EQBENCH: A Dataset of Equivalent and Non-equivalent Program Pairs

Sahar Badihi
Univ. of British Columbia, Canada
shrbadihi@ece.ubc.ca

Yi Li
Nanyang Technological University, Singapore
yi_li@ntu.edu.sg

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

***Abstract*—Equivalence checking techniques help establish whether two versions of a program exhibit the same behavior. The majority of popular techniques for formally proving/refuting equivalence are evaluated on small and simplistic benchmarks, omitting "difficult" programming constructs, such as non-linear arithmetic, loops, floating-point arithmetic, and string and array manipulation. This hinders efficient evaluation of these techniques and the ability to establish their practical applicability in real scenarios. This paper addresses this gap by contributing EQBENCH – the largest and most comprehensive benchmark for equivalence checking analysis, which contains 147 equivalent and 125 non-equivalent cases, in both C and Java languages. We believe EQBENCH can facilitate a more realistic evaluation of equivalence checking techniques, assessing their individual strength and weaknesses. EQBENCH is publicly available at: https://osf.io/93s5b/.**

***Index Terms*—Equivalence checking, benchmark, Java, C.**

I. INTRODUCTION

Equivalence checking establishes whether two versions of a program have identical behavior and is used in a variety of tasks, such as verifying the correctness of software upgrades, refactoring, and optimizations [1]. The most common form of equivalence used in practice is *functional equivalence* (or input-output equivalence), which establishes whether two terminating versions of a program produce the same output for any identical input [2], [3]. Several equivalence checking techniques have been recently proposed [3], [4], [5], [6], [7], [8], [9]. These techniques accept as input two versions of a program and either formally prove equivalence or produce a counterexample to show that the input programs are non-equivalent.

The techniques are typically evaluated on small and simplistic benchmarks: RÊVE [10] was evaluated on 23 EQ and 9 NEQ C programs, with an average of 13.2 lines of code in each program. ModDiff [7] was evaluated on 16 EQ and 12 NEQ C programs with an average of 7.4 lines of code per program. CLEVER [8] used the ModDiff benchmark and further extended it with Python programs, bringing the total numbers to 29 EQ and 21 NEQ cases, with an average of 9.5 lines of code per program. Our earlier work, ARDIFF [9], substantially extended these benchmarks, using 73 EQ and 69 NEQ programs with an average of 33.9 lines of code per program. Yet, these benchmarks are written in Java and most of them do not have a C equivalent.

The lack of a representative and common benchmark hinders the comparison between the tools. Moreover, as existing benchmarks are relatively small, that makes it difficult to realistically assess the strengths and weaknesses of existing approaches. This paper aims at addressing this gap by contributing EQBENCH – the largest and most comprehensive benchmark for equivalence checking analysis. It unifies and extends benchmarks used in earlier works [10], [7], [8], [9], contributing 147 *equivalent* and 125 *non-equivalent* cases, which we capture in both C and Java, to facilitate applicability for different equivalence checking techniques.

Moreover, a distinct feature of our benchmark is that it includes complex language constructs, such as non-linear arithmetic, loops, floating-point arithmetic, and string and array manipulations, which challenge most formal methods/equivalence checking techniques. We borrow programs containing these constructs from existing literature on evaluating formal method techniques in the presence of complex path conditions and non-linear functions [11] and adapt them to the equivalence checking context, as discussed in Section II. In comparison with our earlier ARDIFF benchmark, EQBENCH includes additional cases that handle string and array manipulations, systematically collected meta-data on each benchmark which describes the types of changes it contains, and both Java and C version of each benchmark to facilitate comparison between different language-specific tools.

II. DATASET GENERATION

We now describe the process we followed for creating EQBENCH.

Data Selection. We started from including in our dataset benchmarks proposed by recent work on symbolic-execution-based equivalence checking [10], [7], [8], [9]. Besides the ARDIFF benchmark [9], which we base EQBENCH on, the remaining benchmarks are relatively small and contain no complex constraints. As such, we introduced additional complexity and more diverse language constructs into the dataset. To this end, we adapted benchmarks collected by Li et al. [11] from the literature on evaluating symbolic and concolic execution methods in the presence of complex path conditions and non-linear functions [12], [13]. The methods of those benchmarks are classical numerical computation functions used in real-world distributions. For example, *stat* computes the mean and standard deviation of a list of numbers and *tsafe* is an aviation safety program that predicts and resolves the loss of separation between airplanes.

Change Injection. As benchmarks by Li et al. [11] were not originally designed for the equivalence checking problem, we injected changes in the programs to create their *equivalent* (EQ) and *non-equivalent* (NEQ) versions. Specifically, for each program, we created one EQ and one NEQ version of each of its methods with at least three lines of code (we cannot effectively inject changes in shorter methods). We then created several EQ and NEQ versions of each program: first, we created a program version per a changed method, i.e., modifying one method only and leaving the remaining ones unchanged. Then, we created one EQ version where *all* methods are modified to their equivalent versions and one NEQ version where *all* methods are modified to their non-equivalent versions. For example, for the *tsafe* benchmark that contains three methods, we created four EQ and four NEQ versions: three EQ + three NEQ versions for EQ or NEQ changes in one of the original methods and one EQ + one NEQ versions for EQ or NEQ changes in all methods together. Other combinations of EQ and NEQ methods are also possible, as discussed in Section IV.

When creating EQ and NEQ versions of a method, we controlled for the number, type, and location of the inserted changes. We used a random number generator to automatically pick the number of changes to inject in each method: between one and three. Given the size of the methods in our benchmark (31.4 LOC on average), we believe such a number of changes is realistic and applicable for the main equivalence checking usage scenario – regression verification, where equivalence of successive, closely related versions of a program is established.

To generate NEQ methods, we randomly selected whether to introduce each change within a loop or not. Based on that, we randomly picked an *assignment* or a *control* statement to change (we did not modify method calls, loop control statements, i.e., *break* and *continue*, etc.) Inspired by the mutation testing techniques [14] that generate faulty programs (mutants) by injecting simple syntactic changes, we inserted/deleted/updated parts of the selected statements. Specifically, we represented the selected statement as an Abstract Syntax Tree (AST) whose nodes are program variables and operators and further randomly picked an AST edit operation: insertion (Ins), deletion (Del), or update (Upd) [15].

If a selected statement was an assignment, Ins added an expression to its right-hand side, which is a set of AST nodes with a randomly selected operator (e.g., $+$, $-$, and $/$) and an operand of the corresponding type (e.g., Integer, Float, and String). For example, the statement `c=a-b` could be modified to `c=a-b+10`. Del removed a random set of AST nodes from the right-hand side of the statement or removed the statement altogether, in case it was not an initialization of a variable. For example, the statement `int c=a+b` could become `int c=a` or `int c`. Upd replaced a set of randomly-selected operators and/or operands in the right-hand side of the assignment statement by a set of new values. For example, the statement `c=a-b` could be modified to `c=a+10` where the AST nodes `-` and `b` are replaced with `+` and `10`, respectively.

The treatment of control statements, i.e., *if* and *loop* conditions, is similar, except that the changes were made to the control condition rather than the right-hand side of an assignment. We considered two types of control statement operators: logical (e.g., && and ||) and comparison (e.g., `<=` and `!=`) [16], randomly selecting which one to modify. Ins added random comparison and/or logical operators with the corresponding operands selected at random to the control condition. For example, the statement `if(a<b)` could become `if(a<b||a==0)` by adding one logical and one comparison operators with the corresponding operands. Del removed random logical and/or comparison operators and, if the control condition became empty, replaced the condition by `true`. For example, the statement `if(a<b&&b==0)` could be modified to `if(a<b)` or to `if(true)`. Upd replaced a set of logical and/or comparison operators and their corresponding operands at random. For example, Upd of `if(a<b)` could result in the statement `if(a<=b+10)`.

TABLE I: NEQ: change types and their frequencies.

<table>
<tr><th>Stmt.</th><th>Type</th><th>Descriptions</th><th>Example</th><th># Cases</th></tr>
<tr><td rowspan="3">Assignment</td><td>Ins</td><td>Adding a random expression to the right-hand side of the statement</td><td>c=a-b; ⇒ c=a-b+10;</td><td>46</td></tr>
<tr><td>Del</td><td>Removing a random expression from the right-hand side of the statement or removing the whole statement</td><td>c=a-b; ⇒ c=a;</td><td>35</td></tr>
<tr><td>Upd</td><td>Replacing a random expression in the right-hand side of the statement</td><td>c=a-b; ⇒ c=a+10;</td><td>31</td></tr>
<tr><td rowspan="3">Control</td><td>Ins</td><td>Adding random comparison and/or logical operators with the corresponding operands selected at random</td><td>if(a<b) ⇒ if(a<b || a==0)</td><td>37</td></tr>
<tr><td>Del</td><td>Removing random logical and/or comparison operators or replacing the condition by <code>true</code></td><td>if(a<b) ⇒ if(true)</td><td>20</td></tr>
<tr><td>Upd</td><td>Replacing a random set of logical and/or comparison operators and their operands</td><td>if(a<b) ⇒ if(a<=b)</td><td>28</td></tr>
<tr><td colspan="2">Total</td><td colspan="2">–</td><td>197</td></tr>
</table>

Table I summaries these possible changes, together with an example for each change type and the number of changes of each type that we injected during our benchmark generation process. Overall, we injected 197 changes to generate 83 NEQ methods, with 112 and 85 changes to assignment and control statements, respectively; 42% of the injected changes (83/197) were insertions, 28% (55/197) were deletions, and 30% (59/197) were updates. After applying the changes, we verified the effect of the change on the return value of the method and generated a test case exemplifying the difference between the original and produced versions of the program containing the method. We added these test cases to each benchmark in our dataset.

To generate EQ methods that represent realistic software evolution scenarios, we applied method-level code refactoring techniques, such as, extracting and inlining variables, replacing magic numbers with variables, removing dead code, and decomposing conditionals [17], [18]. We also renamed variables and inserted *dead* or *unreachable* code [19]. Specifically, dead code is code that is executed but has no effect, e.g., adding `x++` followed by `x--`. Unreachable code is the code that is guarded by conditions that cannot hold in practice, e.g., `if(false)`. Such redundant code, albeit not explicitly added by developers, appears in real scenarios as a result of program optimization, modification, and reuse [20]. Similar to

TABLE II: EQ: change types and their frequencies.

Type and Description	Example	# Cases
Refactoring		
Extracting variables to simplify assignments	*return q*p - p*0.05;* ⇒ *base=q*p; discount=p*0.05;* *return base-discount;*	40
Inlining variables to simplify assignments	*c=a*b;* *return c;* ⇒ *return a*b;*	25
Replacing magic numbers with variables	*c=a*b*9.81;* ⇒ *gravity=9.81;* *c=a*b*gravity;*	17
Removing dead code	*c=a*b;* *return a/b;* ⇒ *return a/b;*	8

Decomposing conditionals	if(*a>b* && *a<c*){} ⇒
	if(*inRange*(*a*)){}
	bool inRange(*a*){ *return a>b* && *a<c;* }
		5

Combining conditions having same results	if(*a>b*){ *return a;* }
	if(*a>c*){ *return a;* } ⇒
	if(*a>b* || *a>c*){ *return a;* }
		2

Replacing nested conditions by a flat list of conditions	if(*isDead*){ *c=a;* }
	else{ if(*isRetired*){ *c=b;* } }
	return c; ⇒
	if(*isDead*){ *return a;* }
	if(*isRetired*){ *return b;* }
	return c;
		4

Redundant code		
Adding dead code	⇒ *c=c;*	63
Adding unreachable code	⇒ if(*false*){ *c=10;* }	5
Renaming variables	*t=height*width;* ⇒ *area=height*width;*	7
Total		176

NEQ cases, we randomly selected whether to introduce each change within a loop or not and further randomly selected between applying refactorings, renaming variables, or inserting redundant code.

Table II shows the number of changes of each type that were introduced in EQ methods. Out of 176 changes injected to create the 83 EQ methods, 57% (101/176) changes were refactorings, 36% (63/176) were insertion of dead code, 3% (5/176) were insertion of unreachable code statements, and 4% (7/176) were variable renaming.

To validate the equivalence of the original and the produced EQ versions, an additional member of our research group independently reviewed all programs created by the first author of this paper, validating their equivalence to the original program. When possible, we also ran an equivalence checking technique to assert equivalence [9]. We further manually translated each NEQ and EQ cases from Java to C and from C to Java, as necessary.

III. Dataset Characteristics

Dataset Statistics A summary of the benchmark is given in Table III. The first four columns of the table show the name of each benchmark, the number of EQ and NEQ versions it includes, the type of non-linear operations it contains, and the method size in lines of code (LOC) – minimum, maximum, and mean. The fifth and sixth columns show the fraction of complex non-linear arithmetic and loops in a program, averaged across EQ and NEQ versions of the benchmarks. For example, the *bess* benchmark contains 18 EQ and 18 NEQ versions ranging from 4 to 96 LOC, with 41.8% of complex statements on average.

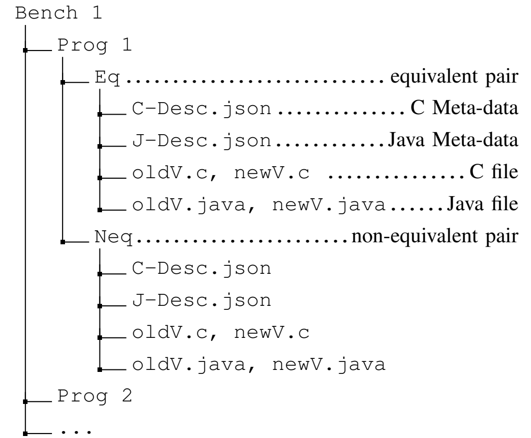

Fig. 1: The structure of the benchmark.

Overall, considering programs from all benchmarks together, 58.8% of the programs (160 out of 272) contain at least one loop and there are 0.89 loops per program on average. Similarly, 57.3% of the programs (156 out of 272) contain at least one statement with complex non-linear arithmetic and there are 25.1% of statements with non-linear arithmetic per program on average. The last column of Table III shows the fraction of changed statements in each of the benchmarks, averaged across all EQ and NEQ versions of a benchmark. There are 14.4% changed program statements per version, on average.

Data Representation. Our benchmarks are publicly available in an archived repository, as a ZIP file [21]. For easier distribution, we also stored a version of the repository in GitHub [22]. The structure of the repository is shown in Figure 1. It is organized in 18 directories corresponding to 18 benchmarks, where each benchmark takes a row in Table III. Inside each benchmark directory, we have one folder for each program. Since we have both equivalent and non-equivalent pairs for each program, we provide two sub-folders within each program directory, i.e., EQ and NEQ. Inside each of the program pair sub-directory, there are four versions: two written in Java (oldV.java and newV.java) and two written in C (oldV.c and newV.c).

For each program version, we also provide two files that contain the meta-data describing the program, in JSON [23] format. The files are named *C-Desc.json* and *J-Desc.json* and correspond to C and Java versions of the program, respectively. Figure 2 shows an example of the meta-data for the Java version of `sign` program in the `airy` benchmark. The file is structured as follows:

- *benchmark name*: the name of the benchmark, e.g., *airy*;
- *program name*: the name of the program, e.g., `sign`;
- *LOC*: program lines of code;
- *#loops*: the number of bounded or un-bounded loops;
- *#non-linear arithmetic*: the number statements with non-linear arithmetic expressions;
- *changes*: description of each injected change (line number, type, and operation);
- *counter-example* (for NEQ only): values for input parameters making two programs non-equivalent.

TABLE III: Overview of the EQBENCH dataset.

Bench.	# EQ/NEQ Vers.	Operations	LOC			% Non-Linear Exp.	# Loops	% Changed Stms.
			Min.	Max.	Mean			
RÊVE	23/9	Polynomials	16	53	13.2	0.4	1.3	9
CLEVER	29/21	Polynomials	4	14	9.5	0	0.9	27.5
airy	6/6	logarithms, arrays	5	57	12.4	6.5	0.5	36.5
bess	18/18	Polynomials, square roots	4	96	22.8	41.8	0.5	8.5
caldat	5/5	trigonometrics, strings	22	76	39.2	25.8	1.5	10.2
dart	1/1	Polynomials		10		9.1	0	21
ell	13/13	Polynomials, trigonometrics	6	116	62.4	37.9	1.5	7.2
frenel	3/3	Object manipulation, trigonometrics	62	168	100.6	21.9	2	3.4
gam	12/12	Logarithms, factorials, strings	7	87	33.3	32.4	1.4	10.8
pow	1/1	Exponentials		22		4.9	0	4
ran	11/11	Polynomials, exponentials, arrays	7	99	39.5	37.8	2.6	5.2
sine	1/1	Bit-vector		148		7.8	0	0.2
tcas	4/4	Constant equality checks	11	54	26.8	0	0	10.4
tsafe	4/4	Trigonometrics	9	39	26.4	31.4	0	6.2
raytrace	6/6	object manipulations, strings	5	73	25.9	35.1	0.6	11.2
statcalc	1/1	Trigonometrics, strings		10		13.2	0	10
optimization	3/3	Strings, exponentials	5	19	9.6	43.5	0	29.2
ej_hash	6/6	object manipulations	5	32	9.5	0	0	28.1
New	95/95	Polynomials, logarithms, Strings, etc.	4	168	33.4	29.3	0.81	11.9
Total	147/125	–	4	168	26.5	25.1	0.89	14.4

```
1  "benchmark name": "airy",
2  "program name": "sign",
3  "LOC": 16,
4  "# Loops": 0,
5  "# non-linear arithmetics":0,
6  "changes": [{
7    "change line": 3,
8    "change type": "Insertion",
9    "change operation": "Arithmetic"
10 },{
11   "change line": 8,
12   "change type": "Update",
13   "change operation": "Arithmetic"
14 },{
15   "change line": 11,
16   "change type": "Insertion",
17   "change operation": "Logical"
18 }],
19 "counter-example":[{"a": 5.0},{"b": 5.0}]
```

Fig. 2: Meta-data of `sign` program in `airy` benchmark.

The C version of the file differs only in the *LOC* and *change line* fields as the changes applied to both versions are identical. The change locations in the program files (.c and .java) are also annotated by the `//change` comments.

IV. LIMITATIONS AND EXTENSIONS

Our main motivation for creating the EQBENCH is to provide evaluation target for equivalence checking techniques, allowing researchers to assess the capabilities of their tools and compare them with each other on a standard benchmark. One limitation of our benchmark is that it targets regression verification case, which assumes that substantial portions of the code is unchanged between two subsequent versions of a program. As such, our dataset might not be applicable to other applications of equivalence checking, such as verification of compiler correctness [24], optimization [25], and program synthesis [26].

Because of the manual effort needed for manually constructing and validating the EQ and NEQ cases, we created one EQ and one NEQ version for each method of the programs and a limited number of EQ and NEQ cases per program (one per each changed methods and one for all changed methods together – in both EQ and NEQ scenarios). As a straightforward extension, other combinations of changed methods, with some, but not all changed methods of both types, can easily be created to enhance the applicability of the benchmark to intra-procedural equivalence checking analysis. Another obvious way of extending our dataset is to include more equivalent/non-equivalent pairs. We invite other researchers to join our effort of extending EQBENCH by contributing to it directly on GitHub [22].

EQBENCH can be further extended by collecting examples of changes from open-source repositories. Such extension will make it possible to investigate the size and complexity of evolutionary changes, comparing them to our manually injected ones. The main challenges making such extension would be (a) dissecting a project to extract a stand-alone sample that can serve as a benchmark, (b) characterizing the change, and (c) manually labeling it as EQ or NEQ.

Another way to extend EQBENCH is by translating the collected cases into additional languages, e.g., Python (required by CLEVER [8]) and Boogie [27] (required by SymDiff [4]).

V. CONCLUSION

This paper contributes EQBENCH: a dataset of 147 equivalent and 125 non-equivalent program pairs from 18 different software projects, which we made available in both Java and C programming languages. To the best of our knowledge, our dataset is the largest, the most comprehensive, and systematically documented one. We hope it will help support equivalence checking research, by allowing researchers to assess the capabilities of their tools and compare them on a common set of programs.

Acknowledgments. Part of this work was funded by Huawei Canada and by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040).

References

[1] A. Kuehlmann and F. Krohm, “Equivalence Checking Using Cuts and Heaps,” in *Proc. of the Design Automation Conference (DAC)*, 1997, pp. 263–268.

[2] B. Godlin and O. Strichman, “Inference Rules for Proving the Equivalence of Recursive Procedures,” *Acta Informatica*, vol. 45, no. 6, pp. 403–439, 2008.

[3] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential Symbolic Execution,” in *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, 2008, pp. 226–237.

[4] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, “Symdiff: A Language-agnostic Semantic Diff Tool for Imperative Programs,” in *Proc. of the International Conference on Computer Aided Verification (CAD)*, 2012, pp. 712–717.

[5] B. Godlin and O. Strichman, “Regression Verification: Proving the Equivalence of Similar Programs,” *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 241–258, 2013.

[6] J. Backes, S. Person, N. Rungta, and O. Tkachuk, “Regression Verification Using Impact Summaries,” in *Proc. of SPIN Workshop on Model Checking of Software*, 2013, pp. 99–116.

[7] A. Trostanetski, O. Grumberg, and D. Kroening, “Modular Demand-driven Analysis of Semantic Difference for Program Versions,” in *Proc. of the International Static Analysis Symposium (SAS)*, 2017, pp. 405–427.

[8] F. Mora, Y. Li, J. Rubin, and M. Chechik, “Client-specific Equivalence Checking,” in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2018, pp. 441–451.

[9] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin, “ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code,” in *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, 2020, pp. 13–24.

[10] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, “Automating Regression Verification,” in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2014, pp. 349–360.

[11] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, “Symbolic Execution of Complex Program Driven by Machine Learning-Based Constraint Solving,” in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2016, pp. 554–559.

[12] M. Souza, M. Borges, M. d’Amorim, and C. S. Păsăreanu, “CORAL: Solving Complex Constraints for Symbolic Pathfinder,” in *Proc. of NASA Formal Methods Symposium*, 2011, pp. 359–374.

[13] P. Dinges and G. Agha, “Solving Complex Path Conditions Through Heuristic Search on Induced Polytopes,” in *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 425–436.

[14] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649–678, 2010.

[15] B. Fluri, M. Wursch, M. PInzger, and H. Gall, “Change Distilling: Tree Differencing for Fine-grained Source Code Change Extraction,” *IEEE Transactions on Software Engineering (TSE)*, vol. 33, no. 11, pp. 725–743, 2007.

[16] Y.-S. Ma and J. Offutt, “Description of muJava’s Method-level Mutation Operators.”

[17] M. Fowler, “Refactoring home page.” https://refactoring.com/catalog/.

[18] “Refactoring GURU.” https://refactoring.guru/remove-assignments-to-parameters.

[19] “Redundant Code,” https://en.wikipedia.org/wiki/Redundant_code.

[20] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer, “How Much Does Unused Code Matter for Maintenance?” in *Proc. of the International Conference on Software Engineering (ICSE)*, 2012, pp. 1102–1111.

[21] (2021) EqBench. https://osf.io/93s5b/.

[22] (2021) EqBench. https://github.com/shrBadihi/EqBench.

[23] JSON. https://www.json.org/json-en.html.

[24] G. C. Necula, “Translation Validation for an Optimizing Compiler,” in *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 83–94.

[25] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron, “From Relational Verification to SIMD Loop Synthesis,” in *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 123–134.

[26] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic Superoptimization,” *Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.

[27] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A Modular Reusable Verifier for Object-oriented Programs,” in *Proc. of the International Symposium on Formal Methods for Components and Objects (FMCO)*, 2005, pp. 364–387.