AutoBench

Comparing the Time Performance of Haskell Programs

Martin A. T. Handley
School of Computer Science
University of Nottingham, UK

Graham Hutton
School of Computer Science
University of Nottingham, UK

Abstract

Two fundamental goals in programming are correctness (producing the right results) and efficiency (using as few resources as possible). Property-based testing tools such as QuickCheck provide a lightweight means to check the correctness of Haskell programs, but what about their efficiency? In this article, we show how QuickCheck can be combined with the Criterion benchmarking library to give a lightweight means to compare the time performance of Haskell programs. We present the design and implementation of the *AutoBench* system, demonstrate its utility with a number of case studies, and find that many QuickCheck correctness properties are also efficiency improvements.

CCS Concepts • **General and reference → Performance**; • **Software and its engineering → Functional languages**;

Keywords Time performance, optimisation, benchmarking

ACM Reference Format:
Martin A. T. Handley and Graham Hutton. 2018. AutoBench: Comparing the Time Performance of Haskell Programs. In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27–28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3242744.3242749

1 Introduction

Correctness and efficiency are two fundamental goals in programming: we want our programs to produce the right results, and to do so using as few resources as possible.

In recent years, property-based testing has become a popular method for checking correctness, whereby conjectures about a program are expressed as executable specifications known as properties. To give high assurance that properties hold in general, they are tested on a large number of inputs that are often generated automatically.

This approach was popularised by QuickCheck [7], a lightweight tool that aids Haskell programmers in formulating and testing properties of their programs. Since its introduction, QuickCheck has been re-implemented for a wide range of programming languages, its original implementation has been extended to handle impure functions [8], and it has led to a growing body of research [3, 16, 36] and industrial interest [1, 22]. These successes show that property-based testing is a useful method for checking program correctness. But what about program efficiency?

This article is founded on the following simple observation: many of the correctness properties that are tested using systems such as QuickCheck are also *time* efficiency improvements. For example, consider the familiar monoid properties for the list append operator in Haskell, which can all be tested automatically using QuickCheck:

$$\begin{aligned} xs +\!\!+ [\,] &= xs \\ [\,] +\!\!+ xs &= xs \\ (xs +\!\!+ ys) +\!\!+ zs &= xs +\!\!+ (ys +\!\!+ zs) \end{aligned}$$

Intuitively, each of these correctness properties is also a time efficiency improvement in the left-to-right direction, which we denote using the $\trianglerighteq$ symbol as follows:

$$\begin{aligned} xs +\!\!+ [\,] &\trianglerighteq xs \\ [\,] +\!\!+ xs &\trianglerighteq xs \\ (xs +\!\!+ ys) +\!\!+ zs &\trianglerighteq xs +\!\!+ (ys +\!\!+ zs) \end{aligned}$$

For example, the associativity property of append is intuitively an improvement because the left side traverses xs twice, whereas the right side only traverses xs once. However, formally verifying improvements for lazy languages like Haskell is challenging, and usually requires the use of specialised techniques such as improvement theory [27].

In this article, we show how improvement properties can be put into the hands of ordinary Haskell users, by combining QuickCheck with Criterion [35] — a popular benchmarking library — to give a lightweight, automatic means to compare the time performance of Haskell programs. More specifically, the article makes the following contributions:

- We present the design and implementation of the *AutoBench* system, which uses Criterion to compare the

Haskell '18, September 27–28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3242749

runtimes of programs executed on inputs of increasing size generated using QuickCheck (section 3);
- We develop a method for estimating the time complexity of a program by analysing its runtime measurements using ridge regression [21] (section 3.3.2);
- We demonstrate the practical applicability of our system by presenting a number of case studies taken from the Haskell programming literature (section 4).

Surprisingly, this appears to be the first time that QuickCheck and Criterion have been combined, despite this being a natural idea. AutoBench comprises around 7.5K lines of new Haskell code, and is freely available on GitHub [20] along with instructions for installing and using the system.

This article is aimed at readers who are familiar with the basics of functional programming in Haskell, but we do not assume specialised knowledge of QuickCheck, Criterion, or statistical analysis. In particular, we introduce the necessary background on these topics where necessary.

2 Example

We start with a simple example to demonstrate the basic functionality of our system. Consider the following recursive function that reverses a list of integers:

```
slowRev :: [Int] → [Int]
slowRev [ ]       = [ ]
slowRev (x : xs) = slowRev xs ++ [x]
```

Although this definition is simple, it suffers from a poor, quadratic time performance due to its repeated use of the append operator, which has linear runtime in the length of its first argument. However, it is straightforward to define a more efficient version using an accumulator [39], which, although less clear, has linear runtime:

```
fastRev :: [Int] → [Int]
fastRev xs = go xs [ ]
  where
    go [ ]       ys = ys
    go (x : xs) ys = go xs (x : ys)
```

An easy way to check that *slowRev* and *fastRev* give the same results is to use QuickCheck [7]. In particular, given a testable property, the *quickCheck* function will generate a number of random test cases and check the property is satisfied in all cases. Here we can use a simple predicate that compares the results of both functions

```
prop :: [Int] → Bool
prop xs = slowRev xs ≡ fastRev xs
```

and, as expected, the property satisfies all tests:

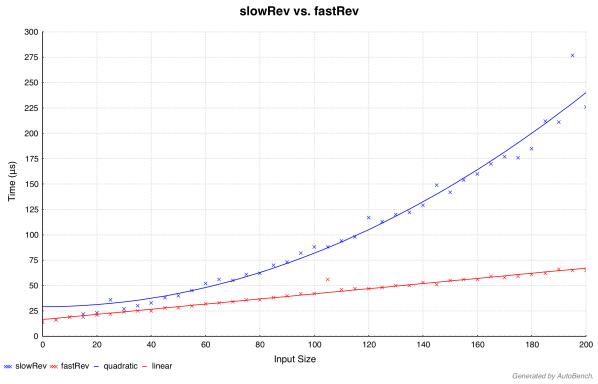
```
> quickCheck prop
+++ OK, passed 100 tests.
```

From a correctness point of view, QuickCheck gives us confidence that *slowRev* and *fastRev* give the same results, but what about their relative efficiencies?

An easy way to compare their time performance is to use AutoBench. Given two or more programs of the same type, the *quickBench* function will generate random inputs of increasing size and measure the runtimes of each program executed on these inputs. The measurements are then analysed to produce time performance results. In the case of our example, the system is invoked by supplying the two functions and their identifiers for display purposes:

```
> quickBench [slowRev, fastRev] ["slowRev", "fastRev"]
```

Two results are produced. The first is a graph of runtime measurements, which is saved to the user's working directory as a portable network graphics (png) file:



The graph illustrates the comparative runtimes of *slowRev* and *fastRev* for each input size. Both sets of measurements also have a line of best fit, which is calculated using regression analysis and estimates the time complexity of the corresponding function. In this case, the graph's legend confirms that *slowRev*'s line of best fit is a quadratic equation and *fastRev*'s line of best fit is linear. While in some cases one may be able to estimate such results 'by eye', in general this is unreliable. For example, it is difficult to determine the exact degree of a polynomial by simply looking at its graph.

Remark Readers are advised to download the article in electronic form so that graphs of runtime measurements can be readily enlarged to show their detail. A larger version of each time performance figure is also available as supplementary material on the ACM website.

The second result produced is a table, output to the user's console, which displays the value of each runtime measurement and the precise equations of the lines of best fit:

Input Size	0	5	10	15	20	...
slowRev (µs)	14.00	16.00	19.00	22.00	23.00	...
fastRev (µs)	14.00	16.00	18.00	19.00	21.00	...
slowRev	$y = 2.91e{-}5 + 2.28e{-}11x + 5.28e{-}9x^2$					
fastRev	$y = 1.65e{-}5 + 2.53e{-}7x$					
Optimisation	*slowRev* ⩥ *fastRev* (0.95)					

Moreover, at the bottom of the table is an optimisation, written $slowRev \trianglerighteq fastRev$, which is derived from the combination of (i) AutoBench's performance results, which show that for almost all test cases *fastRev* is more efficient than *slowRev*, i.e. $slowRev \trianglerighteq fastRev$; and (ii) QuickCheck's correctness results, which show that for all test cases *fastRev* is denotationally equal to *slowRev*, i.e. $slowRev = fastRev$. The decimal 0.95 that appears in parentheses after the optimisation indicates that it is valid for 95% of test cases.

In conclusion, AutoBench suggests that replacing *slowRev* with *fastRev* will not change any results, but will improve time performance. Furthermore, considering each function's derived time complexity, it suggests that $slowRev \trianglerighteq fastRev$ indeed gives a quadratic to linear time speedup. Thus, overall, knowing that one program is more efficient than another, and by how much, allows the user to make an informed choice between two denotationally equal programs.

The *quickBench* function is designed for analysing the time performance of programs using the Haskell interpreter GHCi, in a similar manner to how QuickCheck is typically used. Its primary goal is to generate useful results quickly. Consequently, by default, its performance results are based on *single* executions of test programs on test inputs. For this example, testing takes just a few seconds, with its primary cost being the benchmarking phase. Though this approach sacrifices precision for speed (as demonstrated by the notable 'noise' in the example graph), in practice it is often sufficient for basic exploration and testing purposes.

For more thorough and robust performance analysis, the system provides an executable called *AutoBench*. In contrast to *quickBench*, this tool makes extensive use of the Criterion [35] library to accurately measure the runtimes of *compiled* Haskell programs. Furthermore, its results are based on *many* executions of test programs on test inputs. For this example, testing takes a few minutes, with its primary cost also being the benchmarking phase. The architecture of *AutoBench* largely encapsulates that of *quickBench*. Therefore, for the remainder of the article we focus on the design, implementation, and application of *AutoBench*.

3 AutoBench

In this section, we introduce the core architecture of the AutoBench system. Figure 1 illustrates the system's three major components (data generation, benchmarking, and statistical analysis) and how they are linked together. We discuss the details of each component in a separate subsection along with the key user options it provides.

3.1 Data Generation

Given a number of programs to compare, the system must be able to generate a suitable range of inputs in order to test their time performance. To produce such inputs, our system exploits the random data generation facilities of QuickCheck.

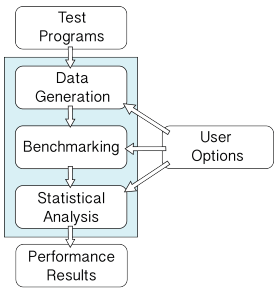


Figure 1. AutoBench's system architecture

These provide generators for all standard Haskell types, together with a rich set of combinators that aid users in defining their own generators for custom data types.

3.1.1 QuickCheck Generators

The notion of a QuickCheck random generator is based on the following simple class declaration:

```
class Arbitrary a where
  arbitrary :: Gen a
```

Intuitively, this class contains types that support the generation of arbitrary values. In turn, the type *Gen a* can be seen as a pseudo-random generator for values of type *a*.

For example, the following instance declaration in the QuickCheck library allows us to generate an arbitrary integer using a primitive $choose :: (Int, Int) \to Gen\ Int$ that randomly chooses an integer in a given interval:

```
instance Arbitrary Int where
  arbitrary = choose (−100, 100)
```

Sampling this generator then yields random integers:

```
> sample' (arbitrary :: Gen Int)
[0, 2, −4, 3, −3, −7, 10, −1, 2, −1, 15]
```

3.1.2 Sized Inputs

Any function to be tested using our system must have an *Arbitrary* instance for its argument type to allow random inputs to be generated using QuickCheck. Furthermore, users must ensure that the generator for this type incorporates a sensible notion of *size*. This is a key requirement, as comparing the time performance of programs relies on benchmarking their runtimes using different sized inputs.

The QuickCheck primitive $sized :: (Int \to Gen\ a) \to Gen\ a$ can be used to define generators that depend on a size parameter, giving users full control over how the size of a data type is realised in practice. However, in some cases the standard generators defined using this primitive are not suitable to use with AutoBench. For example, the arbitrary instance for lists is usually defined as follows:

```
instance Arbitrary a ⇒ Arbitrary [a] where
  arbitrary = listOf arbitrary
```

```
listOf :: Gen a → Gen [a]
listOf gen = sized (λn → do
  k ← choose (0, n)
  vectorOf k gen)
```

That is, an arbitrary list is generated by first generating a random number k in the interval 0 to n, where n is the given size parameter, and then generating a vector of arbitrary values of size k. For performance testing, however, we require finer control over the size of the resulting lists. In particular, rather than generating lists of *varying* size up to the given parameter n, we wish to generate lists of *precisely* this size. In this manner, as the size parameter is increased, we will obtain an even distribution of different sized inputs with which to measure runtime performance.

To this end, we define our own custom generator that produces arbitrary lists with precisely n elements:

```
instance Arbitrary a ⇒ Arbitrary [a] where
  arbitrary = sized (λn → vectorOf n arbitrary)
```

To help users avoid *Arbitrary* instances in the QuickCheck library that are incompatible with our system, we provide a library of instances that override the default versions.

3.1.3 User Options

Random inputs are generated in accordance with a given size range, which is specified by a lower bound, step size, and upper bound. To ensure there are sufficient test results to allow for meaningful statistical analysis, the size range must contain at least 20 values. For example, the default range is 0 to 200 in steps of 5, as illustrated in the example in section 2, but this can be easily modified by the user. The online repository for the paper [20] gives further details on how user options can be specified.

3.2 Benchmarking

The system measures the runtimes of programs executed on random inputs using Criterion [35], a popular Haskell benchmarking library. Two of Criterion's key benefits are its use of regression analysis and cross-validation, which respectively allows it to eliminate measurement overhead and to distinguish real data from noise generated by external processes. As such, measurements made by Criterion are much more accurate and robust than, for example, those made by operating system timing utilities.

3.2.1 Defining Benchmarks

Criterion's principle type is *Benchmarkable*, which is simply a computation that can be benchmarked by the system. A value of this type can be constructed using the *nf* function, which takes a function and an argument, and measures the time taken to fully evaluate the result of applying the function to the argument. For instance, the application of *slowRev* to an example list can be made benchmarkable as follows:

```
nf slowRev [0..200] :: Benchmarkable
```

Evaluating to full normal form ensures runtime measurements reflect the full potential cost of applying a function, as due to Haskell's laziness, computations are only evaluated as much as is required by their surrounding contexts. The standard class *NFData* comprises types that can be fully evaluated, and hence *nf* requires the result type of its argument function to be an instance of this class:

```
nf :: NFData b ⇒ (a → b) → a → Benchmarkable
```

In some situations, however, using *nf* may force undesired evaluation, such as when measuring the runtimes of programs whose outputs are, by design, produced lazily. For this reason, *Benchmarkable* computations can also be constructed using the *whnf* function, which only evaluates results to weak head normal form and requires no extra conditions:

```
whnf :: (a → b) → a → Benchmarkable
```

Finally, a *Benchmark* is given by a *Benchmarkable* computation together with a suitable description, constructed using the *bench* function as in the following example:

```
bench "slowRev, [0..200], nf"
  (nf slowRev [0..200]) :: Benchmark
```

Such a benchmark can be passed to one of Criterion's top-level functions to be executed and have its runtime measured.

3.2.2 Benchmarking with Generated Data

Each *Benchmarkable* computation defined using *nf* or *whnf* requires a function $f :: a \to b$ and an argument $x :: a$. For the purposes of our system, the function will be a test program specified by a user, and the argument will be a random test input generated using QuickCheck.

Because Haskell is lazy, in general we cannot assume that the argument x is already in normal form. This is problematic for benchmarking because measurements may then include time spent evaluating x prior to applying f. Fortunately, Criterion provides an alternative method for defining benchmarks that ensures x is fully evaluated before f is applied, using which we can define the following functions:

```
genNf     :: (Arbitrary a, NFData a, NFData b) ⇒
             (a → b) → Int → String → Benchmark
genWhnf   :: (Arbitrary a, NFData a) ⇒
             (a → b) → Int → String → Benchmark
```

For example, given a function $f :: a \to b$, a size $n :: Int$ and an identifier $s :: String$, the benchmark *genNF f n s* generates a random, fully evaluated argument $x :: a$ of size n and measures the time taken to fully evaluate the result of applying f to this argument. The function *genWhnf* acts similarly, however, the result of the application is only evaluated to weak head normal form. Note that the computation $f\ x$ is evaluated in a lazy manner as usual.

3.2.3 Compiling and Executing Benchmarks

It is preferable to use the GHC compiler rather than the GHCi interpreter when benchmarking. In particular, whereas the compiler generates optimised machine code, the interpreter executes unoptimised bytecode. As such, programs executed in the interpreter are inherently less time efficient than their compiled counterparts. Consequently, Criterion provides a number of utility functions that can be used to construct a top-level *main* action that performs benchmarking tasks in a compiled program. For example, *defaultMain* takes a list of benchmarks and executes them sequentially:

$$defaultMain :: [Benchmark] \rightarrow IO\ ()$$

To ensure testing is fully automated, AutoBench must generate, compile, and execute benchmarks on behalf of users. This is achieved using the *Hint* [19] and *Ghc* [38] packages. The former is used to interpret and validate user input files (containing, for example, test programs), and the latter is used to compile files generated by the system that contain appropriate benchmarks. Benchmarking executables are invoked using the *Process* [30] package.

3.2.4 User Options

Users can configure how benchmarks are compiled by specifying GHC flags in their user options, such as the level of optimisation required (if any). Users can also state whether results should be evaluated to normal form or to weak head normal form, which dictates whether benchmarks are generated using *genNf* or *genWhnf*.

3.3 Statistical Analysis

After benchmarking, runtime measurements are analysed by the system to give time performance results. In particular, high-level comparisons between programs are given in the form of time efficiency improvements. In addition, each program's time complexity is estimated using a custom algorithm. Finally, the runtimes and complexity estimates are graphed to provide a visual performance comparison.

As the results produced by our system are based on random testing, we also collate a number of statistics output by Criterion (such as standard deviation and variance introduced by outliers) to give users a basic overview of the quality of the benchmarking data and, by extension, the reliability of the performance results.

3.3.1 Efficiency Improvements and Optimisations

The system generates improvement results by comparing the runtimes of programs *pointwise*. That is, for each input size, it compares the runtimes of two programs to determine which program performed better. If one program performs better than another for a certain percentage of test cases, then the system concludes that it is more efficient and generates a corresponding improvement result.

For example, when comparing the time performance of the functions *slowRev* and *fastRev* in section 2, the system generated the following improvement result,

$$slowRev \gtrsim fastRev \quad (0.95)$$

which states that *fastRev* was more efficient than *slowRev* in 95% of test cases. By default, 95% of test cases must show one program to be more efficient for an improvement result to be generated. This accounts for the fact that performance may not be improved for small inputs due to start-up costs, and allows for minor anomalies in the benchmarking data.

A fundamental assumption of our system is that the programs being tested are denotationally equal. Nevertheless, a sanity check is performed by invoking QuickCheck to verify that the results of test programs are indeed the same. If this check is passed, any improvement results will be upgraded to correctness-preserving optimisations. This was also exemplified in section 2, as the results table included an optimisation rather than just an improvement:

$$slowRev \triangleright fastRev \quad (0.95)$$

3.3.2 Approximating Time Complexity

Runtime measurements are combined with size information to give sets of (x, y) data points, where x is an input size and y is the runtime of a program executed on an input of size x. The system uses each such set of data points to approximate the time complexity (linear, quadratic, logarithmic, . . .) of the corresponding program. Surprisingly, there appears to be no standard means to achieve this. Moreover, the problem itself is difficult to define precisely. Through experimentation, we have developed a technique based on the idea of computing a line of best fit for a given set of data points, and then using the equation of this line to estimate the time complexity.

In the example from section 2, the system computed the following lines of best fit for *slowRev* and *fastRev*:

$$y = 2.91 \times 10^{-5} + 2.28 \times 10^{-11}x + 5.28 \times 10^{-9}x^2$$

$$y = 1.65 \times 10^{-5} + 2.53 \times 10^{-7}x$$

Using these equations, it then estimated the time complexities as quadratic and linear, respectively. In general, therefore, the system must determine which *type* of function best fits a given data set. Surprisingly, there appears to be no standard means to achieve this either. Our technique is to consider many different types of functions and choose the one with the smallest fitting error according to a suitable measure.

Ordinary least squares To fit different types of functions to a data set, our system uses *regression analysis* [17].

Given a set of (x, y) data points comprising input sizes and runtimes, a regression *model* in our setting is a particular type of function that predicts runtimes using input sizes. For example, we might consider a linear function $\hat{y} = mx + c$, where $\hat{y}$ is the runtime predicted by the model for a given input size x. Using this idea, the *ordinary least squares* (OLS)

method estimates the unknown parameters of the model, for example m and c in the case of a linear function, such that the distance between each y-coordinate in the data set and the corresponding predicted $\hat{y}$-value is minimised:

$$\text{minimise} \sum_{i=0}^{n} (y_i - \hat{y}_i)^2$$

The data points are called the *training data*, and the expression being minimised is the *residual sum of squares* (RSS).

Overfitting When fitting regression models to data sets using the OLS method, those of higher complexity will *always* better fit training data, resulting in lower RSS values. Informally, this is because models with higher numbers of parameters are able to capture increasingly complex behaviour. For example, a polynomial equation of degree $n - 1$ can precisely fit any data set containing n unique points.

Higher complexity models can thus *overfit* training data by fitting small deviations that are not truly representative of the data's overall trend. As such, they may accurately fit the training data, but fail to reliably predict the behaviour of future observations. The size of the training data can also affect the likelihood of overfitting, as, in general, it is more difficult to separate reliable patterns from noise when models are trained using small data sets.

Overall, our system must be able to assess regression models of varying complexities, as different programs can have different time complexities. Furthermore, the benchmarking process can often be time consuming, and hence it is likely that the models will be fitted to comparatively small data sets. Therefore, the possibility of overfitting must be addressed so that comparisons made between models do not naively favour those that overfit training data.

Ridge regression One way to reduce the variance of models and prevent overfitting is to introduce bias. This is known as *bias-variance tradeoff*. Bias can be introduced by penalising the magnitude of the coefficients of model parameters when minimising RSS. This is a form of *regularization*. To achieve this, our system uses *ridge regression* [21], which places a bound on the square of the magnitude of coefficients. That is, for a model $\hat{y} = a_0 + a_1x_1 + a_2x_2 + \cdots + a_px_p$, ridge regression has the following objective function,

$$\text{minimise} \sum_{i=0}^{n} (y_i - \hat{y}_i)^2 \quad \text{subject to} \sum_{j=1}^{p} a_j^2 \leqslant t$$

where $t \geqslant 0$ is a tuning parameter that controls the amount of regularization, i.e. coefficient 'shrinkage'.

The overall effect of constraining the magnitude of coefficients is that models then place more emphasis on their 'influential' parameters. That is, the magnitudes of the coefficients of model parameters that have little effect on minimising RSS values are reduced. Thus, fitting models with superfluous parameters to a data set using this method reduces the effects of those parameters and prevents overfitting.

Fitting polynomial equations of increasing complexity to the runtimes of *slowRev* in section 2 using the OLS and ridge regression methods gives the following results, in which the lowest fitting error in each case is highlighted in bold:

Model	OLS Fitting Error	Ridge Fitting Error
Quartic	$\mathbf{6.52 \times 10^{-13}}$	3.88×10^{-10}
Cubic	1.12×10^{-12}	1.70×10^{-10}
Quadratic	8.40×10^{-12}	$\mathbf{7.21 \times 10^{-11}}$

The results in this table demonstrate the susceptibility of the OLS method to overfitting, as it favours quartic runtime for *slowRev*. In contrast, the results calculated using ridge regression show the quadratic model to be the best fitting, which is what we would expect for *slowRev* given the overall trend in its runtime measurements.

Model selection Selecting a model from a number of candidate models is known as *model selection*. This often involves assessing the accuracy of each model by calculating a fitting error, and then choosing the model with the least error.

As the overall aim is to approximate time complexity, it is good practice to assess each model's predictive performance on *unseen* data, i.e. data not in the training set. This way, time complexity estimates have a higher likelihood of being representative of inputs that are outside of the size range of test data. To achieve this with relatively small data sets, the system uses *Monte Carlo cross-validation* [40].

Cross-validation is a technique used to evaluate a model by repeatedly partitioning the initial data set into a training set T_k to train the model, and a validation set V_k for evaluation. For each iteration k of cross-validation, a fitting error is calculated by comparing the y-values of data points in the evaluation set V_k with the corresponding $\hat{y}$-values predicted by the model trained on set T_k. Errors from each iteration are combined to give a cumulative fitting error for the model.

Model selection is then performed by comparing cumulative fitting errors. By default, the system compares models using *predicted mean square error* (PMSE). The model with the lowest PMSE is chosen and its equation is used to approximate the respective program's time complexity.

In the example in section 2, runtimes were split randomly into 70% training and 30% validation data in every iteration of cross-validation and a total of 200 iterations were performed. The following results were obtained in which models are ranked by decreasing PMSE value:

	slowRev:		*fastRev*:	
Rank	Model	PMSE	Model	PMSE
1	Quadratic	$\mathbf{7.21 \times 10^{-11}}$	Linear	$\mathbf{1.59 \times 10^{-11}}$
2	Cubic	1.70×10^{-10}	$n \log_2 n$	1.06×10^{-10}
3	Quartic	3.88×10^{-10}	$\log_2^2 n$	2.17×10^{-10}
⋮	⋮	⋮	⋮	⋮

The equations of the top ranked models (in bold) were then used to approximate the time complexities of *slowRev* and *fastRev* as quadratic and linear, respectively.

3.3.3 User Options

Users can specify which types of functions should be considered by the system when approximating time complexity and how to compare them. The system currently supports functions of the following types: constant, linear, polynomial, logarithmic, polylogarithmic, and exponential.

For each type of function, a range of parameters is considered, such as degrees between 2 and 10 for polynomials, and bases 2 or 10 for logarithms. By default, all types of functions are considered. These models can be compared using a number of different fitting statistics, including R^2, adjusted R^2, predicted R^2 and predicted square error (PSE).

4 Case Studies

In this section, we demonstrate the use of the AutoBench system with three sets of examples. In each case, the programs being tested were added to a file along with QuickCheck generators to produce necessary inputs, *NFData* instance declarations to ensure test cases could be fully evaluated, and appropriate user options. Unless otherwise stated, the user options specified the size range of the inputs and configured the results of test cases to be evaluated to normal form. Test files were then passed to the AutoBench system and the time performance of the programs compared.

4.1 QuickSpec

Research on property-based testing has also introduced the notion of *property generation*. Given a number of functions and variables, *QuickSpec* [9] will generate a set of correctness properties that appear to hold for the functions based upon QuickCheck testing. This facility gives users the opportunity to gain additional knowledge about their code.

For example, given the append function (+), the empty list [ ], and variables *xs*, *ys*, and *zs*, QuickSpec will generate the following identity and associativity properties:

$$\begin{aligned} xs + [\,] &= xs \\ [\,] + xs &= xs \\ (xs + ys) + zs &= xs + (ys + zs) \end{aligned}$$

In previous work, these equational laws have been formally shown to be time improvements [27]:

$$\begin{aligned} xs + [\,] &\trianglerighteq xs \\ [\,] + xs &\trianglerighteq xs \\ (xs + ys) + zs &\trianglerighteq xs + (ys + zs) \end{aligned}$$

With this in mind, a fitting first case study for our system was to test the equational properties presented in the QuickSpec paper [9] to see which give improvement results. For example, our system indicates that all three of the above properties are correctness-preserving optimisations:

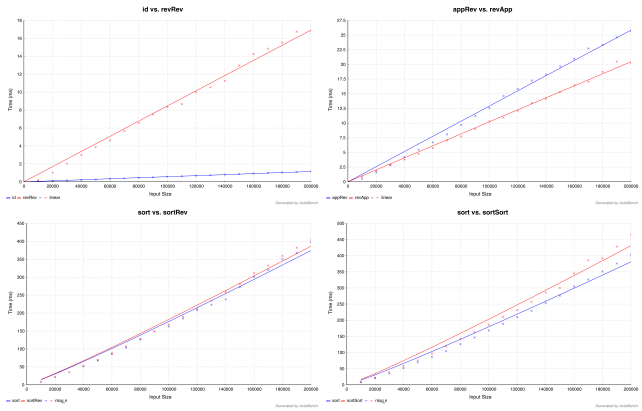


Figure 2. Graphs of results for the QuickSpec examples

$$\begin{aligned} xs + [\,] &\trianglerighteq xs \\ [\,] + xs &\trianglerighteq xs \\ (xs + ys) + zs &\trianglerighteq xs + (ys + zs) \end{aligned}$$

If the QuickSpec inputs are then extended to include the standard *reverse* and *sort* functions, a number of additional properties are generated by the system, including:

$$\begin{aligned} reverse\ (reverse\ xs) &= xs \\ reverse\ xs + reverse\ ys &= reverse\ (ys + xs) \\ sort\ (reverse\ xs) &= sort\ xs \\ sort\ (sort\ xs) &= sort\ xs \end{aligned}$$

For each equation, we compared the time performance of its left-hand side against that of its right-hand side on lists of random integers using two corresponding test programs. For example, the second *reverse* property above was tested by comparing *appRev* and *revApp*, defined as follows:

$$\begin{aligned} &appRev :: ([Int], [Int]) \to [Int] \\ &appRev\ (xs, ys) = reverse\ xs + reverse\ ys \\ &revApp :: ([Int], [Int]) \to [Int] \\ &revApp\ (xs, ys) = reverse\ (ys + xs) \end{aligned}$$

The graphs of runtime measurements produced by our system for each of the *reverse* and *sort* examples are depicted in Figure 2. As before, performance results indicate that all of the properties are correctness-preserving optimisations:

$$\begin{aligned} reverse\ (reverse\ xs) &\trianglerighteq xs \\ reverse\ xs + reverse\ ys &\trianglerighteq reverse\ (ys + xs) \\ sort\ (reverse\ xs) &\trianglerighteq sort\ xs \\ sort\ (sort\ xs) &\trianglerighteq sort\ xs \end{aligned}$$

4.1.1 Baseline Measurements

Consider the graphs in Figure 3 for append's identity laws. The first graph indicates that $xs + [\,] \trianglerighteq xs$ is a linear time improvement, because *xs* must be traversed to evaluate $xs + [\,]$ to normal form. In comparison, we may expect the second graph to indicate that $[\,] + xs \trianglerighteq xs$ is a constant time improvement, because $[\,] + xs$ is the base case of the append

operator's recursive definition. However, the graphs in Figure 3 indicate linear time complexity for both functions. This is because, for each test case, the resulting list must be traversed in order to ensure each computation is fully evaluated. This takes linear time in the length of the list.

Although the system can evaluate test cases to weak head normal form, the true cost of applying append would not be reflected in the runtime measurements of $xs +\!\!+ [\,]$ if this user option was selected. This option is, however, appropriate when testing $[\,] +\!\!+ xs$. How then can users best compare the time performance of $xs +\!\!+ [\,]$ and $[\,] +\!\!+ xs$ directly?

To aid users in these kind of situations, the system provides a *baseline* option, which measures the cost of normalising results in each test case. When this option is selected, the system simply applies one of the test programs to each test input, and then benchmarks the identity function on the results of these applications. Baseline measurements are plotted as a black, dashed line of best fit, as in Figure 3.

If we compare the runtimes of $xs +\!\!+ [\,]$ against the baseline measurements, the line of best fit's gradient suggests additional linear time operations are being performed during evaluation. In contrast, the runtimes of $[\,] +\!\!+ xs$ approximately match the baseline measurements. Thus, interpreting these results with respect to the baseline measurements suggests that $xs +\!\!+ [\,]$ is linear while $[\,] +\!\!+ xs$ is constant.

Our online repository [20] includes many more examples from the original QuickSpec paper [9], each of which gives an improvement or cost-equivalence result, where the latter indicates that two programs have equal runtimes within a user-configurable margin of error.

4.2 Sorting

The *sort* function in Haskell's *Data.List* module was changed in 2002 from a quicksort algorithm to a merge sort algorithm. Comments in the source file [29] suggest that the change occurred because the worst-case time complexity of quicksort is $O(n^2)$, while that of merge sort is $O(n \log_2 n)$. Included in the comments is a summary of performance tests, which indicated that the new merge sort implementation did indeed perform significantly better than the previous quicksort implementation in the worst case.

Performance tests carried out at that time predate the development of systems such as Criterion and AutoBench. As such, runtimes were measured using an OS-specific timing

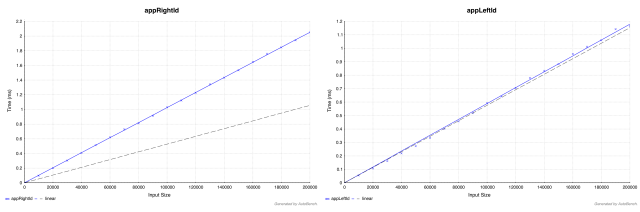
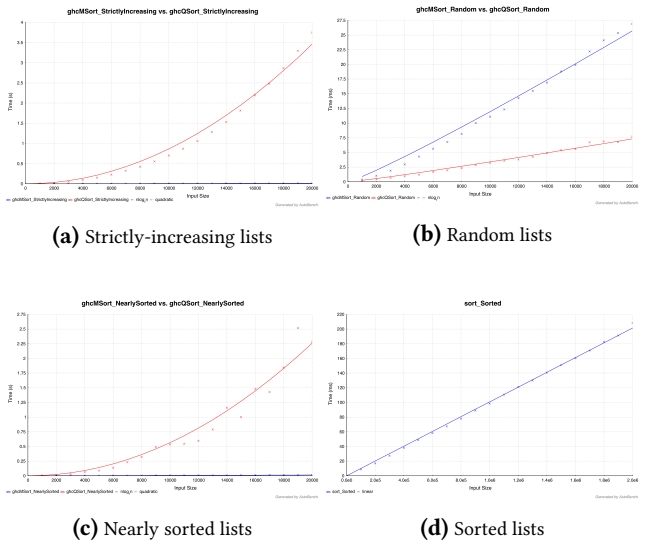


Figure 3. Graphs of results for append's identity laws



(a) Strictly-increasing lists **(b)** Random lists

(c) Nearly sorted lists **(d)** Sorted lists

Figure 4. Graphs of results for merge sort and quicksort

utility and testing was coordinated using a bash script. Our second case study was thus to rework the performance tests using our system. Doing so has a number of advantages over the previous approach. First of all, AutoBench uses Criterion to measure runtimes, which is more accurate and robust. Secondly, testing is fully automated, so there is no need to develop a custom script. And finally, our system produces lines of best fit and estimates time complexities.

When sorting lists that are strictly-increasing or reverse sorted, quicksort suffers from its worst-case time complexity of $O(n^2)$. In contrast, merge sort's time complexity is always $O(n \log_2 n)$ [12]. To put the theory to the test, we compared the time performance of the previous quicksort and new merge sort algorithms from *Data.List* when executed on strictly-increasing and reverse sorted lists of integers. All test files are available on our system's GitHub page [20].

The graph of runtime measurements for strictly-increasing input lists is given in Figure 4a, and shows a significant difference between the runtimes of quicksort and merge sort. Furthermore, for both types of list, our system estimated the time complexity of merge sort as $n \log_2 n$ and quicksort as n^2, and output a corresponding optimisation:

$$ghcQSort_{StrictlyIncreasing} \trianglerighteq ghcMSort_{StrictlyIncreasing} \quad (0.95)$$
$$ghcQSort_{ReverseSorted} \trianglerighteq ghcMSort_{ReverseSorted} \quad (0.95)$$

Hence, the results from our system concur with the previous performance tests, indicating that merge sort performs significantly better than quicksort in the worst case.

Different list configurations It is perhaps unfair to compare two implementations exclusively by their worst-case behaviour. As such, we added further tests to assess the time performance of each implementation when run on sorted, nearly sorted, constant, and random lists of integers.

A key advantage of our system's automated testing is that supplementary tests can be added easily, by simply defining additional generators that produce the necessary inputs. For example, we defined a generator for random lists,

```
instance Arbitrary RandomIntList where
  arbitrary = sized (λn → RandomIntList
    <$> vectorOf n arbitrary)
```

and then added one line test programs to accept values of this type and pass the underlying list of integers to the quicksort and merge sort implementations defined in the test file:

```
ghcQSort_Random :: RandomIntList → [Int]
ghcQSort_Random (RandomIntList xs) = ghcQSort xs
```

Runtime measurements for random and nearly sorted lists are depicted in Figures 4b and 4c, respectively. In all tests, except that of random lists, the merge sort implementation performed best, while for random lists both implementations were polylogarithmic. Thus, overall, our system suggests that the decision to change the implementation from a quicksort algorithm to a merge sort algorithm was a good one.

Smooth merge sort The current implementation of *sort* in *Data.List* is a more efficient version of merge sort than the one that originally replaced quicksort. By exploiting order in the argument list, this *smooth* merge sort algorithm [32] captures increasing and strictly-decreasing *sequences* of elements as base cases for its recursive merge step:

```
> sequences [0, 1, 2, 3]        > sequences [0, 1, 2, 3, 2, 1, 0]
[[0, 1, 2, 3], []]              [[0, 1, 2, 3], [0, 1, 2], []]
```

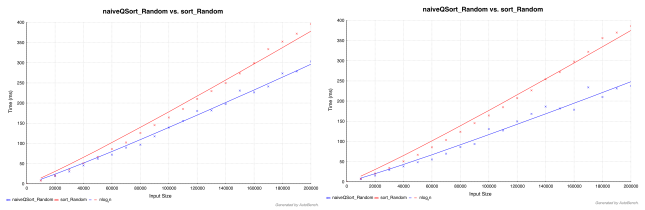
In particular, by taking sequences such as [0, 1, 2, 3] as base cases rather than singleton lists such as [0], [1], [2], and [3], the merge sort algorithm runs in linear time when the initial list is sorted or strictly-decreasing.

To see if this theory is supported by testing, we analysed the time performance of *sort* on sorted and strictly-decreasing lists. At this point, our test file already included all definitions required to test *sort*, so all that was left to do was import *Data.List* and define a one-line test program:

```
sort_Sorted :: SortedIntList → [Int]
sort_Sorted (SortedIntList xs) = sort xs
```

Results in Figure 4d estimate the time complexity of *sort* when executed on sorted lists as linear. Similar linear time estimates were given when testing its performance on strictly-decreasing and nearly sorted input lists.

Sorting random lists While exploring tests for this case study, we came across an interesting result: merge sort had worse time performance than quicksort when sorting random lists of integers (see Figure 4b). We were, therefore, curious to see how *sort* compared to different implementations of quicksort for random lists. Here we focus on the naive implementation often presented in textbooks [23]:



(a) Random lists, no optimisation **(b)** Random lists, O3 optimisation

Figure 5. Graphs of results for the sort function

```
qsort :: [Int] → [Int]
qsort []     = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
                 where
                   smaller = [a | a ← xs, a ⩽ x]
                   larger  = [b | b ← xs, b > x]
```

Two graphs of runtime measurements are displayed in Figures 5a and 5b, which show that, under different levels of optimisation, the library function *sort* performs notably *worse* than the naive quicksort function. Given that in real-life settings lists to be sorted are often 'nearly sorted' [14], this result may only have minor practical significance. Nevertheless, it is an outcome that surprised us, especially given the source of each implementation, and one that we feel underlines the importance of efficiency testing.

4.3 Sieve of Eratosthenes

The Sieve of Eratosthenes is a classic example of the power of lazy functional programming, and is often defined by the following simple recursive program: [23, 26]:

```
primes :: [Int]
primes = sieve [2..]
  where sieve (p : xs) = p : sieve [x | x ← xs, x `mod` p > 0]
```

However, while this definition produces the infinite list of primes, O'Neill [33] demonstrated that it is *not* in fact the Sieve of Eratosthenes. In particular, it uses a technique known as *trial division* to determine whether each candidate is prime, whereas Eratosthenes' original algorithm does not require the use of division. Consequently, the above algorithm performs many more operations than the true version.

In keeping with a list-based approach, the following program by Richard Bird, which appears in the epilogue of [33], implements the true Sieve of Eratosthenes:

```
truePrimes :: [Int]
truePrimes = 2 : ([3..] `minus` composites)
  where
    composites = union [multiples p | p ← truePrimes]
    multiples n = map (n*) [n..]
    (x : xs) `minus` (y : ys)
      | x < y = x : (xs `minus` (y : ys))
      | x ≡ y = xs `minus` ys
```

```
    | x > y = (x : xs) `minus` ys
union = foldr merge []
  where
    merge (x : xs) ys = x : merge' xs ys
    merge' (x : xs) (y : ys)
      | x < y = x : merge' xs (y : ys)
      | x ≡ y = x : merge' xs ys
      | x > y = y : merge' (x : xs) ys
```

Given that the simple but 'unfaithful' sieve performs many more operations than the true sieve, it is natural to ask how *primes* and *truePrimes* perform in practice. For the purposes of this particular case study, therefore, we were interested in comparing their time performance.

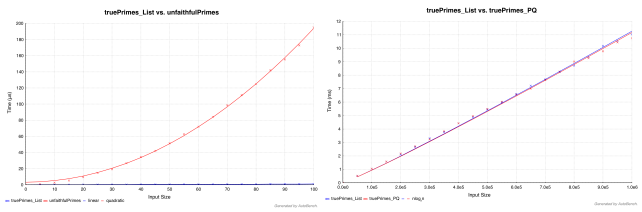
O'Neill's article [33] gives a detailed theoretical treatment of both implementations: to find all primes less than n, the unfaithful sieve *primes* is shown to have $\Theta(n^2/\ln^2 n)$ time complexity and the true list-based implementation *truePrimes* $\Theta(n\sqrt{n}\ \ln\ln n/\ln^2 n)$. Thus, from a theoretical standpoint, *primes* is asymptotically worse than *truePrimes*.

In addition, O'Neill's article includes a number of performance tests whereby implementations are compared according to the number of reductions performed by the Hugs interpreter [25] during their evaluation. In contrast, we wish to compare time performance in GHC using the AutoBench system. To achieve this, we defined the following two test programs to extract the first n prime numbers from each list, and compared them using the *AutoBench* executable.

```
unfaithfulPrimes :: Int → [Int]
unfaithfulPrimes n = take n primes

truePrimes_List :: Int → [Int]
truePrimes_List n = take n truePrimes
```

Test results in Figure 6a demonstrate a clear distinction between the runtimes of the unfaithful sieve and that of the true list-based sieve. Although AutoBench does not currently support time complexities as advanced as $\Theta(n^2/\ln^2 n)$ and $\Theta(n\sqrt{n}\ \ln\ln n/\ln^2 n)$, its prediction of quadratic and linear runtimes for *primes* and *truePrimes* are reasonable approximations. Moreover, when *truePrimes* is tested on larger input sizes (see Figure 6b), the system approximates its time complexity as $n\ \log_2 n$, which is closer to the theory.



(a) Unfaithful and true list-based sieves **(b)** True list- and p.q.-based sieves

Figure 6. Graphs of results for the sieve implementations

Nevertheless, the results in Figure 6a do clearly indicate that *truePrimes* has the better time performance,

$$unfaithfulPrimes \trianglerighteq truePrimes_{List} \quad (1.0)$$

and suggest that the time complexity of *primes* is asymptotically worse than *truePrimes*. Thus, overall, our testing agrees with the theory and also with the corresponding performance results presented in O'Neill's article [33].

4.3.1 An Alternative Data Structure

An implementation of the Sieve of Eratosthenes using *priority queues* is also presented in [33], which is shown to have a better theoretical time complexity than Bird's list-based implementation. As a final test for this case study, we were interested to see whether using a more complex data structure in this instance was worthwhile or not. To this end, we tested the list and priority queue implementations by generating the first million prime numbers.

The graph of runtime measurements for this test is given in Figure 6b and shows that both implementations perform comparably. Here our results differ from O'Neill's, who stated that the priority queue implementation was more time efficient for all primes beyond the $275,000^{\text{th}}$ prime. As a further validation of our results, we then tested both implementations up to the ten millionth prime. In this case, the list implementation performed marginally better.

The Haskell community has sometimes been criticised for overuse of the built-in list type in preference to more efficient data structures. Though the performance results above are very specific, they do illustrate that when used with care, lists *can* give solutions that are efficient. Nonetheless, these results certainly show that the unfaithful sieve's list-based implementation is not efficient, and that both true implementations have a significantly better time performance.

4.3.2 Operational Error

Although the unfaithful sieve produces the right results, it does not correctly implement Eratosthenes' algorithm. However, property-based testing tools such as QuickCheck cannot detect this error, because it is operational in nature, rather than denotational. On the other hand, our system estimated the unfaithful sieve's time complexity as n^2, which is significantly different from the true sieve's known complexity $\Theta(n\ \ln\ln n)$. This difference highlights the implementation error, and demonstrates how AutoBench can be used to uncover operational implementation errors.

5 Related Work

Much research has focused on automatically checking the correctness of Haskell programs, inspired by the development of the QuickCheck system. In contrast, our system appears to be the first aimed at automatically comparing the time performance of two or more Haskell programs. In this

section, we review work from the literature according to how it relates to the three major components of our system.

5.1 Data Generation

Another popular Haskell library for property-based testing is *SmallCheck* [36]. In comparison to QuickCheck's random testing, SmallCheck tests properties for all values up to a certain depth. In theory it would be possible for our system to use SmallCheck to generate inputs instead of QuickCheck. However, a single Criterion benchmark can often take in excess of one minute to run. Hence, measuring the runtimes of programs executed on all possible inputs up to a certain depth is unlikely to be practical.

In the wider literature, *EasyCheck* [5] is a property-based testing library written in the functional logic programming language *Curry*. It takes a similar approach to data generation as SmallCheck, making use of a technique known as narrowing to constrain generated data to values satisfying a given predicate. Research on constrained data generation has also been undertaken by Claessen et al. [6], who introduced a technique to ensure that data generated in this manner has a predictable and useful distribution.

The *Fake* package [2] was recently introduced as an alternative to QuickCheck for random data generation. Unlike QuickCheck, Fake focuses on generating random data that 'appears realistic'. This package was released after AutoBench had been developed, but we are keen to explore whether the data generation mechanism that it provides is useful for the purposes of our system.

5.2 Benchmarking

The system most closely related to ours is *Auburn* [28], which is designed to benchmark Haskell data structures. Similarly to our system it can generate inputs on behalf of users in the form of data type usage graphs (dugs), which combine test inputs with suitable operations on them. Each dug's performance is measured by its evaluation time.

As Auburn analyses sequences of operations, it can give insights into the amortised cost of individual operations. In contrast, our system's performance analysis is more coarse-grained and compares the runtimes of complete programs. Auburn does not compare or extrapolate its time measurements, instead they are simply output in a tabular format.

GHC Cost Centres [31] are used to measure time and space resource usage at particular locations inside functions. When code is compiled with profiling enabled, information regarding resource usage at each location is automatically generated. In comparison to our system, which benchmarks the runtimes of programs for comparative purposes, profiling is more fine-grained, and aims to reveal specific locations of maximum resource usage inside a single function. GHC cost centres could thus be used in conjunction with our system as part of a subsequent optimisation phase.

5.2.1 Comparing Benchmarks

Earlier versions of *Criterion* [35] included a function to compare benchmarks, known as *bcompare*. However, it turned out that this complicated many of the system's internals and as a result was removed, and has yet to be replaced.

Progression [4] is a Haskell library that builds on Criterion. It stores the results of Criterion benchmarks in order to graph the runtime performance of different program versions against each other. Users assign each benchmark a label, and then select which labelled data to compare graphically.

As Progression is a wrapper around Criterion, test inputs and benchmarks must be specified manually. Users are also responsible for compiling and executing their test files. Our system differs in this respect, as inputs are generated automatically and the benchmarking process is fully automated. In addition, Progression uses *Gnuplot* to produce its graphs, which it invokes via a system call. We preferred to use the *Chart* [13] package (similarly to Criterion) in order to keep our implementation entirely Haskell based.

5.3 Statistical Analysis

The field of study aimed at classifying the behaviour of programs (for example, their time complexity) using empirical methods is known as *empirical algorithmics*.

Profiling tools are the primary method of empirical analysis, which assign one or more performance metrics to individual locations inside a single program in order to collect runtime information in specific instances. Unlike our system, which compares the time performance of one or more programs, traditional profiling tools do not characterise performance as a function of input size.

Though some recent articles [10, 11, 41] have aimed to develop profiling tools that do specify runtime as a function of input size, they appear to focus primarily on automatically calculating the size of inputs, rather than describing in detail their methods for model fitting and selection. Nonetheless, input size inference could be useful for our system.

Model fitting and selection is discussed in the work of Goldsmith et al. [18], where the authors introduce a tool for describing the asymptotic behaviour of programs by measuring their 'empirical computational complexity'. In practice, this amounts to measuring program runtimes for different input sizes and then performing regression analysis on the results to approximate asymptotic time complexity.

Although this approach is similar to that used in our work, their system only supports polynomial models. Moreover, their choice of regression method is OLS and their model selection process is user-directed and centred on the R^2 fitting statistic. Both of these approaches favour models that overfit training data, but overfitting is not discussed in the article. In contrast, we use ridge regression and cross-validation to counteract overfitting, and provide a range of fitting statistics that can be used to automatically compare models.

The idea of inferring asymptotic bounds from runtime measurements is one we are keen to explore, but there appears to be no generally accepted solution to this problem [10]. However, some researchers have proposed heuristics for the 'empirical curve bounding' problem, showing their effectiveness on a number of occasions [15]. Some commercial software packages do advertise 'curve fitting' features [24, 34, 37], but as they are not freely available and don't publicise their underlying technologies, unfortunately we are not able to compare them with our system.

6 Conclusion and Further Work

In this article, we have taken two well-established systems, namely QuickCheck and Criterion, and combined them to give a lightweight, fully automated tool that can be used by ordinary programmers to compare the time performance of their Haskell programs. In addition, we have developed a simple but effective algorithm for approximating empirical time complexity based on regression analysis.

There are a number of potential avenues for further work. Firstly, this article has focused on time performance, but space performance is often just as important. Thus, it seems fitting that our system be extended to provide both time and space comparisons. Secondly, inspired by QuickCheck's domain-specific language (DSL) for specifying correctness properties, we would like to develop a DSL for specifying more sophisticated forms of improvement properties. Thirdly, it can be difficult to produce arbitrary inputs of a fixed size in an unbiased way, and an alternative approach would be to select inputs from real-world program runs instead. We are also keen to expand the system to handle multi-argument functions, and investigate how to visualise related results.

Finally, tools such as QuickSpec are able to generate correctness properties in a fully automatic manner. As we have seen in the first case study in section 4.1, many such properties are also time improvements, and it would be beneficial to interface AutoBench directly to such a tool.

Acknowledgements

We would like to thank Thomas Gärtner for many useful discussions, Jennifer Hackett and Iván Pérez for useful ideas on the initial implementation of our system, and the anonymous referees for their useful suggestions. This work was funded by EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency*.

References

[1] T. Arts, J. Hughes, J. Johansson, and U.T. Wiger. 2006. Testing Telecoms Software with Quviq QuickCheck. *Erlang Workshop*.
[2] D. Beardsley. 2018. Fake. https://hackage.haskell.org/package/fake.
[3] J. Bernardy, P. Jansson, and K. Claessen. 2010. Testing Polymorphic Properties. *ESOP*.
[4] N. Brown. 2010. Progression. https://hackage.haskell.org/package/progression.
[5] J. Christiansen and S. Fischer. 2008. EasyCheck - Test Data for Free. *FLOPS*.
[6] K. Claessen, J. Duregård, and M.H. Palka. 2015. Generating Constrained Random Data with Uniform Distribution. *JFP*.
[7] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ICFP*.
[8] K. Claessen and J. Hughes. 2002. Testing Monadic Code with QuickCheck. *Haskell Workshop*.
[9] K. Claessen, N. Smallbone, and J. Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. *TAP*.
[10] E. Coppa, C. Demetrescu, and I. Finocchi. 2012. Input-Sensitive Profiling. *PLDI*.
[11] E. Coppa, C. Demetrescu, I. Finocchi, and R. Marotta. 2014. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. *CGO*.
[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2009. *Introduction to Algorithms*. MIT Press.
[13] T. Docker. 2006. Chart. http://hackage.haskell.org/package/Chart.
[14] V. Estivill-Castro and D.Wood. 1992. A Survey of Adaptive Sorting Algorithms. *ACM Computing Surveys*.
[15] C.McGeoch et al. 2002. Using Finite Experiments to Study Asymptotic Performance. In *Experimental Algorithmics*. Springer.
[16] C. Klein et al. 2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. *POPL*.
[17] J. Fox. 1997. *Applied Regression Analysis, Linear Models, and Related Methods*. Sage Publications.
[18] S.F. Goldsmith, A.S.Aiken, and D.S. Wilkerson. 2007. Measuring Empirical Computational Complexity. *ESEC-FSE*.
[19] D. Gorin. 2015. Hint. https://hackage.haskell.org/package/hint.
[20] M.A.T. Handley. 2018. GitHub Repository for AutoBench. https://github.com/mathandley/AutoBench.
[21] A.E. Hoerl and R.W. Kennard. 1970. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*.
[22] J. Hughes. 2007. QuickCheck Testing for Fun and Profit. *PADL*.
[23] G. Hutton. 2016. *Programming in Haskell*. Cambridge University Press.
[24] D.G. Hyams. 2010. CurveExpert. http://www.curveexpert.net.
[25] M.P. Jones. 1999. Hugs. https://www.haskell.org/hugs.
[26] L.G.L.T. Meertens. 2004. Calculating the Sieve of Eratosthenes. *JFP*.
[27] A.K. Moran and D. Sands. 1999. Improvement in a Lazy Context: An Operational Theory for Call-By-Need. *POPL*.
[28] G.E. Moss. 2000. *Benchmarking Purely Functional Data Structures*. Ph.D. Dissertation. University of York.
[29] The University of Glasgow. 2001. Base: Data.List. http://hackage.haskell.org/package/base-4.11.1.0/docs/src/Data.OldList.html#sort.
[30] The University of Glasgow. 2014. Process. https://hackage.haskell.org/package/process.
[31] The University of Glasgow. 2015. Glasgow Haskell Compiler User's Guide: Profiling. http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html.
[32] R. O'Keefe. 1982. *A Smooth Applicative Merge Sort*. Department of Artificial Intelligence, University of Edinburgh.
[33] M.E. O'Neill. 2009. The Genuine Sieve of Eratosthenes. *JFP*.
[34] OriginLab. 2000. Origin. https://www.originlab.com.
[35] B. O'Sullivan. 2009. Criterion: Robust Reliable Performance Measurement and Analysis. http://www.serpentine.com/criterion.
[36] C. Runciman, M. Naylor, and F. Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. *Haskell Symposium*.
[37] Systat Software. 2002. TableCurve 2D. http://www.sigmaplot.co.uk.
[38] The GHC Team. 2017. GHC. https://hackage.haskell.org/package/ghc.
[39] P. Wadler. 1987. The Concatenate Vanishes. *University of Glasgow*.
[40] Q. Xu and Y. Liang. 2001. Monte Carlo Cross Validation. *Chemometrics and Intelligent Laboratory Systems*.
[41] D. Zaparanuks and M. Hauswirth. 2012. Algorithmic Profiling. *PLDI*.