Type Migration in Large-Scale Code Bases

Ameya Ketkar
Oregon State University
Corvallis, Oregon, USA
ketkara@oregonstate.edu

ABSTRACT

Type migration is a frequent refactoring activity in which an existing type is replaced with another one throughout the source code. Recent studies have shown that type migration is more frequent in larger codebases. The state-of-the-art type migration tools cannot scale to large projects. Moreover, these tools do not fit into modern software development workflows, e.g., in *Continuous Integration*. This paper presents an IDE-independent type migration technique that scales to ultra-large-scale codebases through a MapReduce parallel and distributed process. We have implemented our approach in a tool called T2R. We evaluated it on codebases as large as 790 KLOC for specializing functional interfaces. Our results show that T2R is safe, scalable and useful. Open source developers accepted 70 migration patches spanning over 202 files.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software maintenance tools**;

KEYWORDS

Software Engineering, Refactoring, Type Migration

ACM Reference Format:
Ameya Ketkar. 2018. Type Migration in Large-Scale Code Bases. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3236024.3275434

1 INTRODUCTION

We encountered the need for *type migration* when we studied the adoption of *lambda expressions* in Java [19]. Java 8 retrofitted lambda expressions—a core feature of functional programming—into a mainstream imperative, object-oriented language. In our formative study on 241 open-source Java projects comprising more than 100,000 lambda expressions, we found that developers use them *inefficiently*. Particularly, Java provides 43 *Functional Interfaces* to support lambda expressions, out of which 35 are specialized to work with primitive types. These eliminate *auto-boxing*, thus improving performance. We observed that 20% of the lambda expressions bound to the built-in Functional Interfaces were misused. For example, developers used the generic interface `Function<Integer, T>` instead of the specialized ones, e.g., `IntFunction`, `IntUnaryOperator`, `IntToDoubleFunction`, and `IntToLongFunction`.

Type migration can effectively help in mitigating this problem by automatically converting the uses of generic Functional Interfaces to the specialized ones. However, current type migration techniques [2, 14, 16, 26–28] suffer from two main limitations. First, these techniques (e.g., type-constraints-based ones [2, 26]) do not scale to large codebases, for example, to the projects we used in our formative study with more than 500 KLOC. Second, they depend on IDEs, and consequently, are not suitable for modern software development workflows where code is continuously built and tested with Continuous Integration (CI). Previous studies found an increasing adoption of CI [12, 13], especially because it avoids breaking builds, helps to catch bugs earlier and frees up personal machines from running tests. However, IDE-dependent refactoring tools for type migration cannot take advantage of such a workflow.

To alleviate these problems, we propose an IDE-independent technique for scalable type migration. This technique integrates with build systems (e.g., Gradle, Maven, Ant) and is amenable to distributed computing. Our contributions are:

- A large-scale formative study of 100,000 lambda expressions providing insights into adoption and pitfalls of Functional Interfaces.
- An empirically-motivated approach for type migration in large codebases, which employs a three-step process to collect, analyze, and transform types. Each step is amenable to MapReduce processing, thus making the approach scalable.
- A graph modelling the type structure of the program, facilitating analysis for safe type migration.
- An implementation of the approach, T2R (Type *T* to Type *R*), which specializes Java 8 Functional Interface usages.
- An evaluation of the technique on seven open-source projects which shows that our refactoring is scalable, safe, and useful for real-world developers.

2 APPROACH

A safe type migration technique requires *whole-program analysis*, therefore it is challenging to make it scalable. One possible solution is to make the analysis distributed, but this implies accessing source code files one at a time, possibly in an arbitrary order, thus making it unsuitable for whole-program analysis. To overcome this, we propose a three-step process that, given the source code and the mappings between types and their respective methods, (1) collects relevant type and syntactic information from the source code, (2) analyzes this information to narrow down modification sites, and (3) applies code transformations. The approach bears several intricacies, e.g., collecting complete source code information for accurate analysis, constructing an expressive model to facilitate

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275434

Table 1: T2R Evaluation Results

Project	Patches			#Changed Files	SLOC
	Generated	Passed	Accepted		
CoreNLP	18	18	18	34	576K
SonarQube	14	14	14	47	551K
Speedment	5	5	5	14	127K
Neo4j	17	16	16	46	787K
Apache Cassandra	17	15	15	23	355K
java-design-patterns	2	2	2	2	27K
prestodb	11	11	0 (10*)	36	543K
Total	**84**	**81**	**70**	**202**	**≈3m**

type migration analysis, and effectively translating analysis results into source code transformations.

We build our technique upon an AST visitor [10], translating each visited AST node that is relevant to the *migration source type* T to a model called TRLC (Type-Relevant Language Construct). TRLC effectively captures an Identification for the AST node and its type-dependent nodes, where Identifications allow distinguishing each node related to T in the entire codebase. For example, we capture the name, signature, and owner class of method declarations as its Identification. Also, we capture Identification of its parameters or return expression if their type is related to T. At most one TRLC is constructed per AST node, using the locally-available syntactic and type information, allowing this translation to be distributed.

We then map TRLCs into a model of an abstract semantic graph [7], named *Type-Fact Graph* (TFG), in which nodes represent TRLCs' Identifications and edges are labelled with the type dependence relation between the nodes (e.g., an edge exists between nodes for a method declaration and its return expression). TFG is inspired by the formalization of refactoring using graph transformations [20].

Each of these TFGs are then merged among themselves to form a single unified TFG, possibly containing multiple disconnected subgraphs. Merging graphs preserves nodes and edges, thus helping to discover type dependencies among AST nodes in the code: the set of nodes in each subgraph represents type-dependent AST nodes throughout the codebase. Since graph merge operation is associative and commutative, and an empty TFG acts as an identity element, this analysis can be executed in a parallel manner. These subgraphs are then filtered by applying a set of *preconditions*, which prevents changes from proliferating into external libraries and usages of generic types. The remaining subgraphs are translated into a model that captures the Identification and the transformation instructions for each AST node, i.e., the Refactorables. The AST nodes are visited again, and if the Identification of a visited node is present in the set of the Refactorables, the corresponding transformation instruction is used for in-place AST node rewrites.

3 IMPLEMENTATION

T2R was implemented to migrate six generic functional interfaces to their 33 specialized counterparts. To collect relevant information from the source code and to apply the transformations, T2R uses `error prone` [10]. This provides flexibility to integrate T2R with any build system, in addition to making it usable in any IDE or as a standalone tool. T2R uses `guava` graph library [9] to construct TFGs, and utilizes Java's parallel streams to achieve parallelization when analyzing TFGs. T2R also has additional features to prevent migration to apply on specified packages, which can be extended to classes, methods or annotations.

4 EVALUATION

We analyzed T2R on seven mature and popular projects with thousands of stars on GitHub, which are widely used in industry. The overview of the results is illustrated in Table 1. We answer these research questions through our evaluation:

RQ1 (Accuracy) *How accurate are the performed type migrations?*
Results: After applying type migrations by T2R, we compiled the projects and run all test cases. Overall, T2R produced 81 correct patches out of 84, achieving 96% accuracy. The three patches failed because of Java reflection and dependency on Scala code.

RQ2 (Scalability) *How scalable is this approach?*
Results: We observed that the first and last phases (i.e., information collection and refactoring) do not add significant overhead to the build process. The second phase (i.e., analysis for narrowing down the transformation sites) takes less than five seconds in all projects, with an average of 400 sites to analyze for each project. We anticipate that parallelizing this phase by employing distributed graph libraries (e.g., Apache Giraph [1], Pregel [18]) can further improve its scalability.

RQ3 (Usefulness): *How useful are the patches produced by T2R?*
Results: The patches produced by T2R were sent to the original developers of the projects. Out of the 81 patches sent, 70 were accepted and 10 are still under review (indicated by * in Table 1). The list of patches can be found online [15].

5 RELATED WORK

Previous work has studied library migration in practice [5, 23, 25] and suggested approaches for type-level refactorings, e.g., by facilitating migration via adapters [3], intermediate layers [8], annotations [4], capture-and-replay of change operations [11], type inference algorithms [16], or API migration DSLs [17, 22]. Most related to our work is type migration approach is based on type constraints [2, 26–28] which determine the sites to be updated when replacing types, in order to preserve the type correctness of the program. However, none of the proposed techniques can scale to ultra-large-scale codebases, or they are bound to a specific IDE, hindering their applicability in practice. ClangMR [29] works for C++ codebases and shares a similar goal to ours, i.e., facilitating refactorings in large codebases. However, it requires domain knowledge and program analysis expertise to use it.

6 CONCLUSION

As projects mature, both the need for and the complexity of type migrations increase beyond the capabilities of current solutions. In this paper, we present an automated technique for scalable type migration in large codebases. We implemented T2R to specialize functional interfaces in Java. Our results show that the type migrations performed by T2R is safe, scalable and useful. Inspired from previous work [6, 21, 24, 30, 31], we are currently extending T2R by designing a change-oriented mapping DSL that will enable updating client code for library migration.

REFERENCES

[1] Apache. 2012. Apache Giraph. http://giraph.apache.org/. Accessed: 2018-06-07.

[2] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 265–279. https://doi.org/10.1145/1094811.1094832

[3] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. 2010. Swing to SWT and back: Patterns for API migration by wrapping. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. 1–10. https://doi.org/10.1109/ICSM.2010.5610429

[4] Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*. 359. https://doi.org/10.1109/ICSM.1996.565039

[5] Bradley Cossette and Robert J. Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. 55. https://doi.org/10.1145/2393596.2393661

[6] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Trans. Softw. Eng. Methodol.* 20, 4, 19:1–19:35. https://doi.org/10.1145/2000799.2000805

[7] Premkumar T. Devanbu, David S. Rosenblum, and Alexander L. Wolf. 1996. Generating Testing and Analysis Tools with Aria. *ACM Trans. Softw. Eng. Methodol.* 5, 1 (Jan. 1996), 42–62. https://doi.org/10.1145/226155.226157

[8] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. 2008. *ReBA*: *r*efactoring-aware *b*inary *a*daptation of evolving libraries. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. 441–450. https://doi.org/10.1145/1368088.1368148

[9] Google. 2010. Guava: Google core libraries for Java. https://github.com/google/guava Accessed: 17 July 2018.

[10] Google. 2011. Error Prone. https://github.com/google/error-prone Accessed: 23 March 2018.

[11] Johannes Henkel and Amer Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. 274–283. https://doi.org/10.1145/1062455.1062512

[12] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[13] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[14] JetBrains. 2012. Type Migration Refactoring. https://www.jetbrains.com/help/idea/migrate.html. Accessed: 2018-06-06.

[15] Ameya Ketkar. 2018. https://ameyaketkar.github.io/T2RResults.html Accessed: 13 June 2018.

[16] Raffi Khatchadourian. 2017. Automated refactoring of legacy Java software to enumerated types. *Automated Software Engineering* 24, 4 (01 Dec. 2017), 757–787. https://doi.org/10.1007/s10515-016-0208-8

[17] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. 2015. SWIN: Towards Type-Safe Java Program Adaptation Between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation (PEPM '15)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/2678015.2682534

[18] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[19] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 85 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133909

[20] Tom Mens, Serge Demeyer, and Dirk Janssens. 2002. Formalising Behaviour Preserving Program Transformations. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*. 286–301. https://doi.org/10.1007/3-540-45832-8_22

[21] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A graph-based approach to API usage adaptation. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. 302–321. https://doi.org/10.1145/1869459.1869486

[22] Marius Nita and David Notkin. 2010. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 205–214. https://doi.org/10.1145/1806799.1806832

[23] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2012. Mining Library Migration Graphs. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. 289–298. https://doi.org/10.1109/WCRE.2012.38

[24] Cédric Teyton, Jean-Rémy Falleri, and Xavier Blanc. 2013. Automatic discovery of function mappings between similar libraries. In *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. 192–201. https://doi.org/10.1109/WCRE.2013.6671294

[25] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A Study of Library Migrations in Java. *J. Softw. Evol. Process* 26, 11 (Nov. 2014), 1030–1052. https://doi.org/10.1002/smr.1660

[26] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring Using Type Constraints. *ACM Trans. Program. Lang. Syst.* 33, 3, Article 9 (May 2011), 47 pages. https://doi.org/10.1145/1961204.1961205

[27] Frank Tip, Adam Kiezun, and Dirk Bäumer. 2003. Refactoring for Generalization Using Type Constraints. *ACM SIGPLAN Notices* 38, 11 (2003), 13. https://doi.org/10.1145/949343.949308

[28] Frank Tip and Peter F. Sweeney. 1997. Class Hierarchy Specialization. *ACM SIGPLAN Notices* 32, 10 (1997), 271–285. https://doi.org/10.1145/263700.263748

[29] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. 2013. Large-Scale Automated Refactoring Using ClangMR. In *2013 IEEE International Conference on Software Maintenance*. 548–551. https://doi.org/10.1109/icsm.2013.93

[30] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 325–334. https://doi.org/10.1145/1806799.1806848

[31] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 195–204. https://doi.org/10.1145/1806799.1806831