shrinking is to 'extract the signal from the noise' (Hughes 2016). In this manner, inputs to failing test cases are reduced in size to find smaller instances that also violate the given property: the end goal being a counterexample in which every part is relevant to failure.

A notion of size for an input typically depends on its constituent datatypes. As such, in QuickCheck, a user-defined shrinking function must be provided for each type. Given an input, such a function produces a list of similar but smaller inputs. For example, a shrinking function for a list of integers can be defined as follows:

$$
\begin{aligned}
&\mathit{shrinkList} :: [\mathit{Int}] \to [[\mathit{Int}]] \\
&\mathit{shrinkList}\ [\,] \quad\quad = [\,] \\
&\mathit{shrinkList}\ (x : xs) = [xs] +\!\!+ [\ x : xs' \mid xs' \leftarrow \mathit{shrinkList}\ xs\ ] \\
&\qquad\qquad\qquad\qquad +\!\!+ [\ x' : xs \mid x' \leftarrow \mathit{shrinkInt}\ x\ ] \\
&\mathit{shrinkInt} :: \mathit{Int} \to [\mathit{Int}] \\
&\mathit{shrinkInt} = \ldots
\end{aligned}
$$

Using the above function for shrinking lists of integers, QuickCheck automatically finds an instance of the smallest possible counterexample to the following incorrect property

$$\mathit{prop\_rev\_bad}\ x = \mathit{slowRev}\ xs == xs$$

which is almost always $[0, 1]$ and on occasion $[1, 0]$.

Hughes' 2016 experience report states that "Debugging a test that is 90% irrelevant is a nightmare; presenting the developer with a test where every part is known to be relevant to the failure, simplifies the debugging task enormously." Our experience echoes this.

2.2 Performance testing

When confronted with the term *testing*, we should not be surprised to immediately think of functionality testing. By far, the majority of software testing literature addresses this kind of testing (Bertolino 2007), as we have just described in the previous section. However, certifying functionality is rarely sufficient to guarantee total software reliability. In fact, the major problems usually reported by large-scale projects after release are not incorrect system responses or even system crashes, but rather system performance degradation and

difficulties handling required throughput (Weyuker and Vokolos 2000).

The impact of these 'extra-functional' properties, for example, end-to-end response time, network delay, and the usage of system resources, on the reliability of software is characteristic of each specific application domain. Hence, these properties are perhaps harder to capture in comparison to functional requirements, which often transcend such boundaries and thus can be elicited by more comprehensive methods of requirements engineering (Weyuker and Vokolos 2000). Nonetheless, performance requirements of this kind are clearly an important indicator of software reliability and scalability, and consequently they should be held in similar regard to functional requirements.

To this end, *performance testing* is an umbrella term used to describe many approaches for examining the effects of different performance properties on software quality. An important distinction is between small-scale performance tests at the source code level and large-scale tests that target entire components or systems. In this thesis, we focus on the former case, in particular, performance testing via *microbenchmarking*. Woodside, Franks, and Petriu (2007) overview prominent approaches in the latter case.

Benchmarking

A traditional way of testing the performance of a system is to develop one or more *benchmarks* for it (Dolan and Moré 2002). In short, a benchmark is an abstract workload representing how a system is used in practice. In this manner, the system's behaviour when executed on a benchmark is considered to be indicative of its real-world behaviour.

The notion of a representative workload raises a number of questions that echo those discussed when introducing property-based testing in section 2.1. Primarily, assuming a workload involves executing a system on a sample of test data, how can we be sure that the distribution of the test data reflects that of actual data? In our experience, the situation is much the same as with property-based testing: the emphasis is on the user of a benchmarking test harness to ensure that test data is representative of real-world data.

Typical performance measurements used in benchmarking tests include execution time, memory allocation, throughput, lock contention, and input/output operations. The work

in this thesis primarily focuses on execution time but also touches on allocation.

Microbenchmarking

In comparison to benchmarking a full system, *microbenchmarking* aims to examine the performance of smaller code units, that is, a system's sub-components. The primary distinction between both methods is, therefore, the level of granularity at which performance testing is conducted. Given that entire systems and sub-system components are implemented alike in the realms of functional programming languages, the line drawn between benchmarking and microbenchmarking often blurs in practice.

Bershad, Draves, and Forin (1992) point out two implicit assumptions underlying the use of microbenchmarks. Firstly, it is assumed that the time required for a microbenchmark to traverse along a particular path of execution is the same as when that execution path is traversed in real-world use. Secondly, there is an assumption that a microbenchmark is representative of a system component that is either important in its own right, or which has a measurable impact on overall system performance.

The details of the first assumption in (Bershad, Draves, and Forin 1992) are centred around how cache coherence can affect performance testing: we do not address this concern. Nonetheless, a related insight is of concern: whether or not the same execution path even exists in real-world use. State-of-the-art compilers, such as GHC (GHC Team 2019b), subject source code to a sophisticated optimisation pipeline while transliterating it into machine code. The transformations applied during this phase depend on many variables that are not always self-evident. Thus, it is not necessarily the case that precisely the same machine code is generated in test conditions as in real-world conditions.

This insight is related to the second assumption, which is also relevant. As microbenchmarks examine the performance of increasingly smaller code units, it is increasingly important that those units of code have a measurable effect on the performance of the overall system. This is perhaps obvious in theory. What is less apparent is *when* sub-components of a system have a measurable effect on overall performance in practice. Specifically, optimising compilers may often be able to improve the performance of smaller code units in

the context of larger system components, but not in isolation; or vice-versa.

Overall, then, modern-day benchmarking libraries must be flexible, allowing for a myriad of testing environments to determine the effects of compilation and optimisation in practice. In addition, they must support (uniform) testing of code units at different levels of granularity so as to examine how their performances scale in broader contexts. They must also be statistically robust, and ideally user-friendly. A popular Haskell benchmarking library called *Criterion* has been shown on many occasions to satisfy all of these requirements.

Next, we illustrate the key concepts of microbenchmarking by introducing Criterion.

2.2.1 Benchmarking with Criterion

Criterion (O'Sullivan 2019a) is a microbenchmarking library that is used to measure the performance of Haskell code. It provides a framework for both executing benchmarks and analysing their results. Its high-resolution analysis is able to measure the performance of runtime events with duration in the order of picoseconds.

Two of Criterion's key benefits are its use of regression analysis and cross-validation, which allow it to eliminate measurement overhead and distinguish real data from noise generated by external processes. In particular, the system is robust enough to filter out noise coming from, for example, clock resolution, operating system scheduling, and garbage collection. To achieve this, it measures many 'runs' of a benchmark in sequence and then uses linear regression to estimate the time needed for a single run. In this manner, the outliers become visible. Overall, measurements made by Criterion are far more accurate and reliable than those made by operating system timing utilities.

Given that Haskell's non-strict semantics only evaluates an expression when it is needed, Criterion provides mechanisms to evaluate, or force, the results of benchmarks to different normalisation forms, including weak head normal form and normal form. Initial versions of the library only supported the analysis of pure Haskell code. More recently, however, Criterion has been extended to analyse the performance of code with IO side effects.

Criterion can measure CPU time, CPU cycles, memory allocation, and garbage collection. It has also been adapted to measure energy consumption (Lima et al. 2016). The work in this thesis predominantly focuses on benchmarking the time performance of pure

Haskell code. The remainder of this introduction, therefore, focuses on CPU time, but we note that analysing other performance indicators using Criterion is much the same.

Specifying benchmarks

Recall that functional programming is a style of programming in which the primary method of computation is the application of functions to arguments. As such, a Haskell microbenchmarking library can directly support the analysis of differently sized code units by simply allowing 'any' function and its corresponding arguments to be benchmarked. This is the approach taken by Criterion, whose principal type is *Benchmarkable*. A value of this type can be constructed using the following functions

$$nf :: NFData\ b \Rightarrow (a \to b) \to a \to Benchmarkable$$
$$whnf :: (a \to b) \to a \to Benchmarkable$$

each of which takes a function $f :: a \to b$ and an argument $x :: a$, and measures the time taken to evaluate the computation $f\ x :: b$. In the former case, *nf* measures the time taken to evaluate $f\ x$ to normal form. In the latter case, *whnf* measures the time taken to evaluate $f\ x$ to weak head normal form, which is Haskell's default degree of normalisation.

Unlike with the QuickCheck system, which automatically generates random inputs when checking correctness properties (see section 2.1.1), arguments to functions must be manually specified when measuring the runtimes of computations with Criterion. Despite this difference, it remains the responsibility of the user to ensure that test inputs are representative of real-world use cases, just as with QuickCheck data generators.

As a concrete example, consider the following definition

$$nf\ slowRev\ [0..200] :: Benchmarkable$$

which makes the application of Haskell's naive list-reversing function

$$slowRev :: [a] \to [a]$$
$$slowRev\ [] = []$$
$$slowRev\ (x : xs) = slowRev\ xs +\!\!+ [x]$$

to a list of integers *Benchmarkable*. In this instance, evaluation to normal form ensures

the runtime measurements reflect the cost of fully applying *slowRev*. The standard class *NFData* comprises types that can be fully evaluated, and hence *nf* requires the result type of its argument function—$[Int]$ in this case—to be an instance of this class.

In some situations, using *nf* may force undesired evaluation, such as when measuring the runtimes of programs whose outputs are, by design, produced lazily. The following definition, therefore, measures the time taken to reach weak head normal form:

$$whnf\ slowRev\ [0\,..\,200]\ ::\ Benchmarkable$$

According to the definition of *slowRev*, this is the point at which its result has the form $200 : ([199] +\!\!+ ([198] +\!\!+ ([197] +\!\!+ \ldots)))$, and so we see that the vast majority of the appends are not evaluated. Consequently, the runtimes measured by this *Benchmarkable* are significantly faster than those of the above *Benchmarkable* constructed using *nf*.

In general, however, we should not assume that any *Benchmarkable* constructed using *whnf* performs fewer steps of evaluation than its *nf* counterpart. For example, if measuring the runtime of Haskell's standard *reverse* function, which is implemented using *foldl*, then both of the following *Benchmarkable*s are essentially equivalent

$$nf\ reverse\ [0\,..\,200]\ ::\ Benchmarkable$$
$$whnf\ reverse\ [0\,..\,200]\ ::\ Benchmarkable$$

because *foldl* does not produce a result until its entire input has been completely traversed.

Remark. It may appear odd that a *Benchmarkable* cannot be constructed from a single argument. That is, why not accept $f\ x :: b$ directly, rather than requiring $f :: a \to b$ and $x :: a$ be separate arguments? This is due to memoisation. As stated previously, Criterion measures many runs of a benchmark in order to provide statistically robust results. In turn, this means that each *Benchmarkable* must be evaluated multiple times. In Haskell, any expression is memoised after it has been evaluated once. Repeatedly benchmarking with the same expression would thus mean that all subsequent runs after the first would perform no evaluation. This is clearly problematic for performance analysis, and so Criterion constructs and evaluates a new expression from f and x for each run.

*Benchmarkable*s can also be used to measure the performance of impure code:

$$\begin{aligned}
\mathit{nfIO} &:: \mathit{NFData}\ a \Rightarrow \mathit{IO}\ a \rightarrow \mathit{Benchmarkable} \\
\mathit{whnfIO} &:: \mathit{IO}\ a \rightarrow \mathit{Benchmarkable} \\
\mathit{nfAppIO} &:: \mathit{NFData}\ b \Rightarrow (a \rightarrow \mathit{IO}\ b) \rightarrow a \rightarrow \mathit{Benchmarkable} \\
\mathit{whnfAppIO} &:: (a \rightarrow \mathit{IO}\ b) \rightarrow a \rightarrow \mathit{Benchmarkable}
\end{aligned}$$

Note the first two functions perform their IO actions for each measured run and hence their pure expressions of type a are not affected by the above issue concerning memoisation.

To uniquely identify measurements for purposes of analysis, Criterion does not directly execute *Benchmarkable*s. Instead, users must define *Benchmark*s. A *Benchmark* is simply a *Benchmarkable* computation together with a suitable description, which can be constructed using the *bench* function as in the following examples:

$$\begin{aligned}
&\mathit{bench}\ \texttt{"slowRev, [0..200], nf"}\ (\mathit{nf}\ \mathit{slowRev}\ [0..200]) :: \mathit{Benchmark} \\
&\mathit{bench}\ \texttt{"reverse, [0..200], nf"}\ (\mathit{nf}\ \mathit{reverse}\ [0..200]) :: \mathit{Benchmark}
\end{aligned}$$

Executing benchmarks

Previously, we highlighted the importance of microbenchmarking libraries supporting a wide range of testing environments. For example, it is often useful to analyse the performance of code when subjected to different degrees of optimisation, as this can have a notable effect on machine code generation, and, by extension, efficiency.

To support different testing environments, Criterion benchmarks are executed in the $\mathit{main} :: \mathit{IO}\ ()$ functions of Haskell modules. Haskell modules are typically compiled using the Glasgow Haskell Compiler, which offers a comprehensive set of compiler options. These options affect both the compilation process and the runtime system (GHC Team 2019b), and so, in practice, benchmarks can be executed in a multitude of different ways.

Criterion's top-level functions for executing benchmarks are as follows:

$$\begin{aligned}
\mathit{defaultMain} &:: [\mathit{Benchmark}] \rightarrow \mathit{IO}\ () \\
\mathit{defaultMainWith} &:: \mathit{Config} \rightarrow [\mathit{Benchmark}] \rightarrow \mathit{IO}\ ()
\end{aligned}$$

Each takes a list of benchmarks and executes them in sequence. The first uses a standard configuration. The second allows users to specify a custom configuration, for example, to select which performance indicators to measure. Further details regarding Criterion's *Config* are given on the system's webpage (O'Sullivan 2019a). Both functions parse command-line options, which can, for example, be used to execute a subset of the given benchmarks and produce numerous different reports. We discuss performance reports next.

Analysing performance measurements

The basic form of output from a Criterion benchmark is as follows:

```
benchmarking: slowRev, [0..200], nf
time                    127.9 μs  (127.3 μs .. 128.6 μs)
                        0.999 R²  (0.999 R² .. 1.000 R²)
mean                    128.3 μs  (127.8 μs .. 129.3 μs)
std dev                 2.247 μs  (1.280 μs .. 3.558 μs)
variance introduced by outliers: 11% (moderately inflated)
```

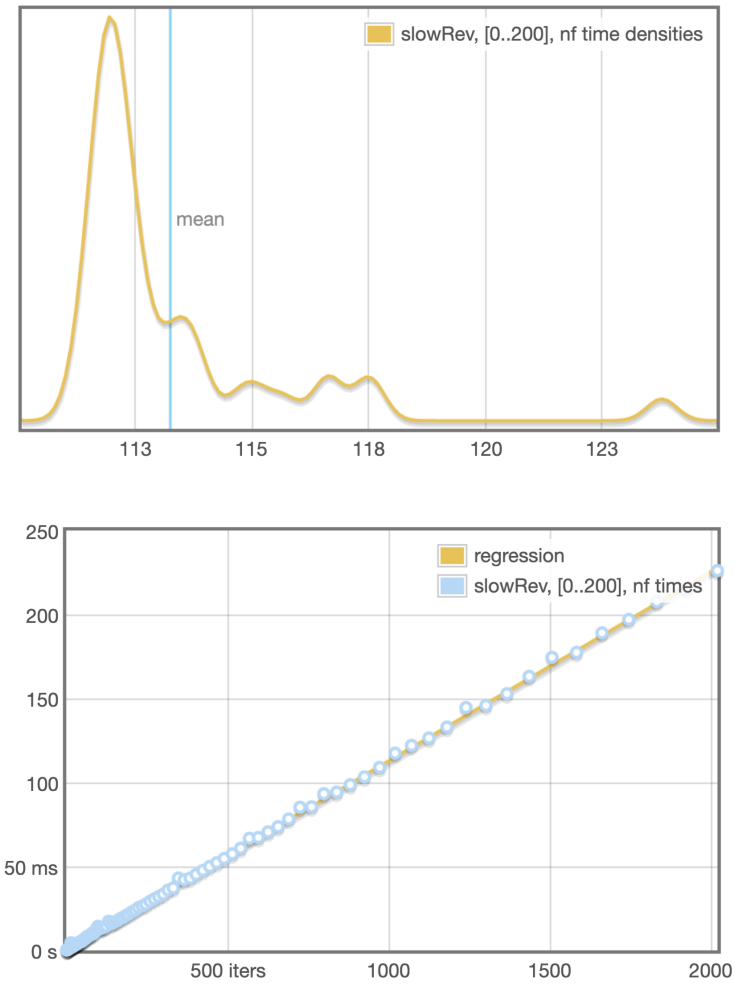
This information is printed to the console when a *Benchmark* is executed by *defaultMain*. It reveals that the *mean* time taken to evaluate the result of *slowRev* $[0\,..\,200]$ to normal form across all measured runs is 128.3 microseconds. The *time* predicted by Criterion's linear regression is more accurate, but should in general be comparable to average time. The accuracy of the regression model is measured by R^2, which is a standard goodness-of-fit indicator used in statistics whose value ideally lies between 0.99 and 1.00.

Standard deviation, *std dev*, and *variance* assess the quality of the measurements taken. In simple terms, both statistics measure the dispersion of the raw data in relation to the mean, with the latter value attributing it to the presence of outliers. In the context of performance testing, high standard deviation and variance indicate a 'noisy' testing environment, which may negatively affect the reliability of the measurements. (From our experience, if the variance is above 25%, then it is advisable to re-run the benchmark.)

Measurements such as those above can be automatically saved to different file formats, including JSON and CSV, when benchmarks are executed. These reports are essentially

'data dumps' whose filepaths are given in Criterion's *Config* datatype.

Visual feedback on performance measurements is provided by Criterion in the form of interactive charts, which are saved as HTML webpages. For example:



The chart at the top is a kernel density estimate (KDE) of runtime measurements. In general, a KDE graphs the probability of any given measurement occurring. In this instance, a spike at a given value indicates a runtime measurement of that particular value was recorded; its height indicates how often the measurement was repeated.

The chart at the bottom displays the raw data from which the kernel density estimate

was built. Recall that Criterion measures many runs of a benchmark. As such, the x-axis of this chart indicates the number of runs, while the y-axis shows the runtime for a given number of runs. The line of best fit predicts runtime using a basic linear regression model. Ideally, all measurements should be on or very close to this line.

Both of the charts are interactive in that they detail individual measurements when a user places their cursor at different points of each line. Finally, note that (as with reports) charts are produced only when their filepaths are specified in Criterion's *Config* datatype.

2.3 Equational and inequational reasoning

2.3.1 Equational reasoning

The equations that are used to define programs in pure functional programming languages are both computational rules and a basis for simple, secondary school algebraic reasoning about the functions and data structures that they define. For example, a function that doubles a given integer x can be defined by the following equation:

$$\begin{aligned} &\mathit{double} :: \mathit{Int} \rightarrow \mathit{Int} \\ &\mathit{double}\ x = x + x \end{aligned}$$

As well as specifying how to compute two times any integer, namely by substituting all occurrences of x in $x + x$ for a given argument, *double* also serves as a logical equation. That is, for any integer x, the expression $\mathit{double}\ x$ is equal to the expression $x + x$. Thus, in the spirit of algebraic reasoning, either may be replaced by the other.

A key property of pure functional programming languages is referential transparency, which states that a denotational semantics—mapping expressions to values in a semantic domain—is concerned exclusively with each expression's denotation (see section A.1.1 for relevant background material). In practice, given that pure expressions have no side effects, this means that a subexpression can be freely replaced by any other subexpression mapped to the same value without affecting the meaning of its surrounding context.

In the case of *double*, the following denotational equalities hold: