Failing Faster: Performance Supplement

Jonathan Fowler Graham Hutton

December 12, 2016

1 Introduction

This report is a performance supplement to the paper *Failing Faster: Overlapping Patterns for Property-Based Testing* [3]. The report assesses the performance of generating data that satisfies a precondition, comparing the difference between using overlapping pattern matching and traditional pattern matching. The comparison is challenging as the use of overlapping patterns alters how we write a precondition and therefore simply evaluating an overlapping precondition in a traditional fashion may be unnecessarily inefficient or even not terminate. In this supplement, we explain our methodology for making the comparison and provide performance results. Overall, in the examples we use we find that overlapping patterns are never substantially detrimental to performance and frequently offer performance benefits.

A key difference in how we write overlapping preconditions is the use of bespoke sizing restrictions. The size restriction is encoded as an additional condition which is included in the precondition. It is simultaneously evaluated along with the rest of the precondition due to the use of overlapping conjunction. However, if we evaluate the precondition using traditional pattern matching the size restriction may no longer be evaluated and the narrowing evaluation may not terminate.

There are a number of ways we could fix this problem. One method would be to adopt the traditional approach to size restriction in narrowing, by imposing a restriction on the depth or number of constructors [4, 5, 1]. However doing so makes it difficult to compare overlapping and traditional methods as the distribution of generated data will not be directly related. Another method would be to move the size restriction to the first constraint in a precondition so it is always evaluated. But this is unfair to traditional evaluation, commonly a value is fully refined by the size restriction before it is even tested on the rest of the precondition.

Instead, we opt to use overlapping patterns within traditional preconditions but only for the purpose of restricting the size. In doing so, we sometimes make traditional and overlapping preconditions equivalent or near equivalent as in our typed expression example. It is important to note that the ability to tailor size restrictions to a problem is itself beneficial and can be essential to limiting backtracking, for example in red-black trees and simply typed lambda calculus.

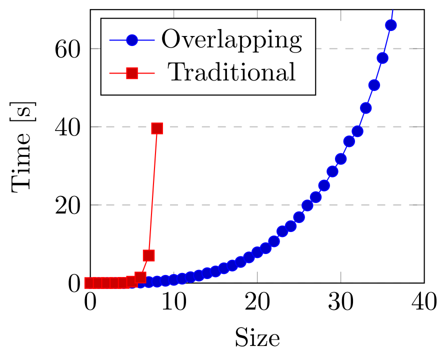

Figure 1: Time taken to randomly generate one hundred permutations

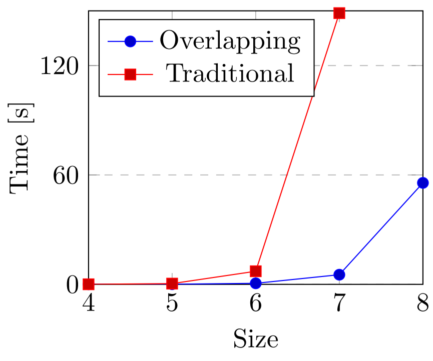

Figure 2: Time taken to enumerate all permutations of a given size

1.1 Evaluation

We use two different types of performance tests, random and enumerative generation. For the random generation of data we record the time taken to produce one hundred test cases of a given size, limiting the time to 240s and averaging over forty runs. When testing enumeration we enumerate all test cases of a given size limiting the time to 480s and taking an average of five runs. In both cases if any single run exceeds the time limit then we consider the generation to have failed at that size. The tests were all run using our prototype implementation, *OverlapCheck* [2]. In all examples the backtracking was limited to depth three.

All results reported were obtained using a quad-core Intel i5 running at 3.2GHz, with 16GB RAM, under 64-bit Ubuntu 16.04 LTS with kernel 4.4.0. The program is written without the use of parallel features and therefore the number of processor cores should have little impact on performance.

2 Permutation

For the permutation condition a direct comparison between overlapping and traditional methods is straightforward. This is because we generate permuations of a given size and therefore the precondition has an implicit size restriciton. Therefore, we can convert the overlapping condition given in our paper [3] into a non-overlapping one simply by replacing overlapping conjunction with non-overlapping conjunction at each instance.

The performance results, given in figures 1 and 2, show overlapping patterns giving a significant improvement for both random and enumerative generation of test data. When generating random data, the tool can typically generate one hundred permutations of size 32 within forty seconds using overlapping patterns, whereas without overlapping patterns it can typically only generate one hundred permutations of size 8. The generation time for traditional permutation grows exponentially at small values, in fact size 8 was the maximum size we were able

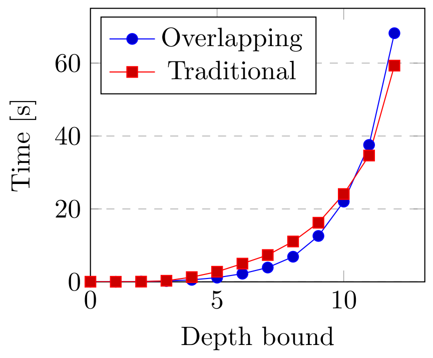

Figure 3: Time taken to generate one hundred ordered trees

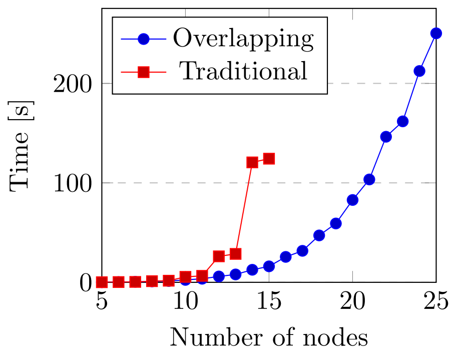

Figure 4: Time taken to enumerate ordered balanced trees

to generate test cases for within our constraints. For enumerating data the results are similar, overlapping patterns being over an order of magnitude faster at generating all size 7 permutations.

3 Ordered Trees

To generate test cases for ordered trees we use a size restriction limiting the depth of the nodes and elements. The sizing restriction is defined using overlapping patterns and we use the same restriction both in our overlapping test and our traditional test. The size restriction is similar to the paper [3] however it contains an additional limit on the depth of elements *depthElem*, which is required for the termination of the traditional version. The complete precondition, including the size restriction, is defined as follows:

$$propOrd\ n\ t = ordered\ t \wedge depthTree\ t \leqslant n \wedge depthElem\ t \leqslant 30$$

The difference between the traditional and overlapping precondition is entirely in the *ordered* constraint, in which the overlapping condition uses overlapping patterns where possible and the traditional precondition only ever used traditional pattern matching. Overlapping patterns are used in both within *depthTree*, *depthElem* and for the conjunction in the precondition above.

The performance of random generation of trees, in figure 3, shows similar performance for both the overlapping and traditional version of the condition. However, the distribution of generated trees varies greatly, with trees generated using the traditional condition being on average much smaller. For example, an average tree generated using the traditional condition with a maximum depth of ten has under three nodes, whereas an average tree produced similarly with the overlapping condition has over forty nodes.

This discrepancy is caused by failed generation attempts: if the constraint fails then it will search the immediate neighbourhood to try and find a continuation; if it does not find one then the generation fails and the generation is reset.

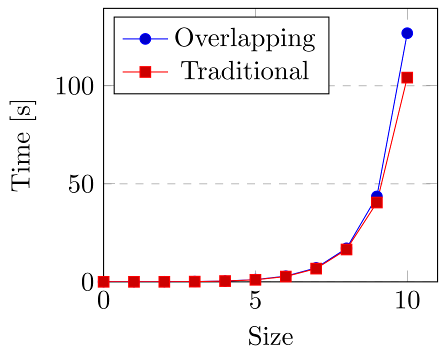

Figure 5: Generation time of one hundred random typed expressions

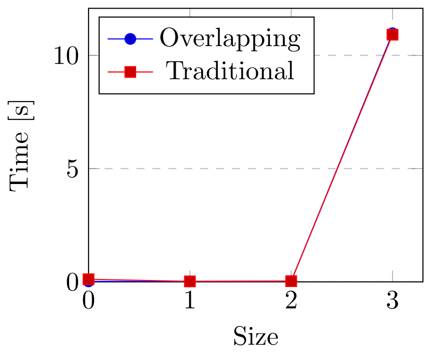

Figure 6: Time take to enumerate typed expressions of a given size

As small trees are far more likely to succeed the distribution is skewed towards these trees. This is not a problem when using the overlapping condition, as the condition *fails fast* every generation attempt will always succeed [3].

In our second performance test we sought to give a fairer performance comparison by fixing the shape of the tree. Instead of using the depth size restriction we restrain to the tree to have a set number of nodes in a balanced configuration:

$$propBalanced\ n\ t = ordered\ t \land balancedTree\ n\ t \land depthNat < s6$$

The *balancedTree* condition fixes the shape of the tree to a left biased balanced tree with n nodes. We measured the time taken to enumerate all trees of a given size. The results, in figure 4, show overlapping patterns giving a significant performance benefit. Without overlapping patterns the time taken increased exponentially and the maximum size enumerated was fifteen. With overlapping patterns the time taken also increased exponentially but at a much slower rate, the maximum size enumerated was twenty five. For comparison, there are $15,504$ size fifteen trees and $142,506$ size twenty five trees.

4 Well-Typed Expressions

For well-typed expressions we used the precondition given in the paper, shown below. Similarly to the conversion for ordered trees we only replace the overlapping patterns within *hastype* for the traditional version.

$$propExpr\ n\ e\ t = hastype\ e\ t \land depthExpr\ t \leqslant n$$

In this case there is little performance difference between the traditional and overlapping version. This is because the clauses of *hastype* are comprised of disjoint conditions. When generating random test cases the use of overlapping patterns results in a minor decrease in performance, with size ten generation taking around twenty percent longer (figure 5). This is due to the overlapping

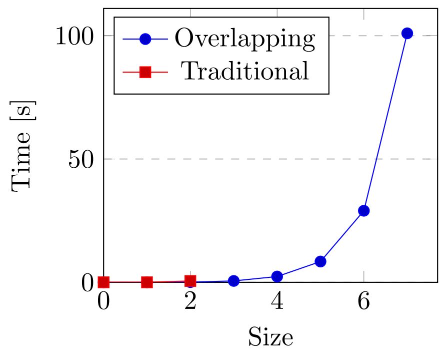

Figure 7: Time taken to generate one hundred random red-black trees

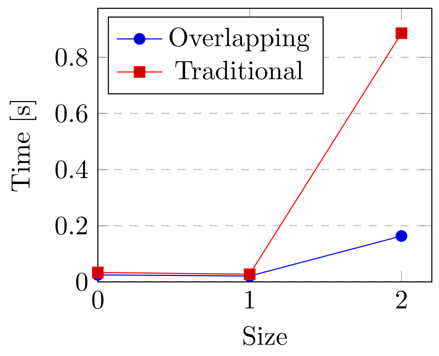

Figure 8: Time taken to enumerate red-black trees of a given size

version having to traverse all the conditions each narrowing step, even though in this case it is not beneficial.

There is no siginificant difference in performance when enumerating test cases, as shown in figure 6. The maximum enumerated size for both was depth three expressions for which they both took around eleven seconds.

5 Red-Black Trees

Finally, we consider red-black trees. As we have already tested the performance of generating ordered data we generate coloured trees, with no elements, that satisfy the red-black criteria. To that end we use the following condition:

$$\mathit{redBlack}\ n\ t = \mathit{rootBlack}\ t \wedge \mathit{black}\ n\ t \wedge \mathit{red}\ t$$

The condition *rootBlack* asserts the first node is coloured black, the condition *black* n that each path in the tree contains n black nodes and the condition *red* that no red node has a red child. The traditional version requires a size restraint, $\mathit{depth}\ t \leqslant 2 * n$, which is added to the end with overlapping conjunction.

Overlapping patterns improve performance significantly. The biggest difference is seen in random generation of red-black trees, in figure 7, where the generation was completed up to size seven with overlapping patterns but only up to two without. Although without overlapping patterns the generation of one hundred size two trees was completed in under a second the size three trees were not generated within the constraint of 240s. Further analysis shows that generating a single tree of size 3 takes an average of around ten seconds. For comparision, a size 3 tree has between 7 and 31 nodes, whereas a size 7 tree, for which generation was completed with overlapping patterns, has a minimum of 255 nodes.

There is a less pronounced performance improvement for enumerating trees, as show in figure 8. For both the maximum enumeration size was two with overlapping patterns completing around five times quicker. Neither completed the enumeration at size three within 480s.

References

[1] Koen Claessen, Jonas Duregård, and Michał H Pałka. Generating Constrained Random Data with Uniform Distribution. In *International Symposium on Functional and Logic Programming*, 2014.

[2] Jonathan Fowler. The OverlapCheck system for property-based testing. `https://github.com/JonFowler/OverlapCheck`, 2016.

[3] Jonathan Fowler and Graham Hutton. Failing Faster: Overlapping Patterns for Property-Based Testing. In *19th International Symposium on the Practial Aspects of Declarative Langugaes*, 2017.

[4] Matthew Naylor and Colin Runciman. Finding Inputs that Reach a Target Expression. In *International Conference on Source Code Analysis and Manipulation*, 2007.

[5] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck Automatic Exhaustive Testing for Small Values. In *Symposium on Haskell*, 2008.