How Good are Your Types?

Using Mutation Analysis to Evaluate the Effectiveness of Type Annotations

Rahul Gopinath*, Eric Walkingshaw†
School of EECS, Oregon State University
Email: *gopinatr@oregonstate.edu, †walkiner@oregonstate.edu

***Abstract*—Software engineers primarily use two orthogonal means to reduce susceptibility to faults: software testing and static type checking. While many strategies exist to evaluate the effectiveness of a test suite in catching bugs, there are few that evaluate the effectiveness of type annotations in a program. This problem is most relevant in the context of gradual or optional typing, where programmers are free to choose which parts of a program to annotate and in what detail. Mutation analysis is one strategy that has proven useful for measuring test suite effectiveness by emulating potential software faults. We propose that mutation analysis can be used to evaluate the effectiveness of type annotations too. We analyze mutants produced by the MutPy mutation framework against both a test suite and against type-annotated programs. We show that, while mutation analysis can be useful for evaluating the effectiveness of type annotations, we require stronger mutation operators that target type information in programs to be an effective mutation analysis tool.**

I. Introduction

Reliability is a core concern in software engineering. *Static type systems* address this concern by providing a lightweight way to prove the absence of a restricted class of errors [42, p. 1]. *Software testing* offers a complementary set of trade-offs for addressing software reliability: by evaluating parts of a program with a variety of inputs and checking that the program returns the expected outputs, software testing provides a flexible way of addressing a larger class of errors, but is more labor intensive and cannot prove the absence of bugs [21].

There are several metrics for evaluating the quality of a software test suite. For example, *code coverage* measures the percentage of program code that is executed when running the test suite [2]; in general, higher is better since a test suite will certainly not catch any bugs in untested code.

At first glance, it seems like such program-level metrics do not make sense in the context of static type systems. If a language has a (sound) static type system, then any program in that language is guaranteed to be free from the class of errors excluded by the type system. Of course, there are ways to judge the quality of the type system itself (e.g. more expressive types can express more sophisticated properties and rule out larger classes of bugs), but these are language-level rather than program-level concerns. However, in any language with explicit type annotations and subtyping, there is a question of how *specific* a type annotation should be, and this program-level concern has an impact on software reliability. Additionally, several recent languages feature *optional* or *gradual type systems* [7], which enable mixing static and dynamic typing through the use of optional type annotations. In this setting, it makes sense to ask how the set of type annotations provided impacts the reliability of a particular program.

To illustrate the need for metrics to evaluate the quality of type annotations, consider the following example in Python, taken from the documentation of the optional typing feature.[1]

```
def broadcast(msg: str, srv: List[Any]) -> None:
  ...
```

In this example, the parameter `srv` is annotated to have type `List[Any]`, where `Any` represents any type. Thus, `srv` can be instantiated by any list. A more specific type for this program is illustrated in the following example, where we define a several type aliases to represent connection options, addresses, and servers, then refine the type of `srv` to be a list of servers.

```
ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]
def broadcast(msg: str, srv: List[Server]) -> None:
  ...
```

Obviously the initial `List[Any]` type annotation is easier to write, but the more elaborate annotation is probably more effective at preventing bugs. Observe that there is a partial order among type annotations in terms of specificity that corresponds exactly to the subtyping relation on the corresponding types.

As another example, consider the following program, which is a subject in the study described in Section II.

```
def hashtags(jr: Dict[str, Any])
  -> Tuple[int, List[str]]:
    ht = jr.get('entities', {}).get('hashtags', [])
    hashtags = sorted(set(h['text'] for h in ht))
    return jr['ctime'], hashtags
```

The parameter `jr` is annotated as a dictionary from strings to `Any`, but it is more specifically expected to be a dictionary of JSON objects (encoded as Python records). We could make the effort to define a JSON type alias, which precisely describes the form of the expected argument, but is this effort worthwhile? That is, will it make our program more reliable?

Given the spectrum of possibilities for a type annotation, what do we gain from refining a low-effort, less specific annotation to a high-effort, more specific annotation? Do we at some point suffer from diminishing returns? At what stage should we stop trying to refine our types, but instead focus on writing better tests? Is it better to prioritize high coverage with less specific annotations, or to focus on more

[1] https://docs.python.org/3/library/typing.html

specific annotations in well-chosen places? To answer these questions, we need a way to measure how effective a particular set of type annotations are at preventing bugs, and also to compare the effectiveness of type annotations against other reliability strategies, such as test suites, property-based tests, and contracts.

Mutation analysis is a way of evaluating the quality of a test suite by injecting faults into a program and counting how many of the injected faults are caught by the test suite [36]. Our idea is to repurpose mutation analysis to evaluate the quality of the *type annotations* in a program by counting how many (and observing which kinds) of the injected faults can be statically ruled out by the type system. Our hypothesis is that *higher coverage* and *more specific* type annotations will lead to catching more of the injected faults.

Work on *property-based testing* (PBT) [14], [26] provides tangential evidence that mutation analysis is a good fit for evaluating the quality of type annotations. In PBT, the programmer writes executable properties that should always hold within a program and the PBT framework generates random inputs to test against these properties. PBT is therefore a sort of midway point between testing and types since it expresses general properties of a program, similar to types, but is based on execution-based sampling rather than formal verification, similar to tests. Previous work has shown that the strength of properties can be evaluated using mutation analysis [13], [17].

However, we also foresee some challenges for adapting mutation analysis to evaluate the quality of type annotations. The most significant is that existing mutation frameworks probably do not generate the kinds of mutations needed to best evaluate type annotations. Since early frameworks were written in the context of statically typed languages, most mutation operators are type preserving and so do not generate ill-typed mutants if the original program is type correct. More generally, due to their pedigree in software testing, mutation frameworks focus on generating the kinds of faults that test suites are written to test against, but these may not be the same kinds of faults we expect our type system to catch.

In order to determine whether existing mutation frameworks are already a good fit for evaluating the quality of type annotations, we conducted a small study, described in Section II. In the study, we evaluated two Python programs (*twitter-graph* and *w3lib*) that had both adequate test suites and type annotations that cover almost all of the functions. We used *MutPy*, a Python mutation analysis framework with a comprehensive set of mutation operators often used in academic research [18], to evaluate the resulting mutants both using type checker and test suites. Our results, described in Section III, indicate that while mutation analysis can indeed be used to evaluate the quality of type annotations, the current set of mutation operators offered by *MutPy* are inadequate for this task. Further, it is surprisingly difficult to come up with mutants that actually describe subtle type faults.

Indeed, another possibility is that type annotations as provided by tools such as Mypy provide too little benefit compared to a sufficient test suite. While there seems to be some support for such a conclusion[2], our research is still in a very early stage for any such conclusions to be drawn.

In Section IV, we describe the kinds of mutations that triggered type errors in our analysis and also discuss how the suite of mutation operators can be extended to better support mutation analysis of the quality of type annotations. We discuss threats to validity in Section V, related work in Section VI and conclude with a discussion of future work in Section VII.

II. Methodology for Assessment

To evaluate the utility of mutation analysis with respect to type annotations, we chose two projects: our own *twitter-graph* project [24] (83 lines of code in 10 functions fully type annotated, and 25 test cases), and the open-source *w3lib* project [43] (369 lines of code in 42 functions with 34 type annotations, and 97 test cases) both using Python *unittest* library. The test suite for *twitter-graph* has 99% statement coverage, and 98% branch coverage; the test suite for *w3lib* has 94.7% statement coverage, and 91.7% branch coverage. For *w3lib*, we relied on the type annotations by the library author;[3] for *twitter-graph*, we included as much type information as possible by annotating all function signatures with the strictest types possible.

We generated mutants for both projects using *MutPy* [31], a comprehensive mutation framework available for Python [18]–[20]. The MutPy mutators are listed in Table I. For both projects, we evaluated each generated mutant using both the test suite and the Python type checker *Mypy* [33].

For *twitter-graph*, we measured the number of mutants killed by the type checker over a range of more strict to less strict type annotations. We started with strictest possible type annotations, then progressively generalized them over several steps by first replacing the deepest basic types, such as `int` and `str` with `Any`, then replacing the innermost parameterized types with completely generic parameters, such as `List[Any]` or `Dict[Any,Any]`, with simply `Any` until a single level of type annotations. After each step of generalizing the type annotations, we re-ran the Mypy type checker on the mutants.

III. Results

The results of our analysis are given in Table II. For *twitter-graph*, we found no difference between the mutants killed using the most strict type annotations and the least strict type annotations. Further, for both projects and for all operators, the mutants killed by the type checker was a strict subset of the mutants killed by the test suite.

IV. Discussion

Our analysis yielded two surprising results: (1) the mutants killed by the type checker were a strict subset of the mutants killed by the test suite, and (2) simple type annotations were just as effective at killing mutants as more sophisticated type

[2] http://blog.cleancoder.com/uncle-bob/2017/01/13/TypesAndTests.html

[3] https://www.reddit.com/r/Python/comments/4tl50y/conclusions_about_mypy_after_my_experience/d5iwuzm/

operator	description
AOD	arithmetic operator deletion
AOR	arithmetic operator replacement
ASR	assignment operator replacement
BCR	break continue replacement
COD	conditional operator deletion
COI	conditional operator insertion
CRP	constant replacement
DDL	decorator deletion
EHD	exception handler deletion
EXS	exception swallowing
IHD	hiding variable deletion
IOD	overriding method deletion
IOP	overridden method calling position change
LCR	logical connector replacement
LOD	logical operator deletion
LOR	logical operator replacement
ROR	relational operator replacement
SCD	super calling deletion
SCI	super calling insert
SIR	slice index remove
CDI	classmethod decorator insertion
OIL	one iteration loop
RIL	reverse iteration loop
SDI	staticmethod decorator insertion
SDL	statement deletion
SVD	self variable deletion
ZIL	zero iteration loop

TABLE I: Mutation operators used by MutPy.

Project	Operator	Type Checker	Test Suite	Total
tgraph	AOR	0	5	6
tgraph	CDI	5	9	9
tgraph	COD	0	5	5
tgraph	COI	0	9	9
tgraph	CRP	2	22	37
tgraph	DDL	2	2	2
tgraph	EHD	0	1	1
tgraph	EXS	0	0	1
tgraph	LCR	0	1	1
tgraph	OIL	0	1	3
tgraph	RIL	1	2	2
tgraph	ROR	0	6	8
tgraph	SDI	7	8	8
tgraph	SDL	16	38	46
tgraph	SVD	29	30	30
tgraph	ZIL	0	3	3
w3lib	AOR	11	17	17
w3lib	ASR	2	5	5
w3lib	BCR	0	2	3
w3lib	COD	1	7	7
w3lib	COI	6	48	49
w3lib	CRP	6	220	322
w3lib	EHD	0	2	3
w3lib	EXS	0	0	2
w3lib	LCR	0	12	12
w3lib	LOR	0	1	3
w3lib	OIL	0	9	10
w3lib	RIL	1	8	10
w3lib	ROR	0	8	11
w3lib	SDL	63	172	184
w3lib	SIR	0	8	9
w3lib	ZIL	2	10	10

TABLE II: Mutants killed by the type checker and the test suite. Note that the mutants killed by the type checker are a subset of those killed by the test suite.

annotations. One reason for the first result is simply that our test suites are good, that is, they have high coverage and stronger oracles[4]. Since type errors are a subset of semantics errors, a good test suite can be expected to catch many type errors.

A more interesting reason for these results is that the mutation operators we used are targeted at evaluating test suites, rather than type errors. The kinds of type errors produced by the mutation operators are quite crude, usually raising a dynamic type error as soon as the mutated code is executed. A test suite with good code coverage will find all such type errors. And relatively simple type annotations can rule out such type errors just as well as more sophisticated types.

In the rest of this section, we give a sense for the kinds of type errors caught in our analysis (Section IV-A), discuss limitations of current mutation frameworks for generating realistic type errors (Section IV-B), and provide ideas for addressing these limitations in the future (Section IV-C).

A. Mutations that induce type errors

In the following, we enumerate the mutators that produced ill-typed mutants during our analysis.

1) Arithmetic operator replacement: A typical AOR mutant killed by the type checker is produced by the replacement of the `+` operator by another arithmetic operator, in a context where it is actually used for string concatenation.

```
- print("Hello" + " World")
+ print("Hello" - " World")
```

Another similar example is the replacement of the `%` operator, used for string formatting, by arithmetic operators.

```
- print("Hello %s" % "World")
+ print("Hello %s" * "World")
```

2) Statement deletion: The SDL mutator can introduce a type error by deleting the statement that initializes a variable prior to its use. For example, in the following fragment `y` is unbound, resulting in a type error.

```
def square(x: int) -> int:
-    y = x**2
+    pass
     return y
```

Perhaps surprisingly, deleting a return statement, as in the following fragment, does not introduce a type error since a Python function with no return statement implicitly returns `None`, which is a member of all types.

```
def square(x: int) -> int:
-    return x**2
+    pass
```

3) Self variable deletion: The SVD mutator deletes the `self` receiver from an object field access.

```
class A(object):
    def add(self, x: int) -> int:
-       y = x + self.val
+       y = x + val
        return y
```

[4] Since it is developer written, the test cases can assert the exact value, not just the type of values.

In the example above, the mutation turns the field access into a reference to an unbound variable `val`, resulting in a type error.

4) Decorator deletion and insertion: The DDL and SDI mutators can introduce type errors when `@property` or `@staticmethod` declarations are deleted or inserted, changing the interface of the mutated classes. For example, in the following fragment, the deletion of `@staticmethod` causes a type error since the first argument to a non-static method is a reference to the object itself, in this case of type `A` rather than the declared argument type `int`.

```
class A(object):
    ...
-   @staticmethod
    def mymethod(x: int) -> int:
        return x
...
a = A()
a.mymethod()
```

5) Reverse iteration loop: A rather subtle type error encountered in our experiment was caused by the RIL mutator, which reverses the order that elements are iterated over. This causes a type error when looping over the entries in a dictionary, since the dictionary iterator object doesn't support reversal.

```
def tst(hd: Dict[str, str]):
-   for v in hd.items():
+   for v in reversed(hd.items()):
        pass
```

6) Zero iteration loop: The ZIL mutator replaces a loop body by a break statement. This triggered a type error due to limitations of type inference in Mypy. In the following fragment, after removing the call to `append` Mypy can no longer infer that `lst` is a list of strings, as required when it is passed as an argument to `join`.

```
def mymethod(mystr: str) -> str:
    lst = []
    for k in mystr.split(mystr):
-       lst.append(k)
+       break
    return ','.join(lst)
```

7) Constant replacement: The CRP mutator is restricted to only replace constants with other constants of the same type, so we did not initially expect it to produce any type errors. However, Mypy statically checks format strings; mutating these can produce static errors as illustrated below.

```
-   print("Hello \%s" \% y)
+   print("" \% y)
```

B. Limitations of AST-level mutation

Mutation testing researchers rarely differentiate between source-level mutation and AST-level mutation. Indeed, aside from *c-mutate* [44] and mutation tools that operate on bytecode, most mutation tools (such as Major [28], MutPy, and MuCheck [32]) operate on an AST. An unfortunate consequence is that this precludes many simple errors that programmers make on the source level that correspond to large differences in the AST, and so are out of scope for simple AST mutators.

For example, consider the following edit to fix a program by moving a parenthesis. Although the source-level edit is minor, the two programs have quite different ASTs so the incorrect program will not be generated by a simple AST-level mutator.

```
-   root = (-b + sqrt(b**2 - 4*a*c))/(2*a)
+   root = (-b + sqrt(b**2 - 4*a*c)/(2*a))
```

Another observation is that AST-level operator replacement mutations do *not* correspond to source-level operator replacement mutations, due to operator precedence rules. For example, the following two programs differ by one character but have different shaped ASTs.

```
-   print('Ex %s' % 1 + 2)
+   print('Ex %s' % 1 * 2)
```

The initial version contains a rather subtle type error since `%` binds more tightly than `+`. This seems like a plausible mistake a human could make but would not be generated from the correct version with a simple AST-level operator replacement.

C. Toward new mutation operators

Our analysis revealed that MutPy does not generate the mutants we need to evaluate the quality of type annotations. Therefore, we propose extending MutPy with new mutation operators that affect the types in mutants in new and more varied ways. Related work by Lerner et al. [34] on systematically modifying programs that *already* contain type errors in order to try to find a related well-typed program is relevant here since their modifications necessarily impact the types. As future work, we propose to extract type-affecting mutation operators from their systematic program modifier.

More immediately, we propose several new mutation operators based on the preceding discussion. We illustrate each mutation operator on Python, but the ideas can also be applied to other languages. Further research is needed to verify that these type-affecting mutators are indeed useful for evaluating the quality of type annotations.

1) Enhanced constant replacement operator: Constant replacement should be extended to also substitute constants with different types, as illustrated below.

```
def area(self):
-   return Pi * (self.r**2)
+   return "" * (self.r**2)
```

2) Enhanced operator replacement: Operator replacement should be adapted to simulate source-level operator replacement, which may require restructuring the AST to account for operator precedence.

```
-   print('Ex \%s' \% 1 * 2)
+   print('Ex \%s' \% 1 + 2)
```

3) Mutate parenthesis placement: A new mutator that simulates human faults due to mistakes in parenthesis placement in nested expressions. This mutation also involves locally restructuring the AST.

```
-   print('\%s' \% ("Example " + (no + 1)))
+   print('\%s' \% (("Example " + no + 1)))
```

4) Mutate variable names: A new mutator that modifies the names of variables to simulate spelling mistakes.

```
def area(radius):
-    return Pi * radius**2
+    return Pi * radius_**2
```

5) Mutate nesting: A new mutator that moves statements in and out of nested blocks. In Python, this corresponds at the source level to changing the indentation of a statement.

```
def squares(lst):
  s = []
  for i in lst:
      v = i**2
-     s.append((i, v)
+ s.append((i, v)
  return s
```

6) Reorder function arguments: A new mutator that reorders the arguments passed to a function.

```
- broadcast_message("Ack", [(('::1',10), {})])
+ broadcast_message([(('::1',10), {})], "Ack")
```

7) Disable function call: A new mutator that replaces the invocation of a function by its lookup. In Python, this amounts to removing the trailing parentheses so that the function itself is returned, rather than its result.

```
- val = myfunction()
+ val = myfunction
```

8) Add elements to containers: Mutator that add new elements to containers. Elements may be of different types so that the corresponding container type changes.

```
- val = ['a', 'b']
+ val = ['a', 'b', 1]
```

9) Change type of container: Mutator that replaces one type of container by another, for example, replace a list with a set or dictionary.

```
- val = ['a', 'b']
+ val = {'a', 'b'}
```

V. Threats to Validity

Our results are based on two small programs. These programs were selected because they were the first Python programs to be annotated with types. One of the programs was written by the first author, and this is the only program used in the analysis of strict vs. less-strict types. It is possible that these biases and limitations confound our results, and so they do not generalize to larger and more diverse programs. Our results are also based on analysis in a single language (Python) using a single mutation framework (MutPy), and so it is possible that our observations do not generalize beyond Python and MutPy.

VI. Related Work

The idea of mutation analysis was proposed by Lipton [36], its concepts formalized by DeMillo et al. [16], and was first implemented by Budd [9]. Previous research [10], [37], [41] suggests that it subsumes different test coverage measures, including *statement*, *branch*, and *all-defs* dataflow coverage. Research also shows that mutants are similar to real faults in terms of error trace produced [15], the ease of detection [3], [4], and effectiveness [29]. The foundational assumptions of mutation analysis—"the competent programmer hypothesis" and "the coupling effect"—have been validated both theoretically [25], [46], [47] and empirically [23], [39], [40]. Early work on mutation analysis was in the context of statically typed languages. Recent work has extended this to dynamically typed languages: Derezinska et al. [18] studied mutators for Python, Li [35] for Ruby, and Mirshokraie et al. [38] for Javascript. Aside from evaluating test suites, mutation analysis has been used to evaluate property based tests [17], specifications in theorem provers [6], and comparing static analysis generators [5].

The only previous work that has looked at a metric for type annotations in the context of gradual typing is Takikava et al. [45]. They were focused on comparing the *performance* of gradual type systems with annotations of different specificity, whereas we are interested in investigating whether more specific annotations are able to detect more faults.

In order to improve the ability of mutation frameworks to catch bugs in real software, it is helpful to understand the kinds of errors that programmers make in practice. An early study by Youngs [49] directly lead to the first mutation operators by Budd [9]. Ko et al. [30] enumerate many kinds of errors that programmers make and provide a cognitive model for these errors. The Blackbox is a dataset of errors made by programmers using the BlueJ IDE. A summary of this dataset was published by Jadud [27], which reports that 45.7% of Java compilation errors would be undetected in Python. These are errors that we think should be specifically targeted by mutation operators. Another analysis of Blackbox by Brown et al. [8] showed similar results. A part of this dataset (a years worth of data totalling 37 million compilations) was analyzed by Altadmri et al. [1], who found that invoking functions with the wrong type was the second most frequent kind of error; incompatible return arguments were also common.

Previous work on *variational typing* [11], [12] can provide a means to efficiently type check a large number of mutant programs at once. To support variational typing, mutation operators can introduce fine-grained, local variation points called *choices* [22], [48] between the original code and several different mutated alternatives. Variational typing can then efficiently type check every variant program that can be produced by picking a particular mutation at each choice point.

VII. Conclusion and Future Work

We evaluated the suitability of mutation analysis for measuring the effectiveness of gradual type annotations in Python programs. Our results indicate that while mutation analysis can indeed be used in this manner, at least theoretically, the current mutation operators do produce the kinds of faults needed to elicit useful insights about typing annotations. We suggest new mutation operators that can produce errors commonly seen in type-incorrect programs. These operators need to be

investigated in the context of type annotations to ensure that they are actually useful.

It is surprisingly difficult to come up with mutation operators that can produce mutants with *subtle* type errors that are not killed by the simplest of type annotations. One of the ultimate conclusions of this line of work could be that more specific type annotations do not provide much value over less specific ones. However, much of the discussion in this paper focused on how the limitations of the current generation of mutation operators (adapted from research on statically typed languages such as C and Java) may be biased towards generating mutants that that do not contain interesting type errors.

References

[1] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 522–527.

[2] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[5] C. A. Araújo, M. E. Delamaro, J. C. Maldonado, and A. M. Vincenzi, "Correlating automatic static analysis and mutation testing: towards incremental strategies," *Journal of Software Engineering Research and Development*, vol. 4, no. 1, p. 5, 2016.

[6] S. Berghofer and T. Nipkow, "Random testing in isabelle/hol," in *Software Engineering and Formal Methods*, ser. SEFM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 230–239.

[7] G. Bracha, "Pluggable type systems," in *OOPSLA04 Workshop on Revival of Dynamic Languages*, 2004.

[8] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: ACM, 2014, pp. 223–228.

[9] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1980, pp. 220–233.

[10] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1980.

[11] S. Chen, M. Erwig, and E. Walkingshaw, "An error-tolerant type system for variational lambda calculus," in *ACM SIGPLAN International Conference on Functional Programming*, 2012, pp. 29–40.

[12] ——, "Extending type inference to variational programs," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 1, pp. 1:1–1:54, 2014.

[13] Y. Cheng, M. Wang, Y. Xiong, D. Hao, and L. Zhang, "Empirical evaluation of test coverage for functional programs," in *International Conference on Software Testing, Verification and Validation*, 2016.

[14] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *ACM SIGPLAN Notices*, vol. 46, no. 4, pp. 53–64, 2011.

[15] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1996, pp. 158–171.

[16] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[17] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Quickchick: Property-based testing for coq," in *The Coq Workshop*, 2014.

[18] A. Derezińska and K. Hałas, "Analysis of mutation operators for the python language," in *International Conference on Dependability and Complex Systems DepCoS-RELCOMEX*. Springer, 2014, pp. 155–164.

[19] A. Derezinska and K. Halas, "Experimental evaluation of mutation testing approaches to python programs," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014, pp. 156–164.

[20] A. Derezinska and K. Hałas, "Improving mutation testing process of python programs," in *Software Engineering in Intelligent Systems*. Springer, 2015, pp. 233–242.

[21] E. W. Dijkstra, "On the reliability of programs," http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF, 1971.

[22] M. Erwig and E. Walkingshaw, "The choice calculus: A representation for software variation," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 1, pp. 6:1–6:27, 2011.

[23] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *International Symposium on Software Reliability Engineering*, Nov 2014, pp. 189–200.

[24] R. Gopinath, "A script to compute average degree of a tweet stream," https://github.com/vrthra/twitter-graph.

[25] R. Gopinath, , C. Jensen, and A. Groce, "The theory of composite faults," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2017.

[26] J. Hughes, "Specification based testing with quickcheck," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2003.

[27] M. C. Jadud, "An exploration of novice compilation behaviour in bluej," Ph.D. dissertation, Computing Laboratory, 2007.

[28] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 433–436.

[29] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 654–665.

[30] A. J. Ko and B. A. Myers, "Development and evaluation of a model of programming errors," in *IEEE Symposium on Human Centric Computing Languages and Environments*, ser. HCC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 7–14.

[31] Konrad Halas, "Mutpy 0.4.0 mutation testing tool for python 3.x," https://pypi.python.org/pypi/MutPy/0.4.0.

[32] D. Le, M. A. Alipour, R. Gopinath, and A. Groce, "Mucheck: An extensible tool for mutation testing of haskell programs," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 429–432.

[33] J. Lehtosalo, "Optional static typing for python," http://www.mypy-lang.org/, 2016.

[34] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 425–434.

[35] N. Li, M. West, A. Escalona, and V. H. Durelli, "Mutation testing in practice using ruby," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2015, pp. 1–6.

[36] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep., 1971.

[37] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[38] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 74–83.

[39] A. J. Offutt, "The Coupling Effect : Fact or Fiction?" *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, Nov. 1989.

[40] ——, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[41] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, Tech. Rep., 1996.

[42] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.

[43] S. project, "W3lib library of web-related functions 1.16.0," https://pypi.python.org/pypi/w3lib.

[44] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *International Conference on Software Engineering*. ACM, 2008, pp. 351–360.
[45] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen, "Is sound gradual typing dead?" in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 456–468.
[46] K. S. H. T. Wah, "A theoretical study of fault coupling," *Software Testing, Verification and Reliability*, vol. 10, no. 1, pp. 3–45, 2000.
[47] ——, "An analysis of the coupling effect I: single test data," *Science of Computer Programming*, vol. 48, no. 2, pp. 119–161, 2003.
[48] E. Walkingshaw, "The choice calculus: A formal language of variation," Ph.D. dissertation, Oregon State University, 2013, http://hdl.handle.net/1957/40652.
[49] E. A. Youngs, "Human errors in programming," *International Journal of Man-Machine Studies*, vol. 6, no. 3, pp. 361–376, 1974.