Functional Spectrums - Exploring Spectrum-Based Fault Localization for Haskell

Anonymous Author(s)*

Abstract

Fault localization plays an important role in debugging, one technique thereof is *spectrum-based fault localization*, which uses tests and program coverage to produce a *spectrum* of locations involved in passing and failing tests. Despite its extensive application in Java, this technique remains underexplored within functional programming languages. This gap underscores a critical challenge: adapting spectrum-based fault localization to accommodate the unique characteristics of functional paradigms. Addressing this challenge, we evolve current spectrum-based approaches by extending the spectrums with types and AST structure. We introduce a *rule-based system* tailored to capture more complex attributes of the spectrum. Spectrums are generated using an ingredient for the Tasty test framework, which allows easy adoption and reproducibility. Through an empirical study involving 11 real-world programs, we meticulously investigate the generated spectrums along with the effectiveness of the rule-based system and their correlation to faults. Furthermore, we employ a set of classifiers to evaluate the potential for cross-program extrapolation of our findings. For most bugs, conventional spectrum-based formulas perform promisingly well in a functional context and are only outperformed by classifiers that incorporate these formulas.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Software testing and debugging*.

Keywords: Laziness, Fault Localization, Functional Programming, Haskell

ACM Reference Format:
Anonymous Author(s). 2024. Functional Spectrums - Exploring Spectrum-Based Fault Localization for Haskell. In *Proceedings of Haskell Symposium (Haskell '24).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Functional programming has earned a reputation for being at the forefront of type and programming language research, but we believe it can also be a champion of tooling and software engineering practices.

It remains open *what* tooling functional programmers really want, but tooling they need. Haskell is known for innovating in more niche features like software transactional memory and techniques like property-based testing. It has innovative tools like the Hoogle search engine, but we believe we can innovate further and introduce tools that would be harder to develop for other paradigms.

Tools are quite important to software developlent. According to modern developer surveys, approximately 50% of development is spent debugging, half of which is spent fixing bugs [3]. An important part of the debugging process is *fault localization*, i.e. determining *which* part of the program is at fault, which can be assisted by specialized tools. This challenge spans most software paradigms, including functional programming [10, 17].

One way to assist the fault localization process is to introduce automated tools [11, 24], for example, spectrum-based fault localization (SBFL). A *program spectrum* is a matrix where the rows represent test results and the columns represent code locations. Each entry indicates whether that location was involved in the test or not, with an additional column that indicates whether the test passed or failed. A program spectrum is created by running individual tests and collecting program coverage [27]; thus capturing different aspects of the program by branching over the test suite. Program spectrums have been successfully applied in imperative languages, based on the premise that by comparing elements involved in failing tests and those involved in passing tests, we can deduce which location is at fault. However, they have yet to become established in functional communities.

1.1 Example

Consider the function and properties in fig. 1.

```
foldInt :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldInt _ z [] = 0
foldInt f z (x:xs) = (foldInt f z xs) `f` x

prop_sum, prop_prod, prop_diff :: [Int] -> Bool
prop_sum  xs = foldInt (+) xs 0 == sum xs
prop_prod xs = foldInt (*) xs 1 == product xs
prop_diff xs = foldInt (-) xs 0 == negate (sum xs)
```

Figure 1. A buggy program and associated properties

Here, we intended to implement a `fold`, but made a mistake: we accidentally wrote `0` instead of `z` in line 2. The `prop_sum` and `prop_diff` touch all locations in the spectrum, but `prop_prod` only touches the base case, since QuickCheck's initial test is always `[]`.

Running the properties for the program in fig. 1, produces the spectrum in table 1. A standard spectrum consists of only the tests, whether they pass or fail, and the locations involved in each test. In this paper however, we produce *augmented*

Haskell '24, September 02–07, 2024, Milan, Italy
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1. A spectrum for the code in fig. 1

name	type	result	2:18	3:31	3:35-36	3:22-37	3:43	3:22-43	2:1-3:43
type identifier			Int	Int -> Int -> Int f	[Int] xs	Int	Int x	Int	
sum	QC	True	100	2162	2255	2255	2255	2255	2355
prod	QC	False	1	0	0	0	0	0	1
diff	QC	True	100	2224	2319	2319	2319	2319	2419

spectrums. These augmented spectrums also include the types of expressions and tests involved, the name of the identifier, and the number of evaluations of this location in the test. The notation `2:18` represents line 2 column 18, and – indicates a range of characters.

$$\textbf{Tarantula: } \frac{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}}{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}+\frac{n_{e_p}}{n_{e_p}+n_{t_p}}} \quad \textbf{Ochiai: } \frac{n_{e_f}}{\sqrt{(n_{e_f}+n_{t_f})(n_{e_f}+n_{e_p})}}$$

Figure 2. Standard SBFL formulas. n_{e_f} and n_{e_p} are the number of times the expression e is involved in a failing or passing test respectivley, while n_{t_f} and n_{t_p} is how many total failing and passing tests there were.

Using the formulas detailed in fig. 2 on the spectrum, we can score the locations as detailed in fig. 3. Here, the most suspect location is indeed the underlined `0` in `2:18`: it is involved in more failing tests than other locations, apart from the definition of the `foldInt` that spans lines 2 and 3.

Location	Ochiai	Tarantula
2:18	0.577	0.5
2:1-3:43	0.577	0.5
3:31	0	0

Figure 3. Top 3 suspiciousness scores from classic SBFL formulas, with the bug location in bold.

Although replacing the definition of `foldInt` is certainly an option, the *type information* in the augmented spectrum allows us to distinguish expressions from locations. Using the type information to deduce that `2:18` is an expression, we can break the tie and correctly point to `0` as the most suspicious expression in the spectrum. Still, its not often as clear which location is at fault. If we got the base case correct but had written `f x (foldInt f z xs)` in line 3, we would have the traditional `foldr` instead of a flipped `foldr` as presented here. Running the properties again, this

```
1 foldInt :: (Int -> Int -> Int) -> Int -> [Int] -> Int
2 foldInt _ z [] = z
3 foldInt f z (x:xs) = f x (foldInt f z xs)
```

Figure 4. The program from fig. 1, slightly modified.

accidental `foldr` produces the spectrum in table 2. Here, it is not as clear which location is at fault: while `prop_sum` and `prop_prod` pass, now `prop_diff` fails and touches all except `f` in `foldInt f z xs` in line 3, since QuickCheck tests the empty list and then singleton lists. This exonerates the base case, but does not help us to distinguish the remaining locations. As seen in fig. 5, formulas fall short in this case.

Location	Ochiai	Tarantula
2:18	0.577	0.5
3:24	0.577	0.5
3:37	0.577	0.5
3:39-40	0.577	0.5
3:26-41	0.577	0.5
3:22-41	0.577	0.5
2:1-3:41	0.577	0.5
3:35	0.0	0.0

Figure 5. Classic SBFL formula scores (bug location in bold).

While this would be a challenge to traditional SBFL formulas, our rule-based approach allows us to distinguish these cases, by inspecting the AST structure, types, and identifiers. The rule-based approach is detailed further in section 3.2, but for this example, we could proceed as follows: we can filter out the non-expression by limiting ourselves to only those locations that have a type. In this case, we see that the columns for the remaining faulty expressions look the same, except for `2:18`. We then sort by the AST-based `rTFailFreqDiffParent` rule (see section 3.2), which assigns a value of 0.71 to `z` in `2:18`, 2.29 to `f x (foldInt f z xs)` in `3:22-41`, and 3 to all the others: most locations are evaluated alongside their parent, but `3:22-41` and `2:18` are not always evaluated with their parent (`2:1-3:41`). As the test is a property and properties test the base case first, a failure for the empty list would result in fewer evaluations, similar to what we saw in table 1. With that, we can rank `3:22-41` as the most suspicious. This motivates us to do a detailed analysis to shed light on which attributes are important.

1.2 Contributions

In this paper, we apply spectrums and existing suspiciousness scoring algorithms to Haskell and enrich it with unique, novel features: We use Haskell Program Coverage (HPC) instrumentation to determine whether an expression was touched during a test, but also to extract *how often* the location was evaluated. We note the test-framework (QuickCheck, Hunit, etc.) for later progressing, and capture the *type*, constraints, and identifiers of locations that correspond to *expressions* within the spectrum, forming a richer spectrum than existing literature. We aim to cover many Haskell-specific attributes of test-cases and programs by these changes.

Table 2. A spectrum for the code in fig. 1, with a fixed based case but `f x (foldInt f z xs)` in line 3

name	type	result	2:18	3:24	3:35	3:37	3:39-40	3:26-41	3:22-41	2:1-3:41
type identifier			Int z	Int x	Int -> Int -> Int f	Int z	[Int] xs	Int	Int	
sum	QC	True	100	2654	2560	2654	2654	2654	2654	2754
prod	QC	True	100	2604	2509	2604	2604	2604	2604	2704
diff	QC	False	7	4	0	4	4	4	4	11

We provide the tool for spectrum generation as an `ingredient`[1] for the popular *Tasty* test framework.

To explore the spectras and generate new findings, we implement a rule-based approach to merge novel features and existing approaches. The targets for rules are (1) test attributes (test types, executions, frequency), (2) program attributes (AST structure), (3) existing SBFL formulas, and (4) type-based complexity measures (constraints, arity, order).

The overall goal is to see whether the additional rules beyond existing literature improve fault localization for Haskell programs. To rank the locations for their suspiciousness, we concatenate the rule results into a vector and apply simple machine learning (ML) algorithms such as linear regression, decision trees, and (shallow) neural networks. We chose simple predictors to maintain explainability and ease of comparison. For instance, decision trees provide a transparent view into the rules most effective at isolating specific bugs, while regression models capture the correlations between rule attributes and the presence of faults. They directly correlate with different features and form an insight themselves.

Rather than striving for improved outcomes by selectively interpreting metrics or meta-tuning classifiers, our goal is to offer insights and trends encompassing both successful and unsuccessful techniques. We provide an easy-to-adapt tool for practitioners and researchers to extract rich spectra. Popular open-source projects are used to verify the feasibility of spectrum extraction. Real-world bugs are analyzed in detail by formulating rules that capture spectrum attributes. Known SBFL formulas are re-applied and investigated for suitability, and some simple ML algorithms are tested with rule-based vectors.

The total overview of the pipeline is seen in Figure 6. The novel elements and contributions are marked as bright-blue.

Research Questions. We first investigate the spectrums and look what attributes distinguish them from their non-faulty counterparts.

RQ1.A: Spectrums of functional Programs
What attributes significantly differentiate faulty and non-faulty expressions within spectrums?

Before adaptations, it is worth looking at how existing research performs for typed functional programs. We thus apply common spectrum-based formulas from literature, summarized in table 7 in the appendix.

[1] currently anonymized for peer-review

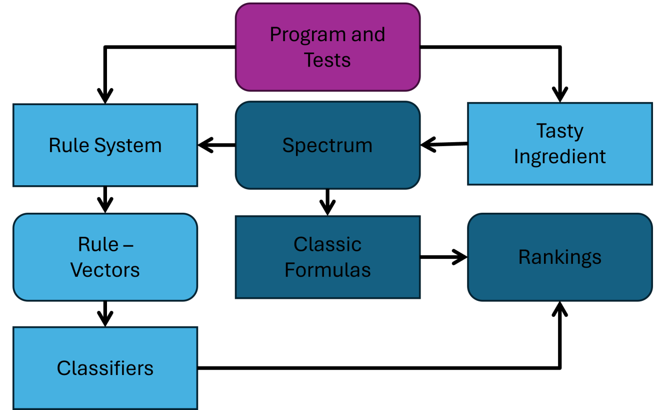


Figure 6. Overview of the fault localization Pipeline

RQ1.B: SBFL Formulas for functional Programs
How well do existing SBFL formulas perform for the given Haskell dataset?

We aim to capture certain attributes of spectrums and their expressions through rules. After implementation, it remains to see if they are applicable. RQ1.C investigates the characteristics of the rules when applied to the spectrum:

RQ1.C: Applicability of Spectrum-Based Rules
What are the most prominent rules triggered by faulty expressions?

As a final subject of investigating the underlying programs, we analyze correlations between the rules and formulas.

RQ1.D: Correlation of Spectrum-Rules
Are there significant correlations between the rules for faulty expressions?

Based on the original data investigation and rules, we apply a set of different simple classifiers and regressors to the data. Due to the exploratory nature, we focus on explainable models and investigate their attributes after fitting.

RQ2.A: Attributes of simple SBFL Models
When fitted to a data point, what rules were the most important for the different models? Are there reoccurring patterns and weights?

A common use of a model is to diagnose faults in (unseen) data, which makes debugging more effective. With RQ2.B we want to see how well the models perform on the programs

331 that they are not fitted for, and if there are recurring patterns,
332 successes, and challenges among them:
333
334 **RQ2.B: Generalization of SBFL Models**
335 How well do the fitted models perform on programs and
336 faults outside of their training data?
337
338 In summary, this research aims to (a) analyze a sample of
339 real world faults and (b) explore directions for predictors
340 that perform better than existing formulas.
341
342
2 Background and Related Work

343 ***Spectrum-based Fault Localization.*** Spectrum-based
344 fault localization (SBFL) was developed as a technique to
345 cover well-testable issues related to the year 2000 problem
346 [27], and is considered one of the most prominent due to its
347 efficiency and effectiveness [29]. After defining a failing test
348 that triggers the Y2k problem of an application, the program
349 tests were executed in order, and their coverage was recorded.
350 Under the assumption that there are (passing) tests covering
351 expected behavior, the issue must originate in statements
352 covered by failing tests without being in passing tests.
353 The Y2K problem consists of straightforward fixes, and
354 thus it is difficult to transfer the techniques developed there
355 to more complex issues. Nevertheless, the idea of collecting
356 per-test coverage to narrow down suspicious statements
357 formed the core of modern SBFL: from the initial concept
358 of intersection, many techniques emerged that use formulas
359 [11, 12, 16, 32] to assign suspiciousness scores to different
360 parts of the program. With differences in the details, all
361 formulas take into account how often a given statement was
362 touched by failing and passing tests, in addition to global
363 attributes of the spectrum (e.g., total number of failing tests).
364 The result of the formulas is used to produce a ranking of
365 (all) statements and report the most suspicious locations.
366 An important piece of work from which we draw is from
367 Naish et al. [22] which discusses the mathematical attributes
368 of spectrum-based formulas. In addition to introducing two
369 new formulas, they prove that some formulas must result in
370 the same ranking (equivalence classes). Within this work, we
371 aim to implement at least one formula from each identified
372 equivalence class.
373
374 ***Other Fault Localization efforts for functional pro-***
375 ***gramming.*** Fault localization in a functional setting has
376 been explored in Liquid Haskell [30], using `refinement`
377 `types`, a type system augmented with logical predicates.
378 They collect constraints and localize faults by mapping a min-
379 imal set of atomic unsatisfiable type constraints to likely bug
380 locations. The work relies on a more powerful type system
381 than Haskell has, namely liquid types, which localize (and
382 repair) errors on the type level. The Liquid Haskell approach
383 requires precise modeling of the expected system-behavior
384 at the type level, which often means giving up type-inference
385

386 In this work, we target programs with existing test suites,
387 and enable developers to get more out of previous testing
388 efforts. Using liquid types, a form of test generation can
389 form supplementary work similar to test generation efforts
390 in program repair.
391
392
2.1 Related Work

393 ***Li et al.*** Comparable work on spectrum-based fault lo-
394 calization for Haskell originates from Li et al. [14]. They
395 collect open source bugs and apply existing SBFL formulas
396 on an expression-level spectrum. To improve generalizabil-
397 ity and introduce an ML approach, the programs were also
398 mutated to extend their data set. Although they publish the
399 dataset which we reuse, the original code is not available.
400 Li et al. have similar goals in introducing SBFL for Haskell,
401 but many of the details differ. On a more fundamental level,
402 our spectrums extend previous work with unique attributes
403 of types, tests, and identifiers. We introduce rules that ex-
404 tend the existing literature to capture more concepts than
405 SBFL formulas currently can. Their approach includes data
406 augmentation, which forms a great venue to synthesize the
407 efforts of both works in future research.
408
409 ***HaskellFL.*** implements the Ochiai and Tarantula algo-
410 rithms for Haskell code [31]. They develop a custom com-
411 piler that compiles the program into SKI-combinators for
412 evaluation, to determine the lines involved in a fault. As
413 they do not integrate with HPC or GHC, an application for
414 real-world programs proves difficult, and no evaluation on
415 large programs is provided.
416
417
3 Implementation & Experiment Setup

418
3.1 Spectrum Generation

419 We introduce a TASTY-SPECTRUM package which adds an
420 *ingredient* to the Tasty test framework that captures cov-
421 erage when tests are run and generates a spectrum. Tasty-
422 Ingredients are a modular way to implement plugins for
423 Tasty to add additional behavior around tests such as re-
424 running, timeouts, or, in this case, data extraction.
425 To generate spectrums, we use the instrumentation pro-
426 vided by Haskell Program Coverage (HPC) and programs
427 compiled with the `-fhpc` flag. This generates `.mix` files that
428 allow HPC to connect the indices it produces to the source
429 locations in the modules. Our implementation also includes
430 a *GHC source plugin*, which integrates with the compiler
431 and extracts type and identifier information from modules
432 during compilation and generates `.types` files.
433
434 ***GHC Source Plugins.*** GHC allows users to define *source*
435 *plugins*, which are run at the end of various stages of compila-
436 tion, including parsing, type-checking, and renaming. These
437 plugins allow the user to modify and interact with the source
438 code after each stage. In the TASTY-SPECTRUM package, we
439 define a plugin that operates at the end of the type-checking
440

441 stage, where we traverse the type-checked expressions, and note their types in the `.types` file. The `.types` files are saved alongside the `.mix` files and later combined with the `.mix` information during spectrum generation.

Haskell Program Coverage (HPC). HPC instrumentation is integrated into GHC, and is based on maintaining an array that counts executions for each source location (which corresponds to expressions) in the module during runtime. Whenever an expression is evaluated, this triggers a "bump" in the array, allowing HPC to track the number of times each expression was evaluated in the module. This array allows access, manipulation and re-initialization at runtime.

Spectrums are generated by running the test suite. As the code has been compiled with the `-fhpc` flag, the RTS will keep the Tix array in memory. Before running each test, we reset the HPC state. After each test, we read the current state of HPC, and track which expressions were evaluated.

After running all tests, the Tix array for each test is combined with the module structure from the Mix files and the type/identifier information from the `.types` files to produce a *type-augmented spectrum* as a `.csv` file. To compress the data, we only include locations that are involved in any of the tests, excluding those that have zero evaluations across the test suite.

3.2 Rules

Fault localization commonly ranks locations based on their *suspiciousness*. To achieve this, the information in the spectrum is quantified and turned into a score. This is traditionally done using formulas that depend on the number of times a location is involved in passing or failing tests, n_{e_p} and n_{e_f} respectively, and the number of total passing and failing tests, n_{t_p} and n_{t_f}. In our analysis we include these classic formulas, but also quantify other elements of the augmented spectrum, aiming to find correlations with faults.

- *Test-type count* the number of tests, passing or failing, that this location is involved in.
- *SBFL-Formulas* apply existing formulas from previous literature; the rule output is the calculated value of the formula.
- *AST structure-based rules* use information based on the distance from a failing location or whether a parent or sibling was executed often.
- *Type-based rules* are based on analysis of the available types and constraints of a location.
- *Meta-rules* operate on the results of the previous, per-module, rules and supplement the data with further analysis. These include the quantile rules and the rules that count how often types, component types, and identifiers appear in failing tests.

Table 4 provides an overview of the type-based rules.

We want to further motivate some of the rules presented in table 7. One general notion is that properties are *stronger* than regular unit tests, as they cover a wider range of input values and have logic beyond an *assert*. It makes sense to rate an expression that is in many passing properties as less suspicious. In a similar, less algorithmic viewpoint, golden tests, i.e. tests that use output comparison, are often written after users report a bug. Thus, it could make sense to rate golden test failures as more suspicious, as they often capture failing behavior, contrary to properties that often test positive program paths. Taking this into account, there is *no one test better than the others* - but there might be patterns that we only find when inspecting them separately.

Table 3. Rules based on AST-based behavior

rASTLeaf	Counts the distance of this node to the nearest leaf.
rFailUniqueBranch	Times this location is touched by failing test that touches none of its sibling expressions.
rFailFreqDiffParent	Ratio of evaluations compared to parent-evaluations.
rDistToFailure	Distance to a location touched in a failing test, by counting links to a common parent.

AST rules (seen in table 3 are based on existing research on active and algorithmic debugging [4, 8, 18]. They aim to capture differences in executions relative to parents and neighbors and reflect control-structures and program flow.

With the group of type rules in table 4, we aim to proxy the complexity of an expression and its context. We expect longer types to indicate a more complex process; especially higher-order functions are a unique case of complexity that is well represented at the type level. `rNumSubTypeFails` aims to connect types seen in failing locations with seemingly un-connected locations — the rationale being that concepts in the program are expressed as types, and there can be a failure in the concept. `rTypeArity` and `rTypePrimitives` allow us to identify correlations of faults with parts of a type and form basis of analysis, e.g. if faults occur in basic elements or complex compositions.

Unlike SBFL formulas, our novel rules are not intended as ranking algorithms, but rather as intermediate results for analysis, model features, and tie breaking (e.g. in table 2).

3.3 Data

We draw data from two Haskell fault datasets, `HasBugs` [1] and `HaFla` [14]. Both datasets provide a similar granularity of faults originating from projects with known faults (based on issues and PRs) whose fault-fixing commits include a test. These tests were extracted to produce a *faulty but tested* version with a failing test suite. We determine faulty expressions as all expressions that are completely within faulty lines, extracted from the git-difference.

Table 4. Rules based on the expressions type

Rule	Description
rTypeArity & rTypeConstraints	Number of arguments and constraints the function has.
rTypeArrows	Number of arrows (`->`) in the type
rTypeFunArgs	Numbers of parentheses in the type to quantify how many *function arguments* there are, and in turn whether it is a higher-order function or not.
rTypeOrder	Counts the number of type applications in the type, such as `Maybe a` or `[[a]]`
rTypePrimitives	Number of primitives, i.e. `String` or `Int`.
rTypeSubTypes	Counts the number of types in the type, i.e., unfolds all constructors and applications.
rTypeLength	Number of Characters of the Type, when represented as `String`.
rNumSubTypeFails	Number of times types which appear in this type are involved in a location involved in a failing test.

A subset of the data was chosen to produce the spectrums that met the required versions of Cabal, tasty (>v1.0), and GHC (>= 8.6). Other limitations excluded projects like Purescript (many of the tests run against compiled Javascript) or Cabal (all bug-asserting tests are package-level tests outside the tasty test suite). This results in a total of 11 programs[2] from 3 projects - **Pandoc**, **Duckling** and an **HLS**-plugin. An overview of the data points used is presented in table 5.

Pandoc is a document converter and, outside of language-specific tooling (GHC, Cabal, HLS, etc.), the biggest Haskell project with over 50k lines of code. The general *flow* of conversion consists of three steps: a reader, an internal representation, and a writer. Most bug reports and issues are based on user-perceived misbehavior, which is commonly captured with a unit or golden test.

HLS is a joint community effort of Haskellers to provide the backbone of a modern Haskell IDE. Most of it is centered on providing a language server in typescript style for the popular Visual Studio Code. Apart from a base framework, many functions are provided as plugins to cover linting, type suggestions, suggested imports, and other features.

Duckling is an open-source Facebook project that extracts structured entities (times, dates, weights, etc.) from texts. The general business logic consists of regex-based rules that are applied in a fine-to-coarse fashion The test suite consists of a domain-specific corpus with examples and *broad* tests that run all examples within a corpus. Generally, the corpus is structured per module, which is why the duckling data points only show one test failure, despite multiple examples being added to a corpus.

Comparison with Defects4J - Comparing the spectra between paradigms is challenging, but to approximate, we consult some data from Defects4J [13]. We draw our data from a public repository shared by René Just[3] that provides statistics from applying GZoltar [28] to a subset of 395 bugs from Defects4J.

The Defects4J bugs inspected have a mean *Source lines of code*(`SLOC`)[23][4] of 57.7k and a median of 62.5k. The mean number of tests in Defects4J is 1439, with a median of 202, with an average of 2 failing tests. Under the assumption that most of the SLOCs represent line-level statements, the resulting spectrums will have a comparable number of elements. The we approximate faulty SLOC for Defects4J as an average of 2.56, based on the lines removed by the patch. In conclusion, the programs and bugs used in this work are comparable in size to Defects4J.

3.4 Experimental Setup

Based on the fault fixing commits of a data point, we revert the source code patch while keeping the changes to the test code, observing a test failure during `cabal test`. At this stage, we also distinguish *noisy* test failures as mentioned in table 5, marking tests that fail before and after the changes as *noisy*. As the next step, the cabal file is altered to include spectrum generation and coverage, following the description in section 3.1 These result files form the basis of a data analysis, done in Python.

RQ1 is answered by investigating the results of their triggered rules. Many of the spectrum attributes are directly captured in rules (e.g., `rTFail` corresponds to *was touched by a failing test*), and thus facilitate the analysis of distributions and proportions.

The primary metric considered for ranking the expressions is the `Top-X-metric` [9]. Within `TopX`, the recommended elements are sorted by their *suspiciousness*, and the *correct* classifications (truly faulty expressions) within the first `X` are counted. For this work, we considered the `Top10`, `Top50` and `Top100`, following previous literature.

Another common metric is `EXAM` [26], assuming that the user follows every recommendation in order until the real fault(s) are fixed. The index of the first correct fault is used to calculate the ratio of the inspected (total) program, with the exam score expressing *how many locations can be skipped when following the recommendations?* The EXAM score is proportional to the *mean reciprocal rank*, another metric commonly reported for FL. For this work, we discarded MRR and EXAM, as we work with different granularity due to our expression level spectrum: when introduced in 2003, EXAM was targeting block-level spectrums, but the sheer difference in the quantity of mostly (benign) expressions would draw

[2] 9 from HasBugs, two from HaFla

[3] https://bitbucket.org/rjust/fault-localization-data/src/master/

[4] SLOC are lines of code, after removing whitespace and comments.

Table 5. Overview of the used data points

Data Point	Issue	Faulty LOC	Faulty Expressions	Total Expressions	Failing Tests	Noisy Test-Failures	Total Tests
pandoc-3be256efb	Wrong application of 'Big Note' highlighting when converting to LaTeX. Reordering necessary.	1	6	88k	6	0	3254
pandoc-4	Failure converting combined code and bold text to LaTeX.	3	12	91k	1	1	3056
pandoc-5	Misinterpretation of code blocks when converting to ROFF MS. Requires escaping.	1	8	61k	2	6	2400
pandoc-6	Misconverting code blocks starting with (1) into enumerations.	5	39	59k	10	13	2365
pandoc-7	Empty multi-cells not picked up when reading LaTeX.	27	72	61k	3	7	2415
hls-2	Issue accounting for relative location "./" instead of expected "."	2	15	269	1	0	6
hls-afac9b18	HLS-Plugins can reformat code, Stylish Haskell was removing the last line of files regardless of whether they had content.	1	17	122	2	0	13
duckling - ea8a4f6d	Wrong pronomina for German million. Regex adjustment.	1	5	288k	1	0	364
duckling - 4cfe88ea	Missing combined durations cases (e.g "2 hours and 20 minutes")	18	4	260k	1	1	342
duckling - 28ddc3bf	Wrong parsing of `1.000,00` for Dutch.	1	5	299k	1	0	346
duckling - 328e59eb	Missing cases for weights (and combinators) in Portuguese language.	19	26	277k	1	1	360

a highly beneficial picture of our approach. Therefore, for ranking evaluations, we focus on the `TopX` metrics [33].

RQ2 is investigated by training classifiers and regressors on the `result files`. Namely we implemented `decision trees`, `random forests`, `linear-` & `logistic regression` and `Multilayer Regressors` from SciKit [25]. At last, we considered a genetic algorithm using Pymoo [2] for an evolutionary search of regressor weights.

To separate the effects of the new rules from existing rules, we assert a total of four configurations: *all*, *classic* (existing sbfl formulas), *original* (only novel rules added by this work) and *cherries* (a handpicked set of rules and formulas). To account for different value ranges, we re-run all experiments with `min-max-scaling`, mapping all features to values between 0 and 1. In the remainder of the paper, this is represented by the terms *scaled* (min-max scaling applied) and *unscaled*.

Fitting the binary classifiers (decision tree, random forest, logistic regression) targets locations to be faulty or not faulty. Regressors are trained to assign faulty locations with a suspiciousness of 1 while other locations have a suspiciousness of 0. In the remainder of the paper models are named after their training data, e.g. *Pandoc-3 model*. For persisting trends of a single project, *pandoc models* refers to all models based on pandoc programs.

GA-based regression. . GAs utilize a custom fitness function to optimize the ranking of the first reported faulty locations, effectively optimizing on `TopX`. For GAs, we set the population to 200 individuals and use Latin Hypercube Sampling [19] to generate the initial population. The population is then evolved trough subsequent generations, by using *binary tournament selection* [20], for selecting the solutions (regression weights) for reproduction based on their fitness. *Simulated Binary Crossover* [7] SBX is used to recombine the selected solutions, and *polynomial mutation* [6] (PM) is used to introduce diversity to the population. We opt for these genetic operators and their recommended parameters values (i.e., SBX with index $\eta_c = 30$, PM with index $\eta_m = 20$ and probability $p_m = 1/n$, with n being the number of regression weights), as they are known to be effective in solving continuous optimization problems [6]. GAs are set to run for 2000 generations or terminate early if no improvement in the fitness function is observed for 100 generations. The solution weights in the final population with the best value of the fitness function is used as the final GA-based regression.

Regressors are evaluated on the resulting `TopX`, while for classifiers, true and false positives are evaluated. A global seed was used to account for inherent randomness.

4 Results

Attributes of Spectrums. The created spectrums range in size from 25Kb (`HLS`), 200 MB (`duckling`) to up to 500 MB (`Pandoc`). Spectrum generation is not a costly addition to the runtime of tests, but compilation time of projects is longer as the `-fhpc` flag is required.

Table 8 groups the expressions into those touched by failing tests and those that are not. This allows for shrinking the spectrum, assuming that statements without failing

tests are *innocent*. When organized in this way, we see that duckling-4cfe88ea, duckling-ea8a4f6d, duckling-1dac46a8 and pandoc-4 do not have faults covered by the tests.

The authors double-checked the test suite, and for duckling, the correct (and expected) corpus tests were failing. We suspect that the tests do not run against the *original source*, but generated code. The generated code is also faulty but is not the origin of the issue, as fixed in the commit. Some of the duckling datapoints, e.g. duckling-328e59eb have faults covered by failing tests. The fix for duckling-328e59eb is more than the adjustment of a regex, and the changes to the structure are successfully tested and represented in the spectrum. For pandoc-4 there are faulty locations on a *reader* that need changes in the data format. The relevant golden test runs with a compiled binary of pandoc (unlike the other pandoc data points) that is invoked by tasty, which is not collected in project coverage. Thus, we have a failing test suite, but the *touched* expressions originate only from noisy test failures.

The existence of faults that are *not directly covered* poses a challenge for this work and a novel aspect of fault localization. Stemming from real projects, the tests are realistic and express community efforts. Although tests cover bugs *semantically*, it does not cover the faulty code and require new spectrum techniques. Due to the common usage of Haskell for domain-specific languages, parsers, and code generation tooling, we expect these types of faults to be more common in functional paradigms than in other languages.

On average, 63.7% of faults are in AST leaves, while 50.5% of expressions are leaves. For duckling, most changes were adjustments to a regex (AST-Leaf) and their wrappers (non-leaf) or required the introduction of a new rule. This results in even distribution of faults in leaves and non-leaves for duckling. Within Pandoc, many faults revolved around combinators and parsers, which involve many higher-order functions. In particular, the program flow in a parser monad produces many non-leaf faulty locations. The combinators (<$>, <|>, etc.) and the patterns (many1Char, noneOf, etc.) are all non-leaf nodes as they require arguments. Due to this structure, the faults in the pandoc programs are proportionally more in non-leaves than leaves.

Most faulty expressions are typed. Usually, one or two faulty locations are untyped, which is a special case of ambiguity that occurs in typing: these are not *expressions*, but rather bindings, e.g. `x = a`. Here, `x` and `a` will have the same type, but the *binding* `x = a` does not have a type.

We see no striking trends in the types of faulty expressions; the most common types are primitives such as Text or UInt, which are also common in non-faulty expressions. The only exceptional types are monadic parsers in pandoc-6 and pandoc-7. The use of monads and the higher-order operators involved is also a reason for the high number of faulty expressions for these data points, as they imply an increased number of function applications per line of code.

Although most expressions are typed, only a few represent an identifier. Less than half of the faulty expressions correspond to an identifier, and 4 data points do not have any faulty expressions that correspond to an identifier. The identifiers encountered match the project vocabulary (e.g., parseMultiCell in pandoc-7) with no trend of shorter identifiers being more faulty. This diverges from existing research's focus on *off-by-one* errors [21] or issues in predicates [15], which also focus on elements with identifiers.

RQ1.A: Attributes of Spectrums

3 Data points (pandoc-4, duckling-4cfe88ea & duckling-ea8a4f6d) do not have faulty expressions covered by a failing test, due to code-generation (duckling) and the test-suite utilizing binaries (pandoc). Two-thirds of expressions are AST-leaves, whereas about half the faults are AST-leaves. Almost all faulty expressions have a type, but identifiers are rare.

Existing SBFL-Formulas. Table 6 shows the Top50 results when applying existing formulas and sorting the statements by their resulting score. We focus on Top50, as Top10 struggled with expression-level granularity and Top100 showed the same trends at a bigger scale. For readability, we reduced table 6 to the best performing formulas.

Ochiai is the formula that performs best with our data, followed by DStar. Ochiai is the only formula with a median Top50 above zero, implying that the other formulas have not found faults for more than half of the data points. We expect that Ochiai performs best as it applies the square root in its denominator, which scales better for large number of expressions and tests. Ochiai, DStar, and Optimal also do not use n_{t_p} (number of total passing tests), which is relatively high for most programs and disproportionate to the number of failing tests.

The best average scores are achieved by the strong performance of some formulas on pandoc-6 and HLS-afac9b18. Our educated guess is that pandoc-6 has a large number of failing tests that exactly distinguish the faulty from the correct cases. HLS-afac9b18 has a much more favorable ratio of faulty expressions to expressions, and the newly added tests primarily invoke the affected faulty statements. Thus, these two data points play into the strengths of formulas due to their test quality.

The data points duckling-4cfe88ea, duckling-ea8a4f6d, pandoc-4 and pandoc-3be256efb did not result in Top50 for any of the existing formulas. Again, we suggest that this is mostly due to the test suite and its attributes highlighted in the previous subsection. Without faulty expressions that are covered by failing tests, most formulas result in a suspiciousness of 0. Furthermore, formulas that include *passing tests* also struggle with the duckling data point, since most expressions are covered by only one or a few passing tests.

Table 6. Formula Top50 Results

Program	Faults	Tarantula	Ochiai	DStar 3	OptimalP
hls-2	15	2	2	2	2
hls-afac9b18	17	17	17	17	17
duckling-4cfe88ea	4	0	0	0	0
duckling-328e59eb	26	1	1	4	0
duckling-ea8a4f6d	5	0	0	0	0
duckling-28ddc3bf	5	0	0	0	0
pandoc-4	12	0	0	0	0
pandoc-5	8	8	8	0	0
pandoc-6	39	0	21	21	25
pandoc-7	72	3	3	3	0
pandoc-3be256efb	6	0	0	0	0
mean	17	2.8	4.7	4.3	4
median	12	0	1	0	0

These few tests are *rich* as they contain multiple examples, but do not take advantage of the considered formulas.

The overall applicability of the formulas is quite high. The small data points of HLS are especially well predictable with formulas, motivating applications for script-sized programs. For `duckling`, organizing tests into a corpus in combination with code generation makes formulas unsuitable.

RQ1.B: Existing SBFL Formulas

`Ochiai` and `DStar` produce the best `Top50` results with an average of 4.7 and 4.3 errors correctly reported in the first 50 expressions. All formulas struggle with `duckling` and `pandoc-4`, due to the faulty expressions not being touched by failing tests: A challenge to all spectrum-based methods, and not specific to the functional context.

Applicability of Rules and Correlations. To investigate the correlation, we applied the Pearson correlation coefficient after combining the spectrums across projects, shown in fig. 11.

Some correlations verify our assumptions that we considered trivial, e.g. that type lengths correlate with the number of subtypes. For most type-based rules this correlation is not statistically significant, but more complex types tend to be *longer*, have higher arity and order, and result more function application. A second block we see are SBFT formulas from literature with Ochiai, Tarantula, DStar, and OptimalP. This is mathematically plausible, as they are proportional to n_{ef}, the number of failing tests for this expression, in their formulas (see table 7). Most rules do not have a significant correlation with each other, and, except for the two blocks, there are no other visible trends. Although this seems underwhelming, we want to stress that most rules do not correlate. For example, `rTFail` and `rTFailFreq` do not correlate significantly

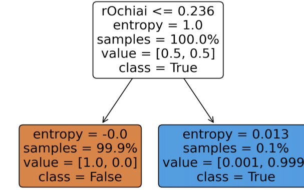


Figure 7. Decision Tree for Pandoc-5 (scaled features, `classic`-rules)

within our data, implying that the execution frequency is not directly proportional to the number of tests (analogue to `rTPass` and `rTPassFreq`). This finding motivates us to investigate formulas focusing on frequency, as they seem more distinct from test failures than expected.

In general, the lack of correlation proofs some *trivial* assumptions wrong, motivating further research and adjustments to existing formulas. We expect imperative programs to have similar patterns, but they can only be found as clear in functional programs. Inferred type information at the expression level is uncommon in other paradigms, and investigating correlations between types, constraints, arity and faults is out of reach for most imperative languages.

RQ1.D: Rule Correlations

Most rules do not correlate according to the Pearson coefficient. Type rules and popular SBFL formulas form (mostly non-significant) trends within the correlations.

Attributes of SBFL Models

Logistic & Linear Regression. In both logistic and linear regression for both scaled and unscaled features, the resulting weights result in significant variance, indicating overfitting. For example, many type rules differ in logistic regression polarity despite being correlated (see RQ1.D).

Decision Trees. Decision trees required a class-balanced fitting using an entropy measure to produce sufficient results. A visible trend is the reproduction of the SBFL formula rankings as in fig. 7. Given the effectiveness of Ochiai, as observed in RQ1.B, this is an understandable result.

For the larger programs (`pandoc-6` & `pandoc-7`), the trees often resulted in configurations that lean left or right with single expression branches. Tree pruning could not address this form of overfitting, as the resulting pruned trees remain with a high entropy.

Explainable conditions, such as *if it is a leaf, use Ochiai; otherwise, Tarantula*, were unfortunately not observed. Trees that were striking to the authors are those that use one of the well-performing formulas (e.g., `DStar`), as root of the tree, and then use a niche rule such as `rHamming`, `rRogot1` or `rNumGoldFails`, which apply to very few locations.

This shortcoming of decision trees is known and motivates the use of a random forest ensemble.

Genetic Algorithms. A key observation is that genetic algorithms (GAs) faced convergence challenges with specific programs: `pandoc-4`, `duckling-4cfe88ea`, `duckling-ea8a4f6d`, and `duckling-28ddc3bf`, exhausting the maximum number of generations allocated without achieving early termination. The non-convergence co-occurs with the absence of *touched* faults. Our educated guess is that ⓐ it is hard for randomly generated weights (that is, the initial population) to produce any correct ranking, and ⓑ for the *untouched* faults the individuals who classify faults are uniquely picking single attributes and the combination skews the weights again. Individuals that rank faults are *fragile*, and mutation and combination lose beneficial attributes, stopping the genetic search to stagnation.

RQ2.A: Development of SBFL Models
Most models struggled with forms of overfitting. Linear and logistic regression, as well as decision trees, struggled with the sparse data. Genetic algorithms face issues converging for programs with untouched faults.

Generalizability of SBFL Models

Classifiers. When investigating the classifiers (decision trees, random forests, and logistic regression), an early finding was that all three generalize better on scaled features. An overview of the transfer performance of the classifiers is shown in fig. 8. We see the trends in which classifiers are grouped according to their false and true positives. Logistic regression produces many true positives and false positives (≈90% false positives). Put in perspective, for many data points, a logistic regressor will give 100 faulty candidates, of which nine will be true faults. Although this is likely frustrating for developers, it can be suitable for tooling (see section 5.1).

The best performance with good precision was achieved by random forests using only SBFL formulas. On average, an ensemble of formula-based decision trees reports five faults, of which ≈2.5 will be true faults. This is a convincing rate for actual usage, given that the reported numbers are averages; for many programs, random forests (and decision trees) were not reporting faults as they were not certain enough. This leads to a low number of true positives, but upon author inspection, most of the actually suggested faults were either true faults or reasonably close.

Throughout the configurations, the classic SBFL formulas performed best in all classifiers. This is due to their good performance on data points with high faults for which the original formulas also performed well (`pandoc-7`). The `all-rules` and `cherries` find fewer faults and produce more false positives, but are better at predicting faults of the most challenging data points `pandoc-4`, `duckling-4cfe88ea` and `duckling-ea8a4f6d`. Depending on the goals, the logistic regression with cherry-configuration is able to predict faults that were not touched by failing tests, at the cost of a high noise ratio.

Regressors. Across the board, the regressors performed better on the unscaled data and primarily produced good `Top50`-scores on the data points with faults covered by failing tests. Due to poor performance, we present only examples of regressors when compared to data points that have locations touched by failing tests. Most regressors performed worse than existing formulas, with the exception of genetic algorithms seen in fig. 9 (and `Top10` in appendix fig. 12). We see that especially for the `Top10` genetic search produces much better averages than the formulas.

We stress that the averaged results only indicate the most fruitful configuration - the results varied greatly from regressor to regressor and per target data point. Thus, we want to highlight two types of well-formed searches in fig. 10. The orange bars indicate the achieved `Top50`-score, while the blue frame indicates the maximum possible faults.

Figure 10a show the results when using the weights originating from the genetic search over `HLS-2` using classic formulas. We observe that this configuration is well suited for a few programs and poor for others, but tops the individual formulas in mean-`Top50`. In general, we noticed that the *small* programs from HLS produced some of the best regressors, probably because the smaller number of entries resulted in smaller weights less prone to overfitting. Figure 10b are the results retrieved from fitting `original` rules (i.e., only rules novel from this work) on `duckling-28ddc3bf`. The resulting weights produce `Top50` suggestions for all data points except `pandoc-6`. This model has broad generalizability across the investigated programs and is one of the drivers of the good median metrics of search-based `Top50` results.

When looking for such individual results, we saw similar trends (uneven and even distributions of predictions) across all regressors, with genetic search producing the most visible trends due to the best predictions.

The best results were achieved for data points without faults executed by failing tests in which three configurations with a `Top50` of 1, when fitting MLPs on `pandoc-4`, `pandoc-5` and `pandoc-3be256efb` with the original rules. Such small variations are in the realm of expected randomness and might not be worth further investigation.

RQ2.B: Applicability of Models
Classifiers performed better for scaled rules, whereas regressors had difficulties with unscaled features. Logistic regression produces high recall, but suffers from false positives. Random forests produced a good ratio of true to false positives, but did not have a high recall. Of the regressors, only genetic search beat the original formulas in average and median, especially in `Top10`.

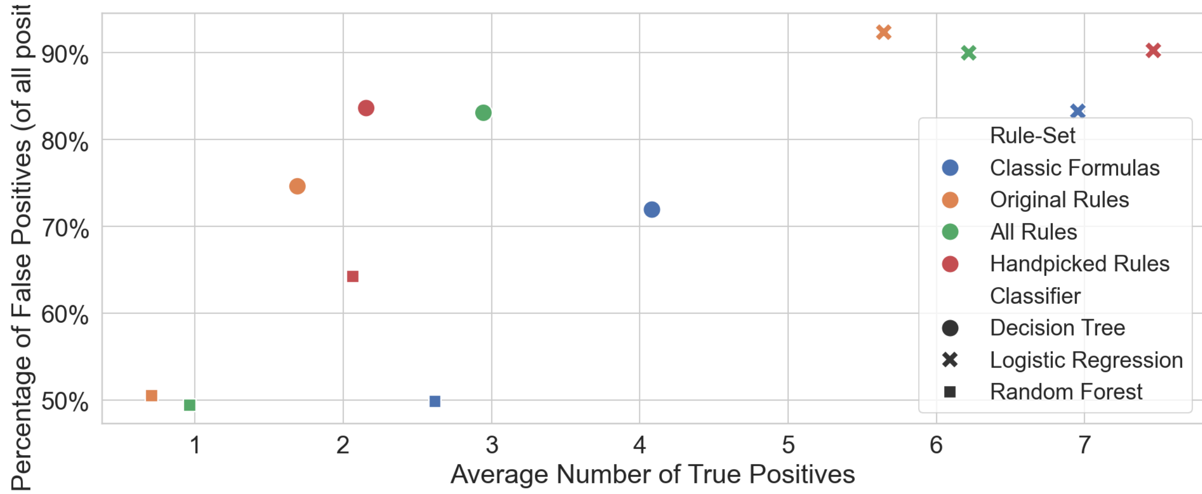


Figure 8. Transfer Performance of Classifiers

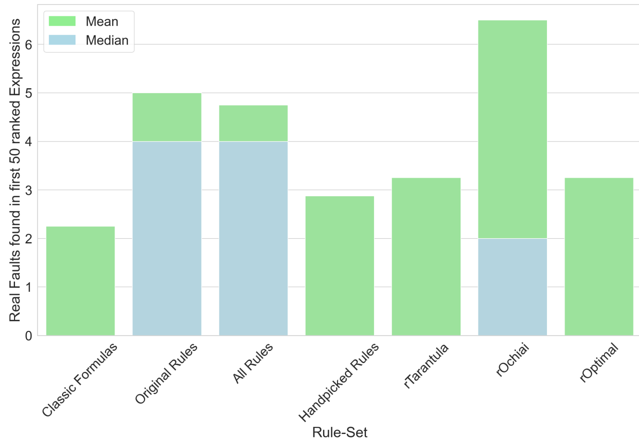
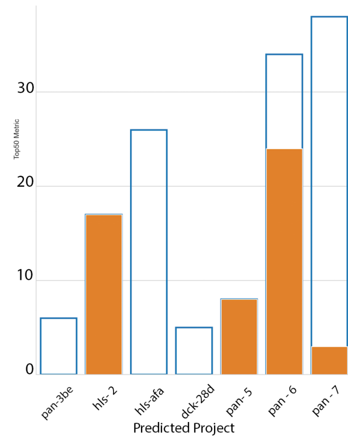
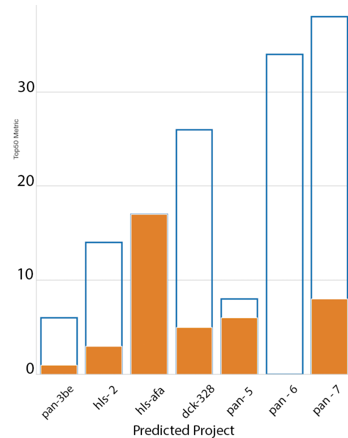


Figure 9. Averaged `Top50`-score for genetic search



(a) Top50 from `HLS-2` with `original` rules



(b) Top50 from `duckling-28ddc3bf` with `original` rules

Figure 10. Example performance of promising search models

5 Discussion

Quality of formulas. In general, the existing SBFL formulas performed well to the point that they might be used in development. The achieved rankings beat some of the existing research in Java, when the difference in granularity is taken into account: most research focuses on statements or blocks, but even the expression level seems reasonable. As formulas do not require training data and are easily applicable, they form attractive targets for Haskell tooling, e.g., suggesting points of interest on a failing PR or highlighting code of failing test runs. It might be possible to adapt existing formulas to Haskell by introducing the frequency of executions instead of binary coverage through tests. Another way to get a better result is the reduction of a spectrum, possibly through filtering for AST properties or types.

However, the results of this work also show that there are unique problems with programs whose faults are not (directly) executed by failing tests. For such programs and maybe other tasks (defect prediction, test generation), novel rules based on types or AST structure can prove successful. With functional programming often used for domain-specific languages or code generation, we expect faults of this kind to be more prominent than in imperative programs. We suggest

that ensemble-style classifiers are used to utilize the best of both worlds. For most bugs, the existing formulas seem sufficient (or, one of them is), while unique features might play a role heavily dependent on the programs structure. Classifier fitted over multiple projects, also including *bug-free* ones, are a promising next step.

Project & Test Structure. One recurring consideration throughout all results was the strong dependency on the project structure and the tests.

Duckling's approach of unifying tests into a corpus of examples makes it easy for contributors and allows for a smoother execution against the generated code, while posing significant challenges for fault localization. Similarly, many contributors (or users) to Pandoc report bugs by providing examples of failing documents that are translated into a system-level regression test. This is very economical for the maintainers, but our results show that pandoc programs with unit-level tests (`pandoc-6` & `pandoc-7`) were the most approachable for all algorithms and formulas. On the other hand, the HLS data points make use of a great degree of modularity; this is already visible, with both programs being plugins. This separation already leads to drastically smaller spectrums, and even more complicated issues (`hls-afac9b18` deleting lines on usage with other plugins) were translatable into side-effect-free unit tests. We understand that not every project can be modular to this extent, but, especially given the size, number of contributors, and changes in Pandoc and Duckling, fault localization can pay off [5].

Closing our thoughts, we would like to stress that functional programming is precisely the domain where excellent modularity can be achieved. The greater the modularity, the greater the applicability of tooling such as SBFL. For projects that have a suitable test suite, even simple SBFL formulas have immediate payoff.

5.1 Future Work

IDE integration. One future path would be to look at the integration of spectrum-based fault localization into IDE tools such as HLS, enabling users to get more out of their test suite than just a pass/fail. Apart from technical challenges in balancing performance and information, experiments can identify user needs when engaging with such tooling.

Innocence. One way to extend this work is to introduce the notion of *innocence*. Here, we focus on the suspiciousness of a given statement, but in a typed setting, we can *verify* certain functions. This could involve functions that are *verified* using tools such as SmallCheck, where we test every possible invocation of a function of type, e.g. `Bool -> a` by applying it to both `True` and `False` and checking that the output is correct. It might be extended to other concepts, e.g. *innocent types* or *innocent modules* from user-declaration. Innocent locations can be excluded from the fault localization process.

6 Conclusion

This paper aims to extend spectrum-based fault localization for Haskell and evaluate its applicability to real-world faults. To achieve this, we implemented a Tasty ingredient that allows the generation of spectrums with expression-level granularity, including additional information on types and identifiers. Making use of the richer information, we implemented *rules* that capture the complexity of types, AST structure, or identifiers and applied them to a total of 11 real-world programs. We used the rules to investigate the attributes of the spectrums and to fit classifiers and regressors. Our exploration uncovered unique kinds of failures: faults that were not covered by failing tests. These failures structured the results into two groups: for most programs, the faults were covered by tests, and existing SBFL formulas performed well and were only outperformed by regression models that also make use of formulas as features. For the faults not touched by failing tests, models based on additional information (e.g., types or identifiers) were necessary to produce any correct prediction. However, these faults remain a challenging case and require further investigation. The contributions of this work hopefully open up a broader discussion of the applicability of SBFL for Haskell. The easy adoption through a plugin allows developers and researchers to experiment and provide information on user needs alongside a greater variety of projects. Further insights in addition to our initial investigation might also form a solid basis for new Haskell-specialized formulas. Especially, the novel type of failures requires approximation not directly based on test failures, but exploits the project structure and types.

Why you should care about SBFL. One of the big selling point of Haskell is the strong type systems and the resulting compiler feedback. But even with strong types, errors can occur (see fig. 1) and require testing. While the compiler assists the program, tools assist the programmer. Especially within the boundaries of a strong type system in a lazy language, the rich information of types and the lack of side effects allow for better localization than imperative languages could dream of. All efforts, whether from developers, fault localization tools, tests, or compilers, can go hand in hand to provide the best program quality with the least effort. Thanks to the ongoing efforts of the Haskell Language Server project, it is high time to introduce new software tooling for Haskell. We hope that the insights provided by our work will provide guidance when designing these tools.

References

[1] Leonhard Applis and Annibale Panichella. 2023. HasBugs-Handpicked Haskell Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 223–227.

[2] J. Blank and K. Deb. 2020. pymoo: Multi-Objective Optimization in Python. *IEEE Access* 8 (2020), 89497–89509.

[3] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* 229 (2013).

[4] Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A survey of algorithmic debugging. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–35.

[5] Tung Dao, Na Meng, and ThanhVu Nguyen. 2023. Triggering Modes in Spectrum-Based Multi-location Fault Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1774–1785. https://doi.org/10.1145/3611643.3613884

[6] Kalyanmoy Deb. 2001. *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons.

[7] K Deb and RB Agrawal. 1995. Simulated binary crossover for continuous search space. *Complex systems* 9, 2 (1995), 115–148.

[8] Maarten Faddegon and Olaf Chitil. 2015. Algorithmic debugging of real-world haskell programs: deriving dependencies from the cost centre stack. *ACM SIGPLAN Notices* 50, 6 (2015), 33–42.

[9] Ronald Fagin, Ravi Kumar, and Dakshinamurthi Sivakumar. 2003. Comparing top k lists. *SIAM Journal on discrete mathematics* 17, 1 (2003), 134–160.

[10] Ruanqianqian Lisa Huang, Elizaveta Pertseva, Michael Coblenz, and Sorin Lerner. [n. d.]. How do Haskell programmers debug? Plateau Workshop.

[11] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[12] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

[13] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[14] Feng Li, Meng Wang, and Dan Hao. 2022. Bridging the Gap between Different Programming Paradigms in Coverage-based Fault Localization. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*. 75–84.

[15] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 286–295.

[16] David Lo, Lingxiao Jiang, Aditya Budi, et al. 2010. Comprehensive evaluation of association measures for fault localization. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[17] Justin Lubin and Sarah E Chasins. 2021. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[18] Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. 2007. A lightweight interactive debugger for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. 13–24.

[19] Michael D McKay, Richard J Beckman, and William J Conover. 2000. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 42, 1 (2000), 55–61.

[20] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[21] Balázs Mosolygó, Norbert Vándor, Gábor Antal, and Péter Hegedűs. 2021. On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 495–499.

[22] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[23] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.

[24] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[26] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 30–39.

[27] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. 432–449.

[28] André Riboira and Rui Abreu. 2010. The gzoltar project: A graphical debugger interface. In *International Academic and Industrial Conference on Practice and Research Techniques*. Springer, 215–218.

[29] Qusay Idrees Sarhan and Árpád Beszédes. 2022. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access* 10 (2022), 10618–10639. https://doi.org/10.1109/ACCESS.2022.3144079

[30] Anish Tondwalkar, Rolph Recto, Westley Weimer, and Ranjit Jhala. 2016. Finding bugs in liquid haskell,-. (2016).

[31] Vanessa Vasconcelos and Mariza AS Bigonha. 2021. HaskellFL: A Tool for Detecting Logical Errors in Haskell. *International Journal of Computer and Systems Engineering* 15, 8 (2021), 479–493.

[32] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[33] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

Appendix

Table 7. Overview of the rules in the rules-based system.

Rules	Description
Test-type count	
rTFail & rTPass	Total number of failing tests involving this location
rPropFail & rPropPass	Number of failing QuickCheck tests involving this location
rUnitFail & rUnitPass	Number of failing unit tests involving this location
rGoldenFail & rGoldenPass	Number of failing golden tests involving this location
rOtherTestFail & rOtherTestPass	Number of other failing tests involving this location
rTFailFreq & rTPassFreq	Sums the number of evaluations in failing and passing tests involving this location.
Formulas from SBFL literature	n_{e_p}/n_{e_f} is the number of passing/failing tests the expression is involved in, while n_{t_p}/n_{t_f} is the total number of passes/fails.
rJaccard	$\frac{n_{e_f}}{n_{e_f}+n_{t_f}+n_{e_p}}$
rHamming	$n_{e_f} + n_{t_p}$
rOptimal	$\begin{cases} -1 \; if \; n_{t_f} > 0 \\ n_{t_p} \; otherwise \end{cases}$
rOptimalP	$n_{e_f} - \frac{n_{e_p}}{n_{e_p}+n_{t_p}+1}$
rTarantula	$\frac{\frac{n_{e_f}}{n_{e_f}+n_{t_f}}}{\frac{n_{e_f}}{n_{e_f}+n_{t_f}} + \frac{n_{e_p}}{n_{e_p}+n_{t_p}}}$
rOchiai	$\frac{n_{e_f}}{\sqrt{(n_{e_f}+n_{t_f})(n_{e_f}+n_{e_p})}}$
rDStar k	$\frac{(n_{e_f})^k}{n_{t_f}+n_{e_p}}$
rRogot1	$\frac{1}{2}(\frac{n_{e_f}}{2n_{e_f}+n_{t_f}+n_{e_p}} + \frac{n_{t_p}}{2n_{t_p}+n_{t_f}+n_{e_p}})$
AST structure-based rules	See table 3
Type-based formula rules	See table 4

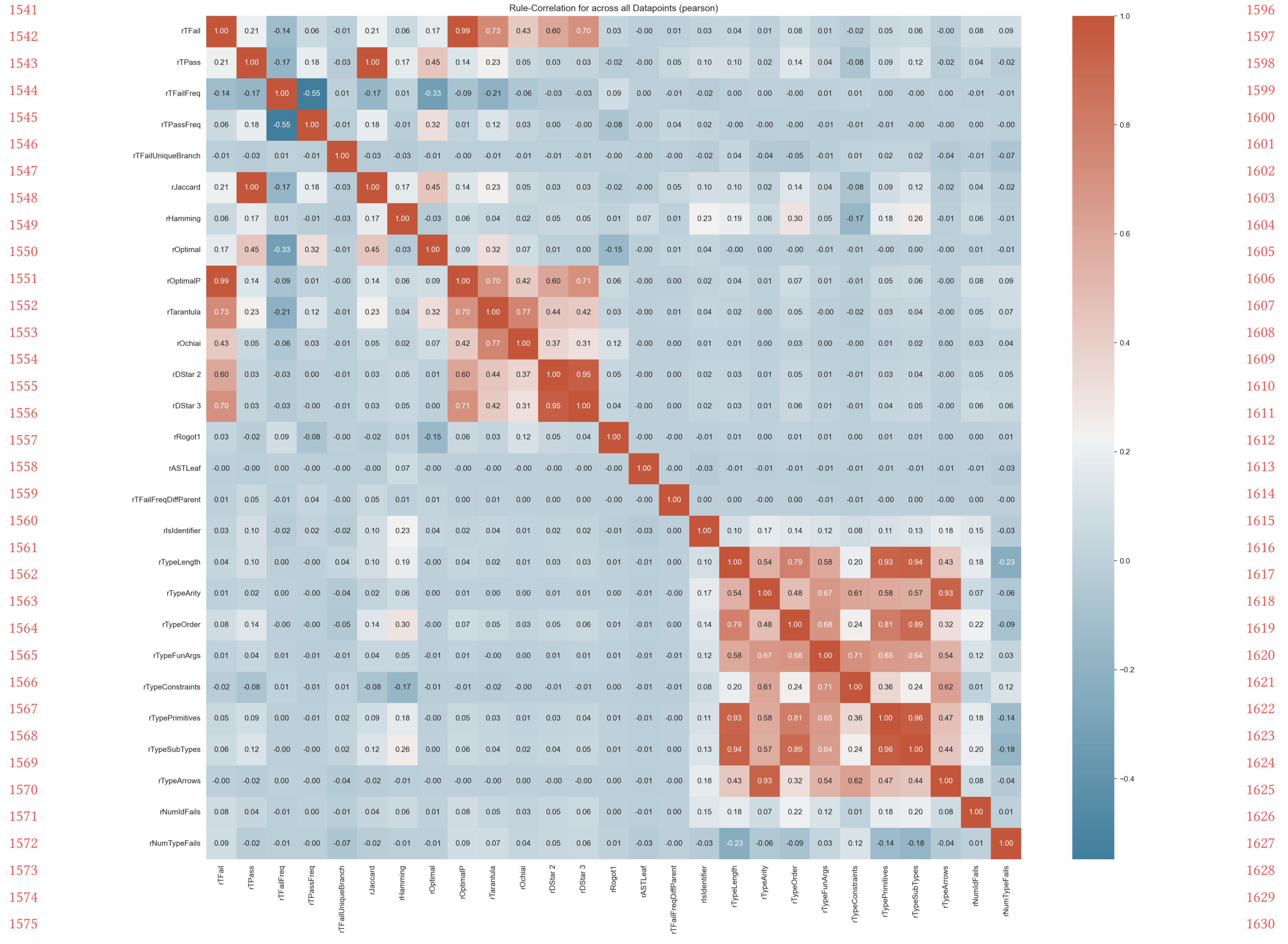


Figure 11. Pearson Correlation Matrix

Table 8. Test-coverage within gathered spectrums

Program	Expressions covered by failing Tests	Expressions untouched by failing tests	Faulty Expressions not covered by failing tests	Faulty Expressions covered by failing tests
hls-2	205	64	1	14
hls-afac	35	87	0	17
duckling-4cfe88ea	1791	297705	4	**0**
duckling-328e59eb	1165	275942	0	26
duckling-ea8a4f6d	2541	286195	5	**0**
duckling-28ddc3bf	2256	260307	0	5
pandoc-4	419	90669	12	**0**
pandoc-5	238	60410	0	8
pandoc-6	2175	57203	34	5
pandoc-7	2235	58839	34	38
pandoc-3be256efb	623	88149	0	6

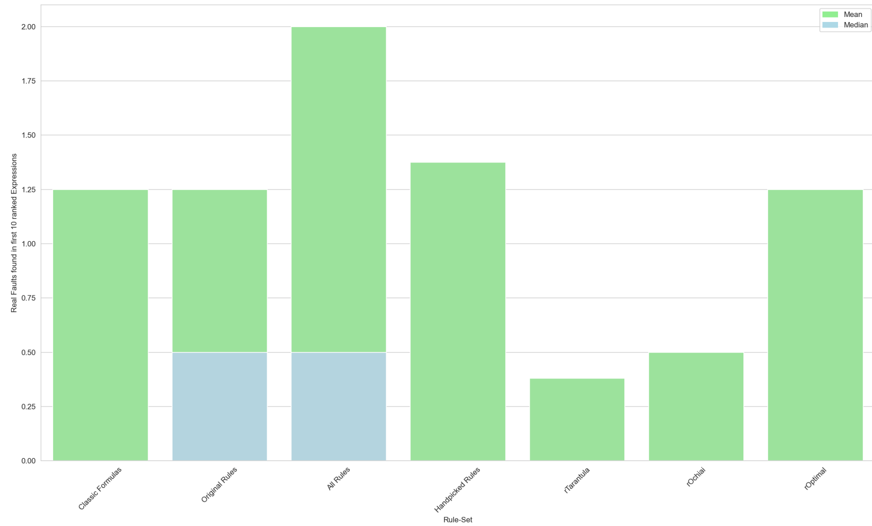


Figure 12. Averaged `Top10`-score for genetic search