Failing Faster: Performance Supplement

Jonathan Fowler   Graham Hutton

November 16, 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 only using traditional pattern matching. The comparison is challenging as the use of overlapping patterns alters how we write a precondition, converting overlapping functions to their traditional counterparts in a precondition may result in a non-terminating data generator. 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 detrimental to performance and frequently offer substantial performance benefits.

The key difference in how we write overlapping preconditions is the use of bespoke sizing constraints. With overlapping patterns we can add a tailored size restriction to a precondition using conjunction, as both sides of a conjunction are evaluated simultaneously the size restriction will be enforced. However if we convert to non-overlapping traditional constraints the size restriction will no longer be evaluated and a 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 when evaluating traditional preconditions, either a restriction on the depth or number of constructors [4, 5, 1]. However doing so makes it difficult to compare overlapping and traditional preconditions as the distribution of generated data will have no direct comparison. Another method would be to move the size restriction to the first constraint in a precondition, so it is always evaluated. However this is unfair to traditional preconditions, 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, that is within the sizing restriction and for the conjunction adding the sizing restriction to the rest of the precondition. 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 therefore that the ability to tailor size restrictions to

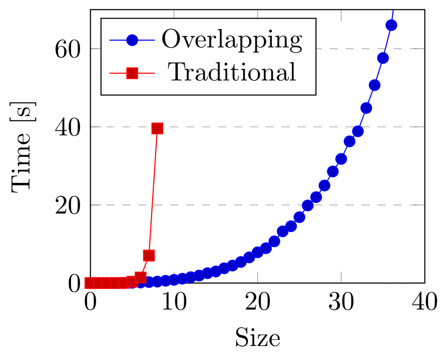

Figure 1: Time taken to randomly generate one hundred permutations

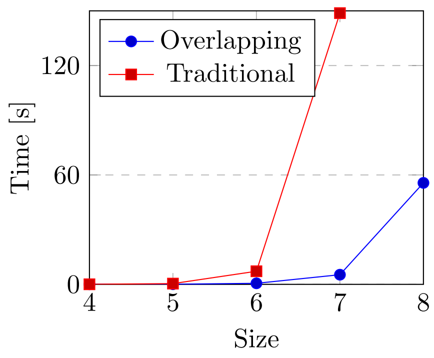

Figure 2: Time taken to enumerate all permutations of a given size

a problem is itself beneficial and can be essential to limiting backtracking, for example in red-black trees and simply typed lambda calculus.

1.1 Evaluation

We use two performance 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 proto-type implementation, *overlapcheck* [2]. In all examples the backtracking was limited to depth three.

All results reported were obtained using a quad-core Intel i5 clocked at 3.1GHz, with 8GB RAM, running 64-bit Ubuntu 14.04 LTS with kernel 3.13.0. The measured performance should not have been significantly affected by the number of processor cores as the tool is single-threaded.

TODO: *check details*

2 Permutation

For the permutation condition a direct comparison is easy. A size restriction is implicit in the precondition, the size of the list and the maximum value of an element are given by the size of the permutation. Therefore we can convert the overlapping condition into a non-overlapping one simply by replacing overlapping conjunction with non-overlapping conjunction everywhere.

The performance results, displayed 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

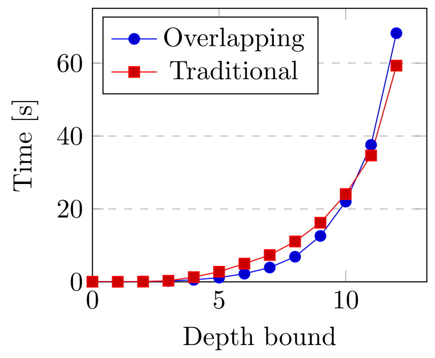

Figure 3: Time taken to generate one hundred ordered trees

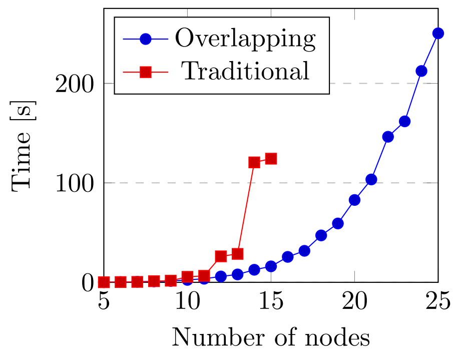

Figure 4: Time taken to enumerate ordered balanced trees

one hundred permutations of size 32 within forty seconds using overlapping patterns whereas without overlapping patterns it typically can only generate one hundred size eight permutations. The generation time increases exponential for traditional permutation, in fact size eight was the maximum size we were able to generate test cases for within our constraints. For enumerating data there is a similar story, overlapping patterns is over a magnitude faster at generating all size seven 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 one in 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 in *depthTree*, *depthElem* and for the conjunction in the precondition above.

The performance of random generation of trees, figure 3, shows similar performance using both the overlapping and traditional 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 max depth of ten has under

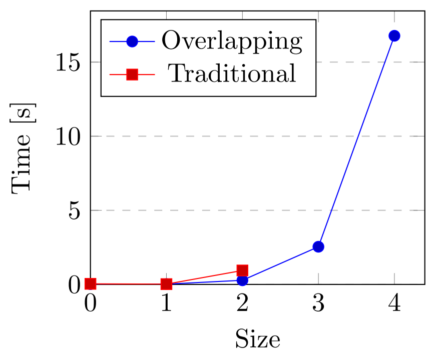

Figure 5: Time taken to generate one hundred random red-black trees

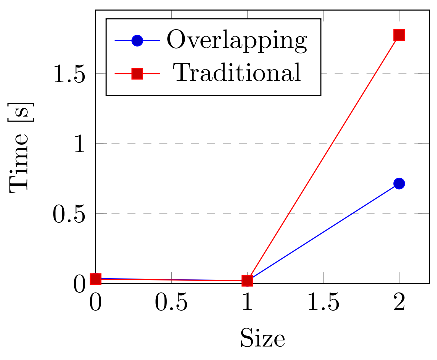

Figure 6: Time taken to enumerate red-black trees of a given size

three nodes whereas an average tree of 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. 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.

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:

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

The *balancedTree* condition fixes the shape of the tree to the left biased balanced tree with n nodes. We measured the time taken to enumerate all trees of a given size. The results, figure 3, 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 Red-Black Trees

Finally we consider red-black trees. As we have already tested the performance of generating ordered data we focus on generating coloured trees that satisfy the red-black criteria. To that end we use the following condition:

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

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. Similarly to permutation the sizing restriction is given naturally by the *black* condition and therefore we use do not need to use any overlapping patterns in the traditional version.

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.