execution time depends on the user in the other cases. Therefore it is excluded from the comparison. P5 requires an interactive command-line interface and cannot be instrumented in a web environment or through function calls alone. P6 fails unexpectedly due to missing symbol files for the parser. The solution necessitates a concept to deploy dependency files in addition to dependency code which is not part of our design.

The first observation beyond the exclusions is that the EC2 instance is consistently slower than the notebook which can be attributed to the rather coarse-grained configuration options offered by IaaS providers. A second observation is that the two layers of indirection through Xinetd and a wrapper shell scripts do often not cause a significantly higher execution time.

These results give an answer to RQ_1. While the applications perform slower by about an order of magnitude compared to typical IaaS or PaaS deployments, the economic feasibility is still in range for services which are not permanently invoked. A concrete example would be P2 whose *main* method invokes the *sum* method once with a memory consumption of 34 MB. While the total execution time from the client is 8.77s, the Lambda execution takes only 1.67ms which leads to an effective billing period of only 100ms. Beyond the free tier, the associated cost for a million calls per month in the region $us - west2$ would be 20.8 US$. The same workload could be delivered by an on-demand virtual machine in EC2 at a minimum of 4.39 US$ (t2.nano). The fewer calls are forecasted, the more the Lambda pricing model becomes effective, with the cutting point at around 210000 calls per month.

Table 7 concludes the observations with a comparison of the source code size before and after the transformation. Due to generated boilerplate code and local method code duplication, the overhead relative to the original size becomes sometimes significant.

Table 7: Application source code size comparison.

Flavour	**P1:S**	**P2:S**	**P3:S**	**P4:S**	**P5:S**	**P6:S**
Original	20 kb	32 kb	44 kb	40 kb	40 kb	96 kb
Lambdafied	548 kb	12436 kb	988 kb	2960 kb	–	1796 kb
Overhead	2640%	38763%	2145%	7300%	–	1771%

4 Discussion

Our findings in automated Java code to Lambda units transformation look promising for future cloud application engineering. The results are also beneficial to programming education where rather simple object-oriented applications are in wide use and educators regularly struggle to keep up with new application hosting formats and platform services.

Difficulties originate from code which is not prepared for individual function access. According to a recent study, at least 20% of Java methods are too accessible (*public* instead of *protected* or *private*) [14], while for our work, they are sometimes too inaccessible, although the solution to both is the same: powerful refactoring tools for software engineers. Further difficulties originate from interfacing with the Java virtual machine, for instance through the classloader, and with the command-line interface, as well as with data access with differing file paths.

Future work identified by limitations of our approach encompasses the handling of dynamic classloading, server-side state handling, FaaSification beyond Java as input language and AWS Lambda as target service, as well as optimisations for attribute and method dependencies.

Acknowledgements

This research has been supported by an AWS in Education Research Grant which helped us to run our experiments on AWS Lambda as representative public commercial FaaS.

References

[1] Frank Fowley, Divyaa Manimaran Elango, Hany Magar, and Claus Pahl. Software System Migration to Cloud-Native Architectures for SME-Sized Software Vendors. In *43rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 498–509, Limerick, Ireland, January 2017.

[2] Joab Jackson and Lawrence Hecht. TNS Guide to Serverless Technologies: The Best of FaaS and BaaS. online: http://thenewstack.io/guide-serverless-technologies-functions-backends-service/, 2016.

[3] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, September 2016.

[4] Alexander Binun and Günter Kniesel. Joining Forces for Higher Precision and Recall of Design Pattern Detection. Technical Report IAI-TR-2012-01, University of Bonn, January 2012.

[5] David Ameller, Xavier Burgués Illa, Oriol Collell, Dolors Costal, Xavier Franch, and Mike P. Papazoglou. Development of service-oriented architectures using model-driven development: A mapping study. *Information & Software Technology*, 62:42–66, 2015.

[6] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold E. Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182, 2016.

[7] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. Modelling and Managing Deployment Costs of Microservice-Based Cloud Applications. In *9th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, Shanghai, China, December 2016.

[8] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Denver, Colorado, USA, June 2016.

[9] Martin Plümicke. Functional Interfaces vs. Function Types in Java with Lambdas. In *Software Engineering (Workshops)*, number 1129 in CEUR-WS, pages 146–147, Kiel, Germany, February 2014.

[10] Nikita Salnikov-Tarnovski. Java version statistics: 2015 edition. online: https://plumbr.eu/blog/java/java-version-statistics-2015-edition, 2015.

[11] Josef Spillner. Exploiting the Cloud Control Plane for Fun and Profit. unreviewed preprint: arχiv CoRR abs/1701.05945, January 2017.

[12] Andrew K. Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a Typed Lambda-Calculus: A Preliminary Report. In *Types in Compilation – 2nd International Workshop*, volume 1473 of *LNCS*, pages 9–27, Kyoto, Japan, March 1998.

[13] Chen Harel. GitHub's 10,000 Most Popular Java Projects: Here are the Top Libraries They Use. online: https://dzone.com/articles/github's-10000-most-popular, 2013.

[14] Santiago A. Vidal, Alexandre Bergel, Claudia Marcos, and J. Andrés Díaz Pace. Understanding and addressing exhibitionism in Java empirical research about method accessibility. *Empirical Software Engineering*, 21(2):483–516, April 2016.