Efficiency Three Ways
Tested, Verified, and Formalised

by Martin Adam Thomas Handley

Supervised by
Professor Graham Hutton

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

December 2019

Abstract

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

One of the key benefits of the functional programming paradigm is the ability to reason about programs as if they are pure mathematical functions. In particular, programs can often be proved correct with respect to a specification by exploiting simple algebraic properties akin to secondary school mathematics. On the other hand, program efficiency is not immediately amenable to such algebraic methods used to explore program correctness.

This insight manifests as a reasoning gap between program correctness and efficiency, and is a foundational problem in computer science. Furthermore, it is especially pronounced in lazy functional programming languages such as Haskell, where the on-demand nature of evaluation makes reasoning about efficiency even more challenging.

To aid Haskell programmers in their reasoning about program efficiency, the work in this thesis seeks to partially bridge the reasoning gap using three different approaches: automated testing, semi-formal verification, and formal verification.

Acknowledgements

First and foremost I am thankful to my supervisor, Graham Hutton, who has been instrumental in my learning and research for over seven years, throughout both my Master's and PhD courses. As a first-year undergraduate, Graham introduced me to Haskell. Since then he has taught me to think logically, write technically, and program functionally.

Thanks to my examiners, Professor John Hughes and Dr. Henrik Nilsson, for an extremely pleasant (online) viva, and for their useful comments and suggestions. Thanks also to Dr. Milena Radenkovic for stepping in last minute to chair my viva.

Thanks to my collaborator, Niki Vazou, for sharing her knowledge and passion for Liquid Haskell with me so that we could use the system for resource analysis. I was also fortunate enough to spend a week visiting Niki at the IMDEA Software Institute, which was a very rewarding experience. Thank you, Niki, for inviting me.

I am grateful to all members of the FP lab at the University of Nottingham, past and present, for their support, insights, and feedback. In particular, the faculty members Thorsten Altenkirch, Venanzio Capretta, and Henrik Nilsson; the postdoctoral students Paolo Capriotti, Laurence E. Day, Jennifer Hackett, and Nicolai Kraus; and my peer Colm Baston. Special thanks go to Ivan Perez for his ongoing guidance and inspiration. Special thanks also go to my peer, good friend, and former housemate, Jonathan Thaler, who provided endless entertainment in my last year of study.

This work was funded by EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency*.

I am indebted to the Starbucks coffee shop on UoN's Jubilee Campus, which served as my primary base for the duration of my PhD. Specifically, thank you to Jazz for serving me endless cups of tea and for always asking about my research. I hope you are keeping well in your new job role. This coffee shop will always be a place of fond memories.

I am thankful to my best friend and former housemate, Fred Etchells, for always looking out for me and keeping me in check. Thanks also to my good friend and former housemate, Jamie Johnson, for always getting that extra 20%, every day.

Last but by no means least, I am deeply grateful to my beloved family, for their unconditional love and support throughout all my life. My Grandma, Eva, to whom this thesis is dedicated; my parents, Beryl and Adrian; my sisters, Clare and Nicola, and their spouses, Damon and Glenn. Finally, to my nieces, Isobel, Robyn, and Heidi, and nephew, William, thank you for always reminding me that life is for living.

For Eva

Table of Contents

Chapter 1

Introduction

Two fundamental goals in programming are correctness and efficiency: we want our programs to produce the right results, and to do so using as few resources as possible. While these characteristics are often equally desirable, they are just as often in contention with each other. Programs that are 'clearly correct' may rely on inefficient abstractions to aid their comprehension, whereas programs that have enhanced performance may avoid such abstractions in favour of specialised efficient operations. To strike a balance between these two extremes, we must reason about both the correctness and the efficiency of our programs.

One of the key benefits of the functional programming paradigm is the ability to treat programs as pure mathematical functions. This property helps to make aspects of functional programming languages especially easy to reason about. In particular, such programs can often be proved correct with respect to a specification, or even derived from a specification, by exploiting simple algebraic properties akin to secondary-school mathematics.

Functional programs are suited to these activities due to the ease with which their extensional behaviour can be understood. On the other hand, intensional program behaviour, that is, properties of how a program computes, not what it computes, is often made more opaque by the high-level nature of the functional paradigm. More specifically, important intensional properties such as efficiency are not immediately amenable to the algebraic methods of reasoning used to readily explore extensional properties such as correctness.

This insight manifests as a *reasoning gap* (Harper 2014) between program correctness and efficiency, and is a foundational problem in computer science. Furthermore, it is es-

pecially pronounced in lazy functional programming languages such as Haskell, where the on-demand nature of evaluation makes reasoning about efficiency even more challenging.

To aid Haskell programmers in their reasoning about program efficiency, the work in this thesis seeks to partially bridge the reasoning gap using three different approaches: automated testing, semi-formal verification, and formal verification. For each approach, we are inspired by previous work on reasoning about program correctness and aim to bring about a comparable method for addressing questions of efficiency. To examine the practical applicability of our work, we implement each method in a new Haskell system that builds upon popular tools used for software testing and verification.

1.1 Contributions

This thesis makes the following contributions:

- We present the design and implementation of the *AutoBench* system (section 3.3), which combines the Criterion benchmarking library and the QuickCheck testing library to provide a lightweight means to compare the time performance of Haskell programs. Our system incorporates a custom method for approximating time complexity (section 3.3.4), based on linear regression analysis.

- We show how the Kansas University Rewrite Engine, which forms the basis of the semi-formal equational reasoning assistant Hermit, can be adapted to form the basis of a semi-formal inequational reasoning assistant called *Unie* (section 4.4). Our system aids in mechanically constructing improvement proofs by implementing Moran and Sands' tick algebra (section 4.3.4), an inequational theory allowing execution costs to be moved around within programs while maintaining or improving efficiency.

- We present the design and implementation of the *RTick* library (section 5.3), which enables Liquid Haskell to be used as a formal means to reason about the abstract resource usage of pure Haskell programs. Liquid Haskell can be viewed as an extension to the type system of Haskell that supports formal reasoning about program correctness by encoding logical properties as refinement types. Our system supports reasoning

about correctness and efficiency in a combined, uniform manner (section 5.3.3). Finally, we prove that our approach is correct with respect to an underlying model of execution cost using the metatheory of Liquid Haskell (section 5.5).

– We demonstrate the practical applicability of all of our implementations. Regarding the AutoBench system, we present a number of case studies taken from the Haskell programming literature (section 3.4). Regarding the Unie system, we mechanically verify all the improvement results in (Hackett and Hutton 2014), which is the article that renewed interest in improvement theory, and a number from the original article (Moran and Sands 1999) (section 4.6). Finally, regarding the RTick library, we provide a wide range of case studies, ranging from standard sorting algorithms to sophisticated relational cost properties, including all examples from Aguirre et al. (2017), Çiçek et al. (2017), and Radiček et al. (2018) (section 5.4).

Some of the work in this thesis has been published previously, in particular:

– The material in chapter 3, which presents the AutoBench system, is based on the work previously published in (Handley and Hutton 2018a).

– The material in chapter 4, which presents the Unie system, is based on the work previously published in (Handley and Hutton 2018b).

– The material in chapter 5, which presents the RTick library, is based on the work previously published in (Handley, Vazou, and Hutton 2020).

The author of this thesis was the lead author of all three papers. Moreover, the author developed the AutoBench system introduced in chapter 3, the Unie system introduced in chapter 4, and was the lead developer of the RTick library introduced in chapter 5. Overall, the work in this thesis resulted in approximately 40,000 lines of new Haskell code.

1.2 Organisation

The remainder of this thesis is organised as follows:

– Chapter 2 provides background material necessary for this thesis to be reasonably self-contained. We introduce property-based testing, microbenchmarking, equational and inequational reasoning, and the formalisation of such reasoning using refinement types. A number of these discussions include an introduction to a system that implements the respective topic in Haskell, for example, QuickCheck. In subsequent chapters, we then build upon such systems to provide new implementations for efficiency analysis. For completeness, we overview the topic of program semantics, specifically operational and denotational semantics, in appendix A.

– Chapter 3 is concerned with testing. We introduce the AutoBench system: a lightweight, fully automated tool for analysing and comparing the time performance of pure Haskell programs. Furthermore, we describe AutoBench's custom method for approximating time complexity based on linear regression analysis. A number of case studies are presented to demonstrate the system's applicability, taken from the existing Haskell programming literature. Finally, we note that some of the future work discussed at the end of this chapter is partially complete.

– Chapter 4 is concerned with semi-formal reasoning. We introduce the Unie system: an inequational reasoning assistant that provides mechanical support for proofs of program improvement. We discuss how Unie implements Moran and Sands' tick algebra for improvement theory, which allows unit time costs to be moved around soundly within terms. Furthermore, we explain how program contexts, a central aspect of improvement theory, are automatically managed by the system, and show how this simplifies reasoning steps in mechanised proofs. We demonstrate Unie's practicality by mechanically verifying a number of key results from the relevant literature.

– Chapter 5 is concerned with formal reasoning. We introduce the RTick library, used to formally reason about abstract resource usage in Liquid Haskell. We show how the library builds upon Liquid Haskell's existing features in a lightweight manner, and demonstrate how harnessing such features improves the precision of the library's cost analysis. In addition, we show how our approach supports reasoning about correctness and efficiency properties in a combined, uniform manner. A large number

of case studies are provided to draw comparisons against related systems.

– Chapter 6 concludes with a summary and discussion on potential future work.

As chapters 3–5 have a specific focus, namely testing, semi-formal reasoning, and formal reasoning, each ends with a discussion on related and further work, and with a short conclusion. The main points from each of these sections are then brought together in chapter 6.

Finally, this thesis is aimed at readers who are familiar with the basics of functional programming in a language such as Haskell, but we don't assume any specialised knowledge on topics such as regression analysis, inequational reasoning, improvement theory, static cost analysis, refinement types, QuickCheck, Criterion, or Liquid Haskell.

Chapter 2

Background

In this chapter, we present background material on a range of different topics that are central to this thesis. The content is divided into three parts, according to how it relates to automated testing, semi-formal reasoning, and formal reasoning. We note that standard background material on program semantics is provided in appendix A.

Automated testing

The first part of this chapter discusses property-based testing (section 2.1) and performance testing (section 2.2), specifically microbenchmarking. The former is used to check correctness properties of programs. The latter is used to examine, among other things, the resources required to execute programs and their sub-components. We introduce the primary Haskell implementation for each testing methodology, highlighting intricacies in their designs that ensure testing is both accurate and reliable.

Subsequently, in chapter 3, we build upon the methodologies of property-based testing and microbenchmarking to provide a fully automated means to analyse and compare the time performance of Haskell programs. In practice, we concretise our approach by introducing a tool that combines the Haskell systems discussed in sections 2.1 and 2.2.

Semi-formal reasoning

The second part of this chapter discusses equational and inequational reasoning (section 2.3). Equational reasoning (section 2.3.1), widely studied and practiced by the func-

tional programming community, is a simple yet powerful method for verifying correctness properties of programs, and can also be used to derive programs from high-level specifications. Inequational reasoning (section 2.3.2), lesser-known, somewhat more involved, but very similar to its equational counterpart in its overall approach, provides a similar proof technique for verifying efficiency properties of programs.

Subsequently, in chapter 4, we introduce a system for mechanising semi-formal reasoning about the time efficiency of Haskell programs. The theory underpinning the system is operational in nature, building on the material presented in section A.1.2. In addition, the style of reasoning supported by the system—prompted by the desire to improve runtime performance—is necessarily inequational, just as in section 2.3.2.

Formal reasoning

The final part of this chapter discusses formal reasoning about programs (section 2.4), specifically by way of refinement types. A popular system called *Liquid Haskell*, which enables formal reasoning about Haskell programs, is introduced in section 2.4.1. In short, Liquid Haskell can be seen as an extension to the type system of Haskell that allows for expressing logical properties as refinement type specifications.

Subsequently, in chapter 5, we build upon Liquid Haskell's existing features to develop a library that can be used to formally reason about the abstract resource usage of pure Haskell programs. A notable feature of the library is that it can be used to construct unified proofs of program correctness and efficiency, which echo the style of proof presented in section 2.3.1, but which are fully formal due to Liquid Haskell's use of refinement types.

2.1 Property-based testing

Functionality testing (Bertolino 2007), usually referred to as just *testing*, aims to determine if the intended and actual behaviours of a program differ, or to provide a high degree of confidence that they do not. To achieve this, testing seeks to uncover observable differences between the behaviour of a program's implementation in practice and the proposed behaviour of the implementation as expressed by the program's requirements.

Often it is not possible to execute a program on its entire input domain because it has very large cardinality. Hence, the choice of inputs used to test a program has a notable effect on the reliability of test results. Furthermore, software bugs may only (and frequently do) manifest when multiple components of a system interoperate. Testing methodologies that inspect sub-components of a system in isolation, for example, unit testing (Daka and Fraser 2014), therefore, cannot rule out the presence of errors existing at a higher level of granularity than their test coverages admit. Finally, when an error does occur, interpreting the failure may require significant effort. Doing so frequently involves probing the program's execution trace to uncover the root cause of the error, thereby simplifying the failing case.

Overall, these insights can make thoroughly testing a code base an arduous task: test inputs should be comparable to expected (user) inputs, but also be sufficiently diverse as to reach the extremities of a program's input domain; testing procedures must inspect sub-components of a system working independently, but also cooperatively in a way that reflects the system's real-world use; and finally, failing test cases may often be hard to comprehend and, therefore, require a notable degree of simplification. In 2000, Claessen and Hughes outlined a testing methodology that addresses these three seemingly fundamental limitations of software testing. This approach, known as *property-based testing*, is prompted by the following tag line from Hughes (2016): *don't write test cases—generate them!*

Claessen and Hughes concretised their approach to property-based testing in a system called *QuickCheck*, which aids users in formulating and *randomly* testing correctness properties of Haskell programs. In the remainder of this section, we introduce the key concepts of property-based testing by highlighting their applications in the QuickCheck system.

2.1.1 Property-based testing with QuickCheck

Specifying correctness properties

Distinguishing a desired (correct) program behaviour from an undesired (incorrect) behaviour is known as the *oracle problem* (Barr et al. 2014). Tools for automated testing must provide a test oracle that can decide whether a particular program behaviour is desirable or undesirable. In other words, they must provide a decision procedure for determining

whether a test case has passed or failed. In property-based testing, a *formal specification* of program properties functions as such an oracle. In particular, specifications drive the testing process by checking whether the respective program satisfies each given property.

QuickCheck (Claessen and Hughes 2000a) provides a simple domain-specific language of testable specifications, which can be used to define correctness properties of programs. For example, a specification of a naive list-reversing function, *slowRev*, is as follows

$$\begin{array}{rcrcl} \mathit{prop\_rev\_one}\ x & = & \mathit{slowRev}\ [x] & == & [x] \\ \mathit{prop\_rev\_app}\ xs\ ys & = & \mathit{slowRev}\ (xs\ +\!\!+\ ys) & == & \mathit{slowRev}\ ys\ +\!\!+\ \mathit{slowRev}\ xs \\ \mathit{prop\_rev\_rev}\ xs & = & \mathit{slowRev}\ (\mathit{slowRev}\ xs) & == & xs \end{array}$$

in which all inputs must be finite, that is, total and terminating.

Property-based testing assumes that this specification captures everything compelling about *slowRev* because, in practice, testing only validates these properties. Nonetheless, these properties afford a much more general specification of *slowRev* than standard automated test cases typically admit. For example, the first two properties are a characterisation of *slowRev*. That is, instead of being properties emergent of a specific implementation of *slowRev*, they define the elementary requirements of any finite list-reversing function.

The above properties of *slowRev* are assumed to hold for its entire input domain, rather than for one or more specific examples. In particular, the first property, *prop_rev_one*, states that for *any* finite value x, reversing a singleton list containing x is an identity operation. Similarly, the last property, *prop_rev_rev*, states that reversing any finite list twice is also an identity operation. For this reason, there is no need to define more than one QuickCheck correctness property for each logical property being tested.

Similarly to the above, QuickCheck specifications can formalise medium-level expectations of interconnected system components and high-level expectations of complete systems. For examples of medium- and high-level specifications, see (Hughes 2016). Thus, overall, we see that property-based testing is highly applicable to software verification.

Checking correctness properties

In practice, a correctness property can be checked by merely evaluating it on a specific input. As we have discussed, it is seldom possible to test a program on its entire input domain: certainly, it is impossible for *slowRev*. Therefore, property-based testing with QuickCheck also involves automatically generating suitable test data.

QuickCheck's approach to data generation is fundamental to its testing methodology: test inputs are generated randomly and so specifications are tested randomly. We previously observed that an inability to test a program on all possible inputs emphasises the importance of suitably chosen test data. This observation resurfaces here, as random testing is most effective when the distribution of test data reflects that of real-world data (Claessen and Hughes 2000a). If we assume comparable distributions of test data and actual data, then we can expect an increasing number of errors to be uncovered when properties are repeatedly tested, simply because every re-test allows for new test cases to be generated. This expectation has been demonstrated in practice (Hughes 2016).

QuickCheck cannot select test data distributions on behalf of users. This is because (good coding practice dictates that) sub-system components are typically reused throughout a codebase. As such, the distributions of the actual data in all subsequent reuses cannot be determined a priori. In consequence, QuickCheck requires that users manually specify test data distributions by way of a data generation domain-specific language. This language provides generators for all standard Haskell types, together with a rich set of combinators that can be used to define generators for user-defined datatypes.

For example, a generator for a list of integers can be defined as follows:

```
generateLists :: Gen [Int]
generateLists = frequency
  [ (1, pure [])
  , (3, (:) <$> elements [1..100] <*> generateLists) ]
```

In this definition, the *frequency* and *elements* combinators choose a random value from a list. The *frequency* combinator does so according to the specified weights 1 and 3, and so this generator produces an empty list 25% of the time, and 75% of the time places a

randomly chosen integer between 1 and 100 at the head of a recursively generated list.

Checking the previously defined properties of *slowRev* thus amounts to evaluating those properties on random data generated in this manner. For example, we can execute the following test to check that the *slowRev* function is an involution:

```
> quickCheck (forAll generateLists prop_rev_rev)
+++ OK, passed 100 tests.
```

In this instance, the QuickCheck system generates one hundred random lists of integers and finds that the property is satisfied in all cases. As *slowRev* is a polymorphic function, we might infer that this result is representative of all finite input lists.

QuickCheck provides two top-level functions for checking *Testable* correctness properties, which can be intuitively thought of as predicates, as per the example above:

```
quickCheck :: Testable prop => prop -> IO ()
quickCheckWith :: Testable prop => Args -> prop -> IO ()
```

The first uses a default set of arguments. The second allows users to specify custom arguments, for example, to check more test cases than the default number of one hundred. Often, these functions are used inside GHCi, which is the Glasgow Haskell Compiler's interpreter and debugger (GHC Team 2019). Further details regarding QuickCheck's *Args* are given on the system's webpage (Claessen 2000).

Shrinking failed test cases

When we find a counterexample to a correctness property, the program in question must be debugged. As previously mentioned, this can be a difficult task, depending on the size of the counterexample. This is because the complexity of a program's execution trace is often proportional to the size of its input. Undoubtedly this affects how easily an error can be spotted. This insight is particularly relevant to property-based testing, as random data generation (and thus, random testing) often produces large counterexamples.

To address this, *shrinking* (Claessen and Hughes 2000a; Claessen 2012) is a useful technique in the context of property-based testing that frequently results in small counterex-

amples that greatly facilitate debugging. The basic idea behind 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 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:

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

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 hit ratio 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 2019), subject source code to a sophisticated optimisation pipeline while translating 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, op-

timising 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 2014a) is a microbenchmarking library that is used to measure the performance of Haskell code. It provides a framework for defining and executing benchmarks as well as 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 mandates that an expression is only evaluated when needed, Criterion provides mechanisms to '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

$$\begin{aligned} slowRev &:: [a] \to [a] \\ slowRev\ [\,] &= [\,] \\ slowRev\ (x : xs) &= slowRev\ xs +\!\!+ [x] \end{aligned}$$

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 lazy evaluation. 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, a reducible expression (redex) is overwritten with its result 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 redex 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 execute *Benchmarkable*s directly. 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 2019), 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 2014a). 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)
```

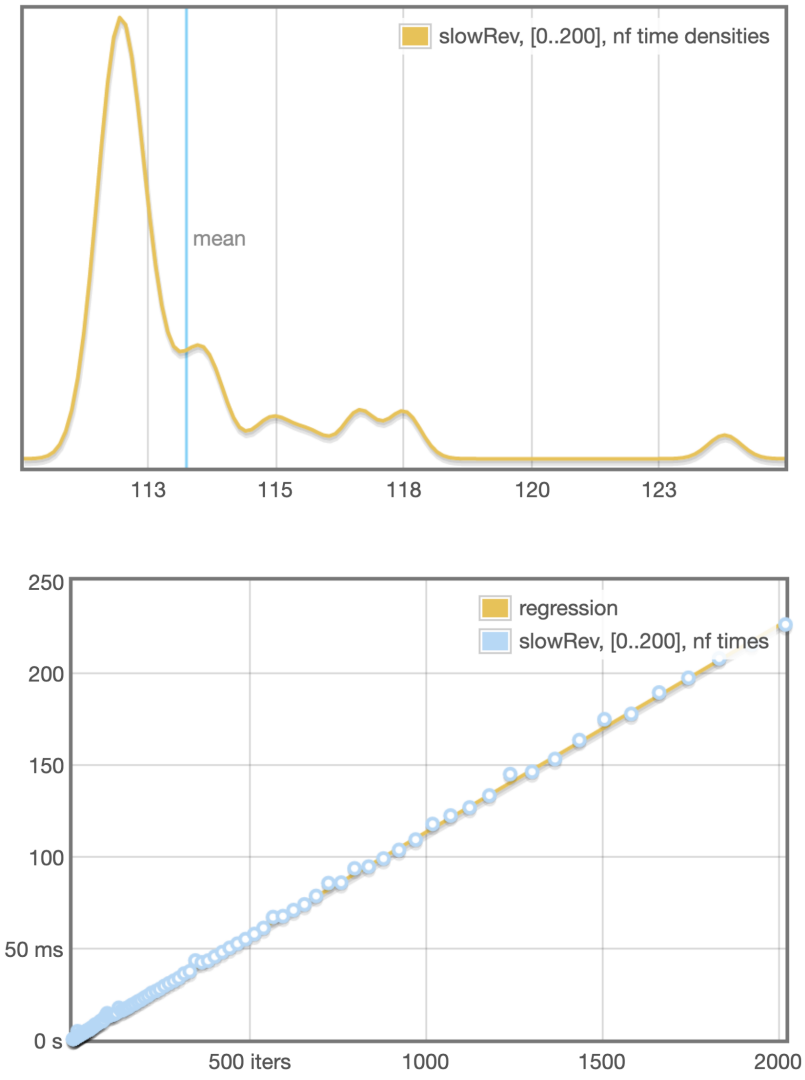
This information is printed to the console when a *Benchmark* is executed by *defaultMain*. It reveals that the *mean* time taken to evaluate the result of *slowRev* $[0\,..\,200]$ to normal form across all measured runs is 128.3 microseconds. The *time* predicted by Criterion's linear regression is 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 additional 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:

$$\begin{array}{lcl} [\![double\ (double\ x)]\!] & = & [\![\quad double\ (x\ +\ x) \quad]\!] \\ [\![double\ (double\ x)]\!] & = & [\![double\ x\ +\ double\ x]\!] \end{array} \quad \Longleftarrow \quad [\![double\ x]\!] \ = \ [\![x\ +\ x]\!]$$

However, reasoning about program definitions in this manner is somewhat indirect. In particular, equating the values of expressions shifts our domain of discourse from the syntactic level of a programming language to the semantic level of its respective denotations. Given that the defining equations of programs in pure functional languages can be directly interpreted as logical equalities, it is more convenient to remain in the former domain.

In fact, referential transparency allows us to do just that: in effect, it 'lifts' equality at the semantic level to the syntactic level. A referentially transparent language in which programs are defined can thus also be the language in which to express and reason about properties of those programs. Consequently, we needn't concern ourselves with a particular semantic domain. Instead, we can adequately reason about pure functional programs by directly transforming their defining expressions, for example, by substituting equals for equals. To this end, we can omit the valuation function $[\![\cdot]\!]$ and simply state that:

$$\begin{array}{lcl} double\ (double\ x) & = & double\ (x\ +\ x) \\ double\ (double\ x) & = & double\ x\ +\ double\ x \end{array}$$

Equational proofs

Proving correctness properties of programs often requires constructing equational proofs. Such proofs equate the left-hand side of a given proof statement with the right-hand side through a sequence, or chain, of proof steps. For example, recall from section 2.1.1 the property stating that naively reversing a singleton list is an identity operation:

$$slowRev\ [x] \quad = \quad [x]$$

A proof of this property in Haskell notation is as follows:

$$\begin{array}{ll} & slowRev\ [x] \\ = & \quad \{\ \text{syntactic sugar}\ \} \end{array}$$

$$
\begin{aligned}
& \mathit{slowRev}\ (x\ :\ [\,]) \\
= \quad & \{\ \text{unfold the definition of } \mathit{slowRev}\ \} \\
& \mathit{slowRev}\ [\,]\ +\!\!+\ [x] \\
= \quad & \{\ \text{unfold the definition of } \mathit{slowRev}\ \} \\
& [\,]\ +\!\!+\ [x] \\
= \quad & \{\ \text{unfold the definition of } (+\!\!+)\ \} \\
& [x]
\end{aligned}
$$

The example above is typical of an equational-style proof. In particular, steps of reasoning are aligned vertically and include a hint or justification in braces $\{\ \cdots\ \}$. In this instance, the first proof step involves desugaring Haskell's list syntax and the remainder of the steps involve rewriting subexpressions using the definitions of functions. Functions are *unfolded* when their names and arguments are replaced by their definitions; applying the inverse transformation is called *folding*. By the transitivity property of equality, we can thus conclude that *slowRev* $[x]$ is equal to $[x]$ *by definition*.

Remark. As per the work of Farmer et al. (2012), we refer to an equational proof as 'semi-formal' if it is constructed using pen-and-paper reasoning, or if it has not been certified using a proof assistant such as Agda (Norell 2008).

Inductive reasoning

Unfolding and folding the definitions of functions is not sufficient to prove correctness properties of programs in the general case. For example, two programs cannot be proved *existentially equal* without (notionally) demonstrating a correspondence between the results obtained from applying them to *all* possible inputs in their domains, which often have infinite cardinality. We must, therefore, appeal to stronger mathematical proof techniques.

One such technique commonly used alongside equational reasoning is *structural induction*, which is both useful and arises precisely because most interesting programs and datatypes are defined recursively. A brief overview of induction is given below, along with an example inductive proof. For a detailed review, we refer readers to one of the many clas-

sic texts (Burstall 1969). Nonetheless, we note that the essential property of pure functional programs that enables the use of induction is, again, referential transparency.

Every recursively defined datatype gives rise to an induction principle. Informally, in the case of lists, to prove a property P for all finite (total and terminating) lists xs using induction, we must: (a) prove that $P\ [\,]$ holds; (b) assuming $P\ xs$, prove that $P\ (x : xs)$ holds. For example, the second property of the *slowRev* function introduced in section 2.1.1

$$slowRev\ (xs\ +\!\!+\ ys) \quad = \quad slowRev\ ys\ +\!\!+\ slowRev\ xs$$

namely that *slowRev* contravariantly distributes over $(+\!\!+)$, is a *propositional equality* that can be proved using induction on lists, as follows:

$$\begin{array}{ll}
& slowRev\ ([\,]\ +\!\!+\ ys) \\
= & \quad \{\ \text{unfold the definition of } (+\!\!+)\ \} \\
& slowRev\ ys \\
= & \quad \{\ \text{right identity of } (+\!\!+)\ \} \\
& slowRev\ ys\ +\!\!+\ [\,] \\
= & \quad \{\ \text{fold the definition of } slowRev\ \} \\
& slowRev\ ys\ +\!\!+\ slowRev\ [\,]
\end{array}$$

$$\begin{array}{ll}
& slowRev\ ((x\ :\ xs)\ +\!\!+\ ys) \\
= & \quad \{\ \text{unfold the definition of } (+\!\!+)\ \} \\
& slowRev\ (x\ :\ (xs\ +\!\!+\ ys)) \\
= & \quad \{\ \text{unfold the definition of } slowRev\ \} \\
& slowRev\ (xs\ +\!\!+\ ys)\ +\!\!+\ [x] \\
= & \quad \{\ \text{inductive hypothesis}\ \} \\
& (slowRev\ ys\ +\!\!+\ slowRev\ xs)\ +\!\!+\ [x] \\
= & \quad \{\ \text{associativity of } (+\!\!+)\ \} \\
& slowRev\ ys\ +\!\!+\ (slowRev\ xs\ +\!\!+\ [x]) \\
= & \quad \{\ \text{fold the definition of } slowRev\ \} \\
& slowRev\ ys\ +\!\!+\ slowRev\ (x\ :\ xs)
\end{array}$$

The proof inducts on the structure of append's first argument, xs. On the left-hand side is the base case, where xs is empty. Here the proof proceeds by unfolding and folding definitions. It also requires the use of a lemma, which states that $xs\ +\!\!+\ [\,] = xs$. This property is known as append's right identity law and is also typically proved using induction.

On the right-hand side is the inductive case, where xs is non-empty. Here the proof begins by unfolding definitions and then appeals to the inductive hypothesis. Another lemma is required to reassociate the two appends. It states that $(xs\ +\!\!+\ ys)\ +\!\!+\ zs = xs\ +\!\!+$

$(ys \mathbin{+\!\!+} zs)$, which formalises that append is an associative operator. Again, this property can be proved using induction. Folding the definition of *slowRev* completes the proof.

Fast and loose reasoning

We have seen how basic notions of equality between expressions can be used to construct proofs of program equality, and how proof techniques from mathematics such as induction can strengthen this approach to reasoning about correctness properties of programs. Indeed, equational reasoning has formed the basis of a whole line of research in which programs are transformed in order to improve their performance, proved correct in regard to high-level specifications, and even derived from such specifications. Nonetheless, there is often an underlying issue with such reasoning when it is performed in practice: in real-world lazy programming languages, such as Haskell, these equational properties are seldom totally correct. This is because they often do not hold in the presence of the undefined value, $\bot$, which represents non-termination and other error conditions.

For example, in order to formally show that *slowRev* $(xs \mathbin{+\!\!+} ys)$ and *slowRev* $ys \mathbin{+\!\!+}$ *slowRev* xs are existentially equal in Haskell, our inductive proof must include another base case that, in general, demonstrates $P\ \bot$ holds. However, it is not true that *slowRev* $(\bot \mathbin{+\!\!+} ys) =$ *slowRev* $ys \mathbin{+\!\!+}$ *slowRev* $\bot$. On the contrary, the latter expression is more defined than the former, which can be seen by partially computing the results of both definitions:

	slowRev $(\bot \mathbin{+\!\!+} ys)$		*slowRev* $ys \mathbin{+\!\!+}$ *slowRev* $\bot$
$=$	{ $(\mathbin{+\!\!+})$ strict in its first argument }	$=$	{ *slowRev* strict in its argument }
	slowRev $\bot$		*slowRev* $ys \mathbin{+\!\!+} \bot$
$=$	{ *slowRev* strict in its argument }		
	$\bot$		

On the left-hand side, we see that *slowRev* $(\bot \mathbin{+\!\!+} ys)$ will never produce a result. In contrast, on the right-hand side, we see that *slowRev* $ys \mathbin{+\!\!+}$ *slowRev* $\bot$ can produce an output, which is at most the reversal of ys given that $(\mathbin{+\!\!+})$ is lazy in its second argument.

The fact that the above proof fails in the presence of $\bot$ means that the property regarding

slowRev is not valid for *infinite* lists. Given that one of the primary advantages of Haskell's call-by-need semantics is that it affords being able to program with (potentially) infinite data structures in a straightforward manner, such a failure appears problematic at face value. However, despite proof failures in the presence of infinite values and non-termination, functional programmers happily derive programs from specifications by rewriting according to partially correct equalities of this kind. That is, they simply overlook $\perp$ and are seemingly confident that, at worst, they are only affecting the definedness of their programs in the less common and more obscure cases, which often do not arise in practice.

For the most part, sacrificing a degree of mathematical rigour for pragmatism in this manner is worthwhile. For example, Danielsson et al. (2006) show that if two closed expressions have the same semantics for a total programming language, then they have related semantics for a non-total (that is, conventional) programming language, and because of this it is impossible to transform a terminating program into a nonterminating one by rewriting expressions according to partially correct equational laws.

In summary, such *fast and loose* equational reasoning (Danielsson et al. 2006) can be of great benefit to the entire functional programming community, from professional compiler writers to amateur theorem provers. Nonetheless, it is important to remember that overlooking the existence of $\perp$ when proving properties about programs written in lazy functional languages sacrifices a degree of mathematical rigour, and hence, fundamentally weakens a formal (totally correct) proof to a *semi-formal* one.

2.3.2 Inequational reasoning

We have seen a number of ways in which two expressions can be equal. For example, they can be defined to be equal, or they can compute the same results. Both of these notions of equality capture an extrinsic property of programs, namely *what* each program computes, which is referred to as its meaning in the context of denotational semantics.

When reasoning about efficiency, we are not just concerned with the results of a program, but also *how* a program computes its results. This can be seen as an intrinsic property, which may quantify, for example, the number of steps taken during its evaluation. Such properties are, therefore, typically captured by an operational semantics.

From an operational point of view, there are many ways in which (denotationally equal) programs can be unequal. In the work of this thesis, we are primarily concerned with their unequal utilisation of system resources. For example, on many occasions, we look to improve the time performance of expressions by applying meaning-preserving transformations. In chapter 5, we also touch on improving memory usage.

Notions of *operational inequality*, though expressed using different relations, are mostly comparable to standard numerical inequalities, such as less than or equal $\leqslant$ and greater than or equal $\geqslant$, both in the properties that they satisfy and how they are used in practice. We omit the details of any such inequality relation here, as they are commonly tied to a specific operational semantics and thus require thorough treatment. Instead, we refer readers to Sands' (1997) overview of *improvement theory*, which provides a detailed introduction to an inequational theory used to reason about efficiency in call-by-need languages such as Haskell. In fact, our contributions presented in chapter 4 are based on this work.

Inequational proofs

As we have yet to introduce any notions of operational inequality, we highlight the basics of inequational proofs involving standard numerical inequalities. Given that these inequalities are comparable to many notions of operational inequality, this material is foundational to understanding all inequational proofs presented in chapter 4 onwards.

In contrast to equational reasoning, which only utilises the equality relation, we must take care when using different relations to prove inequalities. To provide some intuition, consider the *triangle inequality*, which states that for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side. In Euclidean geometry, the triangle inequality is usually stated as

$$\|x \ + \ y\| \quad \leqslant \quad \|x\| \ + \ \|y\|$$

and proved as follows, assuming that $\|x + y\| \geqslant 0$ and $\|x\| + \|y\| \geqslant 0$:

$$\|x \ + \ y\|^2$$

$$
\begin{aligned}
&= \quad \{\ \|x\| \ = \ \sqrt{\langle\ x,\ x\ \rangle}\ \} \\
&\quad \langle\ x\ +\ y,\ x\ +\ y\ \rangle \\
&= \quad \{\ \text{expansion, inner product is symmetric}\ \} \\
&\quad \langle\ x,\ x\ \rangle\ +\ \langle\ y,\ y\ \rangle\ +\ 2\langle\ x,\ y\ \rangle \\
&= \quad \{\ \|x\| \ = \ \sqrt{\langle\ x,\ x\ \rangle}\ \} \\
&\quad \|x\|^2\ +\ \|y\|^2\ +\ 2\langle\ x,\ y\ \rangle \\
&\leqslant \quad \{\ \langle\ x,\ y\ \rangle\ \leqslant |\ \langle\ x,\ y\ \rangle\ |\ \} \\
&\quad \|x\|^2\ +\ \|y\|^2\ +\ 2\ |\ \langle\ x,\ y\ \rangle\ | \\
&\leqslant \quad \{\ \text{Cauchy–Schwarz inequality}\ \} \\
&\quad \|x\|^2\ +\ \|y\|^2\ +\ 2\|x\|\|y\| \\
&= \quad \{\ \text{factorisation}\ \} \\
&\quad (\|x\|\ +\ \|y\|)^2
\end{aligned}
$$

The details of the above proof are not particularly important here. However, the relationship between the less than or equal and equality relations is crucial to understanding how to construct valid inequational proofs. In this case, we see that proving properties involving $\leqslant$ may include substituting equals for equals using $=$. This is because the equality relation is a proper subset of the less than or equal relation: $= \subset \leqslant$. In other words, $x = y$ implies that $x \leqslant y$. Consequently, steps of reasoning in the above proof involving $=$ can be replaced with the same steps using $\leqslant$ and the proof still holds.

Similarly to equality, the less than relation $<$ is also a proper subset of the less than or equal relation: $< \subset \leqslant$. Hence, substitutions can be made using the $<$ operator when reasoning about properties defined using $\leqslant$ and such calculations remain valid. On the other hand, the reverse implication does not hold in both cases. That is, $x \leqslant y \not\Rightarrow x = y$ and $x \leqslant y \not\Rightarrow x < y$. In other words, $\leqslant$ is not a proper subset of either $=$ or $<$. As a result, it is not possible to make valid substitutions (or rewrites) using the $\leqslant$ operator when reasoning about properties defined in terms of $=$ or $<$.

For familiar order relations, such as $<$ and $\leqslant$, the main points raised in the above discussion might appear obvious. However, the order relations used to reason about the

efficiency of expressions in practice are usually somewhat more intricate. Typically, they are defined contextually, according to an extended notion of observational equivalence (see section A.1.2). As such, it is easy to misinterpret or even overlook the inclusion ordering on such relations. Nonetheless, doing so can lead to incorrect calculations. This is one of the key insights that led to the work presented in chapter 4, which, among other things, aims to safeguard against this kind of operator misuse.

Inductive reasoning

Methods of induction can be used alongside inequational properties to reason about program efficiency, just as with equational reasoning for program correctness. However, such methods are typically more intricate than simple structural induction introduced previously (and also highly specialised). For example, multiple induction premises may be required, each defined at a contextual level. This reflects the fact that most notions of operational inequality entail contextual definitions. We omit the details of such induction methods for brevity but note that one particular method is introduced in chapter 4.

2.4 Formal reasoning

In the previous section, we exemplified fast and loose reasoning by overlooking Haskell's undefined value, $\bot$, when partially proving that $\mathit{slowRev}\ (\mathit{xs} +\!\!+ \mathit{ys}) = \mathit{slowRev}\ \mathit{ys} +\!\!+ \mathit{slowRev}\ \mathit{xs}$. Despite Danielsson et al. (2006) showing that this style of equational reasoning can be 'morally correct', there are many situations where ignoring undefined/infinite values and other subtle semantic properties (either intentionally or unintentionally) is not acceptable. Indeed, there is no shortage of publications describing how *strictly* formal methods play a crucial role in the design and implementation of safety-critical systems (Bowen and Stavridou 1993). In this setting, semi-formal proofs are an insufficient means of demonstrating the accuracy, consistency, and correctness of such systems.

It can be challenging to describe formal reasoning as an activity that is distinct from other kinds of (semi-formal) reasoning. Even a proof that most would agree is formal and mathematically rigorous often does not fully detail the underlying knowledge and ex-

pertise that gives the argument a precise meaning, let alone allows it to be judged as 'correct' (Lapets 2010). Throughout this thesis, we view formal reasoning as any reasoning activity that involves the manipulation of concepts according to a consistent set of rules. In particular, these concepts are assumed to be those that are traditionally employed within the fields of pure/discrete mathematics and theoretical computer science.

While this description of formal reasoning can be interpreted simply as the act of 'writing a proof in a formal language', reformulating semi-formal equational reasoning (such as that for *slowRev*) in a formal manner is often far from simple. Typically, there is a notable gap between the more conventional languages used for everyday programming and those used for formal proofs. As such, the conventional language must typically be modelled either partially or wholly in the formal language, and the proof translated. Moreover, if the goal of the reasoning is to transform a program (rather than just verifying a property), then the resulting program must be translated back before it can be compiled and executed.

The choice of formal language in which to transliterate a semi-formal proof is key in determining how its formal counterpart can be expressed, and, moreover, how it can be proved. A popular choice among the relevant literature is Martin-Lof's intuitionistic type theory (1984), or Type Theory for short, which is a formal system that serves not only as a foundation for constructive mathematics but also as a dependently typed programming language. Type Theory is based on the Curry-Howard isomorphism, which relates dependently typed programs with mathematical proofs. In particular, the introduction of dependent types, which are types that depend on the values of other types, allows every proof in predicate logic to be represented by a term of a suitably typed lambda calculus.

The expressiveness of dependent types not only enables programmers to define properties that cannot be captured by conventional type systems (such as Haskell's), but their connection to predicate logic allows such properties to be formally verified. This connection, known broadly as *Propositions as Types* (Curry 1934; Wadler 2015), has inspired the ever-increasing list of dependently typed programming languages (McBride and McKinna 2004; Norell 2008; Bertot and Castéran 2013; Brady 2013), which support computer-aided formal reasoning and allow users to write *verified* functional programs (Stump 2016).

As with most things, the expressiveness of dependent types also comes at a cost: type

checking dependent types is intricate, and, in the general case, undecidable. In short, this is because dependent types remove the isolation between values and types that exists in simply-typed languages. Consequently, the general halting problem is lifted to the level of the type checker. In practice, dependently typed programming languages such as Agda (Norell 2008) provide expressive program annotations that can be used to guide the type checker manually. For example, Agda is a total language (as all formal languages must be) and so a user may have to show that their recursive program terminates.

With respect to equational reasoning, having to be explicit about all the details of a proof formalised in a language like Agda may be seen as advantageous, especially in a learning environment. However, more generally, having to show that even the simplest of invariants hold, such as basic arithmetic equalities, can become tedious. This has led to the development of restricted forms of dependent types that sacrifice some expressiveness for the sake of decidable and, in some cases, low-complexity type checking. Such systems are often called *lightweight dependent types* or *refinement types*.

Haskell's type system has recently been extended with refinement types by a popular framework called *Liquid Haskell*, which enables formal reasoning about Haskell programs 'within the language itself'. In the remainder of this section, we introduce the key concepts of refinement types by highlighting their applications in Liquid Haskell.

2.4.1 Formal reasoning in Liquid Haskell

We note that the first chapter of (Vazou 2016) inspired this background section. Readers are also referred to this work for a more thorough introduction to Liquid Haskell.

Liquid Haskell can be seen as a formal verification system for (total) Haskell programs. It takes as input Haskell source code, annotated with correctness specifications in the form of refinement types, and checks whether the code satisfies the specifications. The annotations provided are designed to be as expressive as possible while ensuring that type checking remains decidable. This allows many properties to be proved automatically.

Liquid Types

In Liquid Haskell, a Liquid Type (Rondon, Kawaguci, and Jhala 2008) has the form $\{\ v : a\ \mid\ e\ \}$, where a is a standard Haskell type and e is a boolean expression that may refer to the variable v and any free variables occurring in a program of that type. This type represents the set of all values u of type a such that the expression $e[u/v]$ evaluates to *true*. For example, $\{\ v : \mathit{Int}\ \mid\ v\ >\ 0\ \}$ is the type of all integers greater than zero. This is called a *refinement type* of the type *Int*, where the *value variable* v is assumed to range over all possible values of the refinement type.

Liquid Types also support dependent function types, where a function's result type can depend on the values of its arguments. Furthermore, the type of an argument can also depend on the values of any preceding arguments. For example, the following function *get* takes as arguments a polymorphic array of type $\mathit{Array}\ a$ and an index within the expected range, and gets the array element $v : a$ at that position:

$$\mathit{get}\ ::\ \mathit{arr} : \mathit{Array}\ a\ \rightarrow\ \{\ i : \mathit{Int}\ \mid\ 0\ \leqslant\ i\ <\ \mathit{length}\ \mathit{arr}\ \}\ \rightarrow\ \{\ v : a\ \mid\ v\ =\ \mathit{arr}[i]\ \}$$

Another example is *max*, which computes the maximum value of its two arguments:

$$\mathit{max}\ ::\ x : \mathit{Int}\ \rightarrow\ y : \mathit{Int}\ \rightarrow\ \{\ v : \mathit{Int}\ \mid\ v\ \geqslant\ x\ \wedge\ v\ \geqslant\ y\ \}$$

Liquid Type theory provides typing rules (Vazou et al. 2014) expressing the relationships that must hold between the types to ensure that programs are well-typed. The most important relationship is that of subtyping: intuitively, a Liquid Type a_1 is a subtype of another type a_2, expressed as $a_1 \preceq a_2$, if the set of values of a_1 is a subset of the values of a_2. In logical terms, if $a_1 = \{\ v : a\ \mid\ e_1\ \}$ and $a_2 = \{\ v : a\ \mid\ e_2\ \}$, then this is equivalent to showing that the formula $e_1 \Rightarrow e_2$ is universally valid.

In order to verify a subtyping query of the above form, refinement type systems such as Liquid Haskell generate verification conditions (VCs), which are logical formulae that stipulate—under a number of assumptions involving the given typing environment Γ—the refinement in the subtype implies the refinement in the super-type: $[\![\Gamma]\!] \wedge e_1 \Rightarrow e_2$.

Generally speaking, refinement type systems are engineered so that, unlike with full dependent types, refinements only includes formulas from decidable logics. Liquid Haskell

uses the quantifier-free logic of linear arithmetic and uninterpreted functions (QF-EUFLIA). Doing so allows verification conditions (and hence subtyping queries) to be efficiently and automatically verified by an SMT solver (Moura and Bjørner 2008). In this manner, refinement type systems set themselves apart from dependent type systems.

To generate the subtyping queries necessary to verify program properties, the type signature of the function being checked is typically the only additional user-input required, that is, the refinement types of the arguments and that of the result. In this signature, the user must express any dependence between each argument and also how the result depends on the arguments. This amounts to giving the function preconditions and postconditions, which is known as a *specification* in program verification terminology.

Specifications

In Liquid Haskell, refinement type specifications are provided by users in their input files (one or more Haskell module) as annotations of the form $\{-@ \cdots @-\}$. Such annotations are regarded as comments by GHC and are thus ignored. The output of Liquid Haskell is simply *SAFE* in the case where every annotated function in the input file type checks. Otherwise, type errors are reported at different source locations, indicating any (partially) inferred types and the subtyping constraints that have been violated.

A range of Liquid Haskell specifications are exemplified below.

Refining values. The following code defines a Haskell integer x that equals one.

$$\{-@\ x\ ::\ \{\ v : Int\ \mid\ v\ =\ 1\ \}\ @-\}$$
$$x\ ::\ Int$$
$$x\ =\ 1$$

In turn, x's refinement type specification states that its value is indeed one. Saving this code to a file in the current directory called *Example.hs* and passing it to Liquid Haskell

```
> liquid ./Example.hs
```

gives the following result: *SAFE*. On the other hand, if we modify the contents of *Example.hs*, setting $x = 0$, then Liquid Haskell returns the following error

```
  3    | x = 0
^^^^
  VV : { v : GHC.Types.Int | v = 0 }
not a subtype of Required type
  VV : { VV : GHC.Types.Int | VV = 1 }
```

which essentially says that it cannot prove $0 = 1$.

Type aliases. It is often convenient to define abbreviations for particular refinement predicates. For example, we can define an alias for non-zero integers as follows:

```
{-@ type NonZero = { v : Int | v /= 0 } @-}
```

It is then straightforward to use this alias as part of other specifications:

```
{-@ x, y, z :: NonZero @-}
x = 1 :: Int
y = 2 :: Int
z = 3 :: Int
```

Preconditions. Preconditions of functions are given by refining the types of their arguments. For example, we can avoid runtime errors when dividing by zero

```
{-@ safeDiv :: Int -> NonZero -> Int @-}
x `safeDiv` y = x `div` y
```

or when taking the head of a list with no elements:

```
{-@ type NonEmpty a = { xs : [a] | length xs > 0 } @-}
{-@ safeHead :: NonEmpty a -> a @-}
safeHead xs = head xs
```

In both cases, *callers* of *safeDiv* and *safeHead* must prove that the arguments they provide to each function satisfy the corresponding specification.

Postconditions. Postconditions of functions are given by refining the type of their results. For example, we can ensure that the absolute value of an integer is always positive

```
{-@ type Nat = { x : Int | x >= 0 } @-}
```

```
{−@ abs :: Int → Nat @−}
```

that the length of a list is preserved when it is reversed

```
{−@ reverse :: xs : [a] → { zs : [a] | length zs = length xs } @−}
```

and that the length of two lists appended together is the sum of their individual lengths:

```
{−@ (++) :: xs : [a] → ys : [a] →
      { zs : [a] | length zs = length xs + length ys } @−}
```

Postconditions must be satisfied by the definitions of the corresponding functions. For example, the following definition of *abs* returns only positive results:

```
abs n
  | n ⩾ 0     = n
  | otherwise = 0 − n
```

Liquid Haskell can analyse both guarded expressions to verify that this is true. Other, more sophisticated postconditions (and indeed, preconditions) can be expressed using the system. In chapter 5, we present a number of such examples, including 'sortedness' constraints on lists. Alternatively, readers are referred to (Vazou 2016) for further examples.

Measures. To allow Haskell functions to appear in refinement types, they are *reflected* into the system's underlying logic of QF-EUFLIA. Liquid Haskell provides a simple mechanism to support this for a particular class of functions called *measures*.

Measures are unary functions whose parameters must be algebraic datatypes. Their definitions must contain a single equation for each applicable data constructor; and furthermore, they can only be defined using arithmetic functions and/or other measures. For this restricted class of functions, refinements can be checked automatically.

Previously, we used the *length* function to prevent runtime errors when taking the *head* of an empty list. The *length* function is indeed a measure:

```
{−@ measure length @−}
{−@ length :: [a] → Int @−}
length []       = 0
length (_ : xs) = 1 + length xs
```

Totality

As we mentioned previously, any language used for formal reasoning must be total. Without this characteristic, the underlying logic of the language becomes inconsistent, which allows arbitrary statements (including those that are false) to be proved. To this end, one of the main differences between conventional Haskell and Liquid Haskell is that all functions defined in Liquid Haskell must be provably total.

In the relevant literature, *total functional programming* (Turner 2004) precludes the use of partial functions, which are only defined on a subset of their domains. Partiality arises quite naturally in programs that are defined using pattern matching by way of non-exhaustive patterns. For example, the *head* function (as defined in the Haskell *Prelude*)

$$
\begin{aligned}
&\mathit{head} :: [a] \rightarrow a \\
&\mathit{head}\ (x : \_) = x
\end{aligned}
$$

does not provide a case alternative for when its input list is empty.

Any non-terminating Haskell function is by definition also partial, as its result, $\bot_a$ for any type a, is not a well-defined value of type a in the traditional mathematical sense.

To ensure that all pattern matches are exhaustive, Liquid Haskell extends GHC's pattern completion mechanism (Vazou 2016). In particular, pattern cases are checked in conjunction with preconditions to verify full coverage of input domains. For example, the *safeHead* function defined above only requires a case alternative for non-empty input lists due to its refinement precondition. In turn, to ensure that all functions terminate, Liquid Haskell uses structural or semantic termination checking.

Structural termination. Structural termination detects common recursion patterns in which the argument to a recursive call is a direct or indirect subterm of the given function's argument. This was exemplified in the definition of the *length* function above. If a function has multiple arguments, then at least one argument must get structurally smaller, and the size of all arguments before it (lexicographically) must remain unchanged.

Semantic termination. When structural termination checking fails, Liquid Haskell attempts to prove termination with a user-defined semantic argument: an expression that

calculates a natural number from the given function's arguments, which must decrease in each recursive call. For example, the recursive calls to the following *merge* function do not satisfy the preconditions for structural termination:

```
merge :: [a] → [a] → [a]
merge (x : xs) (y : ys)
  | x ⩽ y     = x : merge xs (y : ys)
  | otherwise = y : merge (x : xs) ys
```

In particular, the length of each argument list fails to decrease in *both* recursive calls. However, the sum of the lengths of both lists always decreases:

```
{-@ merge :: xs : [a] → ys : [a] → [a] / [length xs + length ys] @-}
```

Semantic termination has two main benefits over structural termination. Firstly, not every function is naturally structurally recursive: adding parameters to achieve this can obscure a function's definition. Secondly, termination is checked by an SMT solver, and so semantic termination conditions can utilise refinement properties if necessary. Despite this, structural termination checking is automated and works for most simple cases.

Equational reasoning

Up until this point, we have seen how Liquid Haskell uses refinement types to automatically prove preconditions and postconditions. These properties can be expressed using measures such as *length* and *nonEmpty*, which are Haskell functions that can be reflected into the underlying logic of Liquid Haskell while retaining fully automatic reasoning.

Haskell functions that are not measures can also be reflected into the logic. However, for technical reasons it is not possible to prove properties about arbitrary functions in a fully automatic way using Liquid Haskell (Vazou et al. 2018). The system still allows us to reason about such functions, but typically we must supply proofs of any desired properties ourselves, which Liquid Haskell then verifies. Properties that cannot be proved automatically by Liquid Haskell are called *deep* program properties. In turn, manual proofs of deep properties involve, for example, unfolding and folding function definitions and the use of mathematical induction. That is, they involve *formal* equational reasoning.

As a concrete example, consider the monoid laws for Haskell's list-appending operator:

$$
\begin{array}{rcll}
[\,] \mathbin{+\!\!+} ys & = & ys & \textit{left identity} \\
xs \mathbin{+\!\!+} [\,] & = & xs & \textit{right identity} \\
(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs & = & xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) & \textit{associativity}
\end{array}
$$

Append does not satisfy the requirements of a Liquid Haskell measure but can still be *reflect*ed into the system's underlying logic as an 'uninterpreted function':

$$
\begin{array}{l}
\{-@\ \textit{reflect}\ (\mathbin{+\!\!+})\ @-\} \\
\{-@\ (\mathbin{+\!\!+})\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow \\
\qquad \{\ zs : [a]\ \mid\ \textit{length}\ zs\ =\ \textit{length}\ xs\ +\ \textit{length}\ ys\ \}\ @-\} \\
[\,] \qquad \mathbin{+\!\!+}\ ys\ =\ ys \\
(x\ :\ xs) \mathbin{+\!\!+}\ ys\ =\ x\ :\ (xs \mathbin{+\!\!+}\ ys)
\end{array}
$$

In contrast to a measure, which is fully translated into QF-EUFLIA, only the *result* of append is translated when it is applied to given arguments xs and ys of type $[a]$. Enforcing this restriction on all non-measures ensures that refinement type checking remains decidable.

To verify append's monoid laws, we must specify each proof statement in a refinement type specification and then manually define proof terms that inhabit each type. Such theorems are expressed as refinements of the unit type. For example, the following type

$$\{-@\ \textit{appLeftId}\ ::\ ys : [a]\ \rightarrow\ \{\ p : ()\ \mid\ [\,] \mathbin{+\!\!+}\ ys\ =\ ys\ \}\ @-\}$$

can be seen as a mapping from any list ys to a proof that $[\,] \mathbin{+\!\!+} ys = ys$. Intuitively, this corresponds to universally quantifying over ys in predicate logic: $\forall ys$. In fact, we may omit the binding $p : ()$ to make the correspondence clearer:

$$\{-@\ \textit{appLeftId}\ ::\ ys : [a]\ \rightarrow\ \{\ [\,] \mathbin{+\!\!+}\ ys\ =\ ys\ \}\ @-\}$$

It is easy to see that this property follows immediately from the definition of append:

$$
\begin{array}{l}
\textit{appLeftId}\ ys \\
\quad =\ [\,] \mathbin{+\!\!+}\ ys \\
\quad ==.\ ys \\
\quad ***\ \textit{QED}
\end{array}
$$

Here, the body of the proof term *appLeftId* is reminiscent of the pen-and-paper proofs exemplified in section 2.3. In general, equational reasoning formalised in Liquid Haskell is highly reminiscent of semi-formal reasoning carried out with pen and paper. However, the advantage of such formalised reasoning is precisely that: it is formally verified.

Append's right identity law can be expressed similarly:

$$\{-@\ appRightId\ ::\ xs : [a]\ \rightarrow\ \{\ xs\ +\!\!+\ [\,]\ =\ xs\ \}\ @-\}$$

However, this property does not follow immediately from the definition of append. Instead, as append is recursively defined on *xs*, we can use induction to prove it. A simple inductive proof in Liquid Haskell—requiring no auxiliary lemmas—can be fully automated using the system's *Proof by Logical Evaluation* (PLE) technique (Vazou et al. 2017). In short, PLE unfolds functions for as many steps as is required to provide to the underlying logic all equations necessary to prove a property. In this instance, the base case of *appRightId* is fully automated by PLE, while in the inductive case we must make a recursive call to appeal to the inductive hypothesis, but the rest is taken care of by PLE:

$$\begin{aligned}
&\{-@\ ple\ appRightId\ @-\}\\
&appRightId\ [\,]\ \ \ \ \ \ \ \ =\ ()\\
&appRightId\ (_\ :\ xs)\ =\ appRightId\ xs
\end{aligned}$$

The final monoid law for append captures the operator's associativity property:

$$\begin{aligned}
&\{-@\ appAssoc\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow\ zs : [a]\ \rightarrow\\
&\qquad \{\ xs\ +\!\!+\ (ys\ +\!\!+\ zs)\ =\ (xs\ +\!\!+\ ys)\ +\!\!+\ zs\ \}\ @-\}
\end{aligned}$$

Reassociating two append's according to this property is a common trick used to improve the performance of recursive programs that construct lists in their results. In fact, this property appears on a number of occasions throughout this thesis. To prove it, we may also induct over the structures of *xs*. We omit the details of this proof in favour of a more familiar example, which we spell out in full: $slowRev\ (xs +\!\!+ ys) = slowRev\ ys +\!\!+ slowRev\ xs$.

$$\begin{aligned}
&\{-@\ reflect\ slowRev\ @-\}\\
&\{-@\ slowRev\ ::\ xs : [a]\ \rightarrow\ \{\ zs : [a]\ \mid\ length\ zs\ =\ length\ xs\ \}\ @-\}\\
&slowRev\ [\,]\ \ \ \ \ \ \ \ =\ [\,]\\
&slowRev\ (x\ :\ xs)\ =\ slowRev\ xs\ +\!\!+\ [x]
\end{aligned}$$

Recall the naive list-reversing function, which is defined above in Liquid Haskell. Notice that the system automatically proves that the length of the input list *xs* is preserved, which is specified as a refinement of the result *zs* (a postcondition). Previously, we used pen-and-paper reasoning to show that *slowRev* contravariantly distributes over append, that is, $slowRev\ (xs +\!\!+ ys) = slowRev\ ys +\!\!+ slowRev\ xs$. We now wish to formalise this proposition in Liquid Haskell, by defining a proof term for the following refinement type:

$$\{-@\ slowRevDistr\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow$$
$$\{\ slowRev\ (xs\ +\!\!+\ ys)\ =\ slowRev\ ys\ +\!\!+\ slowRev\ xs\ \}\ @-\}$$

Just as before, we use induction to prove this property, considering cases where *xs* is empty and non-empty. In the base case, the proof term is defined as follows:

$$\begin{aligned}
&slowRevDistr\ [\,]\ ys \\
&\quad =\ slowRev\ ([\,]\ +\!\!+\ ys) \\
&\quad ==.\ slowRev\ ys \\
&\quad\ ?\ appRightId\ (slowRev\ ys) \\
&\quad ==.\ slowRev\ ys\ +\!\!+\ [\,] \\
&\quad ==.\ slowRev\ ys\ +\!\!+\ slowRev\ [\,] \\
&\quad ***\ QED
\end{aligned}$$

When unfolding and folding the definitions of $(+\!\!+)$ and *slowRev* above, we needn't justify each step of reasoning because Liquid Haskell automatically proves definitional equality. On the other hand, when reasoning with the aid of an auxiliary theorem (utilising a different notion of equality) such as append's right identity law, $xs +\!\!+ [\,] = xs$, we must justify such reasoning by appealing to the proof of this theorem using the (?) operator. In doing so, we must instantiate the theorem's proof term, *appRightId*, using arguments that relate to its context of use: $appRightId\ (slowRev\ ys)$ maps to a proof that $slowRev\ ys +\!\!+ [\,] = slowRev\ ys$. Finally, note that equalities provide two rewrite rules: one in the left-to-right direction and the other right-to-left. We may freely rewrite in either direction.

In the inductive case, where *xs* is non-empty, the proof begins by unfolding definitions:

$$slowRevDistr\ (x : xs)\ ys$$

$$
\begin{aligned}
&= \mathit{slowRev}\ ((x : xs)\ +\!\!+\ ys) \\
&==. \mathit{slowRev}\ (x : (xs\ +\!\!+\ ys)) \\
&==. \mathit{slowRev}\ (xs\ +\!\!+\ ys)\ +\!\!+\ [x]
\end{aligned}
$$

At this point, the argument to *slowRev* is a subterm of the initial argument, and so the inductive hypothesis can be used. Due to the intimate correspondence between induction and recursion, justifying the use of the inductive hypothesis amounts to a recursive call:

$$
\begin{aligned}
&? \ \mathit{slowRevDistr}\ xs\ ys \\
&==. (\mathit{slowRev}\ ys\ +\!\!+\ \mathit{slowRev}\ xs)\ +\!\!+\ [x]
\end{aligned}
$$

We now have two appends associated to the left. These can be reassociated using *appAssoc*:

$$
\begin{aligned}
&? \ \mathit{appAssoc}\ (\mathit{slowRev}\ ys)\ (\mathit{slowRev}\ xs)\ [x] \\
&==. \mathit{slowRev}\ ys\ +\!\!+\ (\mathit{slowRev}\ xs\ +\!\!+\ [x])
\end{aligned}
$$

Finally, we fold the definition of *slowRev*

$$
\begin{aligned}
&==. \mathit{slowRev}\ ys +\!\!+ \mathit{slowRev}\ (x : xs) \\
&*** \ \mathit{QED}
\end{aligned}
$$

and use $(*** \mathit{QED})$ to complete the proof (see figure 2.1).

Summary

In this subsection, we have introduced refinement types in Liquid Haskell as a means of automatically proving preconditions and postconditions of programs. In addition, we have demonstrated that properties of arbitrary (reflected) programs can be proved using formalised equational reasoning. Like with pen-and-paper proofs, this reasoning is carried out manually. Unlike such semi-formal proofs, however, equational reasoning in Liquid Haskell is translated into the logic of QF-EUFLIA and formally verified by an SMT solver.

The proofs in this subsection, in particular, the proof of *slowRev*'s distributivity property, aim to demonstrate the strong correspondence between semi-formal proofs carried out using pen and paper, and the syntax of proof terms defined in Liquid Haskell. The paper introducing Liquid Haskell as a formal equational reasoning assistant states that:

> The correspondence is so close that we claim proving a property in Liquid Haskell can be just as easy as proving it on paper by equational reasoning—but the proof in Liquid Haskell is machine-checked! (Vazou et al. 2018)

Overall, our experience using Liquid Haskell for both equational and inequational reasoning (discussed subsequently in chapter 5) echoes this statement.

To finalise this background section, we clarify a number of points regarding the implementation of Liquid Haskell. Firstly, the $(==.)$ operator used to connect steps of equational reasoning in proof terms is defined in figure 2.1, along with the $(?)$ and $(***\ QED)$ functions and the relevant datatypes. In short, the type of $(==.)$ is refined to ensure that both of its arguments are equal; it returns its second argument to allow many steps to be chained together. This corresponds to $=$'s transitivity property. At first glance, the $(?)$ function appears to be a constant function for a given argument x. In reality, the refinements of the auxiliary $p : \mathit{Proof}$ term are combined with those of the term invoking $(?)$, which—as we've seen—enables rewriting according to p's theorem. The $(***\ QED)$ function signals the end of a proof by returning unit to ensure that the proof term is type-correct.

Our second clarification concerns the 'distance' between reasoning about conventional Haskell programs with pen and paper and formalised reasoning about Liquid Haskell programs. One of the main advantages of Liquid Haskell is that it allows proofs regarding (a subset of) Haskell code to be developed using (a subset of) the language itself. This effectively takes referential transparency one step further than pen-and-paper proofs, allowing *formal* proofs to be constructed using the syntax of Haskell. This is seemingly very useful because, for the most part, it *appears* to avoid having to translate semi-formal proofs into a different language in order to formalise them. However, strictly speaking, this is not true because conventional Haskell allows for partiality, whereas Liquid Haskell does not.

Recall that any language used to formalise mathematical proofs must be total so that its underlying logic is consistent. This is true of Liquid Haskell, which precludes non-terminating functions. As such, the proof of $\mathit{slowRev}\ (xs +\!\!+ ys) = \mathit{slowRev}\ ys +\!\!+ \mathit{slowRev}\ xs$ above is indeed formally correct for Liquid Haskell, but, strictly speaking, it is only semi-formally correct for conventional Haskell because—as we saw previously—it does not hold

Equality	$\{-@\ (==.)\ ::\ x : a\ \rightarrow\ \{\ y : a\ \mid\ y\ =\ x\ \}$ $\rightarrow\ \{\ z : a\ \mid\ z\ =\ x\ \wedge\ z\ =\ y\ \}\ @-\}$ $\_\ ==.\ y\ =\ y$
Theorem invocation	$(?)\ ::\ a\ \rightarrow\ \mathit{Proof}\ \rightarrow\ a$ $x\ ?\ \_\ =\ x$
Proof finalisation	$(***)\ ::\ a\ \rightarrow\ \mathit{QED}\ \rightarrow\ \mathit{Proof}$ $\_\ ***\ \mathit{QED}\ =\ ()$
Proof type	**type** $\mathit{Proof}\ =\ ()$
QED definition	**data** $\mathit{QED}\ =\ \mathit{QED}$

Figure 2.1: Liquid Haskell proof combinators introduced in (Vazou et al. 2018)

when $xs = \bot$. Thus, replacing *slowRev* $(xs +\!\!+ ys)$ with *slowRev* $ys +\!\!+$ *slowRev* xs, or vice-versa, in Liquid Haskell does not change any results, but the same replacements in conventional Haskell can change the results when xs is a non-terminating computation.

It is difficult to determine how big an issue overlooking Haskell's bottom value is in practice, and we have not addressed this question in the work of this thesis. Nonetheless, any system that is used to formalise equational reasoning in a similar manner must also preclude non-termination. Consequently, it appears that there must always be some distance between formal proof systems that regard Haskell programs as pure and total mathematical functions and their actual implementations in the language itself.

Partiality aside, the examples in this subsection aim to demonstrate the many advantages of using Liquid Haskell to formalise equational reasoning, including automatic reflecting of programs into a formal logic, automation of proofs regarding preconditions and postconditions of functions, automation of simple inductive proofs using PLE, machine-checked steps of reasoning, compositionality of theorems, and strong correspondence with the style of proofs-on-paper. We refer readers to (Vazou 2016) for further information regarding the implementation of Liquid Haskell, as well as more example use cases.

Chapter 3

AutoBench

Comparing the Time Performance of Haskell Programs

Property-based testing tools such as QuickCheck provide a lightweight means to test the correctness of Haskell programs, but what about their efficiency? In this chapter, 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 correctness properties are also efficiency improvements.

3.1 Introduction

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 generated automatically.

This approach was popularised by QuickCheck (Claessen and Hughes 2000a), 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 (Hughes 2016), its original implementation has been extended to handle impure functions (Claessen and Hughes 2002), and it has led to a growing body of research (Runciman, Naylor, and Lindblad 2008; Bernardy, Jansson, and Claessen 2010;

Klein et al. 2012) and industrial interest (Arts et al. 2006; Hughes 2007; Hughes 2016). These successes show that property-based testing is a useful method for checking program correctness, but what about program efficiency?

The work in this chapter is founded on the simple observation that many of the correctness properties tested using systems such as QuickCheck can also be interpreted as time efficiency improvements. For example, consider the familiar monoid properties of Haskell's list-appending operator, $(+\!\!+)$, which can all be tested using QuickCheck:

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

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

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

For example, the associativity property of append is an efficiency improvement because the left-hand side, $(xs +\!\!+ ys) +\!\!+ zs$, traverses xs twice, whereas the right-hand side, $xs +\!\!+ (ys +\!\!+ zs)$, only traverses xs once. However, formally verifying such improvement properties for lazy languages like Haskell is challenging, and usually requires the use of specialised techniques such as improvement theory (Moran and Sands 1999).

In this chapter, we show how improvement properties can be put into the hands of ordinary Haskell programmers, by combining the QuickCheck system with the Criterion benchmarking library, to give a lightweight, fully automated means to compare the time performance of Haskell programs, which we call *AutoBench*.

Surprisingly, this appears to be the first time that QuickCheck and Criterion have been combined, despite this being a natural idea. The AutoBench system comprises approximately 7,500 lines of new Haskell code and is freely available on GitHub (Handley 2019).

3.2 AutoBench in practice

To begin, we illustrate the basic functionality of our system with two simple examples from Hutton's (2016) introductory textbook on Haskell.

3.2.1 Quickly reversing a list

Recall Haskell's naive reverse function for lists of integers:

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

Although this definition is straightforward, it suffers from a poor, quadratic runtime performance due to its repeated use of the append operator, (++), which has linear runtime in the length of its first argument. Nonetheless, it is easy to define a more efficient version using an accumulating parameter (Burstall and Darlington 1977). Despite being somewhat less clear, this new definition, which we call *fastRev*, has linear time performance:

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

Before comparing the efficiency of *slowRev* with that of *fastRev*, it is good practice to ensure that both functions give the same results. An easy way to test this is to use QuickCheck. In particular, given a testable property, the *quickCheck* function will generate one hundred random test cases and check that the property is satisfied in all cases. In this instance, we can use a simple predicate that compares the results of both functions

```
revsEq :: [Int] → Bool
revsEq xs = slowRev xs == fastRev xs
```

and, as expected, the property satisfies all the tests:

> *quickCheck revsEq*

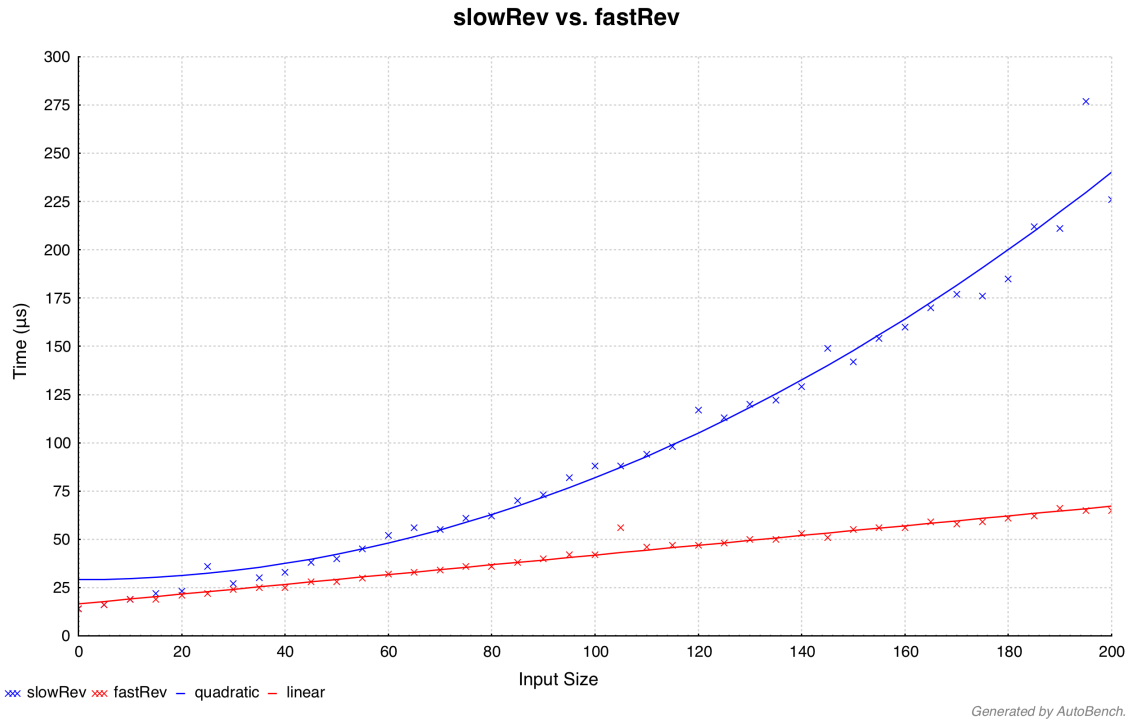
+++ OK, passed 100 tests.

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 the time performance of *slowRev* with that of *fastRev* is to use the AutoBench system. In particular, given two or more programs of the same type, the *quickBench* function generates a number of random inputs that are increasing in size and measures the runtimes of each program when executed on those inputs. The runtime measurements are then automatically analysed to produce time performance results. In this instance, *quickBench* is invoked by supplying it with the two functions to compare as well as their names for display purposes, as follows:

> *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:



This graph compares the runtime measurements of *slowRev* and *fastRev* for each input size. Both sets of measurements also have a line of best fit, which is calculated using linear regression analysis and estimates the time complexity of each 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. In particular, it is difficult to determine the exact degree of a polynomial equation by merely looking at its graph.

The second result produced by the *quickBench* function is a table, which is output to the user's command line. The table displays the value of each runtime measurement and the precise equations of the calculated lines of best fit displayed on the graph above:

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	$y = 5.28e{-}9x^2 + 2.28e{-}11x + 2.91e{-}5$					
fastRev	$y = 2.53e{-}7x + 1.65e{-}5$					
Optimisation	*slowRev* $\trianglerighteq$ *fastRev* (0.95)					

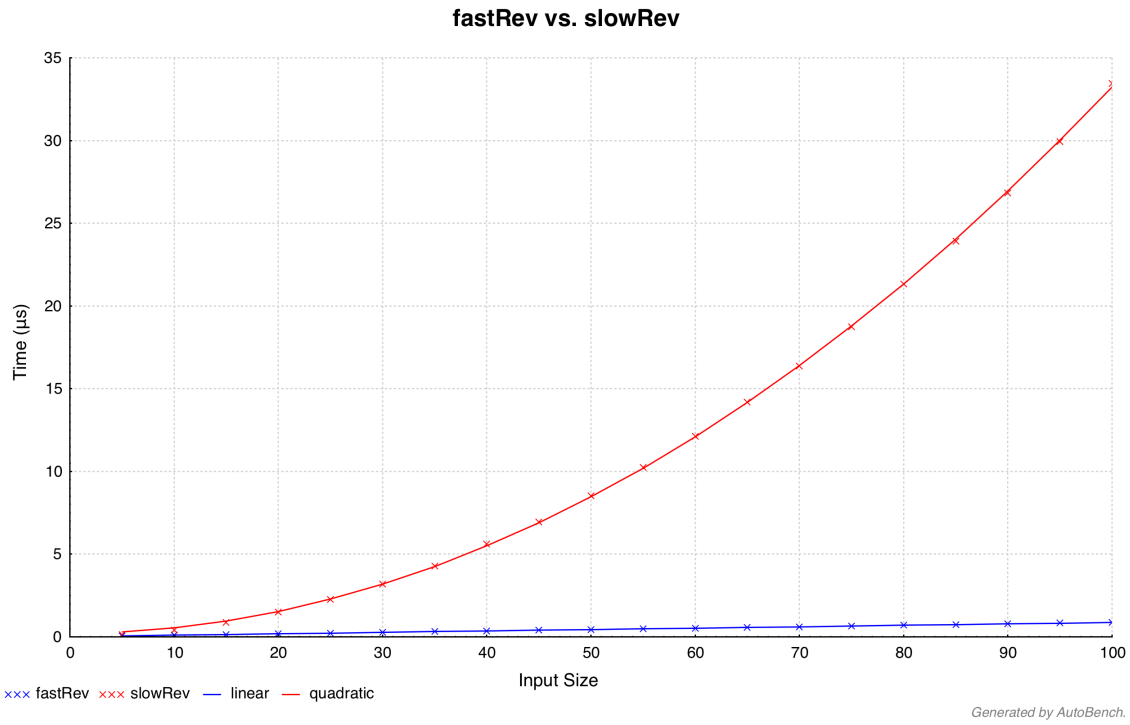
Moreover, at the bottom of the table is an optimisation, written *slowRev* $\trianglerighteq$ *fastRev*, which is derived from the combination of AutoBench's performance results, which show that for *almost* all test cases, *fastRev* is more time-efficient than *slowRev*, that is, *slowRev* $\gtrsim$ *fastRev*; and QuickCheck's correctness results, which show that for all test cases, both functions give the same results, that is, *slowRev* $=$ *fastRev*. The decimal appearing in parentheses after the optimisation indicates that the result 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 the optimisation *slowRev* $\trianglerighteq$ *fastRev* indeed gives a quadratic to linear time speedup, as we expected. Overall, knowing that one program is more (time) efficient than another, and by how much, is key to making an informed choice between two denotationally equal implementations.

Quick results

The *quickBench* function is designed to analyse time performance using the Haskell interpreter GHCi, in a similar manner to how QuickCheck is often used. Its primary goal is to generate useful results quickly. Consequently, by default, its runtime measurements are based on *single* executions of test programs on test inputs. For this example, testing takes just a few seconds, with the primary cost being its benchmarking phase. Although this approach sacrifices precision for speed—as demonstrated by the notable 'noise' in the graph above—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 library (O'Sullivan 2014a) to accurately measure the runtimes of *compiled* Haskell programs. Furthermore, its results are based on *many*, repeated executions of programs on their inputs. In regard to this example, testing takes a few minutes. However, the results better demonstrate the difference in time complexity and a reduction in 'noise':



The architecture of the AutoBench executable fully encapsulates that of *quickBench*, as it is essentially a super-system thereof. As such, the remainder of this chapter focuses on the design, implementation, and application of AutoBench.

3.2.2 Robustly flattening a tree

In the previous example, the key insight used to derive *fastRev* from *slowRev* is to 'make append vanish' (Wadler 1987), which is the primary source of *slowRev*'s inefficiency. Making append vanish, by replacing it with the (:) operator and an accumulating parameter, is a common trick used to improve the time efficiency of recursive functions that construct lists in their results. For example, this same technique can be used to derive an optimised function for flattening binary trees with integers in their leaves:

```
data Tree = Leaf Int | Node Tree Tree
```

Similarly to *slowRev*, a simple but inefficient function that flattens the *Tree* datatype to list form can be defined by repeatedly appending singleton lists:

```
slowFlatten :: Tree → [Int]
slowFlatten (Leaf n)   = [n]
slowFlatten (Node l r) = slowFlatten l ++ slowFlatten r
```

In turn, just as with *fastRev*, a less obvious but more performant version of the above function can be defined using an accumulating parameter:

```
fastFlatten :: Tree → [Int]
fastFlatten t = flatCat t []
  where
    flatCat (Leaf n)   ns = n : ns
    flatCat (Node l r) ns = flatCat l (flatCat r ns)
```

Specifying test inputs

It is straightforward to compare the time performance of *slowFlatten* with that of *fastFlatten* using the AutoBench executable, which is invoked from the command line.

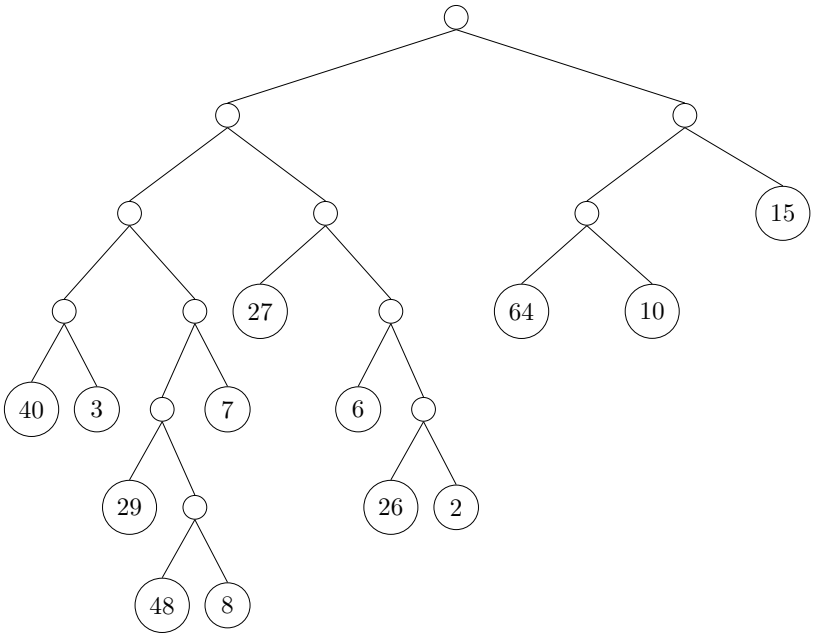


Figure 3.1: A randomly generated binary tree

To do so, both functions are placed in the same Haskell module, *Flatten.hs*, along with the definition of the *Tree* datatype and two instance declarations.

The first instance declaration is required for the standard *NFData* type class, which allows the *Tree* datatype to be evaluated to normal form:

```
instance NFData Tree where
  rnf (Leaf n)   = rnf n
  rnf (Node l r) = rnf l `seq` rnf r
```

Random inputs to each function must be fully normalised before benchmarking commences to ensure that the process of data generation does not distort any subsequent performance measurements. This is discussed in more detail in section 3.3.3.

The second instance declaration is required for generating random, *sized* values in the *Tree* datatype. We use the *SizedArbitrary* type class for this purpose, which can be seen as a 'sized variant' of the *Arbitrary* type class that forms the basis of random data generation in the QuickCheck system. The *SizedArbitrary* type class is introduced in section 3.3.1. Hence, for the purposes of this example, we consider the task of generating random trees of different sizes from a higher level. Unlike with the list datatype, whose notion of size is

> *AutoBench ./Flatten.hs*

► Processing *Flatten.hs* ✓

► Running *defaultTestSuite*:

- QuickChecking the test programs ✓
- Generating the benchmarking file ✓
- Compiling the benchmarking file ✓
- Executing the benchmarking file ...

Figure 3.2: AutoBench execution: set-up phase

intuitively given by the total number of elements, a notion of size for a tree datatype is less obvious. For example, we could define a tree's size to be its height (or depth), the total number of leaves, or the total number of nodes. All of these notions of size are reasonable measures. In this instance, we opt for a tree's total number of nodes:

```
instance SizedArbitrary Tree where
  sizedArbitrary = ...
```

For example, a randomly generated tree of size 25 is shown in figure 3.1. The generation process for this tree is illustrated in section 3.3.2.

Invoking AutoBench

The AutoBench executable requires one parameter to run, which is the path to the Haskell module that contains the programs to analyse (as well as any system settings, datatype definitions, and instance declarations). For this example, the system is invoked as follows when *Flatten.hs* is located in the working directory:

```
> AutoBench ./Flatten.hs
```

Processing *Flatten.hs*. When executed, the system first performs a number of safety checks on the input file. In particular, it ensures the following:

(a) The two functions in the file, namely *slowFlatten* and *fastFlatten*, have the same types;

(b) The input and result types of the functions are members of the *NFData* type class;

(c) The input types of the functions are members of the *SizedArbitrary* type class.

We note that AutoBench provides default *NFData* instances for all standard Haskell types.

As we did not specify a custom test suite in the input file, the default option is selected once the file has been processed, as illustrated by the execution trace in figure 3.2. Test suites configure the functionality of AutoBench, controlling, for example, which programs in the input file to test and the sizes of the random inputs to generate. In many cases, the default test suite is an adequate choice, especially for beginner users. Further information on specifying custom test suites is given in the system's online user manual (Handley 2019).

QuickChecking the test programs. The first action performed by the default test suite is to invoke QuickCheck, which tests whether both functions give the same results. QuickCheck testing confirms that this is true for *slowFlatten* and *fastFlatten*, so the step is marked with a ✓ for success. Note that, unlike with all successive steps, failure at this point does mean testing is aborted. By default, programs under test are checked to see if they give the same results but performance analysis does not necessitate this. On the other hand, it is not obvious as to why a user would wish to compare the performance of programs with (equal types but) different functionalities, and so a warning is issued if this check fails.

Generating the benchmarking file. A new Haskell module is then generated by the system, which is known as the *benchmarking file*. In short, this file contains a number of Criterion benchmarks to analyse the performance of the two functions executed on random inputs. In the default case, 20 benchmarks are generated for each function, with input sizes ranging from 0 to 100 in steps of 5. Both functions are tested on the *same* inputs.

Compiling the benchmarking file. The benchmarking file is automatically compiled using GHC. By default, optimisation is turned off via the -O0 compiler flag. This is the only compilation option used by the default test suite, however, any number of compiler flags can be specified as part of user-defined test suites.

Executing the benchmarking file. The last action performed during AutoBench's set-up phase is invoking the executable compiled from the benchmarking file.

```
benchmarking: input size 1/Flatten.fastFlatten
time                    18.60 ns  (18.51 ns .. 18.73 ns)
                        1.000 R^2 (0.999 R^2 .. 1.000 R^2)
mean                    18.66 ns  (18,58 ns .. 18.84 ns)
std dev                 392.1 ps  (229.1 ps .. 737.3 ps)
variance introduced by outliers: 18% (moderately inflated)

benchmarking: input size 1/Flatten.slowFlatten
time                    20.22 ns  (20.11 ns .. 20.41 ns)
                        1.000 R^2 (0.999 R^2 .. 1.000 R^2)
mean                    20.21 ns  (20.13 ns .. 20.35 ns)
std dev                 358.7 ps  (227.2 ps .. 535.5 ps)
variance introduced by outliers: 23% (moderately inflated)
```

Figure 3.3: AutoBench execution: benchmarking phase

Benchmarking with Criterion

Recall from background section 2.2.1 the default output written to the user's console when running Criterion benchmarks. Figure 3.3 shows an extract of such output for our current example. Just as when using Criterion directly, this information can be suppressed during AutoBench's benchmarking phase via a user option.

The name of each benchmark in figure 3.3 includes the function being tested and the size of the input on which it is being tested. Notice that the size of the first input for both *slowFlatten* and *fastFlatten* is 1 and not 0. This is because the notion of size being used for the *Tree* datatype is the total number of nodes: including internal *Node*s and *Leaf* nodes. Hence, it is not possible to generate a tree of size 0. The system has, therefore, opted for the closest size to 0 that is valid, which happens to be 1.

Analysing the runtime measurements. AutoBench configures Criterion to write all the measurements made during its benchmarking phase (those presented in figure 3.3) to a JSON report file, which the system then parses and interprets. The analysis phase of AutoBench's execution, which is shown in figure 3.4, first confirms that benchmarking is complete, and then notifies the user that these measurements will be analysed.

> *AutoBench ./Flatten.hs*

- ▶ Processing *Flatten.hs* ✓
- ▶ Running *defaultTestSuite*:
  - ...
  - Executed the benchmarking file ✓
  - Analysing the runtime measurements ...

Figure 3.4: AutoBench execution: analysis phase

Once the measurements have been analysed, a number of performance statistics are output to the console. These are illustrated in figure 3.5. Firstly, a number of key test settings are summarised, including the names of the test programs; the type of the test data; whether the QuickCheck test, which compared the results of both programs, succeeded; and any specified compiler flags. Finally, recall from section 2.2.1 Criterion's normalisation setting, which dictates whether the results of test programs are evaluated to full normal form (nf) or to weak head normal form only (whnf).

Underneath the test summary is the analysis section. A results table is displayed for each program, ordered alphabetically by program name. Results tables (extending those of the previous example 3.2.1) display each program's measured runtime for every input size, the standard deviation of all such measurements, and the average variance introduced by outliers. These statistics give a basic assessment of the quality of the raw measurements made by Criterion. This is a useful indicator of the reliability of AutoBench's performance results, as they extrapolate the raw data. See section 3.3.4 for more information.

As with the previous example, lines of best fit are calculated for each table of measurements. In the case of *slowFlatten*, the line of best fit is an $n \log_2 n$ equation, and in the case of *fastFlatten*, a linear equation. Recall that each line of best fit is calculated using linear regression analysis and approximates time complexity.

At the bottom of the analysis section in figure 3.5 is an optimisation result

$$\textit{slowFlatten} \quad \trianglerighteq \quad \textit{fastFlatten} \quad (0.95)$$

indicating that *fastFlatten* performed better than *slowFlatten* in 95% of test cases.

A corresponding graph of results, which plots each set of raw measurements and their

lines of best fit, is presented in figure 3.6. Recall that AutoBench automatically generates this graph and saves it to the working directory as a PNG file.

Overall, the graph and table of results suggest that *fastFlatten* is the more time-efficient of the two programs by a $\log_2$ factor. In the average case, this is to be expected as *slowFlatten* performs approximately linear operations at each of its recursive calls and the height of a binary tree with n total nodes is on average $\log_2 n$. In comparison, *fastFlatten* performs an in-order traversal of its input tree, which is linear in the total number of nodes.

Finally, we note that AutoBench's analysis phase is also customisable by way of user-defined test suites. See the system's user guide for further information (Handley 2019).

3.2.3 Quick versus robust results

The first example, 3.2.1, of this section demonstrates the functionality of the *quickBench* function when comparing the time performance of two programs that reverse lists of integers. In turn, the second example, 3.2.2, showcases the full AutoBench system being used to compare two programs that flatten trees of integers to list form.

While *quickBench* focuses on providing quick results that are sufficient for basic exploration purposes, those produced by the full system are notably more robust, and thus suitable for thorough performance analysis. An important practical difference between such quick and robust results is testing time, specifically, benchmarking time. In the former case, benchmarking *usually* takes orders of seconds, and in the latter case, orders of minutes.

Even though benchmarking with the Criterion library is time-consuming, the example presented in section 3.2.2 demonstrates that utilising the full AutoBench system requires approximately the same user effort as when utilising *quickBench*. This is primarily because the testing process is fully automated. Hence, it is simply the case that users must be prepared to wait longer (and commit additional computing power) to obtain more reliable performance results. We believe that this is a reasonable tradeoff.

Test summary

Programs	*fastFlatten*, *slowFlatten*
Data	random, size range: 1, 5, 9, ..., 99
Normalisation	nf
QuickCheck	✓
GHC flags	-O0

Analysis

fastFlatten

Input size	1	5	9	15	19	25	29	35
	39	45	49	55	59	65	69	75
	79	85	89	95	99			
Time (μs)	0.020	0.045	0.069	0.106	0.130	0.167	0.192	0.229
	0.263	0.300	0.325	0.354	0.379	0.421	0.453	0.484
	0.529	0.548	0.582	0.639	0.658			
Std dev (μs)	0.006							

Average variance introduced by outliers: 22% (moderately inflated)

Fit $y = 6.46e{-}9x + 7.50e{-}9$

slowFlatten

Input size	1	5	9	15	19	25	29	35
	39	45	49	55	59	65	69	75
	79	85	89	95	99			
Time (μs)	0.019	0.062	0.119	0.194	0.272	0.352	0.406	0.520
	0.738	0.771	0.849	0.839	0.999	1.154	1.229	1.246
	1.433	1.723	1.498	1.670	1.889			
Std dev (μs)	0.011							

Average variance introduced by outliers: 19% (moderately inflated)

Fit $y = 2.74e{-}9x\log_2 x + 4.80e{-}8$

Optimisation: *slowFlatten* $\trianglerighteq$ *fastFlatten* (0.95)

Figure 3.5: AutoBench execution: time performance results (a)

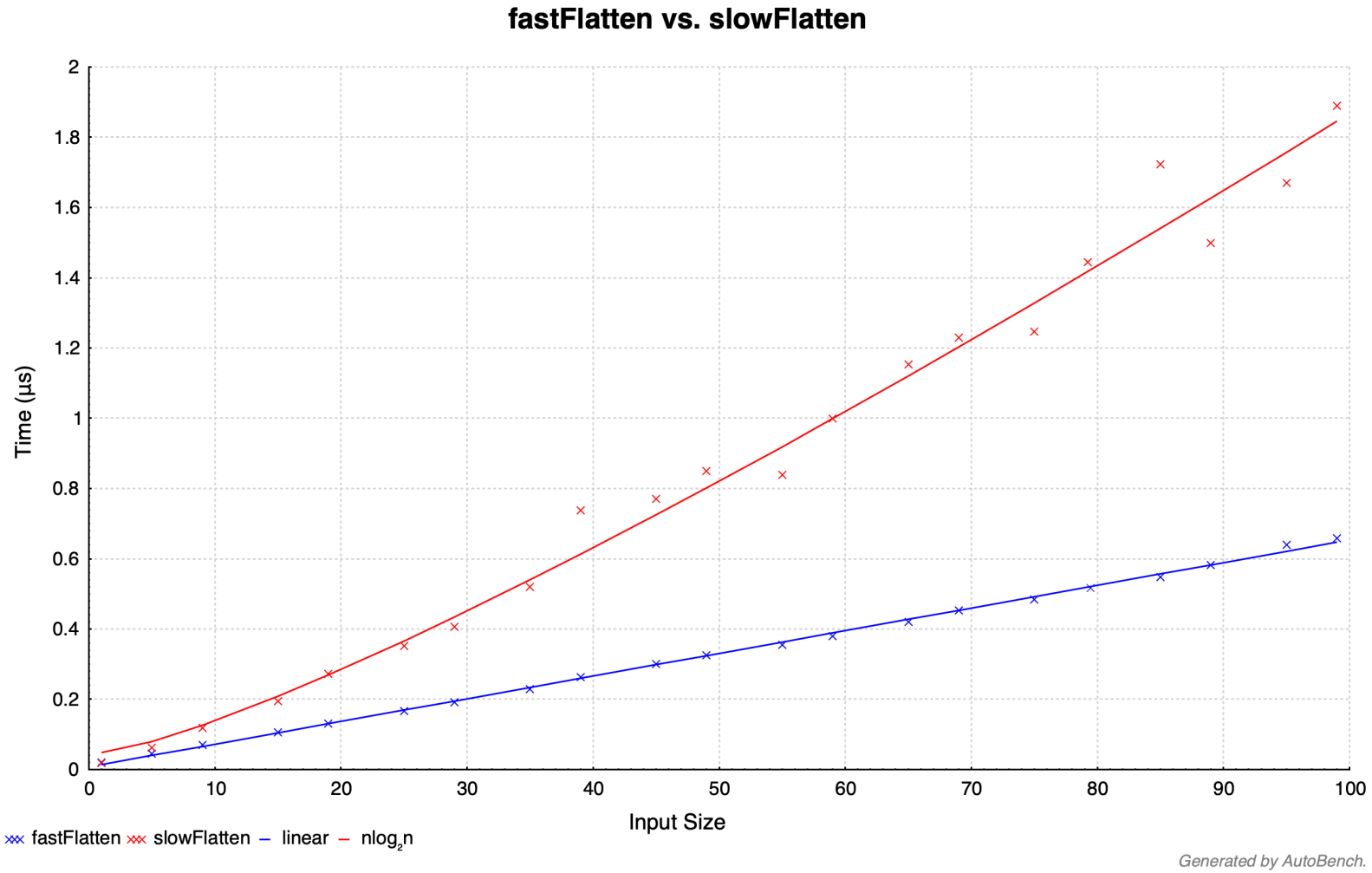


Figure 3.6: AutoBench execution: time performance results (b)

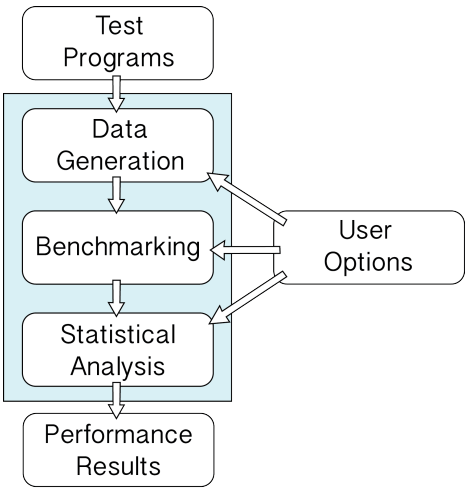


Figure 3.7: AutoBench's core architecture

3.3 Architecture of AutoBench

In this section, we introduce the core architecture of the AutoBench system. Figure 3.7 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 relevant user options.

3.3.1 Data generation

Given a number of programs to compare, the system must generate a suitable range of inputs with which to test their time performance. To produce such inputs, AutoBench exploits the random data generation facilities of QuickCheck. Recall from section 2.1.1 that QuickCheck provides default generators for all standard Haskell types, together with a rich set of combinators that aid users in defining generators for custom datatypes.

First, we briefly review how random values can be generated using QuickCheck, and then explain how our system builds on this approach to generate random values of a particular *size*. Subsequently, in the next section—which is directly related to this one—we overview AutoBench's support for *generic*, random, sized data generation.

QuickCheck generators

QuickCheck's notion of a random data generator is based on the following simple type class

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

which intuitively captures datatypes that support the generation of *arbitrary* values within them. $Gen\ a$ can thus be seen as a pseudo-random generator for values of type a.

For example, the following instance declaration (Claessen and Hughes 2000a), allows us to generate an arbitrary integer using a primitive combinator $choose :: (Int, Int) \rightarrow Gen\ Int$ that randomly chooses an integer in a given interval:

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

Sampling this generator thus yields a number of random integers:

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

QuickCheck's support for data generation goes far beyond integers. Random lists, n-tuples, sets, maps, monoids, and even functions can be generated using default *Arbitrary* instances defined in QuickCheck's standard library (Claessen and Hughes 2000a). Moreover, a rich set of combinators are provided that are useful for defining *Arbitrary* instances for custom datatypes. Some example combinators are as follows:

$oneof :: [Gen\ a] \rightarrow Gen\ a$	one of the given generators
$vectorOf :: Int \rightarrow Gen\ a \rightarrow Gen\ [a]$	a list of the given length
$suchThat :: Gen\ a \rightarrow (a \rightarrow Bool) \rightarrow Gen\ a$	a value that satisfies a predicate
$resize :: Int \rightarrow Gen\ a \rightarrow Gen\ a$	a different sized value

AutoBench sized generators

Analysing the time performance of a program requires measuring its running time on different sized inputs. In this context, the term *efficiency* is commonly understood as the increase

in the runtime of a program in relation to the sizes of its inputs. Any program being tested by AutoBench thus requires not only a method for generating random inputs in its domain (as provided by the *Arbitrary* type class) but a method for generating random inputs with different sizes. AutoBench provides the *SizedArbitrary* type class for this purpose.

Arbitrary versus *SizedArbitrary*

Data generators provided by QuickCheck do in fact have access to an *implicit* size parameter, which can be accessed using $sized :: (Int \rightarrow Gen\ a) \rightarrow Gen\ a$. This is because, when checking correctness properties, QuickCheck begins by generating small test cases and gradually increases the size as testing progresses. The original QuickCheck manual, available online at (Claessen and Hughes 2000b), states the following about the size of test data:

> Different test data generators interpret the size parameter in different ways: some ignore it, while the list generator, for example, interprets it as an upper bound on the length of generated lists. You are free to use it as you wish to control your own test data generators.

This extract is somewhat problematic for performance testing with AutoBench because although it is quite reasonable for a datatype to fail to support a well-defined notion of size (*Bool* for example), it is not appropriate for users to 'freely' control the size of test data 'however they wish'. In particular, it is paramount that any notion of size realised by a datatype's definition is strictly monotonic. In other words, a value $v :: D$ of size 100 *must* be 'bigger' in some meaningful way than a value $v' :: D$ of size 20. If this monotonicity property is not satisfied, then AutoBench's interpretation of efficiency is invalidated, and, in consequence, the system's performance results become meaningless.

To emphasise the importance of size when generating random values of a particular type, the *SizedArbitrary* class makes the size parameter *explicit*:

```
class SizedArbitrary a where
  sizedArbitrary :: Int → Gen a
```

In other words, an AutoBench data generator for a given type is a function from a positive integer to a QuickCheck generator of the same type. Thus, we see how *SizedArbitrary* naturally builds upon *Arbitrary*. The correspondence between both type classes is strengthened by *sizedArbitrary*'s default signature (GHC 7.2.1 onwards), which uses AutoBench's explicit size parameter to *resize* QuickCheck's implicit one:

```
default sizedArbitrary :: Arbitrary a ⇒ Int → Gen a
sizedArbitrary n = resize n arbitrary
```

This default implementation is particularly useful when defining *SizedArbitrary* instances for enumerations, whose sizes are mostly irrelevant. For example, if we wish to generate random lists of booleans, how do we define the size of *True* and *False*? It is likely that this detail is unimportant. In such cases, we can provide an empty instance declaration, which utilises QuickCheck's *Arbitrary* instance, as follows:

```
instance SizedArbitrary Bool
```

We could have opted to define a generator for lists of booleans directly in a similar manner, by utilising flexible instances (GHC 6.8.1 onwards):

```
instance SizedArbitrary [Bool]
```

However, this particular instance declaration is *invalid* because it does not satisfy the monotonicity property discussed above. The source of the problem is QuickCheck's default *Arbitrary* instance for lists, which is 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)
```

According to this definition, a list of integers is generated by first generating a random number k between 0 and the given size parameter n, and then generating a vector of

arbitrary values of size k. Hence, using this generator permits a random list of size $n = 100$ to be empty, while a list of size $n = 50$ can contain 20 elements. In light of this, we must define a new generator for random, sized lists that is compatible with AutoBench. To do so, we can use the *vectorOf* combinator to generate lists with precisely n elements:

```
instance SizedArbitrary a ⇒ SizedArbitrary [a] where
  sizedArbitrary n = vectorOf n (sizedArbitrary n)
```

Notice that the size parameter n is forwarded to the *Arbitrary* instance for booleans. This is not a problem, however, as the size parameter is simply ignored by this instance:

```
> generate (sizedArbitrary 5 :: Gen [Bool])
[False, False, False, True, False]

> generate (sizedArbitrary 10 :: Gen [Bool])
[True, True, True, True, False, False, True, False, True, True]
```

Overall, it should be clear that *SizedArbitrary* type class can be seen as a sized variant of *Arbitrary* (without the shrinkage functionality). As such, careful use of the default implementation for *sizedArbitrary* is convenient on many occasions.

User options

AutoBench generates random inputs 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 given size range must contain at least 20 values with distinct sizes. As illustrated in both examples of section 3.2, the default size range is 0 to 200 in steps of 5, but this can easily be modified by the user.

3.3.2 Generic data generation

In this subsection, we explain how datatype-generic programming can be applied when defining data generators for use with the *SizedArbitrary* type class. Such generators can be used on many occasions, rather than having to define separate ones for each specific occasion: they are *generic* in precisely this sense.

A thorough introduction to generic programming in Haskell requires an equally thorough introduction to type-level programming in Haskell. This subsection is not the place for either introduction. Instead, we focus primarily on high-level intuition and thus simplify relevant explanations where possible. To this end, we provide precise implementation details only when attesting to the concrete contributions of this thesis.

Readers who are keen to delve deeper into applying type-level programming and generic programming in Haskell are referred to relevant course notes by Andres Löh (Löh 2015). The author of this thesis found them to be particularly useful.

Datatype-generic programming

To develop some intuition for datatype-generic programming (Gibbons 2006), we begin by recalling the second example from section 3.2.

```
data Tree = Leaf Int | Node Tree Tree
```

The purpose of this example was to compare the time performance of two functions that flatten the above *Tree* datatype to list form. Doing so required us to generate a number of different sized trees, where the size of a tree was defined to be its total number of nodes.

In section 3.2.2, we overlooked the details of how such binary trees can be randomly generated using the *SizedArbitrary* type class. We return to these details now as they serve as fitting motivation for the use of datatype-generic programming. Throughout, we do not consider the values in the leaves of generated trees to be important because the flattening process is polymorphic in the type of this data.

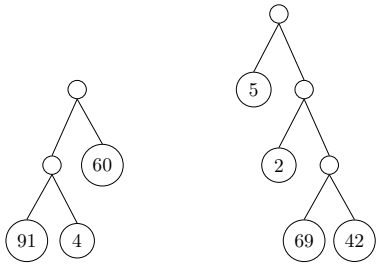
First, we consider a naive generator for random, sized *Tree*s:

```
instance SizedArbitrary Tree where
  sizedArbitrary 1 = Leaf <$> choose (0,100)
  sizedArbitrary n
    | n `mod` 2 == 0 = error "invalid size"
    | otherwise      = Node <$> sizedArbitrary ((n - 1) `div` 2)
                            <*> sizedArbitrary ((n - 1) `div` 2)
```

In the base case of this generator, a *Leaf* node of size 1 is produced containing a random

integer between 0 and 100. In the recursive case, the size parameter n is first scrutinised. If n is even, then it is not possible to produce a tree of this size, and so an error is thrown. If n is odd, then an internal *Node* is produced using two recursively generated subtrees. Each subtree is approximately half the size of its parent.

If we recall that the size parameter n is always positive, then, when it succeeds, this generator produces a random tree of the correct size. Nonetheless, there is a problem with the definition, which is that it can only produce binary trees that are *perfect*. For example, both of the following *Tree*s are valid but are not perfect:



A more satisfactory method for generating random binary *Tree*s thus requires us to account for nonperfection. As such, how can the above definition be modified to cater for this? A solution by Jansson et al. (2006) is to randomly distribute remaining size $n > 1$ between the sizes of both subtrees, n_0 and n_1, respectively. To ensure that the size of each subtree is valid, we simply pick values for n_0 and n_1 that are odd:

```
instance SizedArbitrary Tree where
  sizedArbitrary 1 = Leaf <$> choose (0,100)
  sizedArbitrary n = do
    n0 ← odd (1, n − 1)
    Node <$> sizedArbitrary n0 <*> sizedArbitrary (n − 1 − n0)
```

For any strictly positive, *odd* integer n this generator will thus produce a (perfect or non-perfect) binary tree with n nodes that are randomly configured.

The definition of our generator for the *Tree* datatype is now adequate. However, it lacks generality. For example, if we sought to amend the branching factor of the *Tree* datatype, making it a full ternary tree instead of a full binary tree, then this data generator would be incompatible with the new datatype. This is for two reasons: (a) it is not possible to

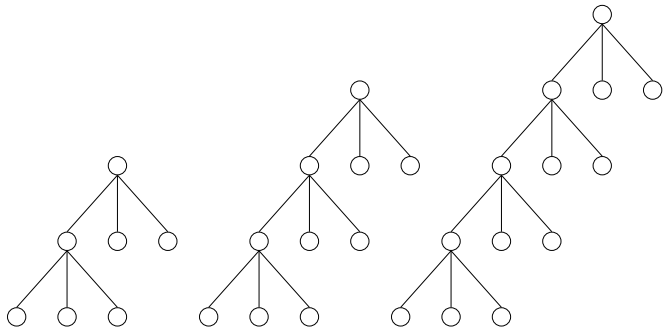


Figure 3.8: Full ternary trees of size 7, 10, and 13

construct a full ternary tree with, for example, 3, 5, 9, or 11 nodes in total, as can be deduced from figure 3.8; (b) ternary trees require an inherently different definition than the one used for binary trees to account for the additional branch.

To devise a generic approach for randomly generating tree-like data structures of a particular size, we seek to overcome both of these issues. Overcoming (a) primarily requires a method for randomly distributing remaining size among arbitrarily many subtrees. In due course, we will see that this can be achieved by solving linear Diophantine equations. However, a solution to (a) is not useful unless we can also overcome the differences in implementation details highlighted by (b). Doing so requires adopting a more uniform view on data, which is the essence of datatype-generic programming (Hinze and Jeuring 2003).

A uniform view on data

First, we seek to overcome the issue raised in point (b) of our previous discussion. Consider the following tree-like data structures containing integers:

$$\begin{aligned}
&\textbf{data}\ \mathit{List} &&= \mathit{Empty} &&\mid \mathit{Cons}\ \mathit{Int}\ \mathit{List}\\
&\textbf{data}\ \mathit{Tree}_2 &&= \mathit{Leaf}_2\ \mathit{Int} &&\mid \mathit{Node}_2\ \mathit{Tree}_2\ \mathit{Tree}_2\\
&\textbf{data}\ \mathit{Tree}_3 &&= \mathit{Leaf}_3\ \mathit{Int} &&\mid \mathit{Node}_3\ \mathit{Tree}_3\ \mathit{Tree}_3\ \mathit{Tree}_3\\
&\textbf{data}\ \mathit{Tree}_* &&= \mathit{Node}_*\ \mathit{Int}\ [\mathit{Tree}_*]
\end{aligned}$$

The first datatype is that of singly-linked lists, which is the standard implementation of lists in Haskell. The last datatype defines multiway branching trees, or rose trees, which are often used to represent game trees, for example, in noughts and crosses and connect four, and when applying basic AI algorithms such as minimax. Our generic approach to random data

generation will ideally account for all of these datatypes (and those of a similar structure) as they frequently arise in practice. However, how can one method be used to generate values for all of these types, given that their implementation-specific details are so distinct? In particular, each type has a different name: *List*, $Tree_2$, $Tree_*$; and different names for its data constructors: *Empty*, $Leaf_2$, $Node_3$. Moreover, the constructors have different arities: *Empty* is nullary, $Leaf_2$ is unary, $Node_*$ is binary; and different types: $Leaf_2 :: Int \to Tree_2$, $Node_3 :: Tree_3 \to Tree_3 \to Tree_3 \to Tree_3$, $Node_* :: Int \to [\,Tree_*] \to Tree_*$.

On the one hand, the differences between these datatype definitions are somewhat immaterial: names of constructors do not dictate how values are generated per se. On the other hand, the names of data constructors identify distinct functions. Therefore, it is not possible for a generation method that must be 'universally polymorphic' (Cardelli and Wegner 1985) to apply such functions in an ad-hoc fashion. Hence, it should be clear that random values for different datatypes cannot be uniformly generated *directly*.

To see how random values can be generated indirectly, we must adopt a more uniform view on data. In particular, each datatype above offers a choice between different data constructors, and the sequence of zero or more arguments used to produce the datatype depends on the chosen constructor. In this manner, we can see that every Haskell datatype has a similar structure, which can be expressed as a *sum of products*:

$$\begin{aligned}
&\textbf{data}\ Unit &&= Unit\\
&\textbf{data}\ a :+: b &&= Inl\ a \mid Inr\ b\\
&\textbf{data}\ a :*: b &&= a :*: b
\end{aligned}$$

For example, a more generic representation of the *List* datatype

$$\textbf{type}\ List^{Rep} = Unit :+: Int :*: List$$

is given by the binary sum (:+:) of the nullary product, *Unit*, and the binary product (:∗:) of an integer and another list. Note that (:∗:) has higher precedence than (:+:). The other tree-like datatypes can also be represented similarly:

$$\begin{aligned}
&\textbf{type}\ Tree_2^{Rep} = Int :+: Tree_2 :*: Tree_2\\
&\textbf{type}\ Tree_3^{Rep} = Int :+: Tree_3 :*: Tree_3 :*: Tree_3\\
&\textbf{type}\ Tree_*^{Rep} = Int :*: [\,Tree_*]
\end{aligned}$$

Notice that these representation types are not recursive. In other words, the representation of a type a only represents the top-level structure of a.

Viewing data this way allows the task of generating random values of different datatypes to be recast as the task of constructing different configurations of sums of products. A 'universal' function of this kind is straightforward to define under the assumption that each argument to (:+:) and (:∗:) can also be randomly generated. Such a function is said to be *structurally polymorphic* (Ruehr 1992) or *shape polymorphic* (Jay and Cockett 1994).

Assuming the existence of such a generator, then all that is needed is a method to convert back and forth between each representation type $List^{Rep}$, $Tree_2^{Rep}$, $Tree_3^{Rep}$, $Tree_*^{Rep}$ and each corresponding type $List$, $Tree_2$, $Tree_3$, $Tree_*$, respectively. This is a task fit for ad-hoc polymorphism, in other words, a Haskell type class: *Generic*.

A generic type class

The basic idea of a generic type class is to provide an interface between an original type a and an isomorphic representation type $Rep\ a$:

```
class Generic a where
  type Rep a
  from :: a → Rep a
    to :: Rep a → a
```

The above class is a type family, whereby $Rep\ a$ is a class-associated type synonym. The specific details of Haskell type families are not crucial for our discussion. Nonetheless, a basic understanding is required: $Rep\ a$ is a user-specified type that may be distinct in each instance of *Generic*. However, $Rep\ a$ may also be similar among multiple instances. The latter case is of interest to us; we return to it below. The *Generic* type class also provides two functions, *from* and *to*, that witness the isomorphism between a and $Rep\ a$.

Given a particular representation type, then, by defining a function that operates on the type constructors of this type, we gain not only a function that operates on the original type, but one that operates on any type representable in a similar manner. This insight is a generalisation of our goal, which is to define a function on the datatypes *Unit*, (:+:), and (:∗:) to randomly generate values of data structures with tree-like representations.

Hence, to achieve this goal using datatype-generic programming, the datatypes *List*, $Tree_2$, $Tree_3$, and $Tree_*$ must be made members of the *Generic* type class, and moreover, all of their instance declarations for *Generic* require associated *Rep* types.

A uniform representation of data

Implementation details of generic type classes in Haskell—of which there are many, including a default provided by GHC—are somewhat involved. Introducing specifics is thus not helpful to our high-level discussion and hence the following explanations are simplified. We refer readers to (Löh 2015) for further information on the following concepts.

The generic data generators provided by AutoBench are defined using the *Generics-Sop* library (Vries and Löh 2014) as we find it particularly easy to use. *Generics-Sop* represents *Generic* types as n-ary sums of n-ary products, which can be seen as generalisations of binary sums and products, (:+:) and (:∗:), respectively. In particular, an n-ary product can be viewed as a sequence of zero or more arguments; and an n-ary sum indexes into a non-empty sequence of arguments, selecting a single option from the one or more possibilities.

Using *Generics-Sop*, the representation types above can thus be defined more generally using the n-ary sums of n-ary products view on data, as follows:

$$\begin{aligned}
&\textbf{type } List^{Rep} &&= (NP\ '[\ '[\,],\ \ '[\ Int\,,\ List\ ]\ ])\ _{\mathbb{I}}\\
&\textbf{type } Tree_2^{Rep} &&= (NP\ '[\ '[\ Int\ ],\ \ '[\ Tree_2\,,\ Tree_2\ ]\ ])\ _{\mathbb{I}}\\
&\textbf{type } Tree_3^{Rep} &&= (NP\ '[\ '[\ Int\ ],\ \ '[\ Tree_3\,,\ Tree_3\,,\ Tree_3\ ]\ ])\ _{\mathbb{I}}\\
&\textbf{type } Tree_*^{Rep} &&= (NP\ '[\ '[\ Int,\ [Tree_*]\ ]\ ])\ _{\mathbb{I}}
\end{aligned}$$

In each of the definitions above, $\mathbb{I}$ indexes into the outer non-empty type-level list, which is written $'[\ a_0,\ a_1,\ a_2,\ \ldots\ ]$ for each a_i a type. This represents a choice between the outer list's elements, each of which is also a (possibly empty) type-level list specifying a sequence of zero or more argument types. Hence, it should be clear that the outer list forms an n-ary product type, *NP*, while the index $\mathbb{I}$ applied to *NP* forms an n-ary sum type, *NS*. The examples below should hopefully clarify these points.

The first choice, index $\mathbb{0}$, of the $List^{Rep}$ datatype corresponds to an *empty* list, which can be defined using an empty n-ary product of arguments. In turn, the second choice,

index $\mathbb{1}$, of the $List^{Rep}$ datatype corresponds to a non-empty list. A *singleton* list can thus be defined by an n-ary product of two arguments, where the second is the empty product:

$$\begin{array}{ll} empty :: List^{Rep} & singleton :: List^{Rep} \\ empty = Nil_{\mathbb{0}} & singleton = (5 :* Empty :* Nil)_{\mathbb{1}} \end{array}$$

Note that $(:*)$ is 'cons' for n-ary products and *Nil* is the empty n-ary product.

Similar definitions can be given for the representation type of binary trees:

$$\begin{array}{ll} leaf :: Tree_2^{Rep} & binary :: Tree_2^{Rep} \\ leaf = (5 :* Nil)_{\mathbb{0}} & binary = (Leaf_2\ 5 :* Leaf_2\ 10 :* Nil)_{\mathbb{1}} \end{array}$$

Given these definitions, it is easy to see that each type and its corresponding representation type are isomorphic. For example, the isomorphism between *List* and $List^{Rep}$ is witnessed by the following two conversion functions:

$$\begin{array}{ll} fromList :: List \to List^{Rep} & toList :: List^{Rep} \to List \\ fromList\ Empty = Nil_{\mathbb{0}} & toList\ Nil_{\mathbb{0}} = Empty \\ fromList\ (Cons\ h\ t) = (h :* t :* Nil)_{\mathbb{1}} & toList\ (h :* t :* Nil)_{\mathbb{1}} = Cons\ h\ t \end{array}$$

In turn, the isomorphism between $Tree_2$ and $Tree_2^{Rep}$ is witnessed as follows:

$$\begin{array}{ll} fromTree_2 :: Tree_2 \to Tree_2^{Rep} & toTree_2 :: Tree_2^{Rep} \to Tree_2 \\ fromTree_2\ (Leaf_2\ n) = (n :* Nil)_{\mathbb{0}} & toTree_2\ (n :* Nil)_{\mathbb{0}} = Leaf_2\ n \\ fromTree_2\ (Node_2\ l\ r) = (l :* r :* Nil)_{\mathbb{1}} & toTree_2\ (l :* r :* Nil)_{\mathbb{1}} = Node_2\ l\ r \end{array}$$

These functions can thus be used to make *List* and $Tree_2$ instances of *Generic*:

```
instance Generic List where
  type Rep List = List^Rep
  from = fromList
    to = toList

instance Generic Tree_2 where
  type Rep Tree_2 = Tree_2^Rep
  from = fromTree_2
    to = toTree_2
```

The remaining tree data structures can be made members of the *Generic* type class in a similar manner. In fact, the two examples above demonstrate that the conversion functions

from and *to* typically have simple definitions, which are determined by the shapes of each datatype's constructors. In consequence, this allows *Generic* instances for regular datatypes to be automatically derived by GHC, as follows:

```
data Tree3                          data Tree*
  = Leaf3 Int                         = Node* Int [Tree*]
  | Node3 Tree3 Tree3 Tree3             deriving Generic
    deriving Generic
```

This feature is available in GHC version 7.10.1 onwards.

Overall, then, the *Generic* instances for all our tree-like data structures have similar representations. In particular, each type is built using precisely the same set of datatype constructors, namely n-ary sums and n-ary products, as defined by the *Generics-Sop* library. Furthermore, each instance declaration provides a method for converting back and forth between each original type and its representation type.

Randomly generating different configurations of n-ary sums of n-ary now becomes our goal. Doing so sidesteps the implementation-specific details that prevent us from generating random values for each tree-like data structure in a direct manner. Nonetheless, as our aim is to randomly generate values of particular sizes, we must first determine how size can affect the generation process. We discuss this next.

Randomly generating sized trees

Point (a) raised at the end of subsection 3.3.2 highlighted that our generic approach to randomly generating values of particular sizes requires a method for randomly distributing remaining size among arbitrarily many subtrees. In this subsection, we outline an algorithm to achieve this. At the heart of this algorithm are solutions to linear Diophantine equations. Hence, we subsequently describe an efficient method for solving such equations.

Recall the notion of size we have decided upon for a tree is its total number of nodes. The total number of nodes for a given tree, and hence its size, is dictated by its one or more *branching factors*, which, for each non-leaf node, determines its number of children. For example, our previous binary tree datatype

data $Tree \;=\; Leaf\ Int \;\mid\; Node\ Tree\ Tree$

has a branching factor of 2, given by its *Node* data constructor. Trees may have non-uniform branching factors, for example, the following datatype

data $Tree_{\leqslant 3} \;=\; Leaf_{\leqslant 3}\ Int \;\mid\; Node_2\ Tree_{\leqslant 3}\ Tree_{\leqslant 3} \;\mid\; Node_3\ Tree_{\leqslant 3}\ Tree_{\leqslant 3}\ Tree_{\leqslant 3}$

represents possibly non-full ternary trees with branching factors of 2 and 3, given by the data constructors $Node_2$ and $Node_3$, respectively.

A generic method for generating a random tree-like data structure of a particular size n can utilise the datatype's branching factors to determine: (a) whether a tree of size n is valid, that is, whether a tree containing n total nodes can be constructed; and (b) assuming a tree of size n is valid, how to recursively generate a tree of that size in a random fashion. In the remainder of this subsection, we informally develop such an approach.

To exemplify our generation algorithm, we describe how a tree of type $Tree_{\leqslant 3}$ with 11 total nodes can be randomly generated. Figure 3.9 illustrates five stages of generation, where the latter four stages correspond to recursive calls. For simplicity, at most one recursive data constructor is selected at each step, with the remainder being leaves, but this need not be the case (and typically isn't). In the first stage of generation, the binary constructor, $Node_2$, is selected; in the second and third stages, the ternary constructor, $Node_3$, is selected; and in the fourth stage, the binary constructor is selected. In the final stage of generation, the base case is reached and leaf nodes terminate the recursive process. Each stage, with the last being an exception, constructs a partial tree definition containing holes, marked $\square$, to be filled recursively by the next stage.

When a hole is filled with a recursive data constructor ($Node_2$ or $Node_3$), the potential size of the subtree filling that hole increases. In other words, each subtree has size $\geqslant 1$ as the smallest possible tree is a $Leaf_{\leqslant 3}$. However, any subtree constructed with $Node_2$ has size $\geqslant 3$: an increase of 2; and any with $Node_3$, size $\geqslant 4$: an increase of 3. This holds recursively, and thus the size n of any $Tree_{\leqslant 3}$ is given by

$$n \;=\; 1 \;+\; 2b \;+\; 3t \tag{3.1}$$

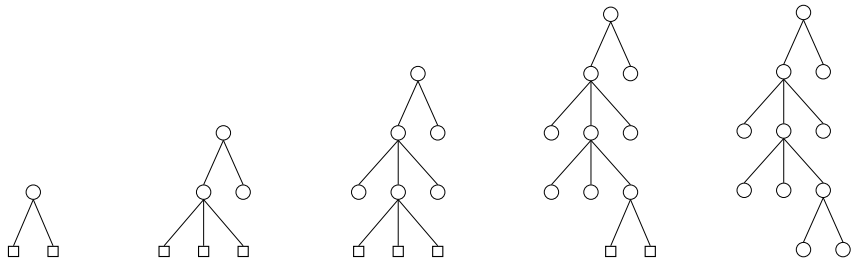


Figure 3.9: Stages of generation: non-full ternary tree of size 11

for b binary nodes of type $Node_2$ and t ternary nodes of type $Node_3$.

Solving equation (3.1) for a given size n not only determines whether a $Tree_{\leqslant 3}$ of that size can be constructed, but the solutions (should they exist) can be used to generate random subtrees of appropriate sizes, such that the overall tree has size n. To see how this works, consider two subtrees s_0 and s_1 of the following sizes:

$$\begin{aligned} |s_0| &= 1 + 2b_0 + 3t_0 \\ |s_1| &= 1 + 2b_1 + 3t_1 \end{aligned}$$

Now suppose that we wish to combine these subtrees to form a larger tree. To do so, we must use the binary data constructor, $Node_2$. If we recall that the size of the new tree must include the root node, then its overall size is calculated as follows:

$$\begin{aligned} |Node_2\ s_0\ s_1| &= 1 + 1 + 2b_0 + 3t_0 \\ &\quad + 1 + 2b_1 + 3t_1 \\ &= 1 + 2 + 2(b_0 + b_1) + 3(t_0 + t_1) \\ &= 1 + 2(1 + b_0 + b_1) + 3(t_0 + t_1) \\ &= 1 + 2b + 3t \end{aligned}$$

Similarly, given three subtrees s_0, s_1, and s_2 with sizes

$$\begin{aligned} |s_0| &= 1 + 2b_0 + 3t_0 \\ |s_1| &= 1 + 2b_1 + 3t_1 \\ |s_2| &= 1 + 2b_2 + 3t_2 \end{aligned}$$

then these must be combined using the ternary data constructor, $Node_3$, and the size of the new tree (including the root) is calculated as follows:

$$\begin{aligned}
|Node_3\ s_0\ s_1\ s_2| &= 1 + 1 + 2b_0 + 3t_0 \\
&\quad + 1 + 2b_1 + 3t_1 \\
&\quad + 1 + 2b_2 + 3t_2 \\
&= 1 + 3 + 2(b_0 + b_1 + b_2) + 3(t_0 + t_1 + t_2) \\
&= 1 + 2(b_0 + b_1 + b_2) + 3(1 + t_0 + t_1 + t_2) \\
&= 1 + 2b + 3t
\end{aligned}$$

Calculating backwards, we see that solutions to (3.1) give sufficient information to generate subtrees of the required sizes. This is perhaps obvious, however, it is worth noting as it has practical significance: random trees generated according to the solutions of equation (3.1) are guaranteed to have size n *by construction*. Hence, there is no need to implement safety checks in the corresponding generation function. As generators are often implemented recursively, this improves efficiency, especially when producing large trees.

Continuing on with our example, we consider solutions to the following equation, which corresponds to the first stage of generation in figure 3.9:

$$\begin{aligned}
& 1 + 2b + 3t = 11 \\
\Leftrightarrow & \\
& 2b + 3t = 10
\end{aligned}$$

A polynomial equation of this form, whereby integer solutions are sought, is called *Diophantine*. Shortly, we will introduce an efficient method for solving linear Diophantine equations, which can be applied in this instance. For now, we just review the results of this method. There are two solutions to the above equation whereby b and t are positive integers:

$$\{\ b = 2,\quad t = 2\ \}$$
$$\{\ b = 5,\quad t = 0\ \}$$

The first solution above states that a tree of size 11 can be generated using two binary nodes and two ternary nodes, while the second solution states that using just five binary

nodes is also a possibility. Either solution is adequate: the first corresponds to our running example and so we continue on with that. The solution $\{\ b = 2,\ t = 2\ \}$ tells us *which* recursive nodes must be used, on the other hand, it does not dictate *how* such nodes should be configured. Therefore, this is a choice that can be made randomly.

Figure 3.9 shows that a binary node is randomly chosen in the first generation stage of our example. The remaining nodes are therefore $\{\ b = 1,\ t = 2\ \}$. Recall that the size n_i of every subtree s_i of type $\mathit{Tree}_{\leqslant 3}$ is calculated as follows:

$$n_i \quad = \quad 1 \ + \ 2b_i \ + \ 3t_i$$

To calculate the size n_0 of the left subtree and that n_1 of the right we can, therefore, randomly distribute the values $b = 1$ and $t = 2$ between the following equations:

$$\begin{aligned} n_0 \quad &= \quad 1 \ + \ 2b_0 \ + \ 3t_0 \\ n_1 \quad &= \quad 1 \ + \ 2b_1 \ + \ 3t_1 \end{aligned}$$

Our running example illustrates a generation process whereby only one recursive constructor (of type Node_2 or Node_3) is chosen at each generation stage. This corresponds to a completely one-sided distribution of the following form:

$$\begin{aligned} n_0 \quad &= \quad 1 \ + \ 2(b_0 = 1) \ + \ 3(t_0 = 2) \\ &= \quad 9 \\ n_1 \quad &= \quad 1 \ + \ 2(b_1 = 0) \ + \ 3(t_1 = 0) \\ &= \quad 1 \end{aligned}$$

However, in practice, this needn't be—and typically isn't—the case.

In figure 3.9, the left and right subtrees in the second generation stage are produced randomly according to these calculated sizes, respectively. In the latter case, a tree of size 1 must be a leaf node. In the former case, a random solution to the equation

$$1 \ + \ 2b \ + \ 3t \quad = \quad 9$$

is sought, and the process repeats. As stated previously, a positive integer solution to $2b \ + \ 3t = \ 8$ is guaranteed to exist at this point. In general, however, a different solution

to the previous one, $\{\ b = 1,\ t = 2\ \}$, might arise in practice, for example, $\{\ b = 4,\ t = 0\ \}$. Hence, the configurations of subtrees are not fixed until the base case (a leaf node) is reached. This design choice does not affect randomness as each solution is chosen uniformly, but it does allow for a more direct recursive implementation. Nonetheless, *repeatedly* solving equations of the above form requires an efficient solver. We introduce this next.

Calculations relating to each generation stage of figure 3.9 are as follows:

Generation stage	Equation	Random solution	Random size distribution
1	$1 + 2b + 3t = 11$	$\{\ b = 2,\ t = 2\ \}$	$\{\ n_0 = 9,\ n_1 = 1\ \}$
2	$1 + 2b + 3t = 9$	$\{\ b = 1,\ t = 2\ \}$	$\{\ n_0 = 1,\ n_1 = 6,\ n_2 = 1\ \}$
3	$1 + 2b + 3t = 6$	$\{\ b = 1,\ t = 1\ \}$	$\{\ n_0 = 1,\ n_1 = 1,\ n_2 = 3\ \}$
4	$1 + 2b + 3t = 3$	$\{\ b = 1,\ t = 0\ \}$	$\{\ n_0 = 1,\ n_1 = 1\ \}$
5	n/a	n/a	n/a

What is important to note is that each equation in the second column of this table is effectively an instantiation of the more general equation

$$a_0 bf_0 + a_1 bf_1 + \cdots + a_k bf_k = n - 1$$

whose solutions for $a_i \in \mathbb{N}$ can be used to randomly generate any *regular* tree-like data structure with branching factors bf_0, bf_1, ..., bf_k for a given size $n \geqslant 1$. In this manner, the steps of the algorithm outlined above generalise in the obvious way.

Solving Diophantine equations

A linear Diophantine equation in two variables is a polynomial equation

$$ax + by = c \tag{3.2}$$

whereby solutions are sought with a, b, and c integers. Such equations can be solved completely and their solutions are closely tied to modular arithmetic. Non-linear Diophantine equations do arise in practice. However, no general algorithm for solving them is possible. Fortunately, we are only interested in the first-order variant.

Congruence methods provide a useful tool in determining the number of solutions to a Diophantine equation. Applied to equation (3.2), these methods show that it either has no solutions or infinitely many, according to whether the greatest common divisor (gcd) of a and b is a factor of c: if not, there are no solutions; if so, there are infinitely many.

Assuming that $\gcd(a, b)$ is a factor of c, then the infinitely many solutions to equation (3.2) can be calculated from any one solution. A common method of finding a single solution is to use the extended Euclidean algorithm. In addition to $\gcd(a, b)$, this algorithm also computes the coefficients of Bézout's identity, which are integers u and v such that:

$$au \ + \ bv \ = \ \gcd(a, b) \tag{3.3}$$

As we assumed that $\gcd(a, b)$ is a factor of c, then there exists an integer k such that $k \ * \ \gcd(a, b) \ = \ c$. Multiplying equation (3.3) by k gives:

$$a(ku) \ + \ b(kv) \quad = \quad k \ * \ \gcd(a, b) \quad = \quad c$$

Hence, one solution to equation (3.2) is $x_0 = ku$ and $y_0 = kv$.

Suppose there exists another solution to equation (3.2), that is:

$$a\bar{x} \ + \ b\bar{y} \ = \ c \tag{3.4}$$

Taking the difference between equations (3.2) and (3.4) gives:

$$a(x_0 \ - \ \bar{x}) \ + \ b(y_0 \ - \ \bar{y}) \quad = \quad 0$$

Thus, we have that $a(x_0 - \bar{x}) = b(\bar{y} - y_0)$, which means that a divides $b(\bar{y} - y_0)$, and, in consequence, that $\frac{a}{\gcd(a,b)}$ divides $\bar{y} - y_0$. Therefore, we have $\bar{y} = y_0 + r\frac{a}{\gcd(a,b)}$ for some integer r. Substituting this into the equation $a(x_0 - \bar{x}) = b(\bar{y} - y_0)$ gives:

$$a(x_0 \ - \ \bar{x}) \quad = \quad rb\frac{a}{\gcd(a,b)}$$

And rearranging the above equation gives:

$$\bar{x} \;=\; x_0 \;-\; r\tfrac{b}{\gcd(a,b)}$$

Thus, if $\{x_0, y_0\}$ is a solution to (3.2), then all other solutions are of the form:

$$\bar{x} \;=\; x_0 \;-\; r\tfrac{b}{\gcd(a,b)} \qquad \bar{y} \;=\; y_0 \;+\; r\tfrac{a}{\gcd(a,b)}$$

Although we need only consider first-order Diophantine equations, we must generalise to account for equations in n variables so that we can solve Diophantine equations derived from multiple non-uniform branching factors, as exemplified previously. This can be achieved by iterating the above method involving the extended Euclidean algorithm. To see how this works, consider solving the following Diophantine equation in three variables:

$$ax \;+\; by \;+cz \;= d \tag{3.5}$$

This is equivalent to solving the following equation in two variables

$$\gcd(a,b)w \;+\; cz \;=\; d$$

as any solution to the latter yields a solution to the former, and vice-versa. This correspondence is due to the fact that the set of possible values of $ax + by$ is precisely the set of all multiples of $\gcd(a, b)$. This explains why we can determine whether equation (3.2) has integer solutions by simply checking if $\gcd(a, b)$ divides c. Moreover, it means that equation (3.5) has solutions if and only if $\gcd(\gcd(a, b), c) = \gcd(a, b, c)$ divides d. Hence, in the general case, a linear Diophantine equation of the form

$$a_0x_0 \;+\; a_1x_1 \;+\; a_2x_2 \;+\; \cdots \;+\; a_kx_k \;=\; d$$

with $k \geqslant 1$ has solutions if and only if $\gcd(a_0, a_1, \ldots, a_k)$ is a factor of d.

In summary, we see that Euclid's extended algorithm provides an effective way of solving linear Diophantine equations in two variables, and those in n variables can be solved by iteratively solving modified equations in two variables. As demonstrated previously, solutions to linear Diophantine equations play an important role in our generic approach to random data generation. As such, the AutoBench system implements an efficient solver for such

equations based on the method described above. In practice, we only seek *positive* integer solutions given that they are used to generate values of strictly positive sizes. Hence, it is not necessarily the case that either zero or infinitely many solutions exist for our purposes. Further implementation details can be found online (Handley 2019).

A generic generator

Finally, we bring together our uniform view on data and our efficient solver for linear Diophantine equations to implement a generic function capable of generating random values of regular tree-like data structures for a given size. Unlike with our previous discussions, we provide specific implementation details to concretise our approach. To make the specifics more accessible, high-level explanations accompany each code segment. Some implementation details are omitted for brevity, nonetheless, full details can be found in the comments of AutoBench's source code, which is available online (Handley 2019).

Calculating branching factors

Recall the following equation from the end of section 3.3.2

$$a_0 bf_0 \ + \ a_1 bf_1 \ + \ \cdots \ + \ a_k bf_k \ = \ n \ - \ 1 \tag{3.6}$$

used to determine whether a value of size n is valid for a tree-like data structure. Furthermore, recall that each bf_i is a branching factor of such a datatype.

To transform the above equation into Diophantine form, we must instantiate each bf_i. For any type a that is both *Generic* and *Typeable*. This is achieved using *recs*:

```
recs :: forall a . (Generic a, Typeable a, ...) ⇒ ...
recs = unPOP (pure matchTy)
  where
    matchTy :: forall b . Typeable b ⇒ K Int b
    matchTy = case eqT @a @b of
      Just _  → K 1
      Nothing → K 0
```

For each data constructor C_i of type a, *recs* essentially outputs a list of lists. Each inner list xs_i is a binary sequence whose values $xs_{i,j}$ represent whether or not the argument x_j to constructor C_i is *recursive*. For example, recall our previous datatype $Tree_{\leqslant 3}$:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}_{\leqslant 3} \\
&\quad = \mathit{Leaf}_{\leqslant 3}\ \underline{\mathit{Int}}_{0} \\
&\quad \mid \mathit{Node}_{2}\ \underline{\mathit{Tree}_{\leqslant 3}}_{1}\ \underline{\mathit{Tree}_{\leqslant 3}}_{1} \\
&\quad \mid \mathit{Node}_{3}\ \underline{\mathit{Tree}_{\leqslant 3}}_{1}\ \underline{\mathit{Tree}_{\leqslant 3}}_{1}\ \underline{\mathit{Tree}_{\leqslant 3}}_{1}
\end{aligned}
$$

The result of applying *recs* to this type is as follows:

$$
\begin{aligned}
&> \mathit{toList}\ (\mathit{recs}\ @\mathit{Tree}_{\leqslant 3}) \\
&[\ [0],\ [1,1],\ [1,1,1]\ ]
\end{aligned}
$$

Notice how each inner list $[0]$, $[1,1]$, and $[1,1,1]$ corresponds to each binary sequence annotating the arguments to $Tree_{\leqslant 3}$'s data constructors above, respectively. In particular, a 0 occurs when an argument is not recursive, for example, *Int*; and a 1 occurs when an argument is recursive, that is, when it is of type $Tree_{\leqslant 3}$.

The implementation of *recs* requires a *Typeable* constraint because the type b of each constructor's argument must be compared—using *matchTy*—with the overall type a to determine whether it is recursive. *Typeable* instances *must* be derived by GHC since version 7.8. To apply *recs* we must use *visible type applications*, which are written $@a$ for some type a, as in $\mathit{recs}\ @\mathit{Tree}_{\leqslant 3}$. Visible type applications were first introduced in GHC version 8.0.

Summing up the values in each inner list returned by *recs* (and filtering) gives us the branching factors of $Tree_{\leqslant 3}$. Each branching factor $bf_i > 0$ is then used to transform equation (3.6) into the form of a linear Diophantine equation.

Validating the size parameter

Once equation (3.6) can be instantiated for a given datatype, it is straightforward to check whether or not a particular size n is valid for that type. That is, whether a value of the datatype with n total nodes can be constructed.

$$
\mathit{linearDiophantine}_{+}\ ::\ [\mathit{Int}]\ \rightarrow\ \mathit{Int}\ \rightarrow\ [[\mathit{Int}]]
$$

Given a list of branching factors *bfs* and a size $n-1$, $linearDiophantine_+\ bfs\ (n-1)$ returns at most 10 (randomly chosen) solutions to equation (3.6). The solutions are returned as lists xs_i where each $xs_{i,j}$ is a positive coefficient for each bf_j, respectively.

$$isValidSize :: (\ldots) \Rightarrow Int \rightarrow Bool$$
$$isValidSize\ n = notNull\ (linearDiophantine_+\ (toList\ (bfs\ @a))\ (n - 1))$$

In turn, given a type a, the function *isValidSize* returns the branching factors *bfs* of a and then checks whether $linearDiophantine_+\ bfs\ (n-1)$ returns any solutions. For example, we know from our previous discussions that binary trees of type $Tree_2$ are always valid for odd sizes, and furthermore, are always invalid for even sizes:

$> isValidSize\ @Tree_2\ 5$ $\qquad$ $> isValidSize\ @Tree_2\ 10$

True $\qquad$ *False*

Conversely, we know that ternary trees of type $Tree_3$ have the opposite validity:

$> isValidSize\ @Tree_3\ 5$ $\qquad$ $> isValidSize\ @Tree_3\ 10$

False $\qquad$ *True*

Finally, we know the following from our worked example in section 3.3.2:

$> isValidSize\ @Tree_{\leqslant 3}\ 11$

True

Remark. In practice, *isValidSize* performs an extremely efficient check. This allows AutoBench to (attempt to) correct invalid sizes by performing a binary search to find the closest valid size in both a positive and negative direction. In fact, this was demonstrated in the second example of section 3.2 when flattening trees to list form.

A generic, random, sized data generator

For any regular tree-like data structure that is both *Generic* and *Typeable*, and a size $n \geqslant 1$, we can now define a function that will uniformly generate random values of the appropriate type containing precisely n nodes in total. We note that the following implementation does not perform any safety checking. This is because AutoBench tests the validity of n using *isValidSize* before this function is called in practice.

Overall, our generation function is rather complex and so we isolate its separate components to present its implementation in a bottom-up fashion. In addition, we demonstrate the use of each component on the $Tree_{\leqslant 3}$ datatype.

Firstly, given a datatype's branching factors *bfs* and a size $n \geqslant 2$, we generate a random set of possible *branches*: at most one for each branching factor.

```
branches :: Int → [Int] → Gen [(Int, [Int])]
branches n bfs = fmap filtPs (mapM (branch n bfs) bfs)
  where
    filtPs = filter (notNull ∘ snd)
    branch = ...
```

That is, we determine whether each *recursive* data constructor of the datatype is a viable option for this stage of generation. If so, we randomly distribute the remaining size, $n - 1$, among its child nodes. For example, applying *branches* to the $Tree_{\leqslant 3}$ datatype, which has branching factors 2 and 3, gives the following results:

```
> generate (branches [2,3] 11)        > generate (branches [2,3] 11)
[ (2,[5,5]), (3,[3,6,1]) ]            [ (2,[7,3]), (3,[4,1,5]) ]
```

As in section 3.3.2, we see that a $Tree_{\leqslant 3}$ with 11 total nodes can be produced by first using either the binary data constructor, $Node_2$, or the ternary data constructor, $Node_3$. On the other hand, a tree with 3 nodes in total can only be produced using the binary constructor, and a tree with precisely 2 nodes is invalid:

```
> generate (branches [2,3] 3)         > generate (branches [2,3] 2)
[(2,[1,1])]                           []
```

For simplicity, we define the *branches* function in terms of *branch*:

```
branch :: Int → [Int] → Int → Gen (Int, [Int])
branch n bfs bf = fmap (bf,) sizes
  where
    sols  = linearDiophantine+ bfs (n - succ bf)
    mults = fmap (succ ∘ sum) ∘ transpose ∘ zipWith (fmap ∘ (*)) bfs
```

```
sizes
  | bf ⩾ n    = pure []
  | null sols = pure []
  | otherwise = do
      sol ← elements sols
      fmap mults (mapM (split 0 bf) sol)
```

Notice that *branch* solves a modified version of equation (3.6), where $n-1$ is replaced with $n - \mathit{succ}\ \mathit{bf}$. This is to ensure that n is sufficiently large to account for the size of the recursive data constructor ($= 1$) and the minimum size of its children ($= \mathit{bf}$). It is easy to see that any solution to this modified equation gives a solution to the original equation (note that this was demonstrated previously in section 3.3.2).

Using a random solution from $\mathit{linearDiophantine}_+$, *branch* returns a list of random, strictly positive *sizes* n_0, ..., n_{bf-1} that sum to $n-1$. Each n_i may be used to subsequently generate each subtree s_i of the data constructor with branching factor bf, hence it is calculated using a *sol* to ensure validity. An empty list of *sizes* is returned by *branch* if the data constructor is invalid. This occurs when either $\mathit{bf} \geqslant n$, as in *branches* $[2,3]$ 2 above, or when *sols* is empty. In the latter case, this occurs for $\mathit{bf} = 2$ in the following example:

```
> generate (branches [2,3] 4)
[(3,[1,1,1])]
```

To calculate each size n_i, *branch* uses the *split* function:

```
split :: Int → Int → Int → Gen [Int]
split min p total = ...
```

which randomly divides a positive integer *total* into p integer parts such that each part has a minimum value *min*. In particular, *split* is used to divide each coefficient a_i from the randomly chosen solution in *sols* among each n_i (as shown in section 3.3.2).

The next component of our generation function is *sizeGens*. It has three arguments: a branching factor *bf*, an n-ary product *bfSeq* of integers, and an n-ary product *sGens* of sized generators. Note that *SizedGen* will be introduced shortly.

```
sizeGens :: ( ... ) ⇒ K Int a → NP (K Int) a → NP SizedGen a → NP Gen a
sizeGens (K bf) bfSeq sGens = ...
```

Previously, we introduced the type *NP* of n-ary products as taking only one argument (a type-level index list). In reality, the situation is more complex. The *NP* type constructor actually takes two arguments, and is defined as follows:

```
data NP (f :: Type → Type) (xs :: [Type]) where
  Nil  :: NP f '[]
  (:*) :: f x → NP f xs → NP f (x ': xs)
```

In simple terms, an n-ary product is a generalised heterogenous list, where each element is given by applying a particular type constructor to one of the types in the index list. More precisely, *NP* also abstracts over a type constructor f, with elements of type $f\ x$ where x is a member of the index list xs (Löh 2015). The isomorphism between n-ary products and heterogenous lists can be seen by taking f as the identity type constructor, I. N-ary products are also isomorphic to homogenous lists, which can be seen by taking f to be the constant type constructor, K, applied to some type a.

```
newtype I a     = I { unI :: a }
newtype K a b = K { unK :: a }
```

The main idea behind the *sizeGens* function is to instantiate the size parameters of generators used to produce arguments for a particular data constructor of a generic datatype. In practice, the size parameter for each branching factor *bf* is calculated using *branches*. Moreover, the n-ary product *sGens* of sized generators has the representation type of a particular data constructor with branching factor *bf*. To clarify this point, recall the previous result from *branches* $[2,3]$ 11 for the $Tree_{\leqslant 3}$ datatype:

```
> generate (branches [2,3] 11)
[ (2,[7,3]),  (3,[4,1,5]) ]
```

Then, the sizes 4, 1, and 5 would be passed to the elements of *sGens* with type

```
NP SizedGen '[ Tree≤3, Tree≤3, Tree≤3 ]
```

respectively. The type $'[\ \mathit{Tree}_{\leqslant 3}, \mathit{Tree}_{\leqslant 3}, \mathit{Tree}_{\leqslant 3}\ ]$ represents $\mathit{Tree}_{\leqslant 3}$'s ternary data constructor, Node_3. Furthermore, each element of *sGens* has type $\mathit{SizedGen}\ \mathit{Tree}_{\leqslant 3}$, where *SizedGen* is simply a wrapper around the function making QuickCheck's size parameter explicit:

```
newtype SizedGen a = SizedGen { unSizedGen :: Int → Gen a }
```

Recursive data constructors can also have arguments that are non-recursive. As such, *bfSeq* is the corresponding sequence of bits calculated by *recs*. It determines when a size from the result of *branches* should be passed to an element of *sGens*, that is, when the current element of the sequence is 1. When the current element is 0, an alternative generation method can be used. In this instance, we simply use a default size of 100:

```
sizeGens :: ( ... ) ⇒ K Int a → NP (K Int) a → NP SizedGen a → NP Gen a
sizeGens _       Nil    Nil      = Nil
sizeGens (K bf) bfSeq sGens = go bfSeq sGens 0
  where
    sizes = fromJust (lookup bf (... branches ...))
    go :: ( ... ) ⇒ NP (K Int) a → NP SizedGen a → Int → NP Gen a
    go Nil Nil _ = Nil
    go (K 0 :* xs1) (sGen :* xs2) idx =
      unSizedGen sGen 100 :* go xs1 xs2 idx
    go (K 1 :* xs1) (sGen :* xs2) idx =
      unSizedGen sGen (sizes !! idx) :* go xs1 xs2 (idx + 1)
```

Parameterising each of *sizeGens*'s arguments by the same type a is particularly useful. This is because it notionally allows all arguments to all data constructors of a generic datatype to be generated easily, by means of a three-way zip:

```
allGens :: ( ... ) ⇒ NP (NP Gen) (Rep a)
allGens = hzipWith3 sizeGens (bfs @a) (recs @a) sGens
  where sGens = unPOP (pure (SizedGen sizedArbitrary))
```

In this manner, the *allGens* function applies *sizeGens* to n-ary products indexed by each list of argument types in the representation type $\mathit{Rep}\ a$ of the generic type a. As an example, consider the simplified representation $\mathit{Tree}^{\mathit{Rep}}_{\leqslant 3}$ of the $\mathit{Tree}_{\leqslant 3}$ datatype:

$$\textbf{type}\ Tree^{Rep}_{\leqslant 3} = (NP\ '[\ '[\ Int\ ],\ '[\ Tree_{\leqslant 3},\ Tree_{\leqslant 3}\ ],\ '[\ Tree_{\leqslant 3},\ Tree_{\leqslant 3},\ Tree_{\leqslant 3}\ ]\ ])$$

In this case, *sizeGens* would be applied to n-ary products indexed by each inner list of $Tree^{Rep}_{\leqslant 3}$: $'[\ Int\ ]$, $'[\ Tree_{\leqslant 3}, Tree_{\leqslant 3}\ ]$, $'[\ Tree_{\leqslant 3}, Tree_{\leqslant 3}, Tree_{\leqslant 3}\ ]$. This may appear problematic because *branches* does not necessarily return a result for every recursive data constructor, as we have seen in previous examples. This issue does not arise in practice, however, as the result of *allGens* is used lazily. In particular, only results for valid constructors are consumed from the output of *allGens*.

Remark. The definition of *allGens* above uses the *sizedArbitrary* function to generate the random arguments for each data constructor. Consequently, every argument type of every data constructor must be a member of the *SizedArbitrary* type class. This is an assumption we previously discussed in section 3.3.2.

The result from *allGens* can be collapsed to a list of generation options:

$$opts :: (\ldots) \Rightarrow [(Int,\ Gen\ a)]$$
$$opts = hcollapse\ (hzipWith\ aux\ (bfs'\ @a)\ (injections\ `hap`\ allGens))$$

Each option is a tuple consisting of a branching factor *bf* and a QuickCheck generator that will produce a value of type *a* using a data constructor with branching factor *bf*. One option is produced for each constructor of the generic type *a*. As previously mentioned, a number of these options may be erroneous. However, only valid options will be selected.

We use *bfs′* for *opts* instead of *bfs* to account for leaf nodes (data constructors with an effective branching factor of 0). In turn, *hap* corresponds to (<∗>) for n-ary products and *injections* essentially converts *allGens* nested n-ary product structure into an n-ary sum of n-ary products, which, as we noted previously, is the underlying structure of *a*'s representation type, *Rep a*. Doing so allows the *aux* function

$$aux :: (\ldots) \Rightarrow K\ Int\ b \rightarrow K\ (NS\ (NP\ Gen)\ (Rep\ a))\ b \rightarrow K\ (Int, Gen\ a)\ b$$
$$aux\ (K\ bf)\ (K\ g) = K\ (bf,\ fmap\ to\ (hsequence\ (Rep\ g)))$$

to convert each data constructor's n-ary product of generators into a single generator for

that constructor of type $Gen\ a$. We omit the details of *aux* for brevity but note its use of the *to* function, which is one half of the isomorphism $Rep\ a \to a$.

In the case of $Tree_{\leqslant 3}$, the options list would consist of three elements:

```
opts :: ( ... ) ⇒ [(Int, Gen Tree≤3)]
opts = [ (0, genLeaf≤3), (2, genNode2), (3, genNode3) ]
```

The generator for the $Leaf_{\leqslant 3}$ data constructor, $genLeaf_{\leqslant 3}$, will always be valid. However, the generators for the $Node_2$ and $Node_3$ data constructors, $genNode_2$ and $genNode_3$, respectively, may be erroneous, depending on the given size parameter n.

To ensure only valid options are used, the erroneous options are filtered:

```
fOpts :: ( ... ) ⇒ Int → [(Int, Gen a)] → [(Int, Gen a)]
fOpts n
  | n == 1 = filter ((== 0)       ∘ fst)
  | otherwise = filter ((∈ validBfs) ∘ fst)
  where
    validBfs :: [Int]
    validBfs = fmap fst (... branches ...)
```

In the base case, when the size parameter n is equal to 1, *fOpts* filters all non-leaf options (those with branching factors > 0). In the recursive case, when $n > 1$, *fOpts* filters all options with branching factors not returned by *branches*.

Finally, a generic generator uniformly picks from the filtered options:

```
gen :: (Generic a, Typeable a, ...) ⇒ Int → Gen a
gen n = snd <$> oneof (fOpts n opts)
```

Assuming the generic datatype a has at least one leaf node and $n \geqslant 1$ is a valid size parameter, then the filtered list of options is guaranteed to be non-empty and thus *gen* will always produce a correctly sized value. We note that both assumptions are checked by AutoBench before this generation function is called.

Generic *SizedArbitrary* instances

The generator above can easily be used to produce random values for *all* of our previous tree-like data structures except $Tree_*$, as follows:

```
instance SizedArbitrary Int

instance SizedArbitrary List where
  sizedArbitrary = gen

instance SizedArbitrary Tree_2 where
  sizedArbitrary = gen

instance SizedArbitrary Tree_3 where
  sizedArbitrary = gen

instance SizedArbitrary Tree_⩽3 where
  sizedArbitrary = gen
```

In the case of $Tree_*$, the above approach fails on two accounts. Firstly, recall the *matchTy* helper function from *recs*, which uses a notion of type equality to determine whether each argument to a data constructor is recursive or non-recursive:

```
matchTy :: forall a b . Typeable b ⇒ K Int b
matchTy = case eqT @a @b of
  Just _  → K 1
  Nothing → K 0
```

This implementation is too restrictive for the $Tree_*$ datatype as its constructor $Node_*$ of type $[Tree_*] \to Tree_*$ is indeed recursive, but not in the type of $Tree_*$ itself. To account for $Tree_*$, *matchTy* must be generalised to determine whether the type of each data constructor's argument b is a 'supertype' of the overall generic datatype a. In practice, this can be achieved by lexically analysing the type representations (*TypeReps*) of a and b.

Assuming a more general implementation of *matchTy*, then generating random values for $Tree_*$ requires one further addition. This is a *SizedArbitrary* instance for *sized* lists of *sized* elements. A prototype implementation for such a generator is provided by the most recent (experimental) version of AutoBench (Handley 2019), which views such a list as

an internal tree-like node with an arbitrary branching factor. Further work is required to devise a suitable theoretical basis for its implementation, however.

To finalise our running example, a tree of type $Tree_{\leqslant 3}$ with 49 total nodes, generated using the above *SizedArbitrary* instance, is given in figure 3.10.

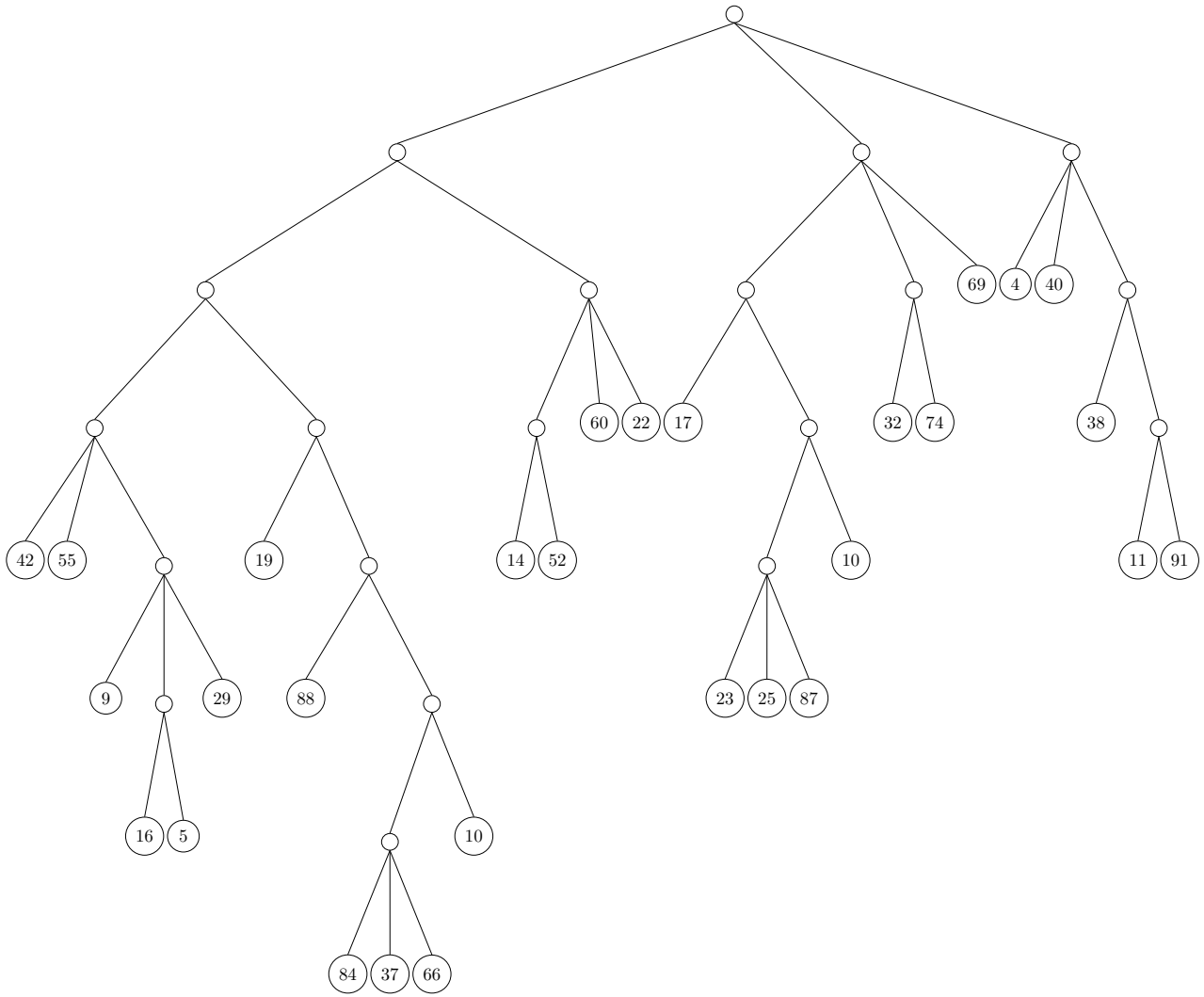


Figure 3.10: A randomly generated tree of type $Tree_{\leqslant 3}$ with 49 total nodes

3.3.3 Benchmarking

AutoBench measures the time performance of programs executed on random test data using the Criterion benchmarking library (O'Sullivan 2014a). Recall from section 2.2.1 that Criterion is popular for many reasons, including its ease of use and because its measurements are notably more accurate than those made by, for example, OS timing utilities.

In this section, we describe how AutoBench automatically generates and executes benchmarks defined using Criterion. The generation process is much the same as how QuickCheck generates random test cases when checking correctness properties. Firstly, we briefly recap how to define benchmarks using Criterion, and then describe the intricacies in benchmarking using generated test data. Secondly, we discuss our system's support for benchmarking functions of any non-zero arity, which we call *polyvariadic benchmarking*. Finally, we explain how randomly generated benchmarks are compiled and executed automatically.

Defining benchmarks

Criterion's principal type is *Benchmarkable*, which is simply a computation that can be benchmarked by the system. A value of this type can be constructed using, for example, 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:

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

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

$$nf\ ::\ NFData\ b\ \Rightarrow\ (a\ \rightarrow\ b)\ \rightarrow\ a\ \rightarrow\ Benchmarkable$$

We also remind readers that *Benchmarkable*s can be defined using the *whnf* function, which evaluates results to weak head normal form only:

$$whnf\ ::\ (a\ \rightarrow\ b)\ \rightarrow\ a\ \rightarrow\ Benchmarkable$$

A *Benchmark* is defined using a *Benchmarkable* along with a suitable description, used to uniquely identify performance measurements for purposes of analysis:

$$bench\ \texttt{"slowRev, [0..200], nf"}\ (nf\ slowRev\ [0\,..\,200])\ ::\ Benchmark$$

Such a benchmark can then be passed to one of Criterion's top-level functions, for example, *defaultMain*, to be executed and have its runtime measured:

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

Benchmarking using generated test data

Each *Benchmarkable* computation defined using *nf* or *whnf* requires a function $f :: a \rightarrow b$ and an argument $x :: a$. In regard to the AutoBench system, the function f will be a program—whose time performance is to be analysed—specified by a user, and its argument x will an input randomly generated using QuickCheck.

Given that Haskell uses a lazy evaluation strategy, in general, we cannot assume that the argument x is in normal form when it is passed to either *nf* or *whnf*. This is problematic for benchmarking because measurements taken by Criterion may then include time spent evaluating (the computations that produce) x prior to applying f. Fortunately, Criterion provides an alternative method for defining benchmarks that ensures x is fully evaluated before f is applied. This alternative method is based on *test environments*:

$$env\ ::\ NFData\ env\ \Rightarrow\ IO\ env\ \rightarrow\ (env\ \rightarrow\ Benchmark)\ \rightarrow\ Benchmark$$

A test environment is created just before its corresponding benchmark is executed. The IO action that creates the environment is run and the newly created environment is evaluated to normal form before being passed to the function that requires it. As such, *env* can be used to safely benchmark functions on randomly generated inputs:

$$env\ (pure\ [0\,..\,200])\ (\lambda dat\ \rightarrow\ bench\ \texttt{"slowRev, [0..200], nf"}\ (nf\ slowRev\ dat))\ ::\ Benchmark$$

Note that the use of *env* imposes a further constraint on programs being analysed by AutoBench, which is that their input types must be members of the *NFData* type class.

Generating random benchmarks

When checking program correctness, the QuickCheck system tests random cases by executing testable properties on randomly generated inputs. In much the same way, AutoBench's approach to checking program efficiency is to test random cases by executing benchmarks constructed using randomly generated inputs of particular sizes. The following two functions can be used to generate such benchmarks:

$$\begin{aligned} \mathit{genNf} &:: (\mathit{SizedArbitrary}\ a,\ \mathit{NFData}\ a,\ \mathit{NFData}\ b) \\ &\Rightarrow (a \to b) \to \mathit{String} \to \mathit{Int} \to \mathit{Benchmark} \\ \mathit{genWhnf} &:: (\mathit{SizedArbitrary}\ a,\ \mathit{NFData}\ a) \\ &\Rightarrow (a \to b) \to \mathit{String} \to \mathit{Int} \to \mathit{Benchmark} \end{aligned}$$

Given a function $f :: a \to b$, an identifier $s :: \mathit{String}$, and a size $n :: \mathit{Int}$, the benchmark $\mathit{genNf}\ f\ s\ n$ uses a random, fully evaluated argument $x :: a$ of size n to measure the time taken to evaluate the computation $f\ x :: b$ to normal form. The latter function, $\mathit{genWhnf}$, acts similarly. However, in this case, $f\ x$ is evaluated to weak head normal form only.

For the purpose of generating random benchmarks to analyse the time performance of a single program in isolation, the above two functions are adequate. However, to *compare* the time performance of two or more programs in this manner, they are not. This is because a fair comparison requires both programs to be tested on the *same* inputs, which cannot, in general, be guaranteed by the above definitions. Again, Criterion provides an alternative approach that can be used in this instance, in the form of *benchmark groups*:

$$\mathit{bgroup} :: \mathit{String} \to [\mathit{Benchmark}] \to \mathit{Benchmark}$$

A benchmark group is simply a list of associated benchmarks with a unique group identifier. Using *bgroup*, the two previous functions can be redefined to accept a non-empty list of program-and-identifier tuples for comparison, as follows:

$$\begin{aligned} \mathit{genNf} &:: (\mathit{SizedArbitrary}\ a,\ \mathit{NFData}\ a,\ \mathit{NFData}\ b) \\ &\Rightarrow [(a \to b,\ \mathit{String})] \to \mathit{Int} \to \mathit{Benchmark} \\ \mathit{genWhnf} &:: (\mathit{SizedArbitrary}\ a,\ \mathit{NFData}\ a) \\ &\Rightarrow [(a \to b,\ \mathit{String})] \to \mathit{Int} \to \mathit{Benchmark} \end{aligned}$$

The result of each new implementation is thus a group of benchmarks, defined by *bgroup*, which tests the runtime of the computation $f\ x :: b$ for each function $f :: a \to b$ in the input list using the *same* randomly generated input $x :: a$ of given the size n.

Polyvariadic benchmarking

When we introduced Criterion previously (section 2.2.1), we explained that its *Benchmarks* are constructed in such a way as to support repeated runs of measurements. Specifically, we remarked that a *Benchmarkable* takes a function $f :: a \to b$ and an input $x :: a$ as *separate* arguments to avoid issues regarding memoisation. For the purposes of AutoBench, this is very fitting as the function $f :: a \to b$ can be specified by the user, and the input $x :: a$ can be randomly generated using QuickCheck. However, what if users wish to compare the performance of non-unary functions, that is, programs with multiple inputs?

Technically speaking, there are no functions of multiple arguments in Haskell, only functions of one argument, some of which may return new functions of one argument. And so when we write $f :: a \to b$ for type parameters a and b, then b may indeed be of the form $c \to d$ for some types c and d. *Currying* is the process of transforming a function that takes multiple arguments in a tuple into a function that takes a single argument—the first from the tuple—and returns another function that accepts a single argument—the second from the tuple—and so on. Hence, we see that all Haskell functions are *curried* functions.

Remark. The above explanation suggests that the type parameters in the signatures of *genNf* and *genWhnf* allow benchmarks for non-unary functions to be generated already.

$$\begin{aligned} genNf\ &::\ (SizedArbitrary\ a,\ NFData\ a,\ NFData\ b) \\ &\Rightarrow\ [(a\ \to\ b,\ String)]\ \to\ Int\ \to\ Benchmark \end{aligned}$$

For example, it is perfectly valid for the type $a \to b$ in the signature above to be instantiated to $Int \to (Bool \to Char)$. This allows a binary function of this curried type to be passed as an argument to *genNf* so long as: (a) *Int* is a member of *SizedArbitrary* and *NFData*; (b) An *NFData* instance exists for function types, in this case, $Bool \to Char$. The latter point is most interesting and has been the topic of some contention. In short, there currently

exists a default *NFData* instance for function types, which evaluates a given function to weak head normal form only. However, functions are already in weak head normal form and so it is essentially a no-op. Benchmarks for non-unary functions generated by *genNf* and *genWhnf* are, therefore, undesirable. Our goal is to rectify this by generating benchmarks that fully saturate functions before they are executed.

Despite a Haskell function of two arguments being curried by default, it is possible to *uncurry* it so that both arguments can be provided at once. More precisely, it is easy to see that a function of type, for example, $Int \to Bool \to Char$ is isomorphic to its uncurried counterpart of type $(Int, Bool) \to Char$, as witnessed by the following two familiar functions:

$$\begin{aligned} curry &:: ((a,\ b) \to c) \to (a \to b \to c) \\ uncurry &:: (a \to b \to c) \to ((a,\ b) \to c) \end{aligned}$$

Using the notion of uncurrying, we can define another pair of benchmark generators that accept functions of (at least) two arguments, as follows:

$$\begin{aligned} genNf_2 &:: (SizedArbitrary\ a,\ SizedArbitrary\ b,\ NFData\ a,\ NFData\ b,\ NFData\ c) \\ &\Rightarrow [(a \to b \to c,\ String)] \to Int \to Benchmark \\ genWhnf_2 &:: (SizedArbitrary\ a,\ SizedArbitrary\ b,\ NFData\ a,\ NFData\ b) \\ &\Rightarrow [(a \to b \to c,\ String)] \to Int \to Benchmark \end{aligned}$$

In practice, each function $f :: a \to b \to c$ is uncurried to the form $f' :: (a, b) \to c$ before being passed to a *Benchmarkable*. This works just fine and only requires an additional instance for *NFData*, which has a straightforward definition:

instance $(NFData\ a,\ NFData\ b) \Rightarrow NFData\ (a, b)$ **where** ...

This approach was implemented in the first version of the AutoBench system, which allowed users to compare the time performance of unary and binary functions only. However, in general, it is not satisfactory due its lack of scalability. This is because an additional pair of generators is required for every non-zero arity, of which there are *many*. Furthermore, each pair of generators requires its own version of *uncurry*:

$$uncurry_3 :: (a \to b \to c \to d) \to (a,\ b,\ c) \to d$$

$$uncurry_4 \;::\; (a \to b \to c \to d \to e) \to (a,\ b,\ c,\ d) \to e$$
$$uncurry_5 \;::\; (a \to b \to c \to d \to e \to f) \to (a,\ b,\ c,\ d,\ e) \to f$$

Polyvariadic uncurrying

To address this scalability issue, the latest version of the system applies the notions of currying and uncurrying *uniformly* at any non-zero arity. That is, as witnesses to the more general isomorphism between a curried function of non-zero arity n and a unary function that takes a non-empty heterogenous list of n arguments:

$$curry_+ \;::\; (HList\ '[\ a_1,\ a_2,\ \ldots,\ a_n\ ] \to r) \to (a_1 \to a_2 \to \cdots \to a_n \to r)$$
$$uncurry_+ \;::\; (a_1 \to a_2 \to \cdots \to a_n \to r) \to (HList\ '[\ a_1,\ a_2,\ \ldots,\ a_n\ ] \to r)$$

Recall from section 3.3.2 that *HList* is the type constructor for heterogenous lists, which are required because arguments to functions need not be uniform in their types.

Defining these two functions in Haskell requires a number of language extensions provided by GHC, including type families and type operators (version 6.8.1 onwards). First, we must express currying at the type level by defining a type family $arguments \to^* results$ to compute a curried function type, taking *arguments* as inputs and returning *result*:

```
type family (arguments :: [Type]) →* (result :: Type) where
  '[]          →* result = result
  (a ': as) →* result = a → (as →* result)
```

We can see this type-level function in action using GHCi's *kind*! command, as follows:

```
> :kind! '[Int, Bool] →* Char
Int → Bool → Char
```

```
> :kind! '[] →* Char
Char
```

Using the type operator $(\to^*)$, we can then specify signatures for our generalised curry and uncurry functions in a type class, which we call *Curry*:

```
class Curry (arguments :: [Type]) where
  curry*   :: (HList arguments → result) → (arguments →* result)
  uncurry* :: (arguments →* result) → (HList arguments → result)
```

Given that *arguments* is a type-level list, and furthermore, by unfolding the definition of $(\rightarrow^*)$, we see that the types of $curry_*$ and $uncurry_*$ are comparable to $curry_+$ and $uncurry_+$, respectively. Nonetheless the functionalities of $curry_*$ and $uncurry_*$ differ slightly from the intended functionalities of $curry_+$ and $uncurry_+$. In particular, the former functions allow their lists of arguments to be empty. Hence, we use the subscript $*$ to mean 'zero or more' arguments rather than +, which means 'one or more'.

We require two instances of the *Curry* class, which are defined recursively. Firstly, in the base case, when the list of *arguments* is empty, $'[\,]$. And then for a non-empty list $(a\ ':\ as)$ assuming the existence of a *Curry* instance for *as*. We omit the implementation details as they are not essential to our discussion.

```
instance Curry '[] where ...
instance Curry as ⇒ Curry (a ': as) where ...
```

Polyvariadic benchmark generation

The $uncurry_*$ function allows us to uncurry a given function of any arity n, returning a unary function that accepts a heterogenous list of n arguments. Using this function, we seek to define two *polyvariadic* benchmark generators. That is, two functions that will generate benchmarks for programs with any number of inputs > 0, whereby programs are fully saturated with all of their required inputs before being executed. The first, $genNf_+$, will ensure results are evaluated to normal form using $nf :: NFData\ b \Rightarrow (a \rightarrow b) \rightarrow a \rightarrow Benchmarkable$, while the second, $genWhnf_+$ will ensure results are evaluated to weak head normal form using $whnf :: (a \rightarrow b) \rightarrow a \rightarrow Benchmarkable$. In addition, all inputs will be randomly generated using QuickCheck, via AutoBench's *SizedArbitrary* type class.

Given a function $f :: a_1 \rightarrow \cdots \rightarrow a_n \rightarrow r$ of arity n, which constraints must be placed on its type for our goal to be realised in practice? Firstly, *nf* requires f's result type to be a member of the *NFData* type class, however, *whnf* does not. Secondly, recall that impartially comparing the time performance of multiple programs requires the use of benchmark environments, which, in turn, introduces an *NFData* constraint on every argument type of f. Thirdly, as f is to be executed on random test data, each of its argument types must

be a member of *SizedArbitrary*. Finally, in order to use $uncurry_*$, the list of f's argument types must be *Curry*able. Thus, for $genNf_+$, we have the following type:

$$\begin{aligned} genNf_+ \;::\; & (\; as \sim Arguments\ (a \to rest),\ r \sim Result\ (a \to rest) \\ & ,\; All\ SizedArbitrary\ as,\ All\ NFData\ as,\ Curry\ as,\ NFData\ r\;) \\ & \Rightarrow [(a \to rest,\ String)] \to Int \to Benchmark \end{aligned}$$

Recall that $a \to rest$ is the type of a curried function of at least one argument, hence, *rest* may indeed be of type $b \to c$. *Arguments* and *Result* calculate the argument types *as* and result type r of $a \to rest$, respectively. The types *as* and r are then constrained in accordance with the above requirements. To this end, $All :: Constraint \to [Type] \to Constraint$ maps a given constraint across a list of types, constraining each type.

Given a list of functions $f :: a_1 \to \cdots \to a_n \to r$, $genNf_+$ generates a random input of each argument type a_i of the given size, and inserts them into a heterogenous list $xs ::$ $HList\ '[\ a_1,\ a_2,\ \ldots,\ a_n\ ]$. A Criterion test environment is then constructed using xs, which is subsequently evaluated to normal form before being passed to each function's *uncurried* counterpart, $f' :: HList\ '[\ a_1,\ a_2,\ \ldots,\ a_n\ ] \to r$, in turn. This step requires the following *NFData* instances for heterogenous lists, which are recursively defined:

instance $NFData\ (HList\ '[\,])$ **where** ...
instance $(NFData\ a,\ NFData\ (HList\ as)) \Rightarrow NFData\ (HList\ (a\ ':\ as))$ **where** ...

Finally, the time taken to evaluate the results of each *fully saturated* function, that is, each computation $f'\ xs$, to normal form is measured.

The definition and functionality of $genWhnf_+$ is similar to $genNf_+$. The only distinction is that $genWhnf_+$ does not require an *NFData* instance for the result type of $a \to rest$ as its degree of evaluation is weak head normal form only:

$$\begin{aligned} genWhnf_+ \;::\; & (\; as \sim Arguments\ (a \to rest),\ All\ SizedArbitrary\ as \\ & ,\; All\ NFData\ as,\ Curry\ as\;) \\ & \Rightarrow [(a \to rest,\ String)] \to Int \to Benchmark \end{aligned}$$

The ideas underpinning the implementations of $genNf_+$ and $genWhnf_+$ form the basis of polyvariadic benchmark generation in AutoBench. In practice, the generators used by the system differ slightly to those described above, in that they allow users to independently

specify the size of each generated input, rather than constraining them to be of equal size. This essentially requires users to specify a list of sizes for each *Benchmark* to be generated. All of AutoBench's code is open source and can be found on GitHub (Handley 2019).

For completeness, the definitions of *Arguments* and *Result* are as follows

```
type family Arguments (f :: Type) :: [Type] where
  Arguments (a -> rest) = a ': Arguments rest
  Arguments _           = '[]

type family Result (f :: Type) :: Type where
  Result (_ -> rest) = Result rest
  Result r           = r
```

and, as before, their functionalities can be tested using GHCi's *kind*! command:

```
> :kind! Arguments (Int -> Bool -> Char)
'[Int, Bool]
> :kind! Result (Int -> Bool -> Char)
Char
```

Finally, we note that polyvariadic currying is referred to as *arity-polymorphic currying* by Foner, Zhang, and Lampropoulos (2018). This article provides the same implementations of $curry_*$ and $uncurry_*$. Our ideas were developed independently, however, we chose the name of the *Curry* type class in light of this work.

Compiling and executing benchmarks

When using QuickCheck, the only difference between checking correctness properties of compiled Haskell code using GHC and checking correctness properties of interpreted code using GHCi is *convenience*. This is because the denotational semantics of Haskell is the same in both cases. Hence, it is preferable to use GHCi as it provides 'instant' feedback.

In contrast, it is preferable to use GHC rather than GHCi when executing benchmarks because 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. This was exemplified in the first example

of section 3.2.1. Furthermore, in background section 2.2, we highlighted the need to assess the performance of programs, for example, subject to different runtime settings. This can only be achieved if source code is compiled using GHC.

Consequently, recall that Criterion provides a number of top-level entry points to be used in the *main* action of a Haskell module. For example, its *defaultMainWith* function takes a list of benchmarks and executes them sequentially:

$$defaultMainWith :: Config \rightarrow [Benchmark] \rightarrow IO\ ()$$

In section 3.2.2, we explained that AutoBench automatically creates a *benchmarking file*, which is a Haskell module containing benchmarks generated in the manner described above. The top-level function of this module executes such benchmarks using *defaultMainWith*.

To ensure testing is fully automated, AutoBench must, therefore, generate benchmarking files, compile them, and then invoke their resulting binary executables on behalf of users. This is achieved using the *Hint* (Martì 2007) and the *GHC API* (GHC Team 2017) packages. The former is used to validate user input files, and the latter is used to compile files generated by the system that contain appropriate benchmarks. Benchmarking executables are invoked using the *Process* (GHC Team 2007) package.

User options

Users can specify any number of command-line flags in order to configure how benchmarks are compiled by GHC and executed by Haskell's runtime environment. Users can also select the degree to which the results of test programs are normalised. That is, whether they are evaluated to normal form or to weak head normal form, which dictates whether benchmarks are generated using $genNf_+$ or $genWhnf_+$, respectively.

Criterion requires a set of user options, which are passed to *defaultMainWith* when executing benchmarks. AutoBench re-exports this configuration in its own user options, allowing Criterion's functionality to be customised as if using the system directly.

3.3.4 Statistical analysis

AutoBench configures the Criterion library to write all of its benchmarking measurements to a JSON report file, which the system then parses. The raw runtimes of test programs are analysed 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.

Efficiency improvements and optimisations

The system generates improvement results by comparing the runtimes of programs pointwise. That is, for each set of input sizes, it compares the runtimes of two programs to determine which program performed better for that test case. If one program performs better than another for a certain percentage of 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 3.2, the system generated the following improvement result

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

which states that *fastRev* was more time efficient than *slowRev* in 95% of test cases. By default, 90% of test cases must show one program to be more efficient than another for an improvement result to be generated. This accounts for the possibility of efficient implementations performing poorly on smaller inputs due to start-up costs, and furthermore, allows for minor anomalies in the raw benchmarking data.

A basic assumption of our system is that the programs being compared are denotationally equal—as we discussed previously, we see no obvious reason as to why users would wish

to compare the performance of functions with the same types but different functionalities. Nonetheless, 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 are 'upgraded' to correctness-preserving optimisations. This was also exemplified in section 3.2, as the results table included an optimisation rather than just an improvement:

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

In order for the QuickCheck test to be applicable, the result type of each program must be a member of the *Eq* type class, the standard Haskell class for checking equality of types. Default *Eq* instances are provided for all standard Haskell types. For custom datatypes, users may provide *Eq* instances inside their input file alongside the programs to be tested. If no *Eq* instance is given, the system will attempt to apply a generic instance, which is implemented using the *Generics-Sop* library (Vries and Löh 2014). However, this may fail, in which case the user is notified that the test could not be performed.

Approximating time complexity

In this subsection, we explain how AutoBench uses a set of runtime measurements and input sizes to approximate the time complexity of the program from which the data was derived. A current limitation of our method is that it is only applicable to two-dimensional data. This means that a time complexity estimate for a program with multiple inputs (and hence, multiple input sizes) is based only on the size of its *first* argument. The remainder of this subsection, therefore, refers to just a single input size. In section 3.5, we discuss how our approach could be extended as part of further work.

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 a random input of size x. The system uses a data set of this kind to approximate the time complexity, for example, 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 calculating a line (or curve) of best fit for a given set of data points, and

then using the equation of this line as an estimate of time complexity.

In the first example of section 3.2, the AutoBench system calculated the following curves of best fit for *slowRev* and *fastRev*, respectively:

$$\begin{aligned} y &= 5.28e{-}9x^2 + 2.28e{-}11x + 2.91e{-}5 \\ y &= 2.53e{-}7x + 1.65e{-}5 \end{aligned}$$

Using these equations, it estimated the time complexities of *slowRev* and *fastRev* as quadratic and linear, respectively. Thus we see that, in general, the system must determine which *type* of function best fits a given data set. Even more surprisingly, there appears to be no standard means to achieve this either. Our method considers many different types of functions and chooses the one with the smallest fitting error according to a suitable measure.

Ordinary least squares

We use a popular method for fitting a function of a particular type to a given data set, which is called *linear regression analysis* (Fox 1997). Given a set of $(x,\ y)$ data points comprising input sizes and runtime measurements, 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 of the form $\hat{y} = mx + c$, where $\hat{y}$ is the runtime predicted by the model for a given input size x. Based on this idea the *ordinary least squares* (OLS) method (Hutcheson 2011) estimates the unknown parameters of the model, which are m and c in the case of the linear model, such that the distance between each y-coordinate in the data set and the corresponding predicted $\hat{y}$-value is minimised:

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

In this context, we refer to the $(x,\ y)$ data points as the *training data*, and the expression being minimised as the *residual sum of squares* (RSS).

Overfitting

When fitting regression models to data sets using the ordinary least squares 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 responding to 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 set also affects the likelihood of overfitting, as, in general, it is more difficult to separate reliable patterns from noise when training with small data sets.

Overall, the AutoBench system must be able to choose between regression models of varying complexities because different programs can have different time complexities. Moreover, the benchmarking process can often be time consuming, and hence it is likely that, in practice, the models will be fitted to comparatively small data sets. The possibility of overfitting must, therefore, be addressed so that comparisons made between models do not naively favour those that overfit training data.

Ridge regression

A regression model that pays a great deal of attention to its training data and, in doing so, does not generalise on data that it hasn't seen before is said to have *high variance*. One way to reduce the variance of models and thus 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 *regularisation*. To achieve this, we use ridge regression (Hoerl and Kennard 1970), 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

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

where $t \geqslant 0$ is a tuning parameter that is *inversely* proportional to the degree of coefficient 'shrinkage': when $t = \infty$ the model parameters are the same as those calculated using OLS; and when $t = 0$ all the coefficients are eliminated.

Constraining the magnitude of coefficients in this manner causes models to place more emphasis on their more influential parameters. In other words, the coefficients of model parameters that have little effect on minimising RSS values are reduced, preventing overfitting. As a concrete example, we can compare the fitting errors of increasingly complex polynomial models when trained on the runtime measurements of *slowRev* given in the first example section 3.2. Using the OLS and ridge regression methods gives the following results, in which the smallest error in each case is highlighted bold:

	Fitting error:	
Model	Ordinary least squares	Ridge regression
Quartic	**6.52e−13**	3.88e−10
Cubic	1.12e−12	1.70e−10
Quadratic	8.40e−12	**7.21e−11**

The results in this table demonstrate the susceptibility of the OLS method to overfitting, as it favours a quartic time complexity for *slowRev*. In contrast, the results calculated using ridge regression indicate a quadratic time complexity, which is what we would 'expect' given the overall trend in *slowRev*'s runtime measurements.

Model selection

Selecting a model from a number of candidates is known as *model selection* (Claeskens and Hjort 2008). This typically involves assessing the accuracy of each model under consideration by calculating a fitting error, and then choosing the model with the least error.

Recall that our overall aim is to use a set of runtime measurements and input sizes to approximate a program's time complexity. As such, it is good practice to assess each model's predictive performance on *unseen* data, that is, data not present 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 the test data. To achieve this on relatively small data sets, the system uses *Monte Carlo cross-validation* (Xu and Liang 2001) alongside ridge regression when assessing model accuracy.

Cross-validation evaluates 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 then combined to give a *cumulative* fitting error.

When all the models have been cross-validated, by default, AutoBench compares them using *predicted mean square error* (PMSE), which is a cumulative error used frequently in model selection involving cross-validation (Claeskens and Hjort 2008). The model with the lowest PMSE is then used to approximate time complexity.

In the first example in section 3.2, the runtime measurements of *slowRev* and *fastRev* were split randomly into 70% training data 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	**7.21e−11**	Linear	**1.59e−11**
2	Cubic	1.70e−10	$n \log_2 n$	1.06e−10
3	Quartic	3.88e−10	$\log_2^2 n$	2.17e−10

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

User options

Users of the system are able to specify which types of functions should be considered when approximating time complexity, and how to compare them. AutoBench currently supports the following types of functions: 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 and parameters are considered.

Regression models can be compared using a number of fitting statistics besides PMSE, including R^2, adjusted R^2, predicted R^2, and predicted square error.

3.4 Case studies

In this section, we further demonstrate the use of AutoBench with four case studies. In each case study, the programs being tested are added to a file along with appropriate *SizedArbitrary* generators to produce necessary inputs, *NFData* instance declarations to ensure test cases can be fully evaluated, and user options. Unless otherwise stated, the user options specify the size range of the test data and configure the results of test cases to be evaluated to normal form. Test files are then passed to AutoBench, as demonstrated in example two of section 3.2, and the time performance of the programs compared.

3.4.1 Case study 1: QuickSpec

Research on property-based testing has also introduced the notion of *property generation*. Given a number of functions and variables, *QuickSpec* (Claessen, Smallbone, and Hughes 2010) will generate a set of correctness properties that appear to hold for the functions based upon QuickCheck testing. Such equations can be used to help better understand the program, or as lemmas to help prove the program correct.

Given the append function ($+\!\!+$), the empty list $[\,]$, and variables xs, ys, and zs, QuickSpec will generate the following identity and associativity properties:

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

In previous work by Moran and Sands (1999), these equational laws have been formally shown to be time efficiency improvements, $\trianglerighteq$, as follows:

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

With this in mind, a fitting first case study for our system is to test the equational properties presented in the QuickSpec paper (Claessen, Smallbone, and Hughes 2010) to see which give improvement results. For example, our system indicates that all three of the above properties are correctness-preserving optimisations:

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

If the inputs from above are extended to include Haskell's standard list *reverse* and *sort* functions (from the *Prelude* and *Data.List*, respectively), then a number of additional properties are generated by the QuickSpec system, including:

$$
\begin{array}{rcl}
reverse\ (reverse\ xs) & == & xs \\
reverse\ xs \mathbin{+\!\!+} reverse\ ys & == & reverse\ (ys \mathbin{+\!\!+} xs) \\
sort\ (reverse\ xs) & == & sort\ xs \\
sort\ (sort\ xs) & == & sort\ xs
\end{array}
$$

For each equation above, the AutoBench system can be used to compare 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 can be tested by comparing *appRev* and *revApp*, defined as follows:

$$appRev :: [Int] \rightarrow [Int] \rightarrow [Int]$$
$$appRev\ xs\ ys = reverse\ xs +\!\!+ reverse\ ys$$

$$revApp :: [Int] \rightarrow [Int] \rightarrow [Int]$$
$$revApp\ xs\ ys = reverse\ (ys +\!\!+ xs)$$

The graphs of runtime measurements produced by our system for each of the *reverse* and *sort* examples are depicted in figure 3.11. Just as before, performance results indicate that all of the properties are correctness-preserving optimisations:

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

Baseline measurements

Consider the graphs in figure 3.12 for append's identity laws. The first graph indicates that $xs +\!\!+ [\,] \gtrsim xs$ 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 $[\,] +\!\!+ xs \gtrsim xs$ is a constant time improvement because $[\,] +\!\!+ xs$ is the base case of the append's recursive definition. However, both graphs show a linear time complexity for the functions, as, for each test case, the resulting list must be traversed in order to ensure each computation is fully evaluated, which takes linear time in the length of the list.

Although AutoBench 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 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 kinds of situations, AutoBench provides a *baseline* option to measure the cost of fully evaluating the *results* of test cases. When this option is selected, the system applies one of the test programs to each input *before* benchmarking, and then benchmarks an identity operation on the results of these applications. Baseline measurements are plotted as a black, dashed line of best fit, as in figure 3.12.

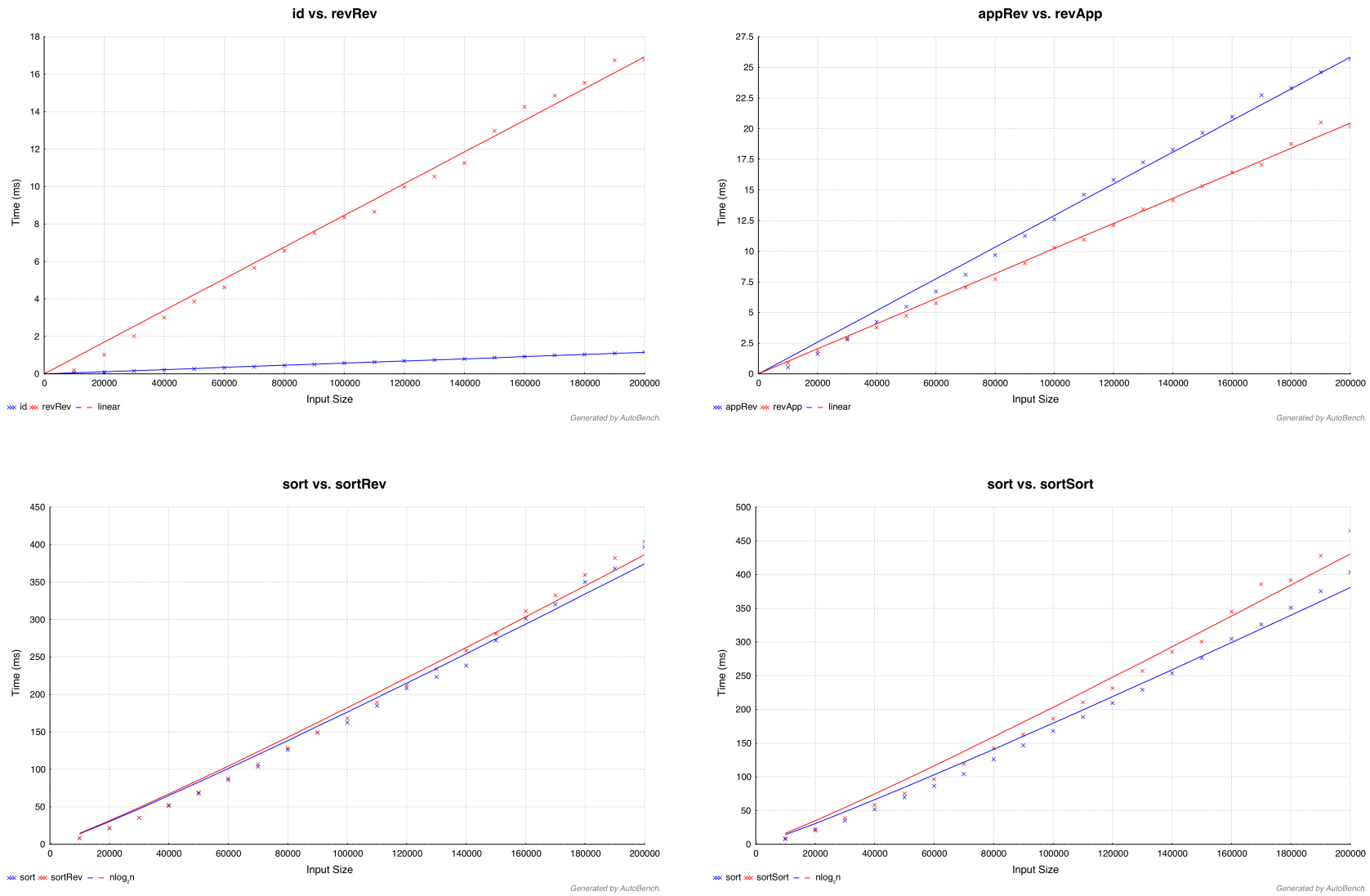
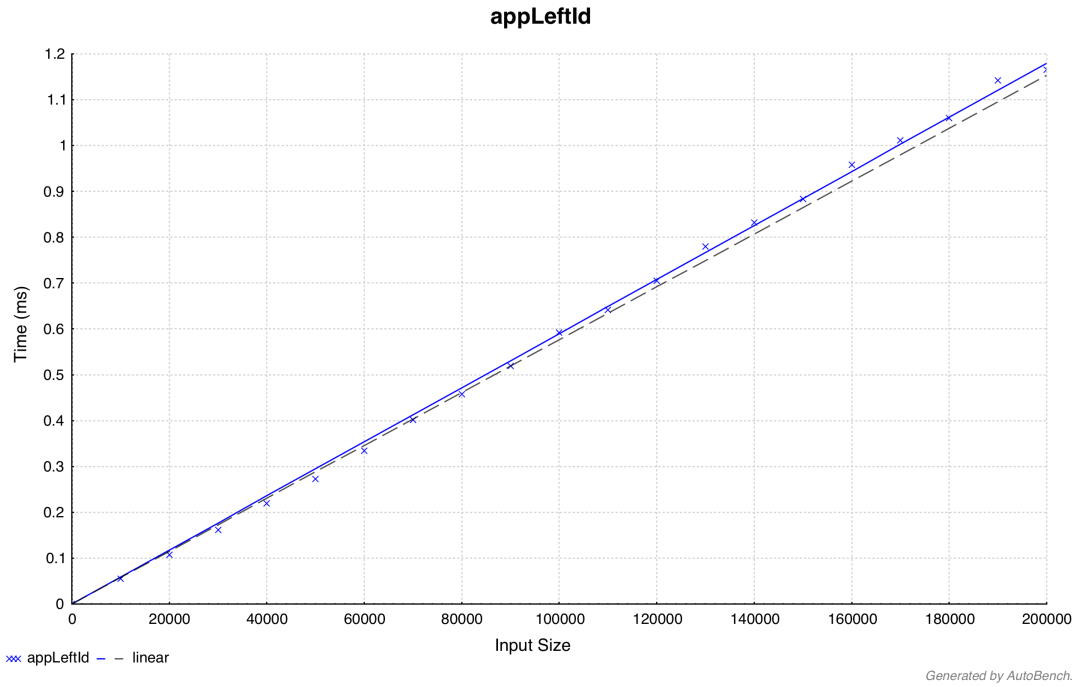
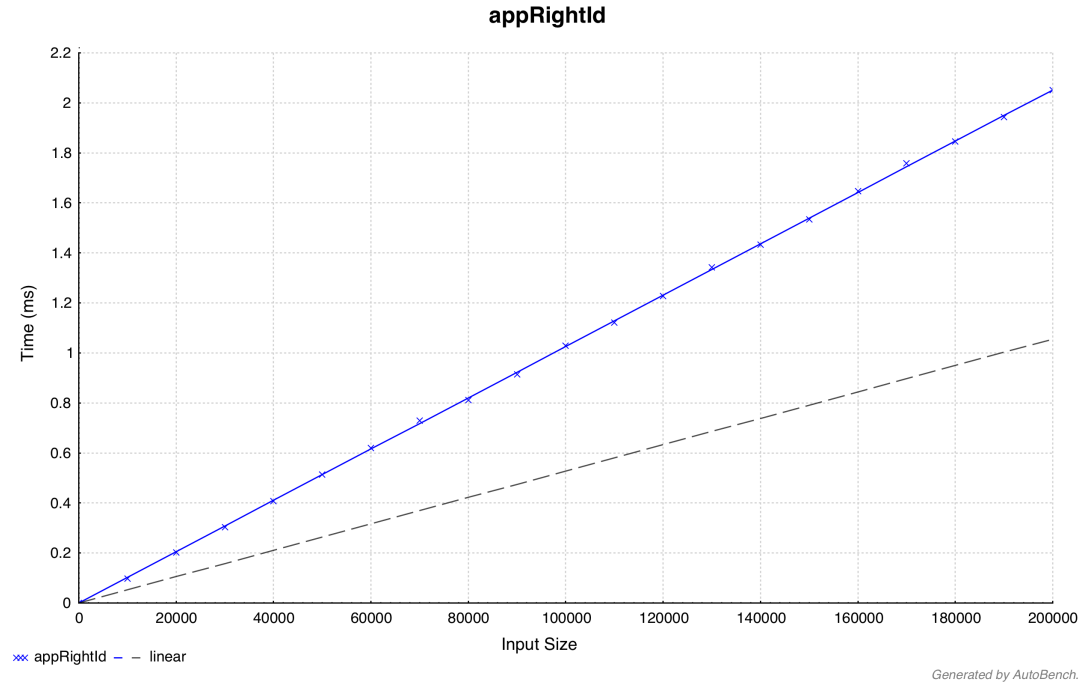


Figure 3.11: AutoBench time performance results: QuickSpec examples



(a) Left identity



(b) Right identity

Figure 3.12: AutoBench time performance results: append's identity laws

If we compare the runtimes of $xs \mathbin{+\!\!+} [\,]$ 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 $[\,] \mathbin{+\!\!+} xs$ approximately match the baseline measurements. Thus, interpreting these results with respect to the baseline measurements suggests that $xs \mathbin{+\!\!+} [\,]$ is linear while $[\,] \mathbin{+\!\!+} xs$ is constant.

AutoBench's GitHub repository (Handley 2019) includes many more examples from the original QuickSpec paper (Claessen, Smallbone, and Hughes 2010), 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.

3.4.2 Case study 2: Sorting

In 2002, the sorting function in Haskell's *Data.List* module was changed from a quicksort algorithm to a merge sort algorithm. Comments in the source file (GHC Team 2001) suggest that the change occurred because the worst-case time complexity of quicksort is $O(n^2)$, while that of merge sort is $O(n \log n)$. Included in the comments is a summary of performance tests, which indicate that the new merge sort implementation does indeed perform significantly better than the previous quicksort implementation in the worst case.

The performance tests carried out at that time predate the development of systems such as Criterion and AutoBench. As such, runtime measurements were taken using an OS-specific timing utility and testing was coordinated using a shell script. Our second case study is, therefore, 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, as stated previously, is much more accurate and robust than an OS timing utility. Secondly, testing is fully automated, so there is no need to develop a custom script. And finally, our system uses linear regression analysis to estimate time complexities, which were not included in the results of the prior analysis.

When sorting lists that are strictly increasing or sorted in reverse order, quicksort suffers from its worst-case time complexity of $O(n^2)$. In contrast, merge sort's time complexity is always $O(n \log n)$. To put this theory to the test, we can compare the time performance of the previous quicksort implementation with that of the new merge sort implementation

from *Data.List* when executed on strictly increasing and reverse sorted lists of integers.

A graph of runtime measurements comparing both implementations sorting random input lists that are strictly increasing is given in figure 3.13a. It shows a significant difference between the runtimes of quicksort and merge sort. Furthermore, for both types of list, our system estimates the time complexity of merge sort as n log n and quicksort as n^2, and outputs a corresponding optimisation:

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

Hence, the results from AutoBench concur with the previous tests, indicating that merge sort performs significantly better than quicksort in the worst case. We note that all test files are available on our system's GitHub page (Handley 2019).

Different list configurations

Overall, it is unfair to draw general conclusions about the relative performance of two implementations based exclusively on their worst-case behaviours. As such, we may incorporate 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 of this kind can be added easily, by simply defining new generators that produce the necessary inputs. For example, a generator for random lists is as follows:

```
instance SizedArbitrary RandomIntList where
  sizedArbitrary n = RandomIntList <$> vectorOf n arbitrary
```

One line test programs can then accept values of this type and forward the underlying list to the quicksort and merge sort functions already present in the file:

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

Graphs comparing the runtime measurements of both implementations when sorting random and nearly sorted lists are depicted in figures 3.13b and 3.13c, respectively. In all

tests, except that of random lists, we can see that the merge sort implementation had the better time performance, while for random lists, the time complexity of each implementation was estimated to be polylogarithmic. Thus, overall, the performance results from AutoBench suggest that the decision to change Haskell's standard sorting implementation from a quicksort algorithm to a merge sort algorithm was an *all-round* good one.

Smooth merge sort

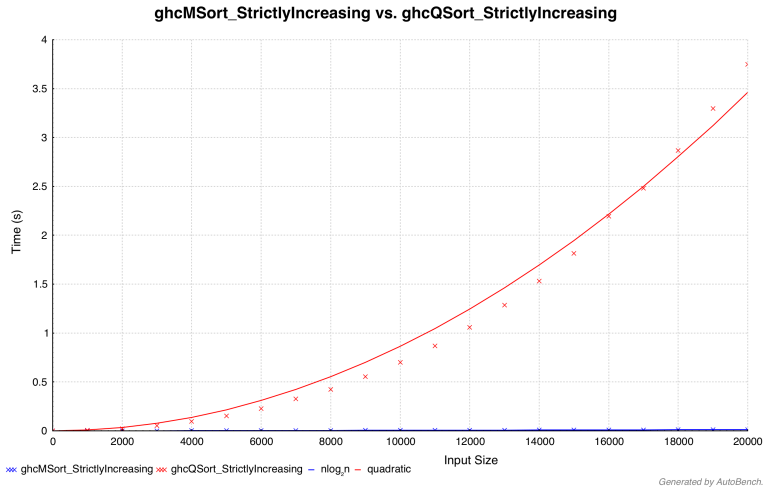
The current implementation of *sort* in Haskell's *Data.List* module is a more efficient implementation of merge sort than the one that replaced quicksort in 2002. By exploiting order in the argument list, this *smooth* algorithm (O'Keefe 1982) captures increasing and strictly decreasing *sequences* of elements as base cases for its recursive merge step:

$$
\begin{array}{ll}
> \mathit{sequences}\ [0,1,2,3] & > \mathit{sequences}\ [0,1,2,3,2,1,0] \\
[\ [0,1,2,3]\,,\ []\ ] & [\ [0,1,2,3]\,,\ [0,1,2]\,,\ []\ ]
\end{array}
$$

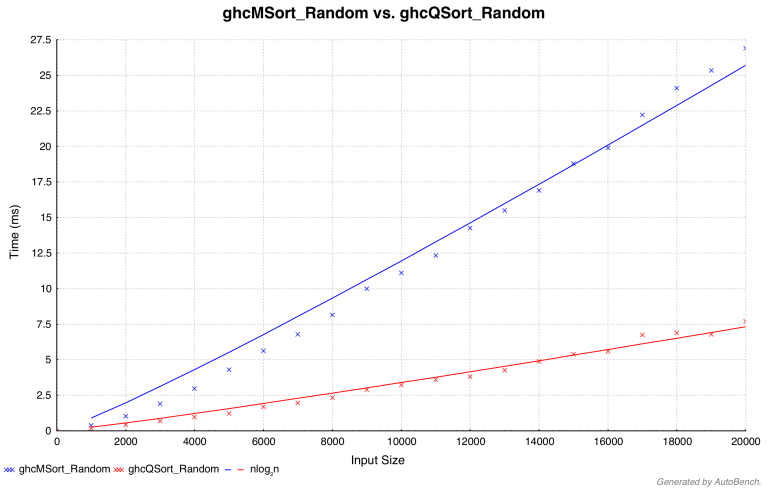
In particular, by taking sequences such as $[0,1,2,3]$ as base cases rather than singleton lists such as $[0]$, $[1]$, $[2]$, and $[3]$, the smooth merge sort algorithm runs in linear time when the initial list is already sorted or in strictly decreasing order. To determine whether this holds in practice, we can analyse the time performance of *sort* on sorted and strictly decreasing lists. At this point, the test file includes all external definitions required to test *sort*, so all that is left to do is import *Data.List* and define a one line test program:

$$
\begin{array}{l}
\mathit{sort}_{\mathit{Sorted}}\ ::\ \mathit{SortedIntList}\ \rightarrow\ [\mathit{Int}] \\
\mathit{sort}_{\mathit{Sorted}}\ (\mathit{SortedIntList}\ \mathit{xs})\ =\ \mathit{sort}\ \mathit{xs}
\end{array}
$$

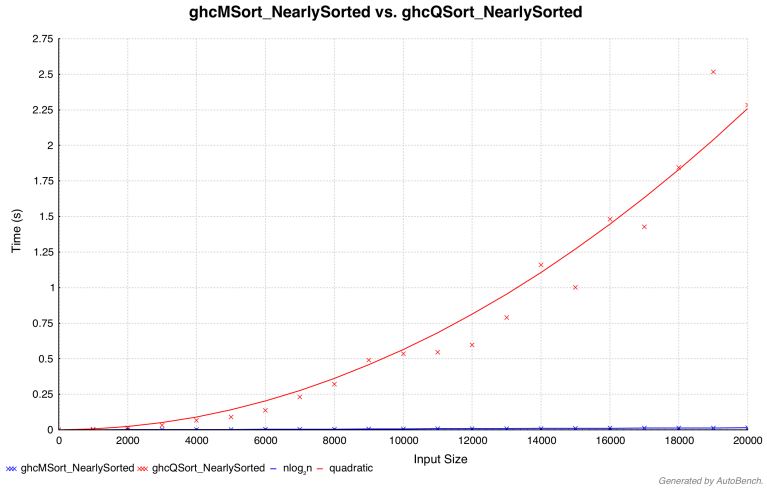
Results in figure 3.13d estimate the time complexity of *sort* when executed on sorted lists to be linear. Similar linear time estimates are also given when testing its performance on strictly decreasing and nearly sorted input lists.



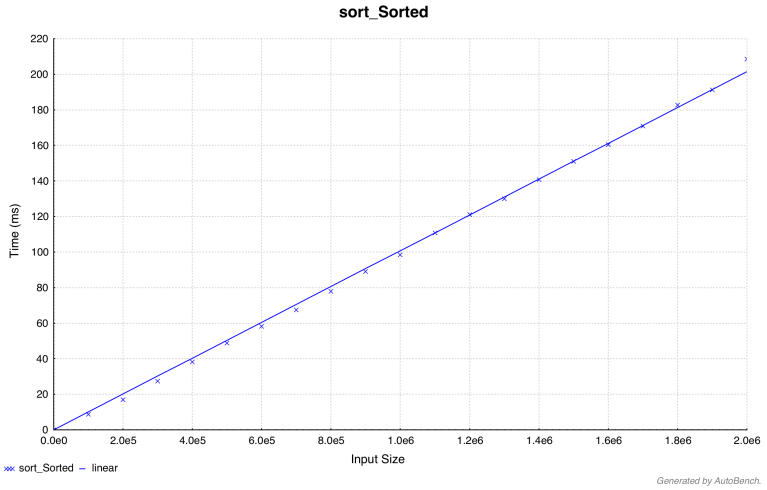
(a) Strictly increasing lists



(b) Random lists



(c) Nearly sorted lists



(d) Sorted lists

Figure 3.13: AutoBench time performance results: merge sort and quicksort

Sorting random lists

While exploring different tests for this case study, we came across an interesting result: merge sort appears to have worse time performance than quicksort when sorting random lists of integers (for example, see figure 3.13b). Given this, we were curious to see how the *sort* function compares to different implementations of quicksort when sorting random lists. Here, we focus on quicksort's naive implementation (Hutton 2016):

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

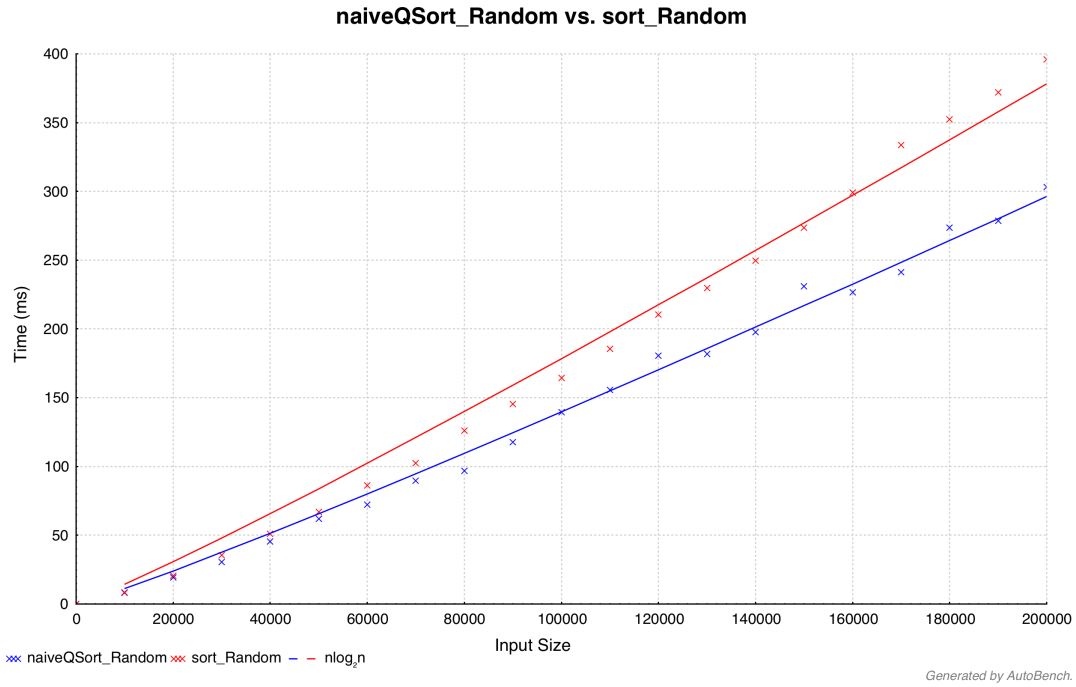
Two graphs of runtime measurements are displayed in figures 3.14a and 3.14b, which show that, under different degrees of optimisation, the library function performs notably *worse* than the naive function. Given that, in real-life settings, lists to be sorted are typically nearly sorted (Estivill-Castro and Wood 1992), this result may only have minor practical significance. Nevertheless, this outcome surprised us—especially given the source of each implementation—and hence we believe it underlines the importance of efficiency testing.

3.4.3 Case study 3: The Sieve of Eratosthenes

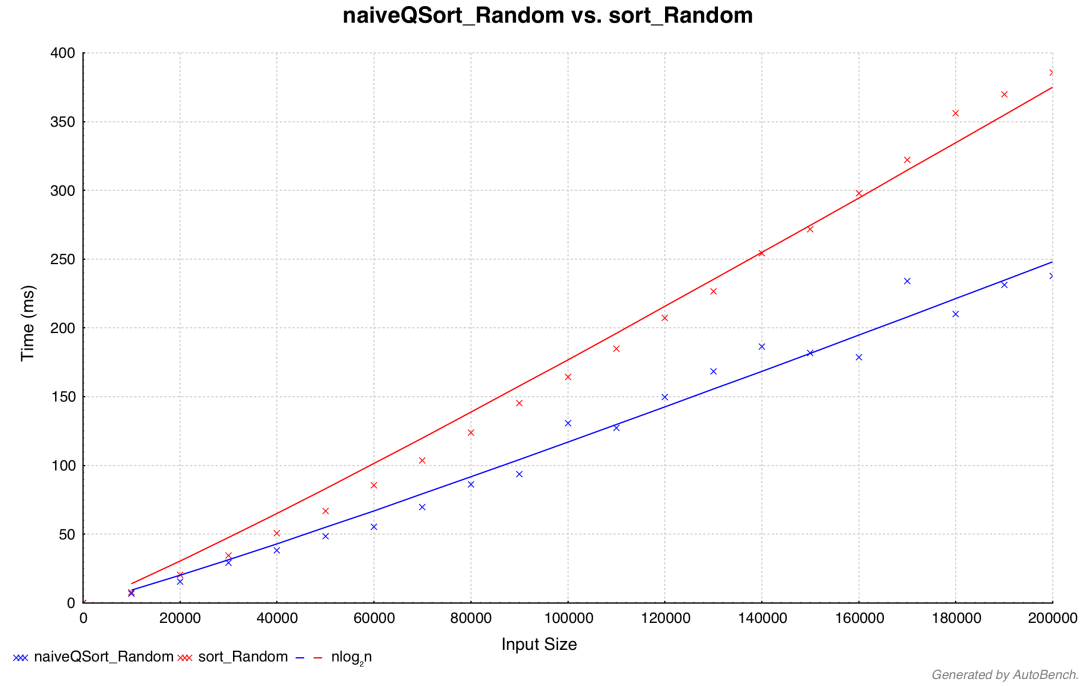
The Sieve of Eratosthenes is a classic example of the power of lazy functional programming, and is often defined by the following simple recursive program:

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

However, while this definition does produce the infinite list of primes, O'Neill (2009) demonstrated that it is *not* the Sieve of Eratosthenes. In particular, it uses a technique known as *trial division* to determine whether each candidate is prime, whereas Eratosthenes' original



(a) Random lists with no optimisation



(b) Random lists with -O3 optimisation

Figure 3.14: AutoBench time performance results: *Data.List.sort* and naive quicksort

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, appearing in the epilogue of (O'Neill 2009), implements the true sieve:

```
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
```

This definition, though far from being a 'one-liner' as in the case of the so-called *unfaithful* sieve, is elegant in its own right: the *composites* are defined to be the union of an infinite list of infinite lists. Moreover, and perhaps more importantly, this approach crosses off multiples of already found primes in the way Eratosthenes had intended, and is optimised to begin crossing off at p^2 for every prime p.

Given that the 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 case study we are, therefore, interested in comparing their time performance. O'Neill's (2009) article gives a detailed theoretical treatment of both implementations. To find all primes less than n the unfaithful sieve, *primes*, is shown to have $\Theta(n^2/\log^2 n)$ time complexity,

and the true list-based implementation, *truePrimes*, $\Theta(n\sqrt{n}\ \log\log n/\log^2 n)$. Thus, from a theoretical point of view, *primes* is asymptotically worse than *truePrimes*.

In addition to the theory, O'Neill's article includes a number of performance tests where various implementations of the Sieve are compared according to the number of reductions performed while being executed by the Hugs Haskell interpreter (Jones 2003). In contrast, we wish to use the AutoBench system to compare time performance of the programs when compiled using GHC. To achieve this, we start by defining the following two test programs to extract the first n prime numbers from each list

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

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

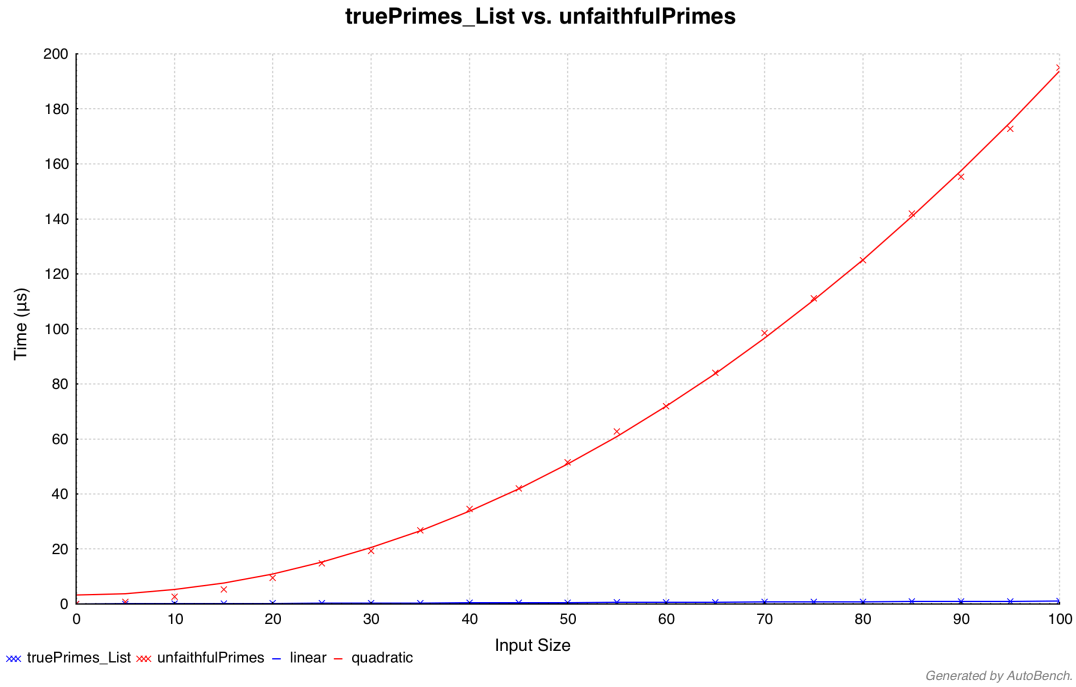
and then execute AutoBench on the corresponding test file.

Test results in figure 3.15a 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 $\Theta(n^2/\log^2 n)$ and $\Theta(n\sqrt{n}\log\log n/\log^2 n)$, 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 3.15b), the system approximates its time complexity as $n\ \log_2\ n$, which is closer to the theory.

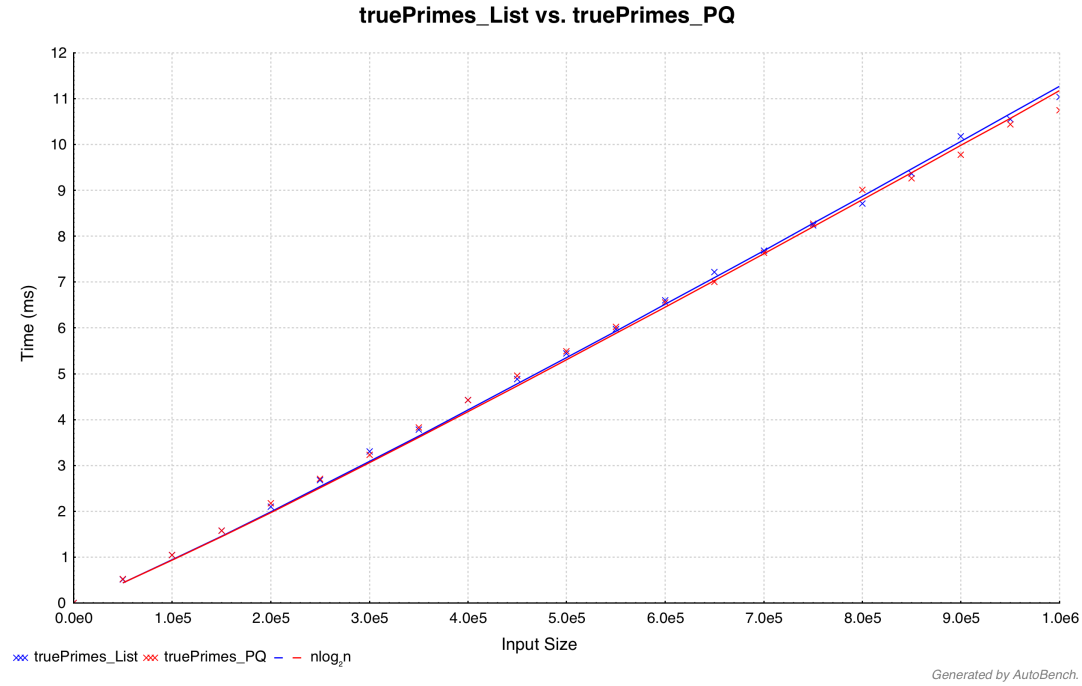
Nonetheless, the results in figure 3.15a do clearly indicate that *truePrimes* has the better time performance. In turn, this suggests that the time complexity of *primes* is indeed asymptotically worse than *truePrimes*:

$$unfaithfulPrimes \quad \trianglerighteq \quad truePrimes_{List} \quad (1.00)$$

Thus, overall, our test results agree with the theory and also with the corresponding performance measurements presented in O'Neill's article.



(a) Unfaithful and true list-based sieves



(b) True list- and priority-queue-based sieves

Figure 3.15: AutoBench time performance results: sieve implementations

An alternative data structure

A priority queue implementation of the Sieve of Eratosthenes is also presented in (O'Neill 2009), which is shown to have a better theoretical time complexity than Richard Bird's list-based solution. As a final test, we are interested to see whether using a more complex data structure in this instance is worthwhile. To this end, we can compare the list and priority queue implementations by using them to generate one million prime numbers.

The graph of runtime measurements for this test is given in figure 3.15b 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. To further validate our results, we tested both implementations up to the 10 millionth prime: the list implementation performed marginally better.

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

Operational errors

It is true that the Unfaithful Sieve of Eratosthenes, *primes*, produces correct results. Yet it does not correctly implement Eratosthenes' algorithm, despite this claim being made on a number of occasions, for example, in (Meertens 2004). As such, technically speaking, *primes* contains a notable implementation error. Property-based testing tools such as QuickCheck cannot detect such errors, however, because they are operational in nature, not denotational. On the other hand, AutoBench estimated the unfaithful sieve's time complexity as n^2, which is significantly different from the true sieve's known complexity $\Theta(n \log \log n)$. This difference arguably highlights the implementation error, and thus demonstrates how AutoBench could be used to uncover operational errors.

3.5 Discussion

Given the nature of the AutoBench system, there are many avenues for further work that relate to each of the system's three main components individually: data generation, benchmarking, and statistical analysis. Similarly, related work is best organised in this manner. In this section, we thus discusses related and further work accordingly.

3.5.1 Data generation

The main difficulty in generating random data using type-driven generators provided by property-based testing tools is ensuring the data has a suitable distribution. This is especially true if it must also enforce one or more invariants. The reason for this is that such generators conflate a number of concerns that ideally should be kept separate: (a) the data's desired structure; (b) the properties it should obey; and (c) its intended distribution.

A number of different solutions to this problem have been proposed. Pałka et al. (2011) show that, with much care and attention, it is possible to manually define an adequate generator for well-typed lambda terms used to test a compiler. This is a particularly difficult problem because type correctness is a global property that must be achieved by a sequence of local choices (as generators are defined on a per-type basis). Claessen, Duregård, and Pałka (2015) introduce a method for automatically deriving generators with uniform distributions from the structure of test data and one or more predicates it must satisfy.

Luck (Lampropoulos et al. 2017) is a domain-specific language for manually writing QuickCheck properties and generators simultaneously, so as to finely control the distribution of test data. This technique employs needed narrowing (Antoy, Echahed, and Hanus 2000) and constraint solving to ensure generated data satisfies one or more given properties. Fetscher et al. (2015) present a generic method for generating random, well-formed expressions from typing judgements. This is used in place of the simple 'generate and test' approach that is largely ineffective for problem domains incorporating even basic types.

Many of the articles cited above address different notions of size for generated test data. For example, Claessen, Duregård, and Pałka (2015) provide a method for generating random values of datatypes that contain specific numbers of constructors, which corresponds to the

notion of size we adopted for our tree-like data structures in section 3.3.2. It achieves this by uniformly sampling values of a particular type and size from a so-called *space*. Spaces allow sampled values to be generated lazily so that all possible values do not need to be constructed upfront. However, in order to generate a single value, the space must be searched in a depth-first manner. Our approach to generating sized data differs in that we don't sample values but instead solutions to Diophantine equations, which are then used to define values that are correctly sized by construction. Thus, our generation method is often more efficient. However, values can be removed from spaces if they fail to satisfy a given predicate, which allows for efficient generation of constrained test data. Our work on generic data generation has yet to address this, but we look to the Haskell implementation in (Claessen, Duregård, and Pałka 2015) for guidance.

Various Haskell libraries have been developed to automatically derive generators for user-defined datatypes, including *Derive* (Mitchell 2007), *Feat* (Duregård, Jansson, and Wang 2013), and *MegaDeth* (Grieco et al. 2017). As per our general approach, these libraries focus on providing random test data for scenarios whereby developers do not necessarily know the properties that the data should satisfy. Consequently, these libraries only address points (a) and (c) above. If developers also wish to address point (b) from above, then it is preferable to manually write specialised generators using, for example, Luck.

The most basic library is Derive, which is implemented using Template Haskell. Similarly to our work presented in section 3.3.2, the library provides generators for regular recursive datatypes. However, such generators permit recursive data constructors to be selected at every recursive generation step, which can lead to infinite loops.

MegaDeth avoids non-termination by placing a maximum bound on the number of times generation functions can recurse. As per our approach, this works by selecting only terminal data constructors when the 'generation size' becomes zero. In contrast, MegaDeth reduces its generation size by a fixed factor at each recursive step, for example, $\frac{n}{2}$, and so can only generate a subset of all possible values for a given datatype. This leads to a non-uniform distribution of test data, which appears to be excessively biased in practice: see figure 3 in (Mista, Russo, and Hughes 2018). Despite the fact that AutoBench's generators must incorporate a notion of size that is monotonically increasing, we believe our underlying

generation method based on solving Diophantine equations serves as a better starting point for generating (uniform) distributions of 'maximally-sized' data.

Feat determines the distribution of generated values similarly to the method described in (Claessen, Duregård, and Pałka 2015). That is, by exhaustively enumerating all possible values of a given datatype up to a particular size, and then uniformly picking between those values. In fact, the former approach (indexing into spaces) was inspired by Feat's implementation. Nonetheless, the particular generation method discussed in (Duregård, Jansson, and Wang 2013) also leads to a non-uniform distribution of test data that appears extremely biased in practice: see figure 3 in (Mista, Russo, and Hughes 2018).

Fundamentally, the Derive, Feat, and MegaDeth libraries support the generation of random test data of size *upto* a given limit (which is infinite in the case of Derive). Hence, even if they we able to generate uniformly distributed test data, their approaches would not be compatible with the needs of AutoBench. As such, none of these libraries can be used in place of our generic approach presented in section 3.3.2.

The same is true for a recent Haskell tool chain called *Dragen* (Mista, Russo, and Hughes 2018), which is able to accurately predict the behaviour of QuickCheck generators using a stochastic model based on *branching processes* (Watson and Galton 1875). Doing so allows the system to automatically derive generators with various distributions. Despite the fact that Dragen produces maximally-sized test data, it does so for non-regular datatypes. We are interested in extending our approach from section 3.3.2 in this respect, as well as to support different distributions of test data, and so look to Dragen's design for guidance.

SmallCheck (Runciman, Naylor, and Lindblad 2008) implements another popular approach to property-based testing. In comparison to QuickCheck's random testing methodology, SmallCheck tests properties for *all* possible inputs up to a certain 'depth', which is a size parameter constraining an exhaustive depth-limited search. It is indeed possible for AutoBench to use SmallCheck to generate test data (we have conducted preliminary experiments). 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 is seldom practical.

In the wider literature, *EasyCheck* (Christiansen and Fischer 2008) 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 narrowing to constrain generated data to values satisfying a given predicate.

Real-world test data

As an alternative option to generating random test data using QuickCheck, AutoBench allows users to manually specify 'real-world' test data. This is particularly useful for three reasons: (a) when testing a program whose domain is intricate, and thus a suitable *SizedArbitrary* instance is not easily definable; (b) when testing a program that is particularly sensitive to the *values* of its inputs, and so testing its performance using randomly generated inputs is not appropriate; (c) to allow performance testing using pre-recorded input traces, for example, as part of game development.

Recall from section 3.3.3 that AutoBench defines benchmarks using test environments

$$env :: NFData\ env \Rightarrow IO\ env \rightarrow (env \rightarrow Benchmark) \rightarrow Benchmark$$

where an *IO env* is taken to be an IO action that produces test data. As such, when testing a program of type $a \rightarrow b$, users must provide a list of suitable inputs of type *IO a*. In addition, the system cannot deduce size information as yet, and so the size of each input must be given alongside it. As a concrete example, the following test data can be used to analyse the performance of *slowRev* and *fastRev* from section 3.2.1:

```
tData :: [(Int, IO [Int])]
tData = [ (0,  pure [])
        , (5,  pure [1..5])
        , (10, pure [1..10])
        , (15, pure [1..15])
        ... ]
```

As in section 3.2.1, the size of each input list is given by its number of elements.

Manual test data must be included in test files along with the programs to be tested. In turn, user-defined test suites can be configured to utilise real-world inputs by referencing such data. Further information is given in the system's user manual (Handley 2019).

With respect to real-world test data, there are two avenues for further work that we are particularly interested in. Firstly, it would be useful for the system to automatically deduce size information. Secondly, we wish to support a module for *fuzzing* inputs, allowing users to determine the *value-sensitivity* of a test program. In both cases, there is a wealth of relevant literature; for brevity, we give only a brief overview.

Sized types (Hughes, Pareto, and Sabry 1996; Vasconcelos 2008), which index types with natural numbers to denote the sizes of their values, appear a promising line of future research for our first goal. In particular, their indices could replace user-specified size information.

As we have seen in section 3.3.2, generic representations (Hinze and Jeuring 2003) of datatypes provide a simple way of prescribing size information, via the notion of 'number of constructors'. Indeed, it is straightforward to define a function of type $Generic\ a \Rightarrow a \to Int$ to calculate size in this manner using the Generics-Sop library (Vries and Löh 2014).

The idea of fuzzing (Godefroid, Levin, and Molnar 2012), which originated as part of security testing, is to randomly mutate well-formed inputs and then test programs using those modified inputs. The degree to which a program is value-sensitive is determined by the relative change in its 'cost' (in our case, time performance) with respect to changes in the values of its inputs. AutoBench could thus use fuzzing and repeated benchmarking to automatically determine a test program's value-sensitivity. Among other things, this can be used to determine whether it is 'safe' to analyse the performance of a program executed on random inputs (as per point (b) at the start of this discussion).

There are many articles on fuzzing, for example, (Sutton, Greene, and Amini 2007; Godefroid, Kiezun, and Levin 2008), as well as libraries for applying fuzzing in Haskell, for example, (Grieco, Ceresa, and Buiras 2016; Grieco et al. 2017).

3.5.2 Benchmarking

The system most closely related to ours is *Auburn* (Moss 2000), which is designed to benchmark purely functional data structures and algorithms (Okasaki 1999) in Haskell. 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 execution time.

Auburn analyses sequences of operations performed on datatypes, allowing it to give insights into the amortised cost of individual operations (as per Okasaki's work). In contrast, our system's performance analysis is more coarse-grained, comparing the runtimes of complete Haskell programs. Auburn does not compare or extrapolate the time measurements it takes, instead they are simply output to the user in a tabular format.

Other performance indicators

As discussed in background section 2.2, typical performance measurements used in benchmarking tests include execution time, memory allocation, throughput, lock contention, and IO operations. It would, therefore, be useful for AutoBench to compare Haskell programs according to additional performance measurements, beyond just execution time. In particular, we are interested in memory usage as this is often difficult to diagnose in Haskell due to its call-by-need semantics. Moreover, we would like to provide runtime and allocation comparisons for both pure and impure Haskell code (that is, to include IO operations).

In the former case, there are two options for extending AutoBench to account for a program's memory allocation. Firstly, the Criterion library—which the system already depends on—can be configured to measure space usage. Hence, the 'front end' of AutoBench need not be amended, and the 'back end' need only be extended to analyse different performance measurements. The second option is to depend on another library specifically designed to measure space usage in Haskell. Among those available, *Weigh* (Done 2016) is the most popular. Seemingly inspired by Criterion, Weigh provides functions to construct its own benchmarks of type *Weigh* () in much the same way:

$$\begin{aligned} \mathit{func} &:: \mathit{NFData}\ b \Rightarrow \mathit{String} \rightarrow (a \rightarrow b) \rightarrow a \rightarrow \mathit{Weigh}\ () \\ \mathit{wgroup} &:: \mathit{String} \rightarrow \mathit{Weigh}\ () \rightarrow \mathit{Weigh}\ () \end{aligned}$$

This is particularly convenient for AutoBench's existing implementation, as it need only be extended laterally to interface with this new library. Weigh can measure total bytes allocated, total number of garbage collections, total amount of live data on the heap, and maximum residency memory in use. In comparison, Criterion can only measure total bytes allocated and total number of garbage collections.

Both Criterion and Weigh provide means to benchmark impure Haskell code, and so can also be used to address the second extension to AutoBench discussed above. Recall from section 2.2.1 that Criterion provides the following functions, for example:

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

In turn, Weigh has a number of comparable functions:

$$\begin{aligned} \mathit{value} &:: \mathit{NFData}\ a \Rightarrow \mathit{String} \rightarrow \mathit{IO}\ a \rightarrow \mathit{Weigh}\ () \\ \mathit{io} &:: \mathit{NFData}\ b \Rightarrow \mathit{String} \rightarrow (a \rightarrow \mathit{IO}\ b) \rightarrow a \rightarrow \mathit{Weigh}\ () \end{aligned}$$

In order to generate polyvariadic benchmarks (see section 3.3.3) using these functions, AutoBench must provide additional generators. For example

$$\begin{aligned} \mathit{genNfIO}_{+} :: (\ & \mathit{as} \sim \mathit{Arguments}\ (a \rightarrow \mathit{rest}),\ \mathit{IO}\ r \sim \mathit{Result}\ (a \rightarrow \mathit{rest}) \\ & ,\ \mathit{All}\ \mathit{SizedArbitrary}\ \mathit{as},\ \mathit{All}\ \mathit{NFData}\ \mathit{as},\ \mathit{Curry}\ \mathit{as},\ \mathit{NFData}\ r\ ) \\ & \Rightarrow [(a \rightarrow \mathit{rest},\ \mathit{String})] \rightarrow \mathit{Int} \rightarrow \mathit{Benchmark} \end{aligned}$$

can be used to generate benchmarks using *nfAppIO*. The only difference between the type signature of this generator and that of $genNf_{+}$ from section 3.3.3 is the constraint $IO\ r \sim Result\ (a \rightarrow rest)$, which ensures the result type of the user-specified test program's of type $a \rightarrow rest$ is an IO action. As such, the implementation of this function is highly comparable to $genNf_{+}$: differing only in the use of *nfAppIO* in place of *nf*. Extending AutoBench to compare impure Haskell code, therefore, requires front end modifications only. In particular, the system must determine whether the tests programs are pure or impure. This can be achieved by simply analysing their type, $a \rightarrow rest$, which is straightforward.

GHC Cost Centres (GHC Team 2019) are used to measure time and space usage at particular locations inside functions. When code is compiled with profiling enabled, information regarding resource usage at each location is generated. In contrast with our system, which benchmarks programs for comparative purposes, profiling is more fine-grained, and aims to reveal specific locations of maximum resource usage. GHC cost centres could thus be used in conjunction with our system as part of a subsequent optimisation phase.

Comparing benchmarks

Earlier versions of Criterion included a function to compare benchmarks, called *bcompare*. However, it turned out that this complicated many of the system's internals. As a result, it was removed and has yet to be replaced (O'Sullivan 2014a).

Progression (Brown 2010) is a popular Haskell library that builds upon 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 unique 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 by users of the system. Users are also responsible for compiling and executing their test files. Our system differs in this respect, as test inputs are generated automatically and the benchmarking is fully automated. Progression uses *Gnuplot* to produce its graphs, which it invokes via a system call. We preferred to use the *Chart* package (Docker 2006), similarly to Criterion, in order to keep our implementation entirely Haskell-based.

Taking inspiration from Progression, AutoBench's results could be labelled and saved to file. This would then allow users to compare data from different tests without having to re-run the benchmarks, which, as we have previously mentioned, can be time consuming. It is straightforward to encode and decode structured data efficiently in Haskell. For example, the *Aeson* library (O'Sullivan 2014b) is optimised for JSON report files.

This idea could be taken one step further, by extending AutoBench to compare programs from different sources. One seemingly useful source of comparison is between different commits to a version control system, for example, GitHub. The continuous integration service known as *Travis* (2019) enables new repository commits to be subjected to a number of QuickCheck-style correctness tests. Such tests ensure updated implementations do not introduce bugs, by requiring that all tests are passed before allowing commits to be merged. In essence, this approach compares the correctness of one program version with that of another. To achieve a similar goal for efficiency, AutoBench requires a method for specifying 'testable efficiency properties', for which we can look to QuickCheck for inspiration.

3.5.3 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 the AutoBench system, which benchmarks a program's time efficiency on different sized inputs, traditional profiling tools do not typically characterise performance as a function of input size.

Although some articles, for example, (Coppa, Demetrescu, and Finocchi 2012; Coppa et al. 2014) aim to develop profiling tools that specify runtime as a function of input size, they focus primarily on automatically calculating the sizes of inputs, rather than describing in details their methods for model fitting and selection. Nonetheless, as we discussed in the previous subsection, input size inference could be useful for our system.

Model fitting and selection is discussed in the work of Goldsmith, Aiken, and Wilkerson (2007), 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 sized inputs and then performing regression analysis on the results to approximate asymptotic time complexity. This approach is similar to that of our work, however, 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 unbiased fitting statistics to compare regression models (see section 3.3.4).

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 (Coppa, Demetrescu, and Finocchi 2012). However, some researchers have proposed heuristics for the 'empirical curve bounding' problem, showing their effectiveness on a number of occasions (McGeoch et al. 2002). Some commercial software packages do advertise 'curve fitting'

features (Systat Software 2019; OriginLab 2019; Hyams 2019), but as they don't publicise their underlying technologies, we are not able to compare them with our system.

LASSO regression analysis

The method we have developed for approximating time complexity uses regularisation to prevent higher complexity models from overfitting training data. In particular, we make use of ridge regression, which constrains the square magnitude of the coefficients of model parameters. Initially, however, we had intended to use the LASSO (least absolute shrinkage and selection operator) regression method (Hans 2009). This differs from ridge regression in that it constrains the *absolute* magnitude of coefficients. The advantage of LASSO over ridge regression is that it can shrink the coefficients of non-influential model parameters to *zero*. Thus, as well as preventing overfitting, LASSO performs *feature selection*.

To demonstrate feature selection in practice, the following table shows the models chosen by the ridge and LASSO regression methods when applied to the runtime measurements of *slowRev* taken from the first example of section 3.2:

Ridge regression	$y = 5.28e{-}9x^2 + 2.28e{-}11x + 2.91e{-}5$
LASSO regression	$y = 5.52e{-}9x^2 + 3.67e{-}5$

In both cases, a quadratic model is chosen. However, the LASSO method has eliminated the linear parameter ($2.28e{-}11x$) present in the equation output by ridge regression. In doing so, the LASSO method gives a clearer indication that *slowRev* is quadratic.

From this example (and many more alike), it becomes clear that the LASSO method is more fitting for our purposes than ridge regression. Unfortunately, however, solving instances of the LASSO algorithm in Haskell is too computationally expensive. In short, this is because LASSO has no closed-form solution. (In comparison, ridge regression's algorithm does have a closed-form solution.) The technical details are somewhat involved but essentially the LASSO method requires solving quadratic optimisation problems. To the best of our knowledge, there are currently no quadratic programming (QP) solvers implemented in Haskell. There are, however, Haskell bindings to QP solvers developed in

other languages (such as Python and R). Nonetheless, a key initial design choice was to keep AutoBench entirely Haskell-based. Hence, the system currently uses ridge regression.

A Haskell library that we hoped would be suitable for AutoBench is *HVX* (Copeland 2017), designed to solve a more general class of convex programming problems, of which the LASSO algorithm is a particular instance. HVX optimises convex models using iterative methods. In practice, this makes calculating model parameters an iterative process. In addition to using an iterative cross-validation technique, this approach makes the analysis phase of AutoBench's execution too time consuming, and hence impractical.

In the future, we hope that an efficient quadratic programming library is developed natively in Haskell. This would allow us to solve instances of the LASSO algorithm more directly, with the aim of improving AutoBench's (time) complexity approximations. Another option is to relax our initial design decision and allow the system to invoke QP solvers developed in other languages. In this case, we have performed some preliminary experiments with *HaskellR* (Boespflug et al. 2014), which enables Haskell and R code to be used in the same source file using quasiquotation. Overall, our experiments—available online (Handley 2019)—indicate that it is reasonably straightforward to invoke R's standard implementation of the LASSO algorithm from impure Haskell code. Furthermore, it appears to be extremely efficient. Further investigations are part of our future work.

Multiple linear regression

Despite the fact that AutoBench can generate benchmarks for functions of any non-zero arity via polyvariadic benchmarking (see section 3.3.3), its complexity analysis currently only supports two-dimensional data. In practice, this means that a time complexity estimate for a program with multiple inputs is based on the size of its *first* argument only.

However, the system's existing method of regression analysis, ridge regression, can be used to determine the relationships between two or more independent variables x_i and a dependent variable $\hat{y}$. More specifically, ridge regression can calculate the unknown parameters a_i in equations of the following form for any p:

$$\hat{y} \quad = \quad a_0 \; + \; a_1 x_1 \; + \; a_2 x_2 \; + \; \cdots \; + \; a_p x_p$$

An example whereby $p = 2$ (that is, for two independent variables) is the time complexity estimate given for *slowRev* in section 3.2.1, which is calculated to be $y = 5.28e{-}9x^2 + 2.28e{-}11x + 2.91e{-}5$. In this instance, the independent variables x_1 and x_2 are instantiated to x^2 and x, respectively, for the same size parameter x. However, this need not be the case. Indeed, x_1, x_2, ..., x_p can freely refer to any available size parameters.

With this in mind, we can imagine that extending AutoBench's complexity approximations to account for multiple input sizes is a viable option. However, it is not clear that extending regression models to include *all* such size parameters is desirable. A simple counterexample to this approach is Haskell's append operator:

$$
\begin{aligned}
&(+\!\!+) \;::\; [a] \;\to\; [a] \;\to\; [a] \\
&[\,] \qquad +\!\!+\; ys \;=\; ys \\
&(x \;:\; xs) \;+\!\!+\; ys \;=\; x \;:\; xs \;+\!\!+\; ys
\end{aligned}
$$

It is often said that $(+\!\!+)$ has linear runtime in the length of its first argument, and, according to AutoBench testing, it does. On the other hand, it should be clear that no matter how large its second argument is, $(+\!\!+)$ will always terminate when its first argument has been traversed. As such, it is *constant* in the length of its second argument. Append's runtime is, therefore, best predicted by the following regression model

$$\hat{y} \;=\; a_0 \;+\; a_1 x_1 \;+\; 0.0 x_2$$

in which the coefficient of the second independent variable, x_2, relating to the size of append's second argument, is zero. This model is equivalent to

$$\hat{y} \;=\; a_0 \;+\; a_1 x_1$$

where x_2 has been removed altogether, meaning that append's runtime is best predicted by the size of its first argument only. This observation can be confirmed by simple strictness analysis, which shows that append is lazy in its second argument (Wadler 1988).

Nonetheless, not all binary functions are lazy in their second arguments. Furthermore, it is easy to imagine that the runtimes of programs that are strict in all of their arguments are affected by all such input sizes. To account for this for an arbitrary binary function, three separate regression models must be considered: (a) running time depending on the

first input size only; (b) depending on the second input size only; (c) depending on both input sizes. It is easy to see that the number of combinations increases exponentially in the number of input sizes and hence this approach is not practical.

A method for extending AutoBench's *current* complexity analysis (based on ridge regression) to account for multiple input sizes is thus not immediately obvious. Nonetheless, an alternative method based on LASSO regression may be more straightforward. In particular, recall that LASSO performs feature selection on independent variables. A single regression model for each type of function (linear, quadratic, cubic, and so on) could thus be used, which incorporates *all* input sizes, in the hope that the algorithm shrinks the coefficients of all non-influential model parameters (that is, input sizes) to zero. We have yet to perform any rigorous tests, but note that this approach does work in the simple case of append. An in-depth investigation is part of our future work.

3.6 Conclusion

In this chapter, we have taken ideas from property-based testing, microbenchmarking, and statistical analysis, and used them to develop a straightforward approach for comparing the time performance of Haskell programs. In doing so, we have combined two well-established systems, namely QuickCheck and Criterion, to give a lightweight, fully automated tool that can be used by ordinary programmers. In addition, we have devised a simple but effective algorithm for approximating time complexity, based on linear regression analysis.

Chapter 4

Improving Haskell

The University of Nottingham Improvement Engine

Lazy evaluation is a key feature of Haskell, but can make it difficult to reason about the efficiency of programs. Moran and Sands' improvement theory addresses this problem by providing a foundation for proofs of program improvement in a call-by-need setting, and has recently been the subject of renewed interest. However, proofs of improvement are intricate and require an inequational style of reasoning that is unfamiliar to many Haskell programmers. In this chapter, we present the design and implementation of an inequational reasoning assistant called Unie, which provides mechanical support for improvement proofs, and demonstrate its utility by verifying a range of results from the existing literature.

4.1 Introduction

In the previous chapter, we introduced the AutoBench system, which provides a simple means to automatically compare the time performance of Haskell programs based on benchmark testing. Although benchmarking libraries such as Criterion (O'Sullivan 2014a) enable programs to be analysed within a wide range of test environments, in general they cannot determine whether one program will outperform another in all program contexts.

In order to achieve such guarantees, we must instead reason more directly about the operational nature of expressions (and their subexpressions) defined in Haskell. However, such reasoning about efficiency is notoriously difficult and counterintuitive. The source

of the problem is Haskell's use of lazy evaluation, or more precisely, the language's call-by-need semantics, which allows computations to be performed with terms that are not fully normalised. In practice, this means that the operational efficiency of a term does not necessarily follow from the number of steps it takes to evaluate to normal form, in contrast to a call-by-value setting where reasoning about efficiency is much simpler.

Moran and Sands' improvement theory (1999) offers the following solution to this problem: rather than counting the number of steps required to normalise a term in isolation, we compare the number of steps required in all program contexts. This idea gives rise to a compositional approach to reasoning about efficiency in call-by-need languages such as Haskell, which is known as operational improvement or just *improvement*.

Improvement theory was originally developed in the 1990s, but has recently been the subject of renewed interest, with a number of general-purpose program optimisations being formally shown to be improvements (Hackett and Hutton 2014; Schmidt-Schauß and Sabel 2015; Hackett and Hutton 2018). In an effort to bridge the so-called correctness/efficiency 'reasoning gap' (Harper 2014), these articles show that it is indeed possible to formally reason about the performance aspects of optimisation techniques in a call-by-need setting.

While improvement theory provides a suitable basis for reasoning about efficiency in Haskell, the resulting proofs are often intricate, and therefore constructing them by hand is challenging. In particular, comparing the cost of evaluating terms in all program contexts requires a somewhat elaborate reasoning process, whose resulting inequational style of calculation is inherently more demanding than the equational style that is familiar to most Haskell programmers (see background section 2.3 for an overview).

To support interactive *equational* reasoning about Haskell programs, the Hermit toolkit was developed by Farmer et al. Its utility has been demonstrated in a series of case studies (Farmer et al. 2012; Sculthorpe, Farmer, and Gill 2013; Adams, Farmer, and Magalhães 2014; Farmer, Siederdissen, and Gill 2014; Adams, Farmer, and Magalhães 2015; Farmer, Sculthorpe, and Gill 2015; Farmer 2015). Despite the fact that inequational reasoning is more involved than its equational counterpart, both approaches share the same calculational style. In addition, the Hermit system and improvement theory are both based on the same underlying setting: the Core language of the Glasgow Haskell Compiler. As such, a

```
 = λxs.λys.✓(case xs of
     []       → ys
     (z : zs) → ((f zs) ++ [z]) ++ ys)

[18]> append-assoc-lr-i
 ≳ λxs.λys.✓(case xs of
     []       → ys
     (z : zs) → (f zs) ++ ([z] ++ ys))

[19]> right
 = λxs.λys.✓(case xs of
     []       → ys
     (z : zs) → (f zs) ++ ([z] ++ ys))

[20]> eval-i
 ≳ λxs.λys.✓(case xs of
     []       → ys
     (z : zs) → (f zs) ++ (z : ys))
```

Figure 4.1: An extract from an improvement proof in the Unie system

system developed in a similar manner to Hermit could prove to be effective in supporting inequational reasoning for proofs of program improvement, just as Hermit has proved to be effective in supporting equational reasoning for proofs of program correctness.

To the best of our knowledge, no such inequational reasoning system existed when our contributions were first published in (Handley and Hutton 2018b). To fill this gap, we developed the University of Nottingham Improvement Engine (Unie): an interactive, mechanised assistant for call-by-need improvement. Our improvement assistant comprises approximately 13,000 lines of new Haskell code, and is freely available on GitHub (Handley 2018). The work in this chapter introduces the system and its underlying theory, and demonstrates its applicability on a number of improvement results taken from the literature.

By way of example, an extract from an improvement proof in our system concerning the familiar *slowRev* function is given in figure 4.1. In each step, the term highlighted in orange is being transformed. In the first step, the append operator ($+\!\!+$) is reassociated to the right, which is an improvement: denoted by $\gtrsim$. We then move to the right and evaluate ($+\!\!+$), which is also an improvement. The 'tick' operator ($\checkmark$) in the proof represents a unit time cost. We will revisit this example in more detail throughout the chapter.

Remark. We note that some of the transformations applied during improvement proofs

(such as that above) can impact the cost of garbage collection. However, garbage collection is not addressed in our work. Related work on space-safe improvement transformations can be found in (Gustavsson and Sands 1999; Hackett and Hutton 2019).

4.2 Improvement theory in practice

To provide some intuition for improvement theory and demonstrate how its technicalities can benefit from mechanical support, we begin with an example that underpins the proof extract in figure 4.1. Recall the following property, which formalises that Haskell's list-appending operator, $(+\!\!+)$, is associative for finite lists:

$$(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \quad = \quad xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) \tag{4.1}$$

As highlighted in the previous chapter, a common informal argument about this equation is that its left-hand side is less time-efficient than its right-hand side, because the former traverses the list xs twice whereas the latter only traverses xs once. In fact, the examples presented in section 3.2 indirectly demonstrate how this property can be exploited when optimising functions defined in terms of append (Wadler 1987). Optimisations of this kind often establish the correctness of (4.1), which can be verified by a simple inductive proof, but fail to make precise any efficiency claims about the equation. Can we do better?

Using improvement theory, we can formally show which side of equation (4.1) is more efficient by comparing the evaluation costs of each term in all program contexts. That is, we can show that one side is 'improved by' the other, written:

$$(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \quad \trianglerighteq \quad xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) \tag{4.2}$$

Remark. Readers may notice that the 'improved by' relation above uses the same notation, namely $\trianglerighteq$, as introduced in the previous chapter when discussing the AutoBench system's optimisation results (see section 3.3.4). This is intentional, as the latter work aims at approximating this relation by way of empirical analysis. Nonetheless, it is important to

note that this new relation is used to formally compare the efficiency of Haskell expressions. Hence, inequations such as (4.2) must be proved.

Before sketching how the above inequation can be proved, we first introduce some necessary background material on improvement theory. As the focus here is on illustrating the basic ideas of program improvement by means of an example, we simplify the theory where possible and will return to the precise details in the next section.

Program contexts and improvement

In the usual manner, program contexts are 'terms with holes', denoted by $[-]$, which can be substituted with other terms. Informally, a term M is *improved by* a term N, written $M \trianglerighteq N$, if, in all program contexts, the evaluation of N requires no more function calls than that of M. If the evaluations require the same number of function calls in all contexts, then they are said to be *cost equivalent*, which is written as $M \lhd\rhd N$.

Recording cost

While reasoning about improvement, it is necessary to keep track of evaluation cost explicitly within the syntax of the source language. This is achieved by means of a tick annotation ($\checkmark$) that represents a unit time cost, that is, one function call. Denotationally, ticks have no effect on terms. Operationally, however, a tick represents a function call. Hence, a term M evaluates with n function calls if and only if $\checkmark M$ evaluates with $n + 1$ function calls. Moreover, for any function definition $f\ x = M$, we have the cost equivalence

$$f\ x \quad \lhd\rhd \quad \checkmark M \tag{4.3}$$

because unfolding the definition eliminates the function call. Removing a tick improves a term, $\checkmark M \trianglerighteq M$, but the reverse $M \trianglerighteq \checkmark M$ is not valid.

Improvement induction

A difficulty with the definition of improvement is that it quantifies over all program contexts. Hence, proving (4.2) notionally requires considering all possible contexts. This often leads to cumbersome proofs: a well-known problem with contextual definitions. As an alternative proof strategy, we will look at improvement induction. Improvement induction allows us to only consider a single program context, which leads to much simpler proofs.

We use improvement induction for this purpose, presented here in a simplified form. For any context $\mathbb{C}$, the following is true:

$$\frac{M \;\trianglerighteq\!\!\!\sim\; \checkmark\mathbb{C}[M] \qquad \checkmark\mathbb{C}[N] \;\trianglelefteq\!\!\!\triangleright\!\!\!\sim\; N}{M \;\trianglerighteq\!\!\!\sim\; N}$$

Intuitively, this rule (Moran and Sands 1999) allows us to prove $M \trianglerighteq\!\!\!\sim N$ by finding a single context $\mathbb{C}$ for which we can 'unfold' M to $\checkmark\mathbb{C}[M]$ and 'fold' $\checkmark\mathbb{C}[N]$ to N. For example, applying improvement induction to inequation (4.2) reduces the problem to finding a single context $\mathbb{C}$ that satisfies the following two properties:

$$(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \qquad \trianglerighteq\!\!\!\sim \qquad \checkmark\mathbb{C}[(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs] \tag{4.4}$$

$$\checkmark\mathbb{C}[xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)] \qquad \trianglelefteq\!\!\!\triangleright\!\!\!\sim \qquad xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) \tag{4.5}$$

Proof of improvement property 4.2

For the purposes of this example, we can assume that the source language of improvement theory's operational model is simply Haskell, with one small caveat: arguments to functions must be variables. Improvement theory requires this assumption and it is easy to achieve by introducing **let** bindings. For example, the term $(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$ can be viewed as syntactic sugar for **let** $ws = xs \mathbin{+\!\!+} ys$ **in** $ws \mathbin{+\!\!+} zs$.

Using improvement induction, we can prove (4.2) by finding a context $\mathbb{C}$ for which properties (4.4) and (4.5) hold. We prove the first of these properties in figure 4.2; the second proceeds similarly. As we have not yet presented the laws of improvement theory,

which permit evaluation costs to be propagated within terms while maintaining or improving efficiency, the reader is encouraged to focus on the overall structure of the reasoning in figure 4.2 rather than the technicalities of each individual step.

Reflection

We chose the associativity of append for our first example because it is a simple property that is widely used to improve the performance of programs operating on lists, for example, in (Wadler 1987; Hackett and Hutton 2014). However, what should be evident from this calculation—and particularly from each step's justification—is that proofs of improvement are non-trivial, even for simple examples. Nonetheless, despite the complexities in dealing with ticks, contexts, and different improvement relations, a general recipe for improvement emerges from studying calculations such as that in figure 4.2:

(a) Unfold function definitions to expose the underlying computations and provide opportunities for optimisation, for example, unfolding (++);

(b) Simplify expressions by applying basic laws about the language primitives, for example, rearranging **let** bindings and **case** statements;

(c) Fold function definitions to compound the effect of previous steps across recursive calls, for example, inlining **let** bindings and folding (++).

In our experience, this recipe can be utilised on many occasions when reasoning about improvement. Thus, given that Unie provides mechanical support for handling ticks, contexts, and improvement relations, it allows users to focus on general recipes and thereby reason about improvement at a higher level of abstraction than afforded by the formalities of the underlying theory. In this manner, users can concentrate on the essential aspects of their reasoning and leave the technicalities of individual steps to the system.

4.3 The theory of improvement

In this section, we return to the formalities of Moran and Sands' call-by-need improvement theory. While explaining the theory, we describe how the Unie system supports, and in

$(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$

$=$ { syntactic sugar }

let $ws = xs \mathbin{+\!\!+} ys$ **in** $ws \mathbin{+\!\!+} zs$

$\lhd\!\rhd$ { unfold $(+\!\!+)$ }

let $ws =$ ✓**case** xs **of**
  $[\,] \to ys$
  $(u : us) \to u : (us \mathbin{+\!\!+} ys)$
in $ws \mathbin{+\!\!+} zs$

$\lhd\!\rhd$ { unfold $(+\!\!+)$ }

let $ws =$ ✓**case** xs **of**
  $[\,] \to ys$
  $(u : us) \to u : (us \mathbin{+\!\!+} ys)$
in ✓**case** ws **of**
  $[\,] \to zs$
  $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { move the tick inside $\mathbb{D}$'s hole, where
  $\mathbb{D} =$ **case** $[-]$ **of**
    $[\,] \to ys$
    $(u : us) \to u : (us \mathbin{+\!\!+} ys)$ }

let $ws =$ **case** ✓xs **of**
  $[\,] \to ys$
  $(u : us) \to u : (us \mathbin{+\!\!+} ys)$
in ✓**case** ws **of**
  $[\,] \to zs$
  $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { move $\mathbb{D}$ inside case, where
  $\mathbb{D} =$ **let** $ws = [-]$ **in** ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$ }

case ✓xs **of**
  $[\,] \to$ **let** $ws = ys$ **in** ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$
  $(u : us) \to$ **let** $ws = u : (us \mathbin{+\!\!+} ys)$
    in ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { move the tick outside $\mathbb{D}$'s hole, where
  $\mathbb{D} =$ **case** $[-]$ **of**
    $[\,] \to$ ...
    $(u : us) \to$ ... }

✓**case** xs **of**
  $[\,] \to$ **let** $ws = ys$ **in** ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$
  $(u : us) \to$ **let** $ws = u : (us \mathbin{+\!\!+} ys)$
    in ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { fold $(+\!\!+)$ }

✓**case** xs **of**
  $[\,] \to$ **let** $ws = ys$ **in** $ws \mathbin{+\!\!+} zs$
  $(u : us) \to$ **let** $ws = u : (us \mathbin{+\!\!+} ys)$
    in ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { inline ws and remove the unused binding }

✓**case** xs **of**
  $[\,] \to$ ✓$ys \mathbin{+\!\!+} zs$
  $(u : us) \to$ **let** $ws = u : (us \mathbin{+\!\!+} ys)$
    in ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { move the tick outside $\mathbb{D}$'s hole, where
  $\mathbb{D} = [-] \mathbin{+\!\!+} zs$ }

✓**case** xs **of**
  $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
  $(u : us) \to$ **let** $ws = u : (us \mathbin{+\!\!+} ys)$
    in ✓**case** ws **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { inline ws and remove unused binding }

✓**case** xs **of**
  $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
  $(u : us) \to$ ✓**case** ✓$(u : (us \mathbin{+\!\!+} ys))$ **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { move the tick outside $\mathbb{D}$'s hole, where
  $\mathbb{D} =$ **case** $[-]$ **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$ }

✓**case** xs **of**
  $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
  $(u : us) \to$ ✓✓**case** $u : (us \mathbin{+\!\!+} ys)$ **of**
    $[\,] \to zs$
    $(v : vs) \to v : (vs \mathbin{+\!\!+} zs)$

$\lhd\!\rhd$ { case of known constructor }

✓**case** xs **of**
  $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
  $(u : us) \to$ ✓✓$(u : ((us \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs))$

$\rhd$ { remove the ticks }

✓**case** xs **of**
  $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
  $(u : us) \to u : ((us \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs)$

$=$ { renaming }

✓**case** xs **of**
  $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
  $(x : xs) \to x : ((xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs)$

$=$ { define $\mathbb{C}$, where
  $\mathbb{C} =$ **case** xs **of**
    $[\,] \to$ ✓$(ys \mathbin{+\!\!+} zs)$
    $(x : xs) \to x : [-]$ }

✓$\mathbb{C}[(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs]$

Figure 4.2: Proof of improvement property 4.4

many cases simplifies, its resulting technicalities.

4.3.1 Syntax and semantics

The operational model that forms the basis of call-by-need improvement theory is an untyped, higher-order language with mutually recursive let bindings. The call-by-need semantics is originally due to Sestoft (1997) and reflects Haskell's use of lazy evaluation. Furthermore, the language is comparable to (a normalised version of) the Core language of the Glasgow Haskell Compiler (GHC Team 2019). We use these similarities to apply results from this theory directly to pure Haskell programs.

Terms of the language are defined by the following grammar, which also comprises the abstract syntax manipulated by the Unie system:

$$
\begin{array}{rcl}
M,\ N & ::= & x \\
& | & \lambda x.M \\
& | & M\ x \\
& | & \textbf{let}\ \{\ \vec{x}\ =\ \vec{M}\ \}\ \textbf{in}\ N \\
& | & c\ \vec{x} \\
& | & \textbf{case}\ M\ \textbf{of}\ \{\ c_i\ \vec{x}_i\ \rightarrow\ N_i\ \}
\end{array}
$$

We use the symbols x, y, and z for variables, c for constructors, and write $\vec{x} = \vec{M}$ for a sequence of bindings of the form $x = M$. Similarly, we write $c_i\ \vec{x}_i \rightarrow N_i$ for a sequence of **case** alternatives of the form $c\ \vec{x} \rightarrow N$. Literals are represented by constructors of zero arity, and all constructors are assumed to be fully applied. A term is a *value*, denoted V, if it is of the form $\lambda x.M$ or $c\ \vec{x}$, which corresponds to the notion of weak head normal form.

The abstract machine for evaluating terms maintains a state $\langle \Gamma, M, S \rangle$ consisting of a heap, Γ, given by a set of bindings from variables to terms; the term M currently being evaluated; and the evaluation stack, S, given by a list of tokens used by the abstract machine. The machine operates by evaluating the current term to a value, and then decides how to continue based on the top token of the stack. Bindings generated by **let**s are added to the heap and only taken off when performing a *Lookup* operation. A *Lookup* executes

$$
\begin{array}{llll}
\langle\ \Gamma\{\ x = M\ \}, x, S\ \rangle & \longrightarrow & \langle\ \Gamma, M, \#x : S\ \rangle & \textit{Lookup} \\
\langle\ \Gamma, V, \#x : S\ \rangle & \longrightarrow & \langle\ \Gamma\{\ x = V\ \}, V, S\ \rangle & \textit{Update} \\
\langle\ \Gamma, M\ x, S\ \rangle & \longrightarrow & \langle\ \Gamma, M, x : S\ \rangle & \textit{Unwind} \\
\langle\ \Gamma, \lambda x.M, y : S\ \rangle & \longrightarrow & \langle\ \Gamma, M[y\,/\,x], S\ \rangle & \textit{Subst} \\
\langle\ \Gamma, \textbf{case}\ M\ \textbf{of}\ \textit{alts}, S\ \rangle & \longrightarrow & \langle\ \Gamma, M, \textit{alts} : S\ \rangle & \textit{Case} \\
\langle\ \Gamma, c_j\ \vec{y}, \{c_i\ \vec{x}_i \to N_i\} : S\ \rangle & \longrightarrow & \langle\ \Gamma, N_j[\vec{y}\,/\,\vec{x}_j], S\ \rangle & \textit{Branch} \\
\langle\ \Gamma, \textbf{let}\ \{\ \vec{x} = \vec{M}\ \}\ \textbf{in}\ N, S\ \rangle & \longrightarrow & \langle\ \Gamma\{\ \vec{x} = \vec{M}\ \}, N, S\ \rangle \quad \vec{x} \not\downarrow \mathrm{dom}(\Gamma, S) & \textit{Letrec}
\end{array}
$$

Figure 4.3: Semantics of Sestoft's call-by-need abstract machine

by adding a token on top of the stack, representing where the term was looked up, and then evaluating that term to a value before replacing it on the heap. This ensures that each binding is evaluated at most once: a key aspect of call-by-need semantics. Restricting function arguments to be variables means that all non-atomic arguments must be introduced via **let** statements and thus can be evaluated at most once.

The transition semantics of the abstract machine are given in figure 4.3. The *Letrec* transition assumes that $\vec{x}$ is disjoint from the domain of Γ and S, denoted using $\not\downarrow$. This can always be achieved by alpha-renaming.

Program contexts

Program contexts are meta-terms representing a family of terms containing *holes*, denoted $[-]$. In turn, if $\mathbb{C}$ is a context, then $\mathbb{C}[M]$ represents $\mathbb{C}$ with the term M substituted for each hole. Contexts are defined by the following grammar:

$$
\begin{array}{rcl}
\mathbb{C},\ \mathbb{D} & ::= & [-] \\
& | & x \\
& | & \lambda x.\mathbb{C} \\
& | & \mathbb{C}\ x \\
& | & \textbf{let}\ \{\ \vec{x}\ =\ \vec{\mathbb{C}}\ \}\ \textbf{in}\ \mathbb{D} \\
& | & c\ \vec{x} \\
& | & \textbf{case}\ \mathbb{C}\ \textbf{of}\ \{\ c_i\ \vec{x}_i\ \to\ \mathbb{D}_i\ \}
\end{array}
$$

Note that **let** and **case** statements admit contexts with multiple holes.

A *value context*, denoted $\mathbb{V}$, is a context that is in weak head normal form. There are also two other forms of contexts, which can contain at most one hole that must appear as the target of evaluation, meaning that evaluation cannot proceed until the hole is substituted. These forms of contexts are known as *applicative contexts* and *evaluation contexts*, and are defined by the following two grammars, respectively:

$$
\begin{array}{rcl}
\mathbb{A} & ::= & [-] \\
& | & \mathbb{A}\ x \\
& | & \textbf{case}\ \mathbb{A}\ \textbf{of}\ \{\ c_i\ \vec{x}_i\ \rightarrow\ \vec{M}_i\ \}
\end{array}
\qquad\qquad
\begin{array}{rcl}
\mathbb{E} & ::= & \mathbb{A} \\
& | & \textbf{let}\ \{\ \vec{x}\ =\ \vec{M}\ \}\ \textbf{in}\ \mathbb{A} \\
& | & \textbf{let}\ \{\ \vec{y}\ =\ \vec{M}; \\
& & \qquad x_0\ =\ \mathbb{A}_0[x_1]; \\
& & \qquad x_1\ =\ \mathbb{A}_0[x_2]; \\
& & \qquad ... \\
& & \qquad x_n\ =\ \mathbb{A}_n\ \}\ \textbf{in}\ \mathbb{A}[x_0]
\end{array}
$$

Given that improvement is a contextual definition, the transformation rules we apply when reasoning about improvement intuitively must also be defined contextually. In general, however, it is not necessarily the case that a given transformation rule is valid for *all* forms of contexts. For example, a tick can be freely moved in and out of an evaluation context using the following rule, but this transformation is not valid for all other forms of contexts:

$$\mathbb{E}[\checkmark M] \quad \lhd\!\rhd \quad \checkmark\mathbb{E}[M] \qquad (\checkmark\text{-}\mathbb{E})$$

Similarly, under certain conditions regarding free (FV) and bound variables (BV), an evaluation context can be moved in and out of a **case** statement using the rule

$$
\begin{array}{c}
\mathbb{E}[\textbf{case}\ M\ \textbf{of}\ \{\ pat_i\ \rightarrow\ N_i\ \}] \\
\lhd\!\rhd \qquad FV(M)\ \not\downarrow\ BV(\mathbb{E}) \qquad FV(\mathbb{E})\ \not\downarrow\ pat_i \\
\textbf{case}\ M\ \textbf{of}\ \{\ pat_i\ \rightarrow\ \mathbb{E}[N_i]\ \}
\end{array}
\qquad (\text{case-}\mathbb{E})
$$

but again, this is not true for all other forms of contexts.

Consequently, when applying a transformation rule to a given term, we must often

ensure that the term is syntactically compatible with a context of a particular form, as stipulated by the chosen rule. When conducted manually, the process of deconstructing a term M into an appropriate context $\mathbb{C}$ and substitution N such that $M = \mathbb{C}[N]$ becomes tedious, time consuming, and highly prone to error.

To address this problem, the Unie system handles all aspects of context manipulation automatically on behalf of users. In particular, each time a rule is applied, the system analyses the syntactic form of the respective term, ensuring it is compatible with the chosen rule's specification. If this is not the case, the system prevents the rule from being applied and reports a suitable error message. The same is also true if a rule's side conditions are not satisfied, such as those regarding free and bound variables for (case-$\mathbb{E}$). Thus, in regard to contexts, not only does Unie make a correct transformation much easier to apply, it makes an incorrect transformation impossible to apply.

4.3.2 Operational improvement

Moran and Sands (1999) showed that the total number of steps taken to evaluate any term M is bounded by a function that is linear in the number of *Lookup* operations (see figure 4.3) required during M's evaluation. Therefore, the evaluation cost of each term can be measured *asymptotically* by just counting uses of *Lookup*. This is the notion of cost used in their work, and so we adopt it for the purposes of our system.

Formally, we write $M{\downarrow^n}$ if the abstract machine of figure 4.3 proceeds from the initial state $\langle \emptyset, M, \epsilon \rangle$ to a final state $\langle \Gamma, V, \epsilon \rangle$ with n uses of *Lookup*. Similarly, we write $M{\downarrow^{\leqslant n}}$ to mean that $M{\downarrow^m}$ for some m such that $m \leqslant n$. Using this cost model, we can now formalise the notion of improvement in our setting. A term M is *improved by* a term N, written $M \trianglerighteq N$, if the following holds for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^n} \quad \Longrightarrow \quad \mathbb{C}[N]{\downarrow^{\leqslant n}}$$

That is, one term is improved by another if the latter takes no more *Lookup* operations to evaluate than the former in all program contexts. In turn, we say that two terms M and N are *cost equivalent*, written $M \lhd\!\rhd N$, if for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^n} \iff \mathbb{C}[N]{\downarrow^n}$$

Standard notions of operational equality (see section A.1.2 for a general overview) state that two terms M and N are *observationally equivalent* if they have the same termination behaviour in all program contexts, which is typically written as:

$$\mathbb{C}[M]{\Downarrow} \iff \mathbb{C}[N]{\Downarrow}$$

Furthermore, *observational approximation*, written as

$$\mathbb{C}[M]{\Downarrow} \implies \mathbb{C}[N]{\Downarrow}$$

states that N terminates in at least as many program contexts as M. In this instance, $M{\Downarrow}$ can be interpreted as the abstract machine of figure 4.3 proceeding from the initial state $\langle \emptyset, M, \epsilon \rangle$ to some final state $\langle \Gamma, V, \epsilon \rangle$ in a finite number of steps.

Improvement theory can thus be viewed as a refinement of the standard theories of observational approximation/equivalence in which the basic observation made about a program's execution (that is, whether or not it terminates) includes intensional information about its computational cost (Sands 1997). In consequence, the definition of improvement entails that N is both more efficient and terminates more often than M. In turn, cost equivalence entails that M and N have the same cost and termination behaviour.

Just as before, we must keep track of evaluation costs explicitly when reasoning about improvement. Our informal introduction viewed the tick operator, $(\checkmark)$, as a syntactic construct that represents a unit time cost. Here, we follow the work of Hackett and Hutton (2014) and define $(\checkmark)$ as a derived operation:

$$\checkmark M \;=\; \mathbf{let}\ \{\, x \;=\; M \,\}\ \mathbf{in}\ x \qquad (x \text{ fresh})$$

This definition takes precisely two steps to evaluate to M: one to add the binding to the heap and the other to look it up. As one of these steps is a *Lookup* operation, the cost of evaluating M is increased by exactly one, as required. The following tick elimination rule

still holds, but, as before, the reverse is not valid:

$$\checkmark M \quad \gtrapprox \quad M \qquad (\checkmark\text{-elim})$$

The $\gtrapprox$ relation formalises when one term is at least as efficient as another in all contexts, but this is a strong requirement. We use the notion of *weak improvement* (Hackett and Hutton 2014) when one term is at least as efficient as another within a constant factor. Formally, M is *weakly improved by* N, written $M \gtrapprox N$, if there exists a function $f(x) = kx + c$ for $k, c \geqslant 0$ such that for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^n} \quad \Longrightarrow \quad \mathbb{C}[N]{\downarrow^{\leqslant f(n)}}$$

This can be interpreted as "replacing M with N may make programs worse, but it will not make them asymptotically worse" (Hackett and Hutton 2014). Analogous to cost equivalence, we also have weak cost equivalence, written $M \lessapprox\gtrapprox N$, which is defined in the obvious manner. As weak improvement ignores constant factors, we can introduce and eliminate ticks while preserving weak cost equivalence:

$$M \quad \lessapprox\gtrapprox \quad \checkmark M \qquad (\checkmark\text{-intro})$$

4.3.3 Inequational reasoning

When constructing an improvement proof, we must pay close attention to the relations used in our calculation. This is because the transformations we apply as part of our reasoning are defined using different notions of improvement: $\gtrsim$, $\gtrapprox$, $\lessapprox\gtrsim$, $\lessapprox\gtrapprox$, some of which may not entail the relation of the given proof statement, in which case their use would lead to an incorrect calculation. For example, as $\gtrsim \subseteq \gtrapprox$, any transformation defined using $\gtrsim$ automatically entails $\gtrapprox$, but the converse is not true. Similarly, removing a tick ($\checkmark$-elim) is an improvement, whereas unfolding a function's definition (4.3) is not.

The Unie system simplifies such inequational reasoning by ensuring that each transformation rule applied by a user entails a particular improvement relation established prior to the start of the reasoning process. If a user attempts to apply a rule that does not entail

this relation, the system will reject it and display an appropriate error message. Similarly to the intricacies in dealing with different forms of contexts discussed previously, Unie's safety mechanisms in this instance make incorrect transformations impossible to apply.

4.3.4 Tick algebra

We conclude this section by discussing Moran and Sands' (1999) *tick algebra*, which is a collection of equational and inequational laws for propagating evaluation costs within terms while preserving or increasing efficiency. These laws make up a large proportion of the transformation rules provided by our system, and are a rich inequational theory that subsumes all the axioms of the call-by-need calculus of Ariola et al. (1995).

We refer the reader to (Moran and Sands 1999) for the full tick algebra, but present two example laws below to illustrate their nature and complexity:

$$\begin{array}{ll} & \textbf{let } \{\ \vec{x} = \vec{L}\ \} \textbf{ in let } \{\ \vec{y} = \vec{M}\ \} \textbf{ in } N \\ \lhd\!\rhd & \quad \vec{x} \not\downarrow \vec{y} \quad \vec{y} \not\downarrow FV(\vec{L}) \\ & \textbf{let } \{\ \vec{x} = \vec{L},\ \vec{y} = \vec{M}\ \} \textbf{ in } N \end{array} \qquad \text{(let-flatten)}$$

$$\begin{array}{ll} & \checkmark\textbf{let } \{\ x = z,\ \vec{y} = \vec{M}[z/w]\ \} \textbf{ in } N[z/w] \\ \rhd & \\ & \textbf{let } \{\ x = z,\ \vec{y} = \vec{M}[x/w]\ \} \textbf{ in } N[x/w] \end{array} \qquad \text{(var-expand)}$$

The (let-flatten) rule is a cost equivalence that allows us to merge the binders of two **let** statements, modulo binder collisions and variable capture. In turn, (var-expand) is an improvement that allows us to replace a binding with its binder provided there is a tick in front of the **let** to pay for this expansion. Also included in the tick algebra are the (✓-$\mathbb{E}$) and (case-$\mathbb{E}$) rules introduced previously in section 4.3.1.

These laws are only a small fragment of the tick algebra. However, it should be evident from these examples that applying such rules manually can be a difficult task. In particular, the use of different improvement relations, different forms of contexts, and each rule having

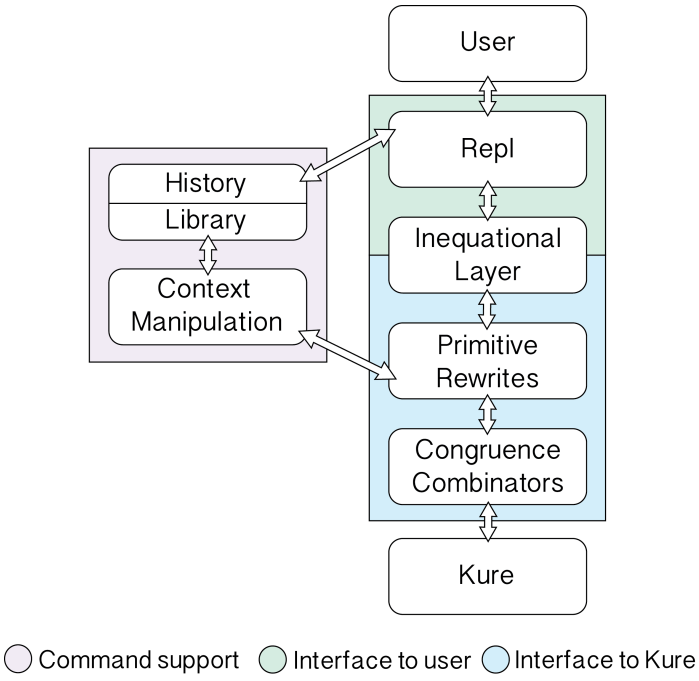


Figure 4.4: Unie's core architecture

a unique syntactic form makes it challenging to know when a rule can be applied correctly. Furthermore, many laws have side conditions, often concerning free and bound variables as with (case-𝔼) and (let-flatten), which must be checked every time they are applied.

A primary strength of the Unie system is that it provides mechanical support for all of the above tasks, which are integral to the advanced style of inequational reasoning used in improvement proofs. Moreover, the system automatically performs, for example, alpha-renaming to enable rules such as (let-flatten) and (case-𝔼) to be applied correctly.

4.4 Architecture of Unie

In this section, we introduce the key features of the Unie system developed to support inequational reasoning for improvement theory. We begin by overviewing the system's core architecture. We then discuss how Unie supports interactive program transformation, automatically matches against and generates program contexts, and how it (semi-formally) validates inequational steps of reasoning. Finally, we discuss how the system's support for cost-equivalent contexts simplifies proofs of improvement.

4.4.1 Overview

The main components of the Unie system and the interactions between each component are illustrated in figure 4.4. A summary of the figure is as follows:

- The *read-evaluate-print loop* handles interactions with the user.
- The *history* records successfully executed commands and the resultant proof state of each command. In turn, the *library* maintains a collection of term, context, and cost-equivalent context definitions for use during transformations, together with a collection of user-defined command scripts.
- The *inequational layer* ensures that transformation rules invoked by the user are safe to apply in the current proof state.
- *Primitive rewrites* and *congruence combinators* are basic building blocks for defining program transformations in a modular fashion.
- The *context manipulation* component supports the automatic generation, matching, and checking of different forms of contexts.

Other notable features of the system include:

- A *pretty-printer* for abstract syntax and a *parser* for source syntax.
- An *abstract machine* for evaluating terms using the semantics of figure 4.3.
- A *help system* in the style of the Unix `man` command.
- Informative *error messages*, which explain why an invalid transformation rule cannot be applied in a particular proof state.

4.4.2 Read-evaluate-print loop

A key aspect of developing improvement proofs interactively is applying rewrite rules to sub-terms. We prioritise this requirement by maintaining a focus into the term being transformed and providing navigation commands for changing the focus. Transformations are then applied to the sub-term currently in focus. By default, only the focused sub-term

is displayed on screen, and updated each time a navigation command is executed. For situations where it may be beneficial to always display the whole term, the system provides an option to highlight the current focus. This feature is demonstrated in figure 4.1.

4.4.3 Inequational layer

Each time a transformation rule is invoked by the user, the system ensures it is safe to apply in the current proof state. An important part of this verification step is to check whether the rule's operator, for example, $\trianglerighteq$ or $\lhd\rhd$, entails the improvement relation of the initial proof statement (see section 4.6 for an example of this). If this is not the case, the transformation rule is rejected and a suitable error message displayed. Another important check determines whether the side conditions of the chosen rule are satisfied.

Not only are these checks essential to ensuring calculations produced by the system are well-formed, they also permit beginner users to safely experiment with improvement rules.

4.4.4 Primitive rewrites and congruence combinators

Similarly to the equational reasoning assistant Hermit (Farmer 2015), our system utilises the Kansas University Rewrite Engine (Kure) (Sculthorpe, Frisby, and Gill 2014) for specifying and applying transformations to the abstract syntax of its operational model.

In brief, Kure is a strategic programming language (Lämmel, Visser, and Visser 2003) that provides a principled method for traversing and transforming datatypes. The fundamental idea behind Kure is to separate the implementations of traversals and the implementations of transformations. Traversal strategies and transformation rules can thus be reused and combined independently. For our system, this allows a sophisticated library of transformation rules, tailored to the needs of improvement theory, to be constructed by composing a small number of basic operations using a selection of Kure's primitive combinators.

In addition, Kure's use of datatype-generic programming (Gibbons 2006) means that traversals can navigate to particular locations in order to apply type-specific transformations, giving fine control over when and where transformations are applied within a data type. This is vital for our implementation, as each reasoning step in an improvement proof typically transforms only a single subpart of an overall term.

In summary, our approach to implementing rewrite rules, using so-called *primitive rewrites* and *congruence combinators*, was heavily inspired by the Hermit system and builds on the work in (Farmer et al. 2012; Farmer 2015). We refer readers to the following articles (Sculthorpe, Frisby, and Gill 2014; Farmer 2015) for a detailed discussion on the relevant concepts, including an introduction to the Kure library.

4.4.5 Cost-equivalent contexts

The Unie system maintains a library of *cost-equivalent contexts*, which syntactically do not satisfy the requirements for a particular form of context (for example, value or evaluation) but are nonetheless cost equivalent to a context of such a form, and hence admit the same inequational laws. Cost-equivalent contexts occur frequently in improvement proofs, such as in (Moran and Sands 1999; Hackett and Hutton 2014), as they lead to simplified reasoning steps. Once added by the user, cost-equivalent contexts are manipulated by the system in the same manner as all other forms of contexts.

In section 4.6, we demonstrate how cost-equivalent contexts can be added to Unie's library and then utilised in a mechanised improvement proof.

Remark. We note that the use of cost-equivalent contexts is potentially unsound, as the system does not check that they are indeed cost-equivalent to syntactically valid contexts. Adding such checks in the form of provable lemmas is part of our future work.

4.4.6 Context manipulation

Managing contexts is one of the primary intricacies in developing improvement proofs. In this subsection, we explain how this is handled by the Unie system.

Context matching

In our setting, a *context pattern* is simply shorthand notation for one or more program contexts, allowing sub-terms to be specified implicitly using wildcards and constructor patterns. For example, the context **let** { $x = a\,;\ \ y = b$ } **in** $[-]$ may be described by

any of the following context patterns, among others:

$$\textbf{let}\ \{\ x\ =\ a\,;\ \ \_\ \}\ \textbf{in}\ [-] \qquad \textbf{let}\ \{\ x\ =\ \mathit{VAR}\,;\ \ \_\ \}\ \textbf{in}\ [-] \qquad \textbf{let}\ \_\ \textbf{in}\ [-]$$

As is standard, the underscores above are wildcards that match with any term, while VAR is a constructor pattern that matches with any variable.

Although context patterns do not represent unique contexts, when used in conjunction with a specific transformation rule, they are often sufficient to determine a unique context. In practice, they are used to simplify and reduce the amount of input required from users when interacting with the system. For example, recall the following rule, (✓-$\mathbb{E}$), allowing ticks to be moved in and out of evaluation contexts:

$$\mathbb{E}[\checkmark M] \quad \lhd\!\rhd \quad \checkmark\mathbb{E}[M]$$

Suppose we wish to apply this rule to the term $\checkmark(a\ b\ c)$. To do so, we must determine an evaluation context $\mathbb{E}$ and a term M for which $\checkmark\mathbb{E}[M] = \checkmark(a\ b\ c)$. In this case it is straightforward, such as by taking $\mathbb{E} = [-]\ b\ c$ and $M = a$. Applying (✓-$\mathbb{E}$) right-to-left then allows us to move the tick inside the context:

$$\begin{array}{ll} & \checkmark(a\ b\ c) \\ \lhd\!\rhd & \{\ \checkmark\text{-}\mathbb{E} \quad \text{where } \mathbb{E} = [-]\ b\ c\ \} \\ & (\checkmark a)\ b\ c \end{array}$$

This transformation can be mirrored almost identically in our system:

```
unie> trans $`(a b c)$
✓(a b c)
[1]> untick-eval $[-] b c$
⋪⋫ ✓a b c
```

In this instance, Unie uses the specified context $[-]\ b\ c$ and the current term $\checkmark(a\ b\ c)$ to verify the preconditions necessary for (✓-$\mathbb{E}$)'s safe application. In particular, it checks that $[-]\ b\ c$ is a valid evaluation context and, by calculating the substituted term $M = a$,

ensures the initial term has the required form $\checkmark\mathbb{E}[M]$. If any of these preconditions were not met, the transformation rule would be rejected.

Suppose that the context $\mathbb{E}$ from the above example was more complex, such as a **let** statement with multiple bindings. In this case, it would be impractical to expect a user to manually enter $\mathbb{E}$'s complete definition. Context patterns address this problem by allowing users to specify contexts by shorthand representations. The system uses these representations to automatically calculate valid contexts on the user's behalf, by *matching* the specified pattern against the syntax of the term being transformed. For example, we can use wildcard patterns to apply the above transformation in a simplified manner:

```
unie> trans $`(a b c)$
✓(a b c)
[1]> untick-eval $[-] _ _$
◁▷ ✓a b c
```

Context generation

Applying the same transformation rule as above but without specifying a context parameter for $\mathbb{E}$ leads to the following system response:

```
unie> trans $`(a b c)$
✓(a b c)
[1]> untick-eval
Select a context/substitution option:
(1) E = [-]       M = a b c
(2) E = [-] c     M = a b
(3) E = [-] b c   M = a
```

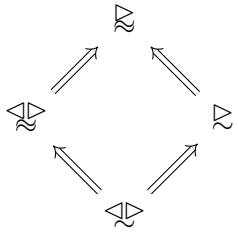
That is, three possible context/substitution pairs have been automatically *generated* on behalf of the user. Each allow the ($\checkmark$-$\mathbb{E}$) transformation rule to be correctly applied to the given term. Option three corresponds to our previous choice:

```
[1]> 3
◁▷ ✓a b c
```

Context generation is available when applying any of the system's transformation rules. In contrast to context manipulation, which affords a simple means to specify contexts precisely, context generation fills in such details automatically. In practice, this feature has proved to be invaluable when validating proofs from the original article (Moran and Sands 1999) on improvement theory, as the calculations presented in this work state which tick algebra rules are applied, but not *how* they are applied.

4.4.7 Inequational reasoning

As we have seen, a central feature of the Unie system is its support for inequational reasoning. The relationships between the different improvement relations that were introduced in section 4.3 are summarised in the following lattice:



In Unie, a proof is initiated by the user entering a proof statement, such as $\checkmark\checkmark x \trianglerighteq x$. The system extracts from the proof statement a 'global' improvement relation: $\trianglerighteq$ in this case. Each time a transformation is applied, its corresponding operator is checked to ensure that it entails this global relation in the above lattice. If this is not the case, the rewrite rule is rejected and an appropriate error message is displayed, for example:

```
unie> trans $``x$ IMP $x$
Global relation set: ▷≈.
Transformation goal set: x.
✓✓x
[1]> tick-elim
▷≈ ✓x
[2]> untick-intro
Relation error: ◁▷≈ ≠ ▷≈.
```

The first step of the above reasoning successfully applies the `tick-elim` rule. This rule is

defined using the $\gtrsim$ relation, which trivially entails the global relation. On the other hand, the error message at the bottom states that the relation of the second rule `untick-intro`, $\lhd\!\!\rhd$, does not entail $\gtrsim$. Hence the transformation is rejected.

4.5 The worker/wrapper transformation

While developing the Unie system, we were guided by the desire to mechanically verify the results from the article that renewed interest in improvement theory (Hackett and Hutton 2014). In this section, we review the main result of this article, which shows that the worker/wrapper transformation is an improvement, and an example application of this result, which shows how the familiar naive reverse function on lists can be improved. In the next section, we show how the latter result can be mechanised in our system.

4.5.1 Formalising correctness

The worker/wrapper transformation is a technique for improving the performance of recursive programs by changing their types (Gill and Hutton 2009). Given a recursive program of some type, the basic idea is to factorise the program into a *worker* program of a different type, together with a *wrapper* program to interface between the original program and the new worker. If the worker type supports more efficient operations than the original type, then this efficiency improvement should result in a more efficient program overall.

More formally, suppose that we are given a recursive program, which is defined as the least fixed point, $\mathit{fix}\ f$, of a function f on some type A. Now, consider a more efficient program that performs the same task, defined by first taking the least fixed point, $\mathit{fix}\ g$, of a function g on some other type B, and then migrating the resulting value back to the original type A by applying a conversion function, abs. The equivalence between the two programs is captured by the following equation:

$$\mathit{fix}\ f \quad = \quad \mathit{abs}\ (\mathit{fix}\ g)$$

This equation states that the original program, $\mathit{fix}\ f$, can be factorised into the application of a wrapper function abs to a worker program $\mathit{fix}\ g$. The validity of the equation depends

on some properties of the functions f, g, and abs, together with a dual conversion function rep. These properties are summarised in the following worker/wrapper correctness theorem (Sculthorpe and Hutton 2014). Given functions $f : A \to A$, $g : B \to B$, $abs : B \to A$, and $rep : A \to B$ satisfying one of the assumptions (a–c) and one of the conditions (1–3)

$$\begin{array}{llcllcl} \text{(a)} & abs \circ rep & = & id_A & \text{(1)} & g & = & rep \circ f \circ abs \\ \text{(b)} & abs \circ rep \circ f & = & f & \text{(2)} & g \circ rep & = & rep \circ f \\ \text{(c)} & \mathit{fix}\ (abs \circ rep \circ f) & = & \mathit{fix}\ f & \text{(3)} & f \circ abs & = & abs \circ g \end{array}$$

then we have the correctness equation: $\mathit{fix}\ f = abs\ (\mathit{fix}\ g)$.

4.5.2 Formalising improvement

The previous subsection formalised that the worker/wrapper transformation is correct, in the sense that the original and new programs have the same denotational meaning. We now formalise that the transformation improves efficiency, in the sense that the new program improves the runtime performance of the original.

To reformulate the correctness theorem as an improvement theorem, we must first make some changes to the basic setup to take account of the differences between the underlying denotational and operational theories. In particular, functions are replaced by contexts, that is, the functions f and g become contexts $\mathbb{F}$ and $\mathbb{G}$; the use of a fix operator is replaced by recursive **let** bindings, that is, $\mathit{fix}\ f$ becomes $\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x$; and the use of equality is replaced by an appropriate improvement relation, that is, $=$ becomes $\trianglerighteq$, $\trianglerighteq$, or $\lhd\!\rhd$. Thus, we have the following worker/wrapper improvement theorem (Hackett and Hutton 2014). Given value contexts $\mathbb{F}$, $\mathbb{G}$, $\texttt{Abs}$, and $\texttt{Rep}$ satisfying one of the assumptions (a–c)

$$\begin{array}{lll} \text{(a)} & \texttt{Abs}[\texttt{Rep}[x]] & \lhd\!\rhd\ x \\ \text{(b)} & \texttt{Abs}[\texttt{Rep}[\mathbb{F}[x]]] & \lhd\!\rhd\ \mathbb{F}[x] \\ \text{(c)} & \mathbf{let}\ x = \texttt{Abs}[\texttt{Rep}[\mathbb{F}[x]]]\ \mathbf{in}\ x & \lhd\!\rhd\ \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x \end{array}$$

where x is free, and one of the conditions (1–3)

$$
\begin{array}{lllll}
(1) & \mathbb{G}[x] & \lessapprox & \texttt{Rep}[\mathbb{F}[\texttt{Abs}[x]]] \\
(2) & \mathbb{G}[\checkmark\texttt{Rep}[x]] & \lesssim & \texttt{Rep}[\checkmark\mathbb{F}[x]] \\
(3) & \texttt{Abs}[\checkmark\mathbb{G}[x]] & \lessapprox & \mathbb{F}[\checkmark\texttt{Abs}[x]]
\end{array}
$$

then we have the improvement: **let** $x = \mathbb{F}[x]$ **in** $x \gtrapprox$ **let** $x = \mathbb{G}[x]$ **in** $\texttt{Abs}[x]$.

The assumptions and conditions above ensuring the original recursive program **let** $x = \mathbb{F}[x]$ **in** x is improved by **let** $x = \mathbb{G}[x]$ **in** $\texttt{Abs}[x]$ are natural extensions of the corresponding properties for correctness. For example, correctness condition (1), $g = \mathit{rep} \circ f \circ \mathit{abs}$, is replaced by improvement condition (1), $\mathbb{G}[x] \lessapprox \texttt{Rep}[\mathbb{F}[\texttt{Abs}[x]]]$.

The proof of the this theorem utilises two other results: a rolling rule and a fusion rule. Both are central to the worker/wrapper transformation (Gill and Hutton 2009), and can be proved using tick algebra laws. Consequently, the worker/wrapper improvement theorem is itself a direct result of the tick algebra's inequational theory. Indeed, all aforementioned results have been verified using our system. The proofs can be found online in the form of command scripts that can be loaded into Unie (Handley 2018).

4.5.3 Improving naive reverse

Recall from the previous chapter the standard definition for naively reversing lists:

$$
\begin{array}{l}
\mathit{slowRev} \;=\; \textbf{let}\ f \;=\; \texttt{Revbody}[f]\ \textbf{in}\ f \\
\\
\texttt{Revbody} \;=\; \lambda xs.\ \textbf{case}\ xs\ \textbf{of} \\
\quad [\,] \qquad \to\ [\,] \\
\quad (y : ys) \;\to\; [-]\ ys\ +\!\!+\ [y]
\end{array}
$$

Here, *slowRev* is defined using a recursive **let** binding rather than explicit recursion, with the context `Revbody` capturing the non-recursive part of the definition. We previously stated that this implementation is inefficient due to its use of the append operator, $(+\!\!+)$, which is linear in the length of its first argument. For the purposes of this example, we would like to use the worker/wrapper technique to improve it.

The first step is to select a new type to replace the original type $[a] \to [a]$, and define contexts to convert between the two types. As before, we utilise the type $[a] \to [a] \to [a]$

to provide an additional argument used to accumulate the resulting list. The necessary contexts to convert between the original and new types are defined as follows:

$$\texttt{Abs} \;=\; \lambda xs.\; [-]\; xs\; [\,] \qquad\qquad \texttt{Rep} \;=\; \lambda xs.\; \lambda ys.\; [-]\; xs \;+\!\!+\; ys$$

We must now verify that the conversion contexts, `Abs` and `Rep`, satisfy one of the worker/wrapper assumptions. We verify assumption (b) as follows:

$\texttt{Abs}[\texttt{Rep}[\texttt{Revbody}[f]]]$

$=$ { unfold the definitions of `Abs` and `Rep` }

$\lambda xs.\; (\lambda xs.\; \lambda ys.\texttt{Revbody}[f]\; xs \;+\!\!+\; ys)\; xs\; [\,]$

$\lhd\!\rhd$ { β-reduction }

$\lambda xs.\; \texttt{Revbody}[f]\; xs \;+\!\!+\; [\,]$

$=$ { unfold the definition of `Revbody` }

$\lambda xs.\; (\lambda xs.\; \textbf{case}\; xs\; \textbf{of}$

$\quad [\,] \qquad \rightarrow\; [\,]$

$\quad (y\; :\; ys) \rightarrow\; f\; ys \;+\!\!+\; [y])\; xs \;+\!\!+\; [\,]$

$\lhd\!\rhd$ { β-reduction }

$\lambda xs.\; (\textbf{case}\; xs\; \textbf{of}$

$\quad [\,] \qquad \rightarrow\; [\,]$

$\quad (y\; :\; ys) \rightarrow\; f\; ys \;+\!\!+\; [y]) \;+\!\!+\; [\,]$

$\lhd\!\rhd$ { case-$\mathbb{E}$ where $\mathbb{E} = [-] \;+\!\!+\; [\,]$ }

$\lambda xs.\; \textbf{case}\; xs\; \textbf{of}$

$\quad [\,] \qquad \rightarrow\; [\,] \;+\!\!+\; [\,]$

$\quad (y\; :\; ys) \rightarrow\; (f\; ys \;+\!\!+\; [y]) \;+\!\!+\; [\,]$

$\lhd\!\rhd$ { associativity of $(+\!\!+)$ }

$\lambda xs.\; \textbf{case}\; xs\; \textbf{of}$

$\quad [\,] \qquad \rightarrow\; [\,] \;+\!\!+\; [\,]$

$\quad (y\; :\; ys) \rightarrow\; f\; ys \;+\!\!+\; ([y] \;+\!\!+\; [\,])$

$\lhd\!\rhd$ { evaluate $[\,] \;+\!\!+\; [\,]$ and $[y] \;+\!\!+\; [\,]$ }

$\lambda xs.\; \textbf{case}\; xs\; \textbf{of}$

$$
\begin{array}{ll}
 & \quad [\,] \quad\quad \to [\,] \\
 & \quad (y : ys) \to f\ ys +\!\!+ [y] \\
= & \ \{ \text{ fold the definition of } \texttt{Revbody} \ \} \\
 & \ \texttt{Revbody}[f]
\end{array}
$$

Note that the above calculation uses the fact that $(+\!\!+)$ is associative up to weak cost equivalence, that is, $(xs +\!\!+ ys) +\!\!+ zs \lessapprox\!\!\gtrapprox xs +\!\!+ (ys +\!\!+ zs)$.

Next, we must verify that one of the worker/wrapper conditions is satisfied. In this case, we use condition (2) as a *specification* for the context $\mathbb{G}$, whose definition can be calculated using laws from the tick algebra. We omit the details for brevity, but they are included in the original article (Hackett and Hutton 2014), and result in the following definition:

$$
\begin{array}{l}
\mathbb{G} = \lambda xs.\, \lambda ys.\ \textbf{case}\ xs\ \textbf{of} \\
\quad [\,] \quad\quad \to ys \\
\quad (z : zs) \to \textbf{let}\ ws = z : ys\ \textbf{in}\ [-]\ zs\ ws
\end{array}
$$

The crucial step in the construction of $\mathbb{G}$ is applying property (4.2), which expresses that reassociating the append operator to the right is an improvement.

Finally, if we define $fastRev = \textbf{let}\ x = \mathbb{G}[x]\ \textbf{in}\ \texttt{Abs}[x]$, then, by applying the worker/wrapper improvement theorem, we have shown that the original version of *reverse* is improved by the new version, that is, $slowRev \gtrapprox fastRev$. Furthermore, by expanding out the definition of *fastRev* and renaming and simplifying the resulting **let** binding, we arrive at the familiar fast version of the original function:

```
fastRev :: [a] → [a]
fastRev xs = revCat xs []
  where
    revCat []       ys = ys
    revCat (x : xs) ys = revCat xs (x : ys)
```

4.6 Mechanising improvement proofs

In this section, we demonstrate how to improve the naive list-reversing function from the previous section mechanically using our system. In doing so, we illustrate a number of

Unie's key features and show how it supports interactive reasoning using transformation and navigation rules. All of the interaction is taken directly from the system itself, with some minor reformatting in light of the paper-based medium.

As in the previous section, we focus on the proof of assumption (b). Prior to constructing the proof, we must ensure that Unie has access to the definitions from section 4.5, which are required at different stages throughout. For convenience, we have stored them in a file whose contents are imported into the system's library using the `import-lib` command, and the names of the definitions displayed by `show-lib defs`. We have also included the definition of append as it is required in a number of proof steps involving evaluation.

```
unie> import-lib ./libs/slowRev
Info: library updated.
unie> show-lib defs
Terms:    (++), slowRev
Contexts: Abs, Rep, Revbody
```

We instruct the system to enter its transformation mode using `trans`. The relevant proof statement `Abs`[`Rep`[`Revbody`[f]]] $\lhd\!\!\rhd$ `Revbody`[f] is supplied as a parameter and determines the proof's global relation and goal. The global relation will prevent rules being applied whose operators do not entail $\lhd\!\!\rhd$, and we will be notified when the goal `Revbody`[f] is reached. When entering expressions into the system, the forms of contexts must be specified: `Abs`, `Rep`, and `Revbody` are value contexts, hence we use the `V_` prefix.

```
unie> trans $V_Abs[V_Rep[V_Revbody[f]]]$ WCE $V_Revbody[f]$
Global relation set: ◁▷.
Transformation goal set: V_Revbody[f].
```

Just as with the paper proof, we begin by unfolding the definitions of `Abs` and `Rep`, and beta-reducing inside the body of the outer abstraction. To reduce the correct sub-terms, we navigate using `left` and `right`, which move the focus to the current term's left and right child, respectively. In turn, `top` restores focus to the full expression.

```
V_Abs[V_Rep[V_Revbody[f]]]
[1]> unfold-def 'Abs ; unfold-def 'Rep
```

```
 = λxs.(λxs.λys.(V_Revbody[f] xs) ++ ys) xs []
[3]> right
 = (λxs.λys.(V_Revbody[f] xs) ++ ys) xs []
[4]> left
 = (λxs.λys.(V_Revbody[f] xs) ++ ys) xs
[5]> beta
◁▷ λys.(V_Revbody[f] xs) ++ ys
[6]> up ; beta
◁▷ (V_Revbody[f] xs) ++ []
[8]> top
 = λxs.(V_Revbody[f] xs) ++ []
```

Next, we unfold the definition of `Revbody` and beta-reduce the resulting redex. We then move `up` to focus on the application of append.

```
[9]> unfold-def 'Revbody
 = λxs.((λxs.case xs of
    []       → []
    (y : ys) → (f ys) ++ [y]) xs) ++ []
[10]> right ; left
 = (++) ((λxs.case xs of
    []       → []
    (y : ys) → (f ys) ++ [y]) xs)
[12]> right
 = (λxs.case xs of
    []       → []
    (y : ys) → (f ys) ++ [y]) xs
[13]> beta
◁▷ case xs of
    []       → []
    (y : ys) → (f ys) ++ [y]
[14]> up ; up
 = (case xs of
    []       → []
    (y : ys) → (f ys) ++ [y]) ++ []
```

Recall the (case-$\mathbb{E}$) rule, which allows an evaluation context to be moved inside a **case**

statement subject to certain conditions regarding free and bound variables:

$$\mathbb{E}[\textbf{case}\ M\ \textbf{of}\ \{\ pat_i\ \rightarrow\ N_i\ \}] \quad \lhd\!\rhd \quad \textbf{case}\ M\ \textbf{of}\ \{\ pat_i\ \rightarrow\ \mathbb{E}[N_i]\ \}$$

In this instance, we would like to use this rule to move $(+\!\!+\ [])$ inside the **case** statement. Knowing that the system can generate evaluation contexts on our behalf, we can attempt to apply the rule without specifying a parameter:

```
[16]> case-eval
Error: no valid evaluation contexts.
```

However, an error results because the context $[-]\ +\!\!+\ []$ we wish to use is not strictly speaking an evaluation context, but only cost equivalent to a context of this form. The equivalence is demonstrated in (Hackett and Hutton 2014). By default, only contexts with valid syntactic forms are accepted by the system, meaning that even if we manually specified the desired context as a parameter to `case-eval`, the rule would still be rejected.

The solution is to add $[-]\ +\!\!+\ []$ to Unie's library of cost-equivalent evaluation contexts, which instructs the system to treat it as if it were an evaluation context. This can be achieved using `add-lib`. In fact, the proof in (Hackett and Hutton 2014) is more general and shows that $[-]\ +\!\!+\ xs$ is cost equivalent to an evaluation context for any list xs. This result can be captured using the constructor pattern *LIST* that matches with any list:

```
[16]> add-lib EVAL $[-] ++ LIST$
Info: library updated.
```

Cost-equivalent contexts are made available to the system's context generation and matching mechanisms (see section 4.4.6), meaning that when we apply `case-eval` again without a parameter, the correct context is used automatically:

```
[16]> case-eval
⋪⋫ case xs of
    []       → [] ++ []
    (y : ys) → ((f ys) ++ [y]) ++ []
```

Notice that, in this case, the context pattern $[-] +\!\!+ LIST$ is instantiated to $[-] +\!\!+ []$ in order to apply the transformation rule correctly.

We have almost completed the proof. All that is left to do is evaluate the applications of append that have resulted from $(+\!\!+ [])$ being moved inside both **case** alternatives. In the second **case** alternative, we wish to evaluate $[y] +\!\!+ []$. In order to do so, we must first reassociate the term using the fact that append is associative up to weak cost equivalence.

```
[17]> right ; rhs
 = [] ++ []
[19]> eval-wce
≪≫ []
[20]> up ; next ; rhs
 = ((f ys) ++ [y]) ++ []
[23]> append-assoc-lr-wce
≪≫ (f ys) ++ ([y] ++ [])
[24]> right ; eval-wce
≪≫ [y]
[26]> top
 = λxs.case xs of
     []       → []
     (y : ys) → (f ys) ++ [y]
```

Finally, after folding the definition of `Revbody`, Unie notifies us that we have reached our transformation goal and hence the proof of the property is complete:

```
[27]> fold-def 'Revbody
Info: transformation goal reached!
 = V_Revbody[f]
```

In conclusion, the above calculation demonstrates how improvement proofs can be constructed mechnically using our system. By following the same pattern of reasoning as in the original paper proof, with the addition of navigation steps to make the point of application of each rule clear, we were able to mechanise the calculation by simply entering the transformation rules as commands into the system. Behind the scenes, the technicalities of each proof step were administered automatically on our behalf to ensure the resulting proof is correct. Moreover, by entering commands without parameters, we allowed the system to

simplify the development of proof steps by automatically generating the necessary contexts.

4.7 Discussion

Several tools have been developed to mechanise *equational* reasoning about Haskell programs (Tullsen 2002; Guttmann et al. 2003; Li, Reinke, and Thompson 2003; Thompson and Li 2013). Most relevant to our work is the Hermit system (Farmer 2015), which builds upon the Haskell Equational Reasoning Assistant (Hera) (Gill 2006). There appears to be no other systems in the literature that directly support *inequational* reasoning about Haskell programs. To the best of our knowledge, our system is the first to provide mechanical support for improving pure Haskell programs.

In the wider literature, the Algebra of Programming in Agda library (Mu, Ko, and Jansson 2009) is designed to encode relational program derivations, which supports a form of inequational reasoning. The Jape proof calculator (Bornat and Sufrin 1997; Bornat and Sufrin 1999) provides step-by-step interactive development of proofs in formal logics, and supports both equational and inequational reasoning. Improvement theory has not been explored within either of these settings, however. More generally, automated theorem provers (Bertot and Castéran 2013; Norell 2007) can be used to provide machine-checked proofs of program properties, but require expertise in dependently typed programming.

Other methods for reasoning about runtime in a lazy setting include (Wadler 1988; Bjerner and Holmström 1989; Madhavan, Kulal, and Kuncak 2017). Most notably, Okasaki (1999) used a notion of *time credits* to analyse the amortized performance of a range of purely functional data structures. This approach has recently been implemented in Agda by Danielsson (2008) and heavily inspired our work on formalising efficiency proofs in Liquid Haskell (Vazou 2016), which is the focus of the next chapter.

Much research has been conducted on type-based methods for statically predicting execution cost, for example, (Hughes, Pareto, and Sabry 1996; Crary and Weirich 2000; Hofmann and Jost 2003; Vasconcelos and Hammond 2003; Brady and Hammond 2005; McCarthy et al. 2017; Wang, Wang, and Chlipala 2017; Çiçek et al. 2017). Given that improvement theory is untyped, comparisons with these works are somewhat difficult. Nonetheless,

from a high-level, a number of these articles present type-and-effect systems for automatically inferring the execution cost of individual programs, which is known as *unary cost analysis*. In contrast, improvement theory is a semantic approach to *relational cost analysis*, which is used to compare the relative performance of two programs.

Furthermore, most work on static cost analysis falls into one of two camps: the first aims at full automation while the second aims at expressiveness. In the former case, restrictions are often placed on the corresponding operational model (or on the analysis itself) to ensure full automation, for example, terms are monomorphic and/or first-order only. In comparison, improvement theory's operational model is much more expressive. In the latter case, precise analysis relies on input from the user, typically in the form of *cost annotations* added to the source syntax, which are used to guide the cost inference engine. As we have seen, the Unie system provides mechanical support for improvement proofs, but such proofs are fundamentally user-directed. Investigating the addition of Coq-style proof tactics, or, more generally, 'proof search', to Unie is part of our future work.

Finally, frameworks for static cost analysis do not in general incorporate a call-by-need semantics. One notable exception is (Jost et al. 2017).

Before discussing our ideas for further work, we that feel it is important to highlight an implementation detail of the Unie system that may be subsequently readdressed in light of recent advancements in the literature. With the rise of datatype-generic programming (Gibbons 2006), a large number of libraries have recently been designed to aid the task of automatically generating traversals of, for example, syntax trees by analysing the generic structure of datatypes. For instance, Kiss, Pickering, and Wu (2018) combine the notions of lenses and traversals to give a flexible approach for querying and modifying complex data structures. As such, we believe that the Kure library could now be replaced by a more recent generic library, which is likely to be more efficient.

We have three main avenues for further work. First of all, we would like to investigate higher level support for navigating through terms and applying transformation to specific locations during improvement proofs. Such support is provided by the Hermit system (Farmer 2015), which also uses the Kure library. Nonetheless, we look to the *Generic-Lens* package (Kiss 2017) for guidance on this, which was created and is currently maintained by

the first author of (Kiss, Pickering, and Wu 2018). Hence, this work could be naturally undertaken while replacing the Kure library with a generic alternative.

Secondly, we wish to incorporate Coq-style proof tactics into Unie to capture and express common recipes for improvement, such as the one discussed in section 4.2. In this instance, we are encouraged by the fact that—on many occasions—rules from the tick algebra can be correctly applied to a term in at most one way. As such, a 'proof search' of some kind may be feasible, but does require further investigation.

Finally, we would like Unie to produce proof objects that can be independently checked using an external proof assistant such as Coq (Bertot and Castéran 2013) or Agda (Norell 2008), to provide a formal guarantee of their correctness. This may require the Unie system to interface with, for example, the AoPA library (Mu, Ko, and Jansson 2009).

4.8 Conclusion

In this chapter, we presented the design of an inequational reasoning assistant called Unie, which provides mechanical support for proofs of program improvement. In doing so, we highlighted a number of difficulties in manually constructing improvement proofs, and described how our system has been developed to address these challenges. We have illustrated the applicability of our system by verifying a range of improvement results from the literature. In particular, we have mechanised all proofs in (Hackett and Hutton 2014), including the proof of the worker/wrapper improvement theorem, which relates to a general-purpose optimisation technique. We have also mechanically verified a number of proofs in (Moran and Sands 1999). All of these proofs are freely available online as scripts that can be loaded into our system, along with the system itself (Handley 2018).

In section 3.2 of the previous chapter, performance results from the AutoBench system indicated that the linear-time reverse function, *fastRev*, optimises the naive reverse function, *slowRev*, in 95% of test cases. In particular, AutoBench output the following optimisation:

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

In section 4.6 of this chapter, we demonstrated how the Unie system can be used to

improve *slowRev*, by interactively developing a calculation that resulted in the definition of *fastRev*, that is, *slowRev* $\gtrapprox$ *fastRev*. Notice that this relation is a weak improvement only: $\gtrapprox$. Interestingly, the reason for this is that *fastRev* is *less* efficient than *slowRev* (by a constant factor) when the input to both functions is the empty list. For all other lists, *fastRev* is the more efficient of both functions, however, *fastRev*'s 'start-up costs' in the empty base case mean that *slowRev* $\gtrsim$ *fastRev* does *not* hold.

Previously we stated that AutoBench's optimisation results aim to predict operational improvements à la improvement theory. In this instance, the system's results show that *fastRev* does not improve *slowRev* in the case of the empty list (see section 3.2.1). Thus, based on these results, we may have anticipated that *slowRev* $\gtrsim$ *fastRev* would not hold.

Chapter 5

Liquidate Your Assets

Reasoning about Resource Usage in Liquid Haskel

Liquid Haskell is an extension to the type system of Haskell that supports formal reasoning about program correctness by encoding logical properties as refinement types. In this chapter, we show how Liquid Haskell can also be used to reason about program efficiency in the same setting, by introducing a library, RTick, for formal cost analysis. We use Liquid Haskell's existing verification machinery to ensure that the results of our cost analysis are valid, together with custom invariants for particular program contexts to ensure that the results are precise. To illustrate our approach, we analyse the efficiency of a wide range of popular data structures and algorithms, and in doing so, explore various notions of resource usage. Our experience is that reasoning about efficiency in Liquid Haskell is often just as simple as reasoning about correctness, and that the two can naturally be combined.

5.1 Introduction

Up until this point, we have presented two different approaches to reasoning about the efficiency of pure Haskell programs. Chapter 3 introduced the AutoBench system, which combines two well-established systems, namely QuickCheck and Criterion, to give a lightweight, fully automated means of comparing time performance. In turn, chapter 4 introduced Unie, a reasoning assistant that provides mechanical support for proofs of program improvement, based on Moran and Sands' improvement theory.

An advantage of the first approach is that it is entirely Haskell-based. In other words, AutoBench operates directly on Haskell source code and its results extrapolate measurements taken from Haskell's runtime environment. On the other hand, the style of reasoning employed by this approach is somewhat informal, as the system's performance results are based entirely on empirical analyses, which provide no formal guarantees.

At the end of the previous chapter, we observed a direct correspondence between AutoBench's empirical analysis and Unie's semi-formal analysis. In this manner, Unie addresses a primary drawback of the AutoBench system, by aiding the construction of proofs of program improvement that are 'universal' in the sense that they account for evaluation in all contexts. Despite being implemented in Haskell, however, Unie operates on a variant of GHC's Core language. Therefore, although the system's operational model is comparable to Haskell's intermediate representation, it is distinct from the language that ordinary Haskell programmers frequently use in practice.

One way to combine the advantages of both approaches is to devise a means to reason about the efficiency of Haskell programs within the language itself. Moreover, to provide formal guarantees, such reasoning should incorporate efficiency proofs rather than extrapolating experimental results. To achieve this, we utilise the Liquid Haskell system (Vazou 2016) to *statically* analyse the resources required to execute pure Haskell programs.

Estimating the amount of resources that are required to execute a program is a key aspect of software development. Nonetheless, performance bugs are as difficult to detect as they are common (Jin et al. 2012). As a result, the problem of statically analysing the resource usage, or execution cost, of programs has been subject to much research, in which a broad range of techniques have been studied, including resource-aware type systems (Hofmann and Jost 2003; Hoffmann, Aehlig, and Hofmann 2012; Jost et al. 2017; Wang, Wang, and Chlipala 2017; Çiçek et al. 2017), program and separation logics (Aspinall et al. 2007; Atkey 2010), and sized types (Vasconcelos 2008; Campbell 2009).

Another technique for statically analysing execution cost, inspired by the work in (Moran and Sands 1999) on improvement theory, is to reify resource usage into the definition of a program by means of a datatype that accumulates abstract computation 'steps'. Steps can either accumulate at the type level inside an index, or at the value-level inside an integer

field. Formal analysis at the type level has been successfully applied in Agda (Danielsson 2008) and, more recently, Coq (McCarthy et al. 2017), while recent work in (Radiček et al. 2018) developed the theoretical foundations of the value-level approach.

In this chapter, we take inspiration from (Radiček et al. 2018) and implement a monadic datatype to measure the abstract resource usage of pure Haskell programs. We then use Liquid Haskell's (Vazou 2016) refinement type system to statically prove bounds on resource usage. Our framework supports the standard approach to cost analysis, which is known as *unary cost analysis* and aims to establish upper and lower bounds on the execution cost of a single program, together with the more recent *relational* approach (Çiçek et al. 2017), which aims to calculate the difference between the execution costs of two related programs or between one program on two different inputs.

Reasoning about execution cost using the Liquid Haskell system has two main advantages over most other formal cost analysis frameworks (Radiček et al. 2018). First of all, the system allows correctness properties to be naturally integrated into cost analysis, which helps to ensure that cost analyses are valid. And secondly, the wide range of sophisticated invariants that can be expressed and automatically verified by the system can be exploited to analyse resource usage in particular program contexts, which often leads to more precise and/or simpler analyses (Radiček et al. 2018).

By way of example, Liquid Haskell can automatically verify that Haskell's standard *sort* function returns an ordered list (of type $OList\ a$) with the same length as its input, even when the result is embedded in the *Tick* datatype that we use to measure resource usage:

$$\{-@\ sort\ ::\ Ord\ a\ \Rightarrow\ xs : [a]\ \rightarrow\ Tick\ \{\ zs : OList\ a\ \mid\ length\ zs\ =\ length\ xs\ \}\ @-\}$$

Applying our cost analysis to this function then allows us to prove that the maximum number of comparisons required to sort any list xs is $O(n \log n)$, where $n = length\ xs$:

$$\begin{aligned}&\{-@\ sortCost\ ::\ Ord\ a\ \Rightarrow\ xs : [a]\ \rightarrow\\ &\qquad\{\ tcost\ (sort\ xs)\ \leqslant\ 4\ *\ length\ xs\ *\ log_2\ (length\ xs)\ +\ length\ xs\ \}\ @-\}\end{aligned}$$

Moreover, we can also combine correctness and resource properties to show that the maximum number of comparisons is linear when the input list is already sorted:

$$\{-@\ sortCost_{Sorted}\ ::\ Ord\ a\ \Rightarrow\ xs : OList\ a\ \rightarrow\ \{\ tcost\ (sort\ xs)\ \leqslant\ length\ xs\ \}\ @-\}$$

The aim of the work in this chapter is thus to develop, prove correct, and evaluate a system that supports the above form of reasoning. As our system builds upon Liquid Haskell, our cost analysis is based on the language's core calculus (Vazou et al. 2014), which models a subset of Haskell's runtime semantics. In particular, our analysis of time and space usage does not account for compiler optimisations or garbage collection. How to interpret the results of our cost analysis in practice is discussed in section 5.2. In addition, a formal operational semantics for Liquid Haskell's core calculus is given in section 5.5.

Finally, our library is available on GitHub, along with the source code for all of the examples presented throughout this chapter (Handley and Vazou 2019).

5.2 Analysing resource usage

In this section, we exemplify our library's two main approaches to analysing resource usage, which are intrinsic and extrinsic and support both unary and relational cost analysis. In addition, each example serves to demonstrate how correctness properties can be naturally integrated into each method of cost analysis in our setting. We conclude this section by discussing how to interpret such analyses in practice.

5.2.1 Intrinsic cost analysis

In the case of *intrinsic* cost analysis, the resources utilised by a function are declared *inside* its refinement type signature and are *automatically* checked by Liquid Haskell.

Example 5.1: Time complexity

We start by analysing the number of recursive calls made by Haskell's list append function (++). First, we define a new operator, ($\underline{+\!+}$), that is similar to append, but lifted into our *Tick* datatype using applicative methods provided by our library:

$$
\begin{aligned}
&(\underline{+\!+}) \;::\; [a] \;\to\; [a] \;\to\; \mathit{Tick}\ [a] \\
&[\,] \qquad \underline{+\!+} \;\; \mathit{ys} \;=\; \mathit{pure}\ \mathit{ys} \\
&(x \;:\; \mathit{xs}) \;\; \underline{+\!+} \;\; \mathit{ys} \;=\; \mathit{pure}\ (x :) \;<\!\!*\!\!>\; (\mathit{xs} \;\; \underline{+\!+} \;\; \mathit{ys})
\end{aligned}
$$

That is, if the first argument list is empty, the second list, *ys*, is embedded into the *Tick* datatype using *pure*, which records zero cost. In turn, if the first list is non-empty, $(x :)$ is embedded using *pure* and applied to the result of the recursive call. To record the cost of the recursive call we use the $(<1>)$ operator, a variant of the applicative operator $(<*>)$ that sums the costs of the two arguments and increases the total by *one*.

Remark. The example code above includes the Haskell type signature for $(\underline{+\!+})$. For brevity, we omit such types in the remainder of the chapter in favour of the corresponding Liquid Haskell type specifications. However, Haskell type signatures are typically required when defining functions and proofs using our library, and are included in the online source code of all of our examples (Handley and Vazou 2019).

Now that we have defined the new operator, we can use Liquid Haskell to encode properties about append's execution cost by means of a refinement type specification:

$$\{-@\ (\underline{+\!+})\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow$$
$$\{\ t : Tick\ \{\ zs : [a]\ \mid\ length\ zs\ =\ length\ xs\ +\ length\ ys\ \}\ \mid\ tcost\ t\ =\ length\ xs\ \}\ @-\}$$

This type states that the length of the output list is given by the sum of the lengths of the two input lists: a correctness property; and that the cost of appending two lists, in terms of the number of required recursive calls, is given by the length of the first list: an efficiency property. Liquid Haskell is able to automatically verify both properties without any assistance from the user. In general, we note that resource bounds can be checked by the Liquid Haskell system but cannot be inferred.

Example 5.2: Memory allocation

Next, we analyse a different resource: the number of thunks allocated when folding lists. As before, we lift the standard *foldl* function into the *Tick* datatype. However, this time we use *step* to manually increment $\underline{foldl}$'s resource usage each time it allocates a thunk:

$$\underline{foldl}\ f\ z\ [\,]\ \ \ \ \ \ \ \ =\ pure\ z$$
$$\underline{foldl}\ f\ z\ (x\ :\ xs)\ =\ \textbf{let}\ w\ =\ f\ z\ x\ \textbf{in}\ 1\ `step`\ \underline{foldl}\ f\ w\ xs$$

As *foldl*'s resource usage increases for each element in the input list, we can use Liquid Haskell to automatically check that the cost of folding is equal to the length of this list:

$$\{-@\ \underline{foldl}\ ::\ (b\ \to\ a\ \to\ b)\ \to\ b\ \to\ xs : [a]\ \to\ \{\ t : Tick\ b\ \mid\ tcost\ t\ =\ length\ xs\ \}\ @-\}$$

In contrast, the strict variant of *foldl*, called *foldl′*, uses Haskell's *seq* primitive to force the evaluation of its intermediate results during execution:

$$\begin{aligned}
&\{-@\ \underline{foldl'}\ ::\ (b\ \to\ a\ \to\ b)\ \to\ b\ \to\ xs : [a]\ \to\ \{\ t : Tick\ b\ \mid\ tcost\ t\ =\ 0\ \}\ @-\}\\
&\underline{foldl'}\ f\ z\ [\,]\ \ \ \ \ \ \ \ =\ pure\ z\\
&\underline{foldl'}\ f\ z\ (x\ :\ xs)\ =\ \textbf{let}\ w\ =\ f\ z\ x\ \textbf{in}\ w\ `seq`\ \underline{foldl'}\ f\ w\ xs
\end{aligned}$$

As before, the *Tick* datatype is used to record the number of allocated thunks. As no thunks are allocated by *foldl′*, we do not increase the cost at each recursive step, and thus Liquid Haskell correctly verifies that *foldl′*'s total execution cost is zero.

Both of these examples are simplified models of execution, in the sense that they only account for the number of thunks allocated by the higher-order folding functions (*foldl* and *foldl′*) and assume that the input function f has no cost. In our subsequent case studies in section 5.4, we give a more accurate account of resource usage that incorporates the number of additional thunks allocated by each f.

Example 5.3: Cost analysis and verification

In this example, we analyse the number of comparisons made when merging two ordered lists. As before, we lift the standard *merge* function into the *Tick* datatype and use the (<1>) operator to increase the cost each time a comparison is made:

$$\begin{aligned}
&merge\ xs\ [\,]\ =\ pure\ xs\\
&merge\ [\,]\ ys\ =\ pure\ ys\\
&merge\ (x\ :\ xs)\ (y\ :\ ys)\\
&\quad \mid\ x\ \leqslant\ y\ \ \ \ \ \ =\ pure\ (x :)\ <1>\ merge\ xs\ (y\ :\ ys)\\
&\quad \mid\ otherwise\ \ \ =\ pure\ (y :)\ <1>\ merge\ (x\ :\ xs)\ ys
\end{aligned}$$

The resource usage of the *merge* function depends on the values of the input lists, and so we cannot easily establish a precise bound on its execution cost. We can, however, use Liquid Haskell to automatically check upper and lower bounds on this cost:

```
{−@ merge :: Ord a ⇒ xs : OList a → ys : OList a →
      { t : Tick { zs : OList a | length zs = length xs + length ys }
      |  tcost t ⩽ length xs + length ys
      ∧  tcost t ⩾ min (length xs) (length ys) }
      /  [length xs + length ys] @−}
```

In the worst case, *merge* performs *length xs* + *length ys* comparisons as both input lists may need to be completely traversed to produce an ordered output. Conversely, the best case requires *min* (*length xs*) (*length ys*) comparisons as *merge* terminates when one of the input lists becomes empty. The above type uses the ordered list type constructor, *OList*, which is defined using abstract refinements (Vazou, Rondon, and Jhala 2013) as follows:

```
{−@ type OList a = [a]<{ λx y → x ⩽ y }> @−}
```

Hence, the refinement type for *merge* also states that merging two ordered lists returns an ordered list whose length is equal to the sum of the two input lengths. Once again, we see that building our cost analysis on top of existing Liquid Haskell features allows us to naturally combine correctness and efficiency properties.

Finally, we note that *merge*'s specification includes the following *termination metric*

```
[length xs + length ys]
```

which enables Liquid Haskell to deduce that *merge* will terminate. All examples up to this point are automatically proved terminating by the system's structural termination checker. However, *merge*'s definition does not satisfy the preconditions for structural termination, and hence a semantic termination metric must be provided. We omit such termination metrics in the remainder of the chapter for brevity, however, they are included in the online source code of our examples (Handley and Vazou 2019).

5.2.2 Extrinsic cost analysis

In the case of *extrinsic* cost analysis, we use refinement types to express theorems about resource usage and then define Haskell terms that inhabit these types to prove the theorems. In contrast to intrinsic cost analysis, this approach does not support fully automated

verification, as proof terms must be provided by users. Nonetheless, this method allows us to specify efficiency properties that are not intrinsic to the definitions of functions. For example, we can relate the costs of multiple functions and analyse the resource usage of functions applied to specific subsets of their domains.

Example 5.4: Unary cost analysis

Using the *merge* function from the previous example, we can define a function that implements merge sort, with the following refinement type:

$$\{-@\ msort\ ::\ Ord\ a\ \Rightarrow\ xs : [a]\ \rightarrow\ Tick\ \{\ zs : OList\ a\ \mid\ length\ zs\ =\ length\ xs\ \}\ @-\}$$

This type captures two correctness properties of merge sort, namely that the output list is sorted and has the same length as the input list. To analyse the cost of *msort* we use the extrinsic approach. That is, we specify appropriate theorems outside of the function's definition and prove them manually. In particular, the following two theorems capture lower and upper bounds on merge sort's execution cost:

$$\begin{aligned}&\{-@\ msortCost_{LB}\ ::\ Ord\ a\ \Rightarrow\ \{\ xs : [a]\ \mid\ pow_2\ (length\ xs)\ \}\ \rightarrow\\ &\qquad\{\ tcost\ (msort\ xs)\ \geqslant\ (length\ xs\ \ /\ \ 2)\ *\ log_2\ (length\ xs)\ \}\ @-\}\end{aligned}$$

$$\begin{aligned}&\{-@\ msortCost_{UB}\ ::\ Ord\ a\ \Rightarrow\ \{\ xs : [a]\ \mid\ pow_2\ (length\ xs)\ \}\ \rightarrow\\ &\qquad\{\ tcost\ (msort\ xs)\ \leqslant\ 2\ *\ length\ xs\ *\ log_2\ (length\ xs)\ \}\ @-\}\end{aligned}$$

Together, the theorems state that the number of comparisons required is $\Theta(n \log n)$, where n is the length of the input list. In both cases, because merge sort proceeds by repeatedly splitting the input list into two equal parts, we assume the input length to be a power of two, specified by pow_2 (*length xs*). This approach highlights the flexibility of the extrinsic method: even though it is reasonable to use this assumption for cost analysis, it would be unreasonable to impose this restriction on all of the inputs to which *msort* is applied. Proofs of these theorems can be constructed using the proof combinators introduced in section 5.3 and are available online (Handley and Vazou 2019).

Note that if we assume the cost of comparison outweighs the cost of all other operations performed during merge sort's execution, we can use the above theorems to infer asymptotic

upper and lower bounds on the algorithm's runtime performance, respectively.

Example 5.5: Relational cost analysis

The extrinsic approach enables us to describe arbitrary program properties, including those that compare the relative cost of two functions or the same function applied to different inputs. This is known as relational cost analysis (Çiçek 2018). Here, we adapt an example from (Çiçek et al. 2017) to demonstrate how relational cost is encoded in our setting.

In cryptography, a program adheres to the 'constant-time discipline' if its execution time is independent of secret inputs. Adhering to this discipline is an effective countermeasure against side-channel attacks, which can allow intruders to infer secret inputs by measuring variations in execution time. Using relational cost analysis, we can prove that a program is constant-time without having to show that it has equal upper and lower bounds on its execution cost (for example, by performing two separate unary analyses). To demonstrate this, we use our library to analyse the execution cost of a function that compares two equal-length password hashes represented as lists of binary digits:

$$\{-@\ \ \textbf{type}\ \mathit{EqLen}\ \mathit{xs}\ =\ \{\ \mathit{ys} : [\mathit{Bit}]\ \mid\ \mathit{length}\ \mathit{ys}\ =\ \mathit{length}\ \mathit{xs}\ \}\ @-\}$$

$$\begin{aligned}
&\{-@\ \mathit{compare}\ ::\ \mathit{xs} : [\mathit{Bit}]\ \rightarrow\ \mathit{ys} : \mathit{EqLen}\ \mathit{xs}\ \rightarrow\ t : \mathit{Tick}\ \mathit{Bool}\ @-\}\\
&\mathit{compare}\ [\,]\ \ \ \ \ \ [\,]\ \ \ \ \ \ =\ \mathit{pure}\ \mathit{True}\\
&\mathit{compare}\ (x\ :\ \mathit{xs})\ (y\ :\ \mathit{ys})\ =\ \mathit{pure}\ (\wedge\ x\ ==\ y)\ \texttt{<1>}\ \mathit{compare}\ \mathit{xs}\ \mathit{ys}
\end{aligned}$$

We assume that the equality ($==$) and conjunction ($\wedge$) functions are both constant-time, therefore, we only measure the number of recursive calls made during *compare*'s execution.

As we have assumed that the computations performed during each recursive step are constant-time, we can prove that *compare* is a constant-time function by showing that it requires the same number of recursive calls when comparing any stored password *pwd* against any two equal-length user inputs, u_1 and u_2:

$$\begin{aligned}
&\{-@\ \mathit{constant}\ ::\ \mathit{pwd} : [\mathit{Bit}]\ \rightarrow\ u_1 : \mathit{EqLen}\ \mathit{pwd}\ \rightarrow\ u_2 : \mathit{EqLen}\ \mathit{pwd}\ \rightarrow\\
&\qquad \{\ \mathit{tcost}\ (\mathit{compare}\ \mathit{pwd}\ u_1)\ =\ \mathit{tcost}\ (\mathit{compare}\ \mathit{pwd}\ u_2)\ \}\ @-\}\\
&\mathit{constant}\ [\,]\ \ \ \ \ \ _\ \ \ \ \ \ _\ \ \ \ \ \ =\ ()\\
&\mathit{constant}\ (_\ :\ \mathit{ps})\ (_\ :\ \mathit{us}_1)\ (_\ :\ \mathit{us}_2)\ =\ \mathit{constant}\ \mathit{ps}\ \mathit{us}_1\ \mathit{us}_2
\end{aligned}$$

The proof of this theorem proceeds by induction on the length of the input lists. Consequently, our proof has a trivial base case and an inductive case that recursively calls the inductive hypothesis. In this instance, Liquid Haskell can deduce the relative cost of both executions from the definition of *compare*. As such, the proof can be handled automatically by Liquid Haskell's proof by logical evaluation (PLE) tactic (Vazou et al. 2017).

Example 5.6: Cost improvement

As a final example, we outline how extrinsic cost analysis can be used to calculate the difference in the execution costs of two related programs. This is also a primary application of relational cost analysis. Consider the familiar monoid laws of the $(+\!\!+)$ operator:

$$
\begin{array}{rcll}
[\,] +\!\!+ ys & = & ys & \textit{left identity} \\
xs +\!\!+ [\,] & = & xs & \textit{right identity} \\
(xs +\!\!+ ys) +\!\!+ zs & = & xs +\!\!+ (ys +\!\!+ zs) & \textit{associativity}
\end{array}
$$

As we know from chapter 3, these properties can be automatically checked using the QuickCheck system. In fact, they can also be proved correct in Liquid Haskell via equational reasoning (Vazou et al. 2018). However, although the two sides of each property give the same results, each side does not necessarily require the same amount of resources (as demonstrated previously in chapter 4). This observation can be made precise by proving the following properties of the annotated append operator, $(\underline{+\!\!+})$:

$$
\begin{array}{rcl}
[\,] \;\underline{+\!\!+}\; ys & \Longleftrightarrow & \textit{pure}\ ys \\
xs \;\underline{+\!\!+}\; [\,] & \geq\!= \textit{length}\ xs \Longrightarrow & \textit{pure}\ xs \\
(xs \;\underline{+\!\!+}\; ys) \gg\!= (\underline{+\!\!+}\; zs) & \geq\!= \textit{length}\ xs \Longrightarrow & (xs \;\underline{+\!\!+}) =\!\ll (ys \;\underline{+\!\!+}\; zs)
\end{array}
$$

Recall from example 5.1 that the $(\underline{+\!\!+})$ operator records the number of recursive calls made during append's execution. Using this notion of cost, the first property above states that the left identity law is a *cost equivalence*. That is, $[\,] +\!\!+ ys$ and ys evaluate to the same result, and moreover, both require the same number of recursive calls to append. We make this precise by relating the annotated version of each side using the cost equivalence relation $\Longleftrightarrow$. Note that ys must be embedded in the *Tick* datatype using *pure* in order for the

property to be type-correct. On the other hand, the right identity and associativity laws are *cost improvements* in the left-to-right direction. That is, both sides of each property evaluate to the same result, but in each case the right-hand side requires fewer recursive calls to append. Again, we make this precise by relating the corresponding annotated definitions. Moreover, we make the cost difference explicit using quantified improvement, written $\succcurlyeq\!= n \Longrightarrow$ for a positive cost difference n, by showing that each right-hand side requires *length xs* fewer recursive calls than its respective left-hand side.

We return to the notions of cost equivalence, cost improvement, and quantified improvement in section 5.3, where we discuss our library's implementation and prove the second property as an example. Subsequently, in section 5.4, we use quantified improvement to construct a unified proof that shows the well-known map fusion technique preserves correctness and improves performance. In turn, we then use quantified improvement to systematically derive an optimised-by-construction reverse function from a high-level specification.

5.2.3 Interpreting cost analysis

Our library allows users to analyse a wide range of resources. Specifically, the *Tick* datatype can measure any kind of resource whose usage is additive, in the sense that the basic operation on costs is addition (and subtraction). Nonetheless, the correctness of a cost analysis relies on appropriate cost annotations being added to a program by the user. As such, it is the user's responsibility to ensure that such annotations correctly model the intended usage of a resource. In section 5.5, we use Liquid Haskell's metatheory to prove the correctness of refinement specifications with respect to annotations.

Assuming that an annotated program does correctly model the intended usage of a particular resource, then the question is: how can a user relate its (intrinsic or extrinsic) cost analysis back to the execution cost of its unannotated counterpart? In other words, what is the interpretation of an annotated expression's cost bound in practice?

Haskell's lazy evaluation

As illustrated in the previous examples, any bound established on the execution cost of an annotated function that manipulates standard Haskell datatypes is a *worst-case* approximation of actual resource usage. For example, consider the annotated append function $(\underline{+\!\!+})$ that measures the number of recursive calls made by $(+\!\!+)$. Then, $tcost\ ([a, b, c]\ \underline{+\!\!+}\ ys) = 3$ implies that the evaluation of $[a, b, c] +\!\!+ ys$ makes three recursive calls to $(+\!\!+)$. Three recursive calls to $(+\!\!+)$ corresponds to $[a, b, c] +\!\!+ ys$ being *fully evaluated*.

An intuitive way to describe our library's cost analysis in this instance is to use terminology from Okasaki (1999): the analysis assumes that functions applied to standard Haskell datatypes are *monolithic*. That is, once run, such functions are assumed to run until completion. This is not true in practice because Haskell's lazy evaluation strategy proactively halts computations to prevent functions from being unnecessarily fully applied.

Moreover, for efficiency, lazy evaluation allows computations to share intermediate results so that expressions are not re-evaluated when needed on multiple occasions. By default, however, annotated expressions do not model sharing. For example, the $\underline{square}$ function below records the resource usage of its input, $n :: Tick\ Int$, twice even though its unannotated counterpart, $square\ n = n * n$, only evaluates $n :: Int$ once:

$$\{-@\ \underline{square}\ ::\ n : Tick\ Int\ \rightarrow\ \{\ t : Tick\ Int\ \mid\ tcost\ t\ =\ 2\ *\ tcost\ n\ \}\ @-\}$$
$$\underline{square}\ n\ =\ pure\ (*)\ <*>\ n\ <*>\ n$$

Thus, overall, our library's default analysis assumes that computations are fully evaluated and overlooks sharing, leading to worst-case approximations of actual execution costs in practice, that is, under Haskell's lazy evaluation strategy.

Explicit laziness

Our library can be used to precisely analyse the execution costs of computations that are *explicitly* lazy. This is achieved by encoding non-strictness into the definitions of datatypes and utilising a 'manual sharing' function, similarly to the approach taken by (Danielsson 2008). We return to these ideas in a case study on insertion sort in section 5.4.

5.3 Implementation

In this section, we present the implementation of our library and discuss two soundness assumptions it makes. The library consists of two modules. The first, *RTick*, defines the *Tick* datatype and functions for modifying resource usage, for example, *pure* and (<1>) from section 5.2. The second, *ProofCombinators*, defines combinators to encode steps of (in)equational reasoning about the values and resource usage of annotated expressions.

5.3.1 Recording resource usage

Our library's principal datatype is *Tick*. For a given type a, *Tick* a consists of an integer, *tcost*, to track the resource usage of a value, *tval*, of type a:

```
{-@ data Tick a = Tick { tcost :: Int, tval :: a } @-}
```

The *Tick* datatype is a monad, with the following applicative and monad functions:

```
{-@ pure, return :: x : a -> { t : Tick a | tval t = x && tcost t = 0 } @-}
pure x   = Tick 0 x
return x = Tick 0 x

{-@ (<*>) :: t1 : Tick (a -> b) -> t2 : Tick a ->
      { t : Tick b | tval t = (tval t1) (tval t2) && tcost t = tcost t1 + tcost t2 } @-}
Tick m f <*> Tick n x = Tick (m + n) (f x)

{-@ (>>=) :: t1 : Tick a -> f : (a -> Tick b) -> { t : Tick b
      | tval t = tval (f (tval t1)) && tcost t = tcost t1 + tcost (f (tval t1)) } @-}
Tick m x >>= f = let Tick n y = f x in Tick (m + n) y
```

The functions *pure* and *return* embed expressions in the *Tick* datatype and record zero cost, while the (<*>) and (>>=) operators sum up the costs of subexpressions.

We have formalised the applicative and monad laws for the above definitions in Liquid Haskell; the proofs can be found online (Handley and Vazou 2019).

5.3.2 Modifying resource usage

The *RTick* module defines various functions to record and modify resource usage. Inspired by the work in (Danielsson 2008), we refer to these (and the applicative and monad functions above) as *annotations*. Next, we present the main annotation functions used throughout this chapter. The most basic way to record resource usage is by using *step*:

$$
\begin{aligned}
&\{-@\ step\ ::\ m : Int\ \rightarrow\ t_1 : Tick\ a\ \rightarrow \\
&\qquad \{\ t : Tick\ a\ \mid\ tval\ t\ =\ tval\ t_1\ \wedge\ tcost\ t\ =\ m\ +\ tcost\ t_1\ \}\ @-\} \\
&step\ m\ (Tick\ n\ x)\ =\ Tick\ (m\ +\ n)\ x
\end{aligned}
$$

A positive integer argument to *step* indicates the consumption of a resource, while a negative argument indicates the production of a resource.

Interestingly, Danielsson (2008) defines a (✓) operator to increment the resource usage of an annotated expression in the spirit of improvement theory (Moran and Sands 1999). Such an operator can be defined in our setting using *step*:

$$
\begin{aligned}
&\{-@\ (\checkmark)\ ::\ t_1 : Tick\ a\ \rightarrow \\
&\qquad \{\ t : Tick\ a\ \mid\ tval\ t\ =\ tval\ t_1\ \wedge\ tcost\ t\ =\ 1\ +\ tcost\ t_1\ \}\ @-\} \\
&(\checkmark)\ =\ step\ 1
\end{aligned}
$$

We often wish to sum the costs of subexpressions and modify the result. For this, we provide a number of resource combinators. One such combinator, ($<_1>$), was used in section 5.2 and is a variant of the apply operator, ($<*>$). Specifically, ($<_1>$) behaves as ($<*>$) at the value-level, but increases the total resource usage of its subexpressions by *one*:

$$
\begin{aligned}
&\{-@\ (<_1>)\ ::\ t_1 : Tick\ (a\ \rightarrow\ b)\ \rightarrow\ t_2 : Tick\ a\ \rightarrow\ \{\ t : Tick\ b \\
&\qquad \mid\ tval\ t\ =\ (tval\ t_1)\ (tval\ t_2)\ \wedge\ tcost\ t\ =\ 1\ +\ tcost\ t_1\ +\ tcost\ t_2\ \}\ @-\} \\
&Tick\ m\ f\ <_1>\ Tick\ n\ x\ =\ Tick\ (1\ +\ m\ +\ n)\ (f\ x)
\end{aligned}
$$

A similar combinator is defined in relation to the bind function:

$$
\begin{aligned}
&\{-@\ (>_1=)\ ::\ t_1 : Tick\ a\ \rightarrow\ f : (a\ \rightarrow\ Tick\ b)\ \rightarrow\ \{\ t : Tick\ b \\
&\qquad \mid\ tval\ t\ =\ tval\ (f\ (tval\ t_1))\ \wedge\ tcost\ t\ =\ 1\ +\ tcost\ t_1\ +\ tcost\ (f\ (tval\ t_1))\ \}\ @-\} \\
&Tick\ m\ x\ >_1=\ f\ =\ \textbf{let}\ Tick\ n\ y\ =\ f\ x\ \textbf{in}\ Tick\ (1\ +\ m\ +\ n)\ y
\end{aligned}
$$

Finally, our library provides functions to embed computations in the *Tick* datatype while simultaneously consuming or producing resources. For example, inspired by Danielsson (2008), we have *wait* and *waitN*, which act in the same manner as *pure* and *return* at the value-level but consume one and n resources, respectively:

```
{-@ wait :: x : a -> { t : Tick a | tval t = x && tcost t = 1 } @-}
wait x = Tick 1 x
{-@ waitN :: n : Nat -> x : a -> { t : Tick a | tval t = x && tcost t = n } @-}
waitN n x = Tick n x
```

Similarly, *go* and *goN* produce one and n resources, respectively:

```
{-@ go :: x : a -> { t : Tick a | tval t = x && tcost t = (-1) } @-}
go x = Tick (-1) x
{-@ goN :: n : Nat -> x : a -> { t : Tick a | tval t = x && tcost t = (-n) } @-}
goN n x = Tick (-n) x
```

From the definitions of ($>_1=$), *step*, and ($\gg=$) above it should be clear that the following equality holds: $(t >_1= f) = step\ 1\ (t \gg= f)$. In fact, all of the resource modification functions provided by the *RTick* module, including ($<*>$) and ($<_1>$), can be defined using *return*, ($\gg=$), and *step*. We make use of this fact in section 5.5 to simplify our proofs of correctness for both intrinsic and extrinsic cost analysis.

Remark. It is important to note that *Tick*'s cost parameter should *not* be modified by any means other than through the use of the functions provided by the *RTick* module, for example, by case analysis. Doing so breaks the encapsulation of *Tick*'s effects, potentially leading to invalid cost analyses. This is discussed in detail at the end of this section.

5.3.3 Proving extrinsic theorems

As exemplified in section 5.2, extrinsic cost analysis requires manually proving that bounds on resource usage hold. In Liquid Haskell, this is formalised as (in)equational reasoning. In other words, a proof of an extrinsic theorem is a total and terminating Haskell function that appropriately relates the left-hand side of the theorem's proof statement to the right-hand

```
Equal          {−@ (==.) :: x : a → { y : a | y = x }
                   → { z : a | z = x ∧ z = y } @−}
               _ ==. y = y

Greater than   {−@ (⩾.) :: m : a → { n : a | m ⩾ n }
or equal           → { o : a | m ⩾ o ∧ o = n } @−}
               _ ⩾. n = n

Theorem        (?) :: a → Proof → a
invocation     x ? _ = x

Proof          (***) :: a → QED → Proof
finalisation   _ *** QED = ()

QED definition data QED = QED
```

Figure 5.1: Proof combinators introduced in (Vazou et al. 2018)

side by, for example, unfolding and folding definitions (Burstall and Darlington 1977) and through the use of mathematical induction (Burstall 1969).

Next, we introduce a number of proof combinators from our library's *ProofCombinators* module that aid the development of extrinsic proofs. As a running example, we show that append's right identity law, $xs +\!\!+ [\,] = xs$, is an optimisation in the left-to-right direction by proving properties about the annotated append function, $(\underline{+\!\!+})$, from section 5.2.

Proof construction

We first recall how to construct (in)equational proofs in Liquid Haskell. To exemplify both the equational and inequational styles of proof, we reason about the results and resource usage of append separately. Readers may refer to section 2.4.1 or to (Vazou et al. 2018) for a more detailed discussion on the following concepts.

Specifying theorems

The *Proof* type is simply the unit type, refined to express a theorem: **type** *Proof* = (). For example, in order to show that append's right identity law is a denotational equivalence, we can express that the values of $xs \ \underline{+\!\!+} \ [\,]$ and *pure xs* are equal:

$$\{-@\ \mathit{rightId}_{\mathit{Val}} :: xs : [a] \rightarrow \{\ p : \mathit{Proof} \mid \mathit{tval}\ (xs \ \underline{+\!\!+} \ [\,]) = \mathit{tval}\ (\mathit{pure}\ xs)\ \}\ @-\}$$

Here the binder $p : \mathit{Proof}$ is superfluous and so we can remove it:

$$\{-@\ \mathit{rightId}_{Val}\ ::\ xs : [a]\ \rightarrow\ \{\ \mathit{tval}\ (xs\ \ \underline{+\!\!+}\ \ [])\ =\ \mathit{tval}\ (\mathit{pure}\ xs)\ \}\ @-\}$$

Equational proofs

The above extrinsic theorem, $\mathit{rightId}_{Val}$, expresses a *value equivalence* between two annotated expressions. In this case, Liquid Haskell cannot prove the theorem automatically on our behalf. To prove it ourselves, we can define one of its inhabitants using a number of proof combinators from figure 5.1, as follows:

$$\begin{array}{ll}
\mathit{rightId}_{Val}\ [] & \mathit{rightId}_{Val}\ (x\ :\ xs) \\
\quad =\ \mathit{tval}\ ([]\ \ \underline{+\!\!+}\ \ []) & \quad =\ \mathit{tval}\ ((x\ :\ xs)\ \ \underline{+\!\!+}\ \ []) \\
\quad ==.\ \mathit{tval}\ (\mathit{pure}\ []) & \quad ==.\ \mathit{tval}\ (\mathit{pure}\ (x :)\ <_1>\ (xs\ \ \underline{+\!\!+}\ \ [])) \\
\quad ***\ \mathit{QED} & \quad ?\ \mathit{rightId}_{Val}\ xs \\
 & \quad ==.\ \mathit{tval}\ (\mathit{pure}\ (x :)\ <_1>\ \mathit{pure}\ xs) \\
 & \quad ==.\ \mathit{tval}\ (\mathit{Tick}\ 0\ (x :)\ <_1>\ \mathit{Tick}\ 0\ xs) \\
 & \quad ==.\ \mathit{tval}\ (\mathit{Tick}\ 1\ (x\ :\ xs)) \\
 & \quad ==.\ \mathit{tval}\ (\mathit{Tick}\ 0\ (x\ :\ xs)) \\
 & \quad ==.\ \mathit{tval}\ (\mathit{pure}\ (x\ :\ xs)) \\
 & \quad ***\ \mathit{QED}
\end{array}$$

Recall that the aim of the proof is to equate the left-hand side of the theorem's proof statement, $\mathit{tval}\ (xs\ \ \underline{+\!\!+}\ \ [])$, with the right-hand side, $\mathit{tval}\ (\mathit{pure}\ xs)$. We split it into two cases. In the base case, where xs is empty, the proof simply unfolds the definition of $(\underline{+\!\!+})$. In the inductive case, where xs is non-empty, the proof unfolds $(\underline{+\!\!+})$ and $(<_1>)$, and unfolds and folds *pure*. It also appeals to the inductive hypothesis using (?), which combines the refinements from its argument theorem with those from the current theorem being proved. In both cases, the $(==.)$ combinator relates steps of reasoning by ensuring that both of its arguments are equal and returns its second argument to allow multiple uses to be chained together. The $(***\ \mathit{QED})$ function signifies the end of each proof.

Inequational proofs

Having proved that the values of $xs \mathbin{+\!\!+} [\,]$ and *pure xs* are equal, the next step is to compare their resource usage. From section 5.2, we know that the costs of both expressions are not equal. In particular, $xs \mathbin{+\!\!+} [\,]$ requires *length xs* more recursive calls to append than *pure xs*. This can be formalised by proving that the execution cost of $xs \mathbin{+\!\!+} [\,]$ is greater than or equal to that of *pure xs* using the $(\geqslant.)$ combinator from figure 5.1:

$$\{-@\ \mathit{rightId}_{\mathit{Cost}}\ ::\ xs : [a]\ \rightarrow\ \{\ \mathit{tcost}\ (xs \mathbin{+\!\!+} [\,])\ \geqslant\ \mathit{tcost}\ (\mathit{pure}\ xs)\ \}\ @-\}$$

$$\begin{aligned}
&\mathit{rightId}_{\mathit{Cost}}\ xs \\
&\quad =\ \mathit{tcost}\ (xs \mathbin{+\!\!+} [\,]) \\
&\quad \geqslant.\ \mathit{tcost}\ (\mathit{pure}\ [\,]) \\
&\quad ***\ \mathit{QED}
\end{aligned}$$

The resource usage of *pure xs* is zero as it requires no recursive calls to $(+\!\!+)$. Furthermore, Liquid Haskell can automatically deduce that $\mathit{tcost}\ (xs \mathbin{+\!\!+} [\,]) = \mathit{length}\ xs$ and that $\mathit{length}\ xs \geqslant 0$. Hence the theorem follows from a single use of $(\geqslant.)$.

The *ProofCombinators* module includes numerous other numerical operators for reasoning about the relative execution costs of annotated expressions, including greater than $(>.)$, less than $(<.)$, less than or equal $(\leqslant.)$, and equal $(==.)$.

Proofs of cost equivalence, improvement, and diminishment

In this subsection, we return to the notions of cost equivalence, improvement, and quantified improvement illustrated in example 5.6. We also introduce two new notions: diminishment and quantified diminishment. It is important to note that these relations are distinct from those relations with the same name introduced in chapter 4. To clarify this, we give them (and their associated Liquid Haskell operators) new notations.

Cost equivalence

Often it is beneficial to reason about the values and resource usage of expressions simultaneously. For example, if we unfold the base case of the annotated append function,

($\underline{+\!+}$), it is easy to show that both expressions are equal:

$$[\,] \;\; \underline{+\!+} \;\; ys \;\; = \;\; pure \; ys$$

However, instead of relating the two expressions using equality, we prefer to use the notion of *cost equivalence*, which better clarifies our topic of reasoning. This cost-equivalence relation is defined as a Liquid Haskell predicate in figure 5.2, and states that two annotated expressions are cost equivalent if they have the same values and resource usage:

$$[\,] \;\; \underline{+\!+} \;\; ys \;\; <\!\!=\!\!> \;\; pure \; ys$$

The above property is thus a 'resource-aware' version of append's left identity law, which formalises that both expressions evaluate to the same result and require the same number of recursive calls to append (as shown previously in example 5.6).

Cost improvement

Previously in this section, we proved that append's right identity law is a value equivalence: $tval\ (xs \;\; \underline{+\!+} \;\; [\,]) = tval\ (pure\ xs)$ and a cost inequivalence: $tcost\ (xs \;\; \underline{+\!+} \;\; [\,]) \geqslant tcost\ (pure\ xs)$. Both of these properties are captured by the cost *improvement* relation defined in figure 5.2. Append's right identity law is thus an improvement—with respect to number of recursive calls—in the left-to-right direction. Following Moran and Sands (1999), we say that "$xs \;\; \underline{+\!+} \;\; [\,]$ is improved by $pure\ xs$".

One way to prove that append's right identity law is a left-to-right improvement is to combine both sets of refinements from $rightId_{Val}$ and $rightId_{Cost}$ using (?):

$$\{-@\ rightId_{Imp} \;::\; xs : [a] \;\rightarrow\; \{\ xs \;\; \underline{+\!+} \;\; [\,] \;>\!\!\simeq\!\!>\; pure\ xs\ \}\ @-\}$$
$$rightId_{Imp}\ xs \;=\; rightId_{Val}\ xs \;\; ?\;\; rightId_{Cost}\ xs$$

However, in general, this approach overlooks a key opportunity afforded by relational cost analysis, which is the ability to precisely relate intermediate execution steps using, for example, inequational reasoning. Crucially, unfolding and folding the definitions of annotated expressions makes resource usage explicit in steps of (in)equational reasoning. Not only does this allow savings in resource usage to be quantified in proofs, but it allows such sav-

ings to be localised. This approach fundamentally requires reasoning about the values and execution costs of annotated expressions simultaneously, however, and thus proofs relating values and costs independently simply cannot exploit it (Çiçek 2018).

Quantified improvement

It is straightforward to show that $xs \mathbin{+\!\!+} [\,]$ is improved by *pure xs* by relating the expressions' intermediate execution steps using cost combinators from figure 5.2. However, we know the exact cost difference between $xs \mathbin{+\!\!+} [\,]$ and *pure xs*, namely *length xs*. This additional information allows us to relate the expressions more precisely using the *quantified improvement* relation, also defined in figure 5.2. Quantified improvement extends cost improvement by taking an additional argument, which is the cost difference between its first and last arguments. Therefore, we can say "$xs \mathbin{+\!\!+} [\,]$ is improved by *pure xs*, by a cost of *length xs*", and make it precise by defining a corresponding theorem, as follows:

$$\{-@\ \mathit{rightId}_{QImp} :: xs : [a] \to \{\ xs \mathbin{+\!\!+} [\,] \succcurlyeq \mathit{length}\ xs \Longrightarrow \mathit{pure}\ xs\ \}\ @-\}$$

To prove this theorem, we can simply extend the previous proof of value equivalence, $\mathit{rightId}_{Val}$, by replacing equality with cost equivalence and by making cost savings wherever possible. Readers are encouraged to note the strong connection between $\mathit{rightId}_{Val}$ and the following proof, which, as before, is split into two cases:

$$\begin{aligned}
&\mathit{rightId}_{QImp}\ [\,] \\
&\quad = [\,] \mathbin{+\!\!+} [\,] \\
&\quad \Longleftrightarrow \mathit{pure}\ [\,] \\
&\quad *** \mathit{QED}
\end{aligned}$$

In the base case, where *xs* is empty, there is no cost saving. This is because $\mathit{length}\ [\,] = 0$ and, therefore, $\mathit{tcost}\ ([\,] \mathbin{+\!\!+} [\,]) = \mathit{tcost}\ (\mathit{pure}\ [\,])$. Hence, it suffices to show that $[\,] \mathbin{+\!\!+} [\,] \Longleftrightarrow \mathit{pure}\ [\,]$, which follows immediately from the definition of $(\mathbin{+\!\!+})$.

$$\begin{aligned}
&\mathit{rightId}_{QImp}\ (x : xs) \\
&\quad = (x : xs) \mathbin{+\!\!+} [\,] \\
&\quad \Longleftrightarrow \mathit{pure}\ (x :)\ <\!1\!>\ (xs \mathbin{+\!\!+} [\,])
\end{aligned}$$

Relations

$$\begin{array}{rl}
\textit{Value equivalence} & t_1 \;\; =!= \;\; t_2 \;\; = \;\; \textit{tval}\ t_1 \;=\; \textit{tval}\ t_2 \\
\textit{Cost equivalence} & t_1 \;\; <=> \;\; t_2 \;\; = \;\; t_1 \;=!=\; t_2 \;\wedge\; \textit{tcost}\ t_1 \;=\; \textit{tcost}\ t_2 \\
\textit{Improvement} & t_1 \;\; >\simeq> \;\; t_2 \;\; = \;\; t_1 \;=!=\; t_2 \;\wedge\; \textit{tcost}\ t_1 \;\geqslant\; \textit{tcost}\ t_2 \\
\textit{Diminishment} & t_1 \;\; <\simeq< \;\; t_2 \;\; = \;\; t_1 \;=!=\; t_2 \;\wedge\; \textit{tcost}\ t_1 \;\leqslant\; \textit{tcost}\ t_2 \\
\textit{Quantified improvement} & t_1 \;>\!\!=\!\!=\; n \;\Longrightarrow\; t_2 \;\; = \;\; t_1 \;=!=\; t_2 \;\wedge\; \textit{tcost}\ t_1 \;=\; n \;+\; \textit{tcost}\ t_2 \\
\textit{Quantified diminishment} & t_1 \;<\!\!=\!\!=\; n \;\Longrightarrow\; t_2 \;\; = \;\; t_1 \;=!=\; t_2 \;\wedge\; n \;+\; \textit{tcost}\ t_1 \;=\; \textit{tcost}\ t_2
\end{array}$$

Combinators

$$\begin{array}{rl}
\textit{Cost equivalence} & \{-@\ (<=>.)\ ::\ t_1 : \textit{Tick}\ a \;\to\; \{\ t_2 : \textit{Tick}\ a \mid t_1 <=> t_2\ \} \\
& \quad \to\ \{\ t : \textit{Tick}\ a \mid t_1 <=> t \wedge t_2 <=> t\ \}\ @-\} \\
\textit{Improvement} & \{-@\ (>\simeq>.)\ ::\ t_1 : \textit{Tick}\ a \;\to\; \{\ t_2 : \textit{Tick}\ a \mid t_1 >\simeq> t_2\ \} \\
& \quad \to\ \{\ t : \textit{Tick}\ a \mid t_1 >\simeq> t \wedge t_2 <=> t\ \}\ @-\} \\
\textit{Diminishment} & \{-@\ (<\simeq<.)\ ::\ t_1 : \textit{Tick}\ a \;\to\; \{\ t_2 : \textit{Tick}\ a \mid t_1 <\simeq< t_2\ \} \\
& \quad \to\ \{\ t : \textit{Tick}\ a \mid t_1 <\simeq< t \wedge t_2 <=> t\ \}\ @-\} \\
\textit{Quantified improvement} & \{-@\ (.>\!\!=\!\!=)\ ::\ t_1 : \textit{Tick}\ a \;\to\; n : \textit{Nat} \\
& \quad \to\ \{\ t_2 : \textit{Tick}\ a \mid t_1 >\!\!=\!\!=\; n \Longrightarrow t_2\ \} \\
& \quad \to\ \{\ t : \textit{Tick}\ a \mid t_1 >\!\!=\!\!=\; n \Longrightarrow t \wedge t_2 <=> t\ \}\ @-\} \\
\textit{Quantified diminishment} & \{-@\ (<\!\!=\!\!=)\ ::\ t_1 : \textit{Tick}\ a \;\to\; n : \textit{Nat} \\
& \quad \to\ \{\ t_2 : \textit{Tick}\ a \mid t_1 <\!\!=\!\!=\; n \Longrightarrow t_2\ \} \\
& \quad \to\ \{\ t : \textit{Tick}\ a \mid t_1 <\!\!=\!\!=\; n \Longrightarrow t \wedge t_2 <=> t\ \}\ @-\}
\end{array}$$

Separators

$$\begin{array}{rll}
\textit{Quantified improvement} & (\Longrightarrow.)\ ::\ (a \to b) \to a \to b & f \Longrightarrow. x = f\ x \\
\textit{Quantified diminishment} & (>\!\!=\!\!=.)\ ::\ (a \to b) \to a \to b & f >\!\!=\!\!=. x = f\ x
\end{array}$$

Figure 5.2: RTick cost relations, combinators, and separators

$$
\begin{aligned}
&? \quad \mathit{rightId}_{\mathit{QImp}}\ \mathit{xs} \\
&.\!>\!=\!=\ \mathit{length}\ \mathit{xs}\ =\!=\!>.\ \ \mathit{pure}\ (x :)\ <\!1\!>\ \mathit{pure}\ \mathit{xs} \\
&<\!=\!>.\ \ \mathit{Tick}\ 0\ (x :)\ <\!1\!>\ \mathit{Tick}\ 0\ \mathit{xs} \\
&<\!=\!>.\ \ \mathit{Tick}\ 1\ (x\ :\ \mathit{xs}) \\
&.\!>\!=\!=\ 1\ =\!=\!>.\ \ \mathit{Tick}\ 0\ (x\ :\ \mathit{xs}) \\
&<\!=\!>.\ \ \mathit{pure}\ (x\ :\ \mathit{xs}) \\
&*** \quad \mathit{QED}
\end{aligned}
$$

In the inductive case, where xs is non-empty, we must save $\mathit{length}\ (x : \mathit{xs})$ cost. We start by unfolding the definition of $(+\!\!+)$ and then replace $\mathit{xs}\ +\!\!+\ [\,]$ with $\mathit{pure}\ \mathit{xs}$ by appealing to the inductive hypothesis using (?), which saves $\mathit{length}\ \mathit{xs}$ resources. This saving is made explicit using the quantified improvement operator, $(.\!>\!=\!=\ \mathit{length}\ \mathit{xs}\ =\!=\!>.)$, which is a combination of two functions from figure 5.2, $(.\!>\!=\!=)$ and $(=\!=\!>.)$, whereby the latter is a syntactic sugar for Haskell's ($) operator, which allows $(.\!>\!=\!=)$ to be used infix. We save one further recursive call by unfolding the definition of $(<\!1\!>)$. Finally, our goal follows from the definition of *pure*. The total resource saving is $1 + \mathit{length}\ \mathit{xs}$, which is equal to $\mathit{length}\ (x : \mathit{xs})$.

By starting at the left-hand side of a resource-aware version of append's right identity law, we have used simple steps of inequational reasoning to derive the right-hand side. Each step of reasoning ensures denotational meaning (value equivalence) is preserved while simultaneously maintaining or improving resource usage. Resource usage is made explicit in steps of reasoning by cost annotations. Furthermore, the location and quantity of each resource saving is made explicit through the use of quantified improvement. We remind readers that Liquid Haskell verifies the correctness of every proof step.

In this particular instance, quantified improvement shows that one recursive call is saved per inductive step of the proof, and hence append's right identity law is a left-to-right optimisation precisely because $\mathit{xs} +\!\!+ [\,]$ evaluates to xs.

Diminishment and quantified diminishment

The combinators introduced up until now can only be used to prove that one expression, $t_1 :: \mathit{Tick}\ a$, is improved by another, $t_2 :: \mathit{Tick}\ a$, by starting at t_1 and deriving t_2. This

is because (quantified) improvement enforces a positive cost difference in the left-to-right direction. However, in some cases it may be easier to derive t_1 from t_2. To support this, we use the notion of (*quantified*) *diminishment*, also presented in figure 5.2, which is dual to (quantified) improvement. For example, it is straightforward to prove that *pure xs* is diminished by $xs \mathbin{+\!\!+} [\,]$, by a cost of *length xs*: simply reverse the calculation steps of $rightId_{QImp}$ replacing instances of quantified improvement with quantified diminishment.

Note that $t_1 >\!\simeq\!> t_2$ if and only if $t_2 <\!\simeq\!< t_1$, and likewise, $t_1 \succcurlyeq n \Longrightarrow t_2$ if and only if $t_2 \Longleftarrow n \preccurlyeq t_1$. Similar relationships also exist between other cost relations. We have formalised all such relationships in Liquid Haskell as part of our implementation.

Notions of improvement

Before discussing the key assumptions of our library, it is instructive to draw a number of comparisons between our new notions of improvement $>\!\simeq\!>$ and cost equivalence $<\!\Longrightarrow\!>$ and those from Moran and Sands' improvement theory, $\trianglerighteq$ and $\lhd\!\rhd$, respectively. Recall that the latter relations were introduced previously in section 4.3.2.

Firstly, note that the $>\!\simeq\!>$ and $<\!\Longrightarrow\!>$ relations are not contextually defined, unlike $\trianglerighteq$ and $\lhd\!\rhd$. As such, comparisons made between execution costs using our library only account for one context of use, which is the empty context, $[-]$. In practice, this means that $M >\!\simeq\!> N$ does not necessarily indicate that N performs better than M in all cases, for example, when N and M appear as subexpressions in larger program contexts.

Secondly, the relations $>\!\simeq\!>$ and $<\!\Longrightarrow\!>$ can be used to compare a wide range of different resources whereas $\trianglerighteq$ and $\lhd\!\rhd$ compare evaluation steps only.

Thirdly, our notions of quantified improvement and diminishment make the cost difference explicit, but there are no corresponding relations in improvement theory.

Finally, all of our improvement relations incorporate denotational meaning. For example, $M >\!\simeq\!> N$ also entails that M and N give the same results. However, this is not true of $M \trianglerighteq N$ because N may terminate more often than M. Hence, in practice, separate proofs of correctness often accompany proofs of improvement à la improvement theory (Hackett and Hutton 2014), while our approach attempts to unify such proofs. (In fact, a separate notion of *strong* improvement $\triangleright$ was used for correctness-preserving optimisations in Moran

and Sands' work. Readers may recall this notation from AutoBench's optimisation results in section 3.3.4 as the system attempts to empirically predict this relation.)

5.3.4 Library assumptions

To ensure its cost analysis is sound, the library makes two key assumptions:

(a) Expressions subject to cost analysis are not defined in terms of *tval* or *tcost*, nor do they perform case analysis on the *Tick* data constructor;

(b) Liquid Haskell's totality and termination checkers are active at all times.

These assumptions are discussed for the remainder of this section.

Projections and case analysis

Annotated expressions subject to resource analysis must not be defined in terms of *Tick*'s projection functions *tval* and *tcost*. This is to preserve the encapsulation of *Tick*'s accumulated cost. For example, *tval* can be used to (indirectly) show that two lists can be appended using $(\underline{+\!+})$ without incurring any cost, as follows:

$$\{-@\ (\underline{+\!+}_0)\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow\ \{\ t : Tick\ [a]\ \mid\ tcost\ t\ =\ 0\ \}\ @-\}$$
$$xs\ \ \underline{+\!+}_0\ \ ys\ =\ pure\ (tval\ (xs\ \ \underline{+\!+}\ \ ys))$$

From the type specification of $(\underline{+\!+})$, we know that $tcost\ (xs\ \ \underline{+\!+}\ \ ys) = length\ xs$. However, the cost of $(\underline{+\!+}_0)$ is zero because it uses *tval* to discard $(\underline{+\!+})$'s incurred cost. Similarly, user-defined functions can freely overwrite accumulated costs using the *tcost* projection or by performing case analysis on *Tick*'s data constructor. Consequently, these primitives are not permitted in the definitions of expressions, as such expressions are not *safe* for cost analysis. (We formally define a safety predicate in section 5.5.) Instead, users should record resource usage *implicitly* using the functions provided by the *RTick* module.

Remark. Despite this assumption, the *tval* and *tcost* projections and the *Tick* data constructor are exported from the *RTick* module. This is because, as we've seen previously, these definitions are required in refinement type specifications and extrinsic proof terms.

Totality and termination

Partial definitions, which Haskell permits, are not valid inhabitants of theorems expressed in refinement types (Vazou et al. 2018). As such, the resource usage of partial definitions should not be analysed using the library. Similarly, partial definitions should not be used to prove theorems regarding the resource usage of total annotated expressions. Haskell can also be used to specify non-terminating computations. Divergence in refinement typing (in combination with lazy evaluation) can, however, be used to prove false predicates (Vazou et al. 2014). Hence, the library's cost analysis is only sound for computations that require *finite* resources. Liquid Haskell provides powerful totality and termination checkers that are active by default (Vazou 2016). Partial and/or divergent definitions will thus be rejected so long as these checkers remain active at all times.

5.4 Case studies

In this section, we present an evaluation of our library, encompassing four detailed case studies: cost analyses of monolithic and non-strict implementations of insertion sort (sections 5.4.1 and 5.4.2); a proof that the well-known map fusion technique is a correctness-preserving optimisation (section 5.4.3); and a derivation of an optimised-by-construction list-reversing function (section 5.4.4). The evaluation concludes with a summary of all of the examples we have studied while developing the library (section 5.4.5), the majority of which have been adapted from the existing literature for purposes of comparison.

5.4.1 Case study 1: Insertion sort

This case study analyses the number of comparisons required by the standard insertion sort algorithm. First, we use intrinsic cost analysis to prove a quadratic upper bound on the number of comparisons needed to sort a list of any configuration. In turn, we then use extrinsic cost analysis to prove a linear upper bound on the number of comparisons needed to sort a list that is already in a sorted order.

To begin, we lift the textbook insertion sort function into the *Tick* datatype:

```
insert x [] = pure [x]
insert x (y : ys)
  | x ⩽ y     = wait (x : y : ys)
  | otherwise = pure (y :) <1> insert x ys

isort []       = return []
isort (x : xs) = insert x =≪ isort xs
```

According to the definition of *isort*, an empty list is already sorted; the result is simply embedded in the *Tick* datatype using *return*. To sort a non-empty list, its head is inserted into its recursively sorted tail. In this case, the 'flipped' bind operator, (=≪), sums up the costs of the insertion and the recursive sorting.

Inserting an element into a sorted list is standard, with each comparison being recorded using the functions *wait* and (<1>) introduced previously in section 5.3.

Intrinsic cost analysis

Refinement types can now be used to simultaneously specify properties about the correctness and resource usage of the above functions. In particular, abstract refinement types (Vazou, Rondon, and Jhala 2013) can be used to define sorted Haskell lists, that is, a list whereby the head of each sublist is less than or equal to every element in the tail:

```
{−@ type OList a = [a]<{ λx y → x ⩽ y }> @−}
```

Here we use the *OList* type constructor to ensure that *insert*'s input list, *xs*, is sorted:

```
{−@ insert :: Ord a ⇒ x : a → xs : OList a → { t : Tick { zs : OList a
      | length zs = 1 + length xs } | tcost t ⩽ length xs } @−}
```

The result type of *insert* asserts that the function's output list, *zs*, is also sorted and contains one more element than *xs*. With respect to efficiency, this type states that an insertion requires at most *length xs* comparisons.

The specification for *isort* states that it returns a sorted list of the same length as its input, *xs*, and that sorting *xs* requires at most $(length\ xs)^2$ comparisons:

$$\{-@\ \mathit{isort}\ ::\ \mathit{Ord}\ a\ \Rightarrow\ xs : [a]\ \rightarrow\ \{\ t : \mathit{Tick}\ \{\ zs : \mathit{OList}\ a \mid \mathit{length}\ zs\ =\ \mathit{length}\ xs\ \}\ \mid\ \mathit{tcost}\ t\ \leqslant\ (\mathit{length}\ xs)^2\ \}\ @-\}$$

Liquid Haskell is able to automatically verify *insert*'s specification. On the other hand, *isort*'s specification is rejected. This is because the cost of $\mathit{insert}\ x =\!\!\ll \mathit{isort}\ xs$ can only be calculated by performing type-level computations that are not automated by the system. At this point, we could switch to extrinsic cost analysis and perform the necessary calculations manually. However, we can also take a different approach that allows us to continue on with our intrinsic analysis. The key to this approach is utilising the following function, which is a variant of the flipped bind operator, $(=\!\!\ll)$:

$$\{-@\ (=\!\!\ll\!\{\cdot\})\ ::\ n : \mathit{Nat}\ \rightarrow\ f : (a\ \rightarrow\ \{\ t_1 : \mathit{Tick}\ b\ \mid\ \mathit{tcost}\ t_1\ \leqslant\ n\ \})\ \rightarrow\ t_2 : \mathit{Tick}\ a\ \rightarrow\ \{\ t : \mathit{Tick}\ b\ \mid\ \mathit{tcost}\ t\ \leqslant\ \mathit{tcost}\ t_2\ +\ n\ \}\ @-\}$$

$$(=\!\!\ll\!\{\cdot\})\ ::\ \mathit{Int}\ \rightarrow\ (a\ \rightarrow\ \mathit{Tick}\ b)\ \rightarrow\ \mathit{Tick}\ a\ \rightarrow\ \mathit{Tick}\ b$$

$$f\ =\!\!\ll\!\{n\}\ x\ =\ f\ =\!\!\ll\ x$$

From an operational point of view, it should be clear that the expression $f\ =\!\!\ll\!\{n\}\ x$ is equal to $f =\!\!\ll x$. However, the refinement type of this 'bounded' version of $(=\!\!\ll)$ restricts its domain to functions $f :: a \rightarrow \mathit{Tick}\ b$ with execution costs no greater than n. The total execution cost of $f\ =\!\!\ll\!\{n\}\ x$ thus cannot exceed that of x plus n.

Using $(=\!\!\ll\!\{\cdot\})$ in the definition of *isort* allows Liquid Haskell to verify its execution cost without performing any type-level computations. Hence, *isort*'s type can be automatically verified by specifying a *length xs* upper bound on the cost of each insertion:

$$\begin{aligned}\mathit{isort}\ [\,] &= \mathit{return}\ [\,]\\ \mathit{isort}\ (x : xs) &= \mathit{insert}\ x\ =\!\!\ll\!\{\mathit{length}\ xs\}\ \ \mathit{isort}\ xs\end{aligned}$$

A more accurate bound

A simple recurrence relation for insertion sort is as follows

$$T(n)\ = \begin{cases} 0 & \text{if } n = 0 \\ T(n-1) + O(n) & \text{if } n > 0 \end{cases}$$

where $n = \mathit{length}\ xs$ for an input list xs. In the base case, where $\mathit{length}\ xs = 0$, no comparisons are made. In the recursive case, where $\mathit{length}\ xs > 0$, we know from the above specification for *insert* that each insertion requires at most $\mathit{length}\ xs - 1$ comparisons. Furthermore, $\mathit{isort}\ (x : xs)$ is defined recursively in terms of $\mathit{isort}\ xs$. The solution to this recurrence relation is the sum of integers between 1 and $n - 1$, which is equal to $\frac{n(n-1)}{2}$. This result provides a more accurate upper bound on the execution cost of *isort*

$$\{-@\ \mathit{isort} :: \mathit{Ord}\ a \Rightarrow xs : [a] \rightarrow \{\ t : \mathit{Tick}\ (\mathit{OList}\ a)\ |\ \mathit{tcost}\ t \leqslant (\mathit{length}\ xs * (\mathit{length}\ xs - 1))\ /\ 2\ \}\ @-\}$$

which is automatically verified by the system. Liquid Haskell is able to intrinsically prove this particular bound due to the correctness property $\mathit{length}\ zs = 1 + \mathit{length}\ xs$ appearing in *insert*'s refinement specification. Again, this demonstrates how building upon Liquid Haskell's existing verification machinery allows for more precise cost analysis.

Extrinsic cost analysis

Next, we prove that the maximum number of comparisons made by *isort* is linear when its input is already sorted. We capture this property with the following extrinsic theorem that takes a sorted list as input. Therefore, the definition of *isort* does not need to be modified.

$$\{-@\ \mathit{isortCost}_{\mathit{Sorted}} :: \mathit{Ord}\ a \Rightarrow xs : \mathit{OList}\ a \rightarrow \{\ \mathit{tcost}\ (\mathit{isort}\ xs) \leqslant \mathit{length}\ xs\ \}\ @-\}$$

To prove this theorem, we consider three different cases: when the input list is empty; when the list is a singleton, which invokes the base case of *insert*; and when the list has more than one element, which invokes the recursive case of *insert*. The first two cases follow immediately from the definitions of *isort* and *insert*, and thus can be proved automatically using Liquid Haskell's PLE feature (Vazou 2016):

$$\{-@\ \mathit{ple}\ \mathit{isortCost}_{\mathit{Sorted}}\ @-\}$$
$$\mathit{isortCost}_{\mathit{Sorted}}\ [\,] = ()$$
$$\mathit{isortCost}_{\mathit{Sorted}}\ [x] = ()$$

When the input list contains more than one element, we begin by unfolding the definitions of *isort* and $(=\!\!\ll\!\{\cdot\})$, and then appeal to the inductive hypothesis:

$$
\begin{aligned}
&\mathit{isortCost}_{\mathit{Sorted}}\ (x\ :\ (\mathit{xs}@(y\ :\ \mathit{ys})))\\
&\quad =\ \mathit{tcost}\ (\mathit{isort}\ (x\ :\ \mathit{xs}))\\
&\quad ==.\ \mathit{tcost}\ (\mathit{insert}\ x\ =\!\!\ll\!\{\mathit{length}\ \mathit{xs}\}\ \ \mathit{isort}\ \mathit{xs})\\
&\quad ==.\ \mathit{tcost}\ (\mathit{isort}\ \mathit{xs})\ +\ \mathit{tcost}\ (\mathit{insert}\ x\ (\mathit{tval}\ (\mathit{isort}\ \mathit{xs})))\\
&\quad ?\ \ \mathit{isortCost}_{\mathit{Sorted}}\ \mathit{xs}\\
&\quad \leqslant.\ \mathit{length}\ \mathit{xs}\ +\ \mathit{tcost}\ (\mathit{insert}\ x\ (\mathit{tval}\ (\mathit{isort}\ \mathit{xs})))
\end{aligned}
$$

At this point, we invoke a lemma that proves *tval* (*isort xs*) is an identity on *xs* when the list is sorted; its proof can be found online (Handley and Vazou 2019).

$$
\begin{aligned}
&\quad ?\ \ \mathit{isort}_{\mathit{SortedVal}}\ \mathit{xs}\\
&\quad ==.\ \mathit{length}\ \mathit{xs}\ +\ \mathit{tcost}\ (\mathit{insert}\ x\ \mathit{xs})\\
&\quad ==.\ \mathit{length}\ \mathit{xs}\ +\ \mathit{tcost}\ (\mathit{insert}\ x\ (y : \mathit{ys}))
\end{aligned}
$$

As the input $x : y : ys$ is sorted, we know that $x \leqslant y$. Consequently, *insert* x $(y : ys)$ will not recurse and unfolding the definitions of *insert* and *wait* completes the proof:

$$
\begin{aligned}
&\quad ==.\ \mathit{length}\ \mathit{xs}\ +\ \mathit{tcost}\ (\mathit{wait}\ (x\ :\ y\ :\ \mathit{ys}))\\
&\quad ==.\ \mathit{length}\ \mathit{xs}\ +\ 1\\
&\quad ==.\ \mathit{length}\ (x\ :\ \mathit{xs})\\
&\quad ***\ \mathit{QED}
\end{aligned}
$$

Overall, this case study exemplifies how our library can be used to establish precise bounds on the resource usage of functions operating on subsets of their domains. In this instance, we imposed a 'sortedness' constraint on *isort*'s input list using an extrinsic theorem, without needing to modify the function's original definition. Furthermore, the proof relies on the fact that *isort*'s result is a sorted list in order to show that *tval* (*isort xs*) is an identity on *xs*. Hence, once more, we have demonstrated how correctness properties can be utilised for the purposes of precise cost analysis.

Resource propagation

The execution cost of any annotated function utilising *isort* will, in general, be at least quadratic. For example, a minimum function defined by taking the head of a non-empty

list that is sorted using *isort* also has a quadratic upper bound:

$$\{-@\ \ \textbf{type}\ \mathit{NonEmpty}\ a\ =\ \{\ \mathit{xs} : [a]\ \mid\ \mathit{length}\ \mathit{xs}\ >\ 0\ \}\ @-\}$$

$$\begin{aligned}&\{-@\ \mathit{minimum}\ ::\ \mathit{Ord}\ a\ \Rightarrow\ \mathit{xs} : \mathit{NonEmpty}\ a\ \rightarrow\\ &\qquad \{\ t : \mathit{Tick}\ a\ \mid\ \mathit{tcost}\ t\ \leqslant\ (\mathit{length}\ \mathit{xs})^2\ \}\ @-\}\\ &\mathit{minimum}\ \mathit{xs}\ =\ \mathit{pure}\ \mathit{head}\ <\!\!*\!\!>\ \mathit{isort}\ \mathit{xs}\end{aligned}$$

This is because, as discussed in section 5.2.3, *isort* is treated as monolithic given that it operates on standard Haskell lists. The cost of $\mathit{pure}\ \mathit{head}\ <\!\!*\!\!>\ \mathit{isort}\ \mathit{xs}$, therefore, includes the cost of *fully* evaluating $\mathit{isort}\ \mathit{xs}$. In practice, however, insertion sort does not need to be fully applied to obtain the least element in the input list. In particular, Haskell's lazy evaluation will halt the sorting computation as soon the head of the list is generated. Next, we see how the *Tick* datatype can be used to explicitly encode such non-strict behaviour.

5.4.2 Case study 2: Non-strict insertion sort

Our cost analysis treats functions operating on standard Haskell datatypes as monolithic. To encode non-strict evaluation, we include *Tick* in the definitions of datatypes in order to explicitly suspend computations. We call datatypes that are defined using *Tick lazy* and functions that operate on them are called *non-strict*.

To illustrate these concepts, in this case study we define a non-strict minimum function to calculate the least element in a non-empty lazy list sorted using insertion sort. The execution cost of the new minimum function has a linear upper bound, which corresponds to the resources required by Haskell's on-demand evaluation.

Refined lazy lists

Following Danielsson (2008), we define lazy lists to be either empty (*Nil*) or constructed (*Cons*) from a $\mathit{lhead} :: a$ and a $\mathit{ltail} :: \mathit{Tick}\ (\mathit{LList}\ a)$:

$$\begin{aligned}&\{-@\ \ \textbf{data}\ \mathit{LList}\ a{<}p\ ::\ a\ \rightarrow\ a\ \rightarrow\ \mathit{Bool}{>}\\ &\qquad =\ \mathit{Nil}\\ &\qquad \mid\ \mathit{Cons}\ \{\ \mathit{lhead}\ ::\ a,\ \ \mathit{ltail}\ ::\ \mathit{Tick}\ (\mathit{LList}{<}p{>}\ (a{<}p\ \mathit{lhead}{>}))\ \}\ @-\}\end{aligned}$$

Notice that the tail of a non-empty lazy list is an *annotated* computation that returns another lazy list. Furthermore, to encode recursive properties into lazy lists, we use an abstract refinement p to capture invariants that hold between the head of a lazy list and each element of its tail, and moreover, that recursively hold inside the tail.

Sorted lazy lists are defined similarly to $OList\ a$, by instantiating the abstract refinement p to express that the head of each sublist is less than or equal to any element in the tail:

$$\{-@\ \ \textbf{type}\ OLList\ a\ =\ LList<\{\ \lambda x\ y\ \rightarrow\ x\ \leqslant\ y\ \}>\ a\ @-\}$$

Non-strict sorting

We can now define a non-strict version of the insertion function using lazy lists. The key distinction between *insert* and $insert_{ns}$ is that, in the definition below, the recursive call to $insert_{ns}$ is *suspended* and stored in the tail of the resulting list.

$$\begin{aligned}
&\{-@\ insert_{ns} :: Ord\ a \Rightarrow a \rightarrow xs : OLList\ a \rightarrow \{\ t : Tick\ (OLList\ a) \mid tcost\ t \leqslant 1\ \}\ @-\}\\
&insert_{ns}\ x\ Nil\ =\ return\ (Cons\ x\ (return\ Nil))\\
&insert_{ns}\ x\ (Cons\ y\ ys)\\
&\quad \mid\ x\ \leqslant\ y\ \ \ =\ wait\ (Cons\ x\ (return\ (Cons\ y\ ys)))\\
&\quad \mid\ otherwise\ =\ wait\ (Cons\ y\ (ys\ \gg\!=\ insert_{ns}\ x))
\end{aligned}$$

When analysing functions that operate on standard Haskell datatypes, we have seen that execution costs correspond to such functions being fully applied. Now we see that the execution costs of non-strict functions correspond to such functions returning the *first part* of their results. In this instance, $insert_{ns}$ returns the first element of its result by making one comparison when its input is non-empty, and zero comparisons otherwise: $tcost\ t \leqslant 1$.

Non-strict insertion sort is analogous to *isort*, however its result is a sorted lazy list:

$$\begin{aligned}
&\{-@\ isort_{ns}\ ::\ Ord\ a\ \Rightarrow\ xs : [a]\ \rightarrow\ \{\ t : Tick\ (OLList\ a)\ \mid\ tcost\ t\ \leqslant\ length\ xs\ \}\ @-\}\\
&isort_{ns}\ [\,]\ \ \ \ \ \ \ \ =\ return\ Nil\\
&isort_{ns}\ (x\ :\ xs)\ =\ insert_{ns}\ x\ =\!\!\ll\{1\}\ \ isort_{ns}\ xs
\end{aligned}$$

Given a standard Haskell list as input, $isort_{ns}$ returns a sorted lazy list of type $OLList\ a$. Hence, it is a non-strict function whose execution cost reflects the maximum number of

comparisons required to produce the first element in its result. Notice that this cost has been intrinsically verified because $(=\!\!\ll\{\cdot\})$ limits the execution cost of each $insert_{ns}$.

Non-strict minimum

The following non-strict minimum function returns the first element in a non-empty list xs *partially* sorted using $isort_{ns}$. As $minimum_{ns}$ only forces the first element of $isort_{ns}\ xs$ to be calculated, it requires at most $length\ xs$ comparisons:

$$\{-@\ minimum_{ns} :: Ord\ a \Rightarrow xs : NonEmpty\ a \to \{\ t : Tick\ a \mid tcost\ t \leqslant length\ xs\ \}\ @-\}$$
$$minimum_{ns}\ xs\ =\ pure\ lhead\ <\!*\!>\ isort_{ns}\ xs$$

Explicit laziness

Lazy lists of type $LList\ a$ are defined such that examining the head is zero-cost, but examining the last element has a cost equal to the sum total of the costs of each suspended computation in the tail. As discussed in section 5.2.3, if such a list is fully evaluated on multiple occasions during a computation, the library's default analysis records the cost of each evaluation independently. However, in practice, once a list is fully evaluated by Haskell, its value is memoised and thus subsequent uses are 'cost-free'.

To explicitly capture such sharing in our analysis, we use *pay* (Danielsson 2008):

$$\{-@\ pay\ ::\ m : Nat\ \to\ \{\ t_1 : Tick\ a\ \mid\ tcost\ t_1\ \geqslant\ m\ \}\ \to$$
$$\{\ t : Tick\ (Tick\ a)\ \mid\ tcost\ (tval\ t)\ =\ tcost\ t_1\ -\ m\ \}\ @-\}$$
$$pay\ m\ (Tick\ n\ x)\ =\ Tick\ m\ (Tick\ (n\ -\ m)\ x)$$

Evaluating $pay\ m\ x \gg\!\!= f$ allows f to use x numerous times while only paying m cost for it once. Therefore, if $m = tcost\ x$ then this effectively models sharing.

We repeated Danielsson's analysis of Okasaki's queues as part of the library's evaluation (section 5.4.5). In this example, non-strictness is captured by a lazy queue datatype and sharing is modelled explicitly by *lazy* functions that are non-strict and use *pay*.

5.4.3 Case study 3: Map fusion

In this case study, we use the proof combinators from section 5.3.3 to simultaneously reason about the correctness and efficiency of *map fusion*. This well-known property states that mapping one function $f :: a \to b$ followed by another function $g :: b \to c$ over a list $xs :: [a]$ gives the same result as mapping the composite function $g \circ f :: a \to c$ over the same list:

$$map\ g\ (map\ f\ xs) \quad = \quad map\ (g\ \circ\ f)\ xs$$

Although the two sides of this equation give the same results, they do not require the same amount of resources. In particular, the left-hand side traverses the list xs twice, whereas the right-hand side traverses xs only once. Thus, replacing the expression on the left-hand side with that on the right preserves correctness while saving $length\ xs$ resources. To prove this, we first define annotated versions of the mapping and function composition operators, and then reason simultaneously about the correctness and efficiency of these definitions.

Definitions

First, we define an annotated mapping function, *mapM*, which takes as input a function $f :: a \to Tick\ b$ returning an annotated result and a list xs. The cost of *mapM*'s result, given by applying f to each element x in the list xs, includes the number of recursive calls made during *mapM*'s execution *and* the cost of each application $f\ x$:

$$\begin{aligned}
&mapM\ ::\ (a\ \to\ Tick\ b)\ \to\ [a]\ \to\ Tick\ [b] \\
&mapM\ _\ [\,] \qquad\quad = \ pure\ [\,] \\
&mapM\ f\ (x\ :\ xs)\ =\ step\ 1\ (liftA2\ (:)\ (f\ x)\ (mapM\ f\ xs))
\end{aligned}$$

In particular, when the input list is empty, no resources are consumed; and when the input list is non-empty, *step* 1 is used to record the recursive call to *mapM* and *liftA2* (defined subsequently) reconstructs the list while recording the cost of the application $f\ x$.

Remark. To the best of our knowledge, it is not possible to define a refinement type for *mapM* that precisely describes its resource usage. This is because f is applied to arbitrary inputs x from the list xs, and thus we cannot specify the cost of each $f\ x$ in the

general case. One way to approximate this cost is to employ the technique described in section 5.4.1, which is to establish an upper bound n on the cost of $f\ x$ for any input x. The total execution cost of *mapM* would then be bounded above by $(n + 1) * \mathit{length}\ \mathit{xs}$. Nonetheless, we can prove that map fusion is an optimisation (a relational cost property) without needing to precisely compute the cost of each $f\ x$ (a unary cost property). This is a notable advantage of relational cost analysis (Çiçek 2018).

The *liftA2* function is defined in the *RTick* library. Similarly to the applicative operator (<∗>), *liftA2* takes as input a binary function f and two annotated arguments and returns an annotated result whose cost is equal to the sum of the costs of its arguments:

$$\{-@\ \mathit{liftA2} :: f : (a \to b \to c) \to t_1 : \mathit{Tick}\ a \to t_2 : \mathit{Tick}\ b \to \{\ t : \mathit{Tick}\ c \mid \mathit{tval}\ t = f\ (\mathit{tval}\ t_1)\ (\mathit{tval}\ t_2) \wedge \mathit{tcost}\ t = \mathit{tcost}\ t_1 + \mathit{tcost}\ t_2\ \}\ @-\}$$
$$\mathit{liftA2}\ f\ (\mathit{Tick}\ m\ x)\ (\mathit{Tick}\ n\ y) = \mathit{Tick}\ (m + n)\ (f\ x\ y)$$

To compose two annotated functions f and g, we define a composition function ($\gg\!\!>$) that, when given an argument x, applies g to the value of $f\ x$:

$$(\gg\!\!>) :: (a \to \mathit{Tick}\ b) \to (b \to \mathit{Tick}\ c) \to a \to \mathit{Tick}\ c$$
$$(\gg\!\!>)\ f\ g\ x = \textbf{let}\ \mathit{Tick}\ m\ y = f\ x\ \textbf{in let}\ \mathit{Tick}\ n\ z = g\ y\ \textbf{in}\ \mathit{Tick}\ (m + n)\ z$$

Notice that ($\gg\!\!>$) returns the total cost of both applications.

Specification

Using the above definitions and cost relations introduced in section 5.3.3, we can now state that the map fusion technique is a cost improvement in the left-to-right direction. Specifically, we can use *quantified improvement* to precisely capture the amount of resources saved by the optimisation, which is given by the length of the list being traversed:

$$(\mathit{mapM}\ f\ \mathit{xs} \gg\!= \mathit{mapM}\ g) \succcurlyeq \mathit{length}\ \mathit{xs} \Longrightarrow (\mathit{mapM}\ (f \gg\!\!> g)\ \mathit{xs})$$

The following extrinsic theorem formalises this property in Liquid Haskell:

$$\{-@\ \mathit{mapFusion} :: f : (a \to \mathit{Tick}\ b) \to g : (b \to \mathit{Tick}\ c) \to \mathit{xs} : [a] \to \{\ (\mathit{mapM}\ f\ \mathit{xs} \gg\!= \mathit{mapM}\ g) \succcurlyeq \mathit{length}\ \mathit{xs} \Longrightarrow (\mathit{mapM}\ (f \gg\!\!> g)\ \mathit{xs})\ \}\ @-\}$$

Proof by inequational rewriting

To prove the *mapFusion* theorem, we must define a (total and terminating) Haskell term that inhabits its type specification. In practice, we define such a term by (in)equationally rewriting on the left-hand side of the proof statement, $mapM\ f\ xs \gg\!= mapM\ g$, ultimately deriving the right-hand side, $mapM\ (f \gg\!\!> g)\ xs$. By utilising the proof combinators from section 5.3.3, we ensure that each rewrite step preserves correctness, that is, value equivalence. Furthermore, such combinators capture the total resource saving, which is calculated 'on the fly' as part of the derivation process. The proof proceeds in the standard manner, by induction on the structure of the argument list.

In the base case, we begin by unfolding the definitions of *mapM* and $(\gg\!=)$. The right-hand side of the proof statement then follows by folding the definition of *mapM*:

$$
\begin{aligned}
&mapFusion\ f\ g\ [\,] \\
&\quad = \quad mapM\ f\ [\,] \gg\!= mapM\ g \\
&\quad \Longleftrightarrow . \quad pure\ [\,] \gg\!= mapM\ g \\
&\quad \Longleftrightarrow . \quad mapM\ g\ [\,] \\
&\quad \Longleftrightarrow . \quad pure\ [\,] \\
&\quad \Longleftrightarrow . \quad mapM\ (f \gg\!\!> g)\ [\,] \\
&\quad *** \quad QED
\end{aligned}
$$

Here we see that the map fusion technique is a *cost equivalence* for empty lists. That is, the costs of both sides of the property are equal. This is to be expected as the length of the input list is zero, and hence no resources can be saved.

The inductive case also begins by unfolding the definition of *mapM*:

$$
\begin{aligned}
&mapFusion\ f\ g\ (x : xs) \\
&\quad = \quad mapM\ f\ (x : xs) \gg\!= mapM\ g \\
&\quad \Longleftrightarrow . \quad step\ 1\ (liftA2\ (:)\ (f\ x)\ (mapM\ f\ xs)) \gg\!= mapM\ g
\end{aligned}
$$

We then use quantified improvement to capture the cost saved by eliminating *step* 1:

$$
.\!>\!== 1 \Longrightarrow . \quad liftA2\ (:)\ (f\ x)\ (mapM\ f\ xs) \gg\!= mapM\ g
$$

To continue, we must unfold the definition of *liftA2*. By deconstructing the results of $f\ x$ and $mapM\ f\ xs$, by pattern matching on the *Tick* datatype in a **where** clause, we can independently refer to the costs and values of *liftA2*'s arguments:

```
  where
    Tick cf fx   = f x
    Tick cfs fxs = mapM f xs
```

Storing these parameters is particularly useful as the remaining cost savings and expenditures can be expressed entirely in terms of *cf*, *cfs*, and constants. This makes manipulating cost straightforward and allows us to focus primarily on correctness.

Using the bindings from the **where** clause, we can unfold the definition of *liftA2* and continue rewriting. First, we save the cost *cf* + *cfs* of *liftA2*'s result:

```
  <=>  Tick (cf + cfs) (fx : fxs) >>= mapM g
  .>=  cf + cfs =>.  pure (fx : fxs) >>= mapM g
```

We then unfold the definitions of $(\gg\!=)$ and *mapM* in turn:

```
  <=>  mapM g (fx : fxs)
  <=>  step 1 (liftA2 (:) (g fx) (mapM g fxs))
```

Eliminating *step* 1 saves an additional resource, however, we must then expend *cfs* resources in order to map *f* over the tail of the input list, *xs*:

```
  .>= 1 =>.  liftA2 (:) (g fx) (mapM g fxs)
  .<= cfs =<.  liftA2 (:) (g fx) (mapM f xs >>= mapM g)
```

At this point, we can appeal to the inductive hypothesis in order to save *length xs* resources, by substituting $mapM\ (f \gg\!> g)\ xs$ for $mapM\ f\ xs \gg\!= mapM\ g$:

```
    ? mapFusion f g xs
  .>= length xs =>.  liftA2 (:) (g fx) (mapM (f >>> g) xs)
```

To finalise the proof, we apply *f* and fold the definition of *mapM*:

$.\Longleftarrow cf \Longrightarrow\!\!<.\ \ liftA2\ (:)\ ((f \gg\!\!> g)\ x)\ (mapM\ (f \gg\!\!> g)\ xs)$

$.\Longleftarrow 1 \Longrightarrow\!\!<.\ \ step\ 1\ (liftA2\ (:)\ ((f \gg\!\!> g)\ x)\ (mapM\ (f \gg\!\!> g)\ xs))$

$\Longleftrightarrow.\ \ mapM\ (f \gg\!\!> g)\ (x : xs)$

$***\ \ QED$

As we have seen throughout the proof, the quantified cost operators are used to explicitly record resource saving, for example ($.\Longrightarrow\!\!> cf + cfs \Longrightarrow.$), and expenditure, for example ($.\Longleftarrow cf \Longrightarrow\!\!<.$). Overall, the cost savings and expenditures involving *cf* and *cfs* cancel out, as do the latter two costs involving *step* 1. The remaining costs are from the initial saving of 1 from *step* 1 and the saving of *length xs* from the inductive hypothesis. Hence, the resulting expression, $mapM\ (f \gg\!\!> g)\ (x : xs)$, requires *length* $(x : xs)$ *fewer* resources than the initial expression, $mapM\ f\ (x : xs) \gg\!= mapM\ g$, as expected. As Liquid Haskell has SMT support for arithmetic, this cost saving is calculated automatically by the system.

In summary, this proof illustrates the power of relational cost analysis in our setting. In particular, the costs of $f\ x$ and $mapM\ f\ xs$ cannot be easily captured by a unary cost analysis of *mapM*. Nevertheless, our extrinsic approach overcomes this restriction by allowing such 'higher-order costs' to cancel out on both sides of the theorem's proof statement. Furthermore, storing the costs of $f\ x$ and $mapM\ f\ xs$ in a **where** clause allowed us to primarily focus on the correctness aspect of the proof. As such, we have not only shown how reasoning about resource usage can be as straightforward as reasoning about correctness, we have shown that the two can in fact *coincide*.

5.4.4 Case study 4: Optimised-by-construction reverse

In chapter 4, we used the Unie system to mechanically improve the naive list-reversing function, *slowRev*. In particular, we proved that $slowRev \lessapprox fastRev$ (see section 4.6). Recall from our previous discussion in section 5.3.3 that efficiency results à la improvement theory do not entail correctness results. In other words, a separate proof is required to show that *slowRev* and *fastRev* give the same results. Such a proof is given in (Hackett and Hutton 2014), which has recently been formalised in Liquid Haskell (Vazou et al. 2018).

The fundamental goal of the proofs in (Hackett and Hutton 2014) and (Vazou et al. 2018)

is to show that the implementation of *fastRev* is *correct-by-construction*. More specifically, the proofs show that the denotational meaning of an initial specification defined using *slowRev* is preserved by the calculation resulting in *fastRev*.

Our previous case study used the proof combinators from figure 5.2 to reason simultaneously about the correctness and efficiency of the map fusion optimisation technique. We apply a similar approach in this case study to go one step further than (Vazou et al. 2018). That is, we prove that the derived implementation of *fastRev* preserves meaning *and* improves efficiency, thereby unifying the separate proofs of $slowRev\ xs = fastRev\ xs$ and $slowRev \lessapprox fastRev$ given by Hackett and Hutton (2014) (albeit in a simplified manner as we do not consider all program contexts). As before, total resource saving and final resource usage are both calculated on the fly during the derivation.

We begin by recalling the naive reverse function, which has been annotated to count the total number of recursive calls, that is, by itself and $(\underline{+\!\!+})$:

$$\begin{aligned} \underline{slowRev}\ [\,] &= return\ [\,] \\ \underline{slowRev}\ (x : xs) &= \underline{slowRev}\ xs >\!\!1\!\!= (\underline{+\!\!+}\ [x]) \end{aligned}$$

The *slowRev* function appends each element of the input list to the end of its reversed tail. As the cost of $(\underline{+\!\!+})$ is linear in the length of its first argument, the total number of recursive calls is quadratic, as expected. This cost is captured by the following extrinsic theorem:

$$\begin{aligned} &\{-@\ \underline{slowRev}_{Cost} :: xs : [a] \to \{\ tcost\ (\underline{slowRev}\ xs) = \\ &\qquad ((length\ xs)^2\ /\ 2) + ((length\ xs + 1)\ /\ 2)\ \}\ @-\} \end{aligned}$$

To improve *slowRev*, we seek to fuse together the processes of appending and reversing. In section 4.5, we achieved this by applying the worker/wrapper theorem. Here we use induction as it allows us to extend the approach taken by Vazou et al. (2018). To this end, we seek to define a new function that reverses its first argument and appends its second, and express this requirement as a Liquid Haskell specification, as follows:

$$\{-@\ \underline{revApp} :: xs : [a] \to ys : [a] \to \{\ t : Tick\ [a] \mid \underline{slowRev}\ xs \gg\!= (\underline{+\!\!+}\ ys) >\!\!\simeq\!\!> t\ \}\ @-\}$$

As we plan to use *revApp* to improve *slowRev*, any implementation we propose for the function *must* record the total number of recursive calls. Furthermore, note that the above

specification is an improvement. This means that *revApp* must give the same results as *slowRev* $xs \gg\!= (+\!\!+\ ys)$ but consume no more resources. We start with a trivial definition for *revApp* that satisfies the specification but makes no cost savings:

$$\begin{aligned} &\textit{revApp}\ [\,] \qquad\quad ys &&= \textit{slowRev}\ [\,] \gg\!= (+\!\!+\ ys) \\ &\textit{revApp}\ (x : xs)\ ys &&= \textit{slowRev}\ (x : xs) \gg\!= (+\!\!+\ ys) \end{aligned}$$

Initial resource usage

While deriving a new implementation for *revApp*, we will calculate the total resource saving—call it s—on the fly using quantified improvement. In order to calculate the resource usage of the final result, however, we must first calculate the initial resource usage of *revApp*, that is, of *slowRev* $xs \gg\!= (+\!\!+\ ys)$—call it u. The final resource usage of the improved definition once derived is then simply $u - s$.

The resource usage of *slowRev* $xs \gg\!= (+\!\!+\ ys)$ is as follows:

$$\begin{aligned} \{-@\ \textit{revApp}_{Cost_0}\ ::\ xs : [a] \to ys : [a] \to \{\ tcost\ (\textit{slowRev}\ xs \gg\!= (+\!\!+\ ys))\ = \\ ((length\ xs)^2\ /\ 2)\ +\ ((3\ *\ length\ xs\ +\ 1)\ /\ 2)\ \}\ @-\} \end{aligned}$$

The proof of this theorem follows immediately from the execution costs of *slowRev* (given previously by $\textit{slowRev}_{Cost}$) and $(+\!\!+)$, and is available on the library's GitHub page (Handley and Vazou 2019) along with the proof of $\textit{slowRev}_{Cost}$.

Proof by inequational rewriting

The next step of the improvement process is to rewrite the right-hand sides of *revApp*'s trivial definition to more efficient forms. As before, we use proof combinators introduced in section 5.3.3 to ensure that each rewrite preserves the result (denotational meaning) of *revApp* while preserving or improving its execution cost.

The rewriting follows the general insight that proofs of improvement are very similar to their counterpart proofs of correctness (as illustrated previously in section 5.3.3). Consequently, we initially focus on correctness by unfolding and folding definitions. Whenever a resource saving can be made, we use quantified improvement to precisely capture it. Finally, at the end of the calculation, we turn our attention to resource usage.

We begin by rewriting the base case of *revApp*. First we unfold the definitions of *slowRev*, ($\gg=$), and ($+\!\!+$); then we inline the **let** binding and fold *return*:

$$
\begin{aligned}
& \mathit{revApp}\ [\,]\ \mathit{ys} \\
&\quad = \ \mathit{slowRev}\ [\,] \gg= (+\!\!+\ \mathit{ys}) \\
&\quad \Longleftrightarrow \ \mathit{Tick}\ 0\ [\,] \gg= (+\!\!+\ \mathit{ys}) \\
&\quad \Longleftrightarrow \ (\mathbf{let}\ \mathit{Tick}\ n\ y\ =\ [\,]\ +\!\!+\ \mathit{ys}\ \mathbf{in}\ \mathit{Tick}\ n\ y) \\
&\quad \Longleftrightarrow \ (\mathbf{let}\ \mathit{Tick}\ n\ y\ =\ \mathit{Tick}\ 0\ \mathit{ys}\ \mathbf{in}\ \mathit{Tick}\ n\ y) \\
&\quad \Longleftrightarrow \ \mathit{return}\ \mathit{ys}
\end{aligned}
$$

In the recursive case, we also begin by unfolding definitions as much as possible:

$$
\begin{aligned}
& \mathit{revApp}\ (x\ :\ \mathit{xs})\ \mathit{ys} \\
&\quad = \ \mathit{slowRev}\ (x\ :\ \mathit{xs}) \gg= (+\!\!+\ \mathit{ys}) \\
&\quad \Longleftrightarrow \ (\mathit{slowRev}\ \mathit{xs}\ >\!1\!=\ (+\!\!+\ [x])) \gg= (+\!\!+\ \mathit{ys}) \\
&\quad \Longleftrightarrow \ (\mathbf{let}\ \mathit{Tick}\ o\ w\ =\ \mathit{slowRev}\ \mathit{xs}\ \mathbf{in}\ \mathit{Tick}\ o\ w\ >\!1\!=\ (+\!\!+\ [x])) \gg= (+\!\!+\ \mathit{ys}) \\
&\quad \Longleftrightarrow \ (\mathbf{let}\ \mathit{Tick}\ o\ w\ =\ \mathit{slowRev}\ \mathit{xs}\ \mathbf{in} \\
&\qquad\quad \mathbf{let}\ \mathit{Tick}\ p\ v\ =\ w\ +\!\!+\ [x]\ \mathbf{in} \\
&\qquad\qquad \mathit{Tick}\ (1\ +\ o\ +\ p)\ v \gg= (+\!\!+\ \mathit{ys})) \\
&\quad \Longleftrightarrow \ (\mathbf{let}\ \mathit{Tick}\ o\ w\ =\ \mathit{slowRev}\ \mathit{xs}\ \mathbf{in} \\
&\qquad\quad \mathbf{let}\ \mathit{Tick}\ p\ v\ =\ w\ +\!\!+\ [x]\ \mathbf{in} \\
&\qquad\qquad \mathbf{let}\ \mathit{Tick}\ q\ u\ =\ v\ +\!\!+\ \mathit{ys}\ \mathbf{in} \\
&\qquad\qquad\quad \mathit{Tick}\ (1\ +\ o\ +\ p\ +\ q)\ u)
\end{aligned}
$$

Notice that in order to unfold the definition of ($>\!1\!=$), we must introduce a **let** binding because *slowRev xs* is not in '*Tick* normal form'.

At this point, there is an addition of a constant cost on the returned *Tick*. As we are in the recursive case of the calculation, we do not save this cost under the assumption that we will recurse on *revApp*. Instead, we 'bank' the recursive call using *step*.

$$
\begin{aligned}
&\quad \Longleftrightarrow \ \mathit{step}\ 1\ (\mathbf{let}\ \mathit{Tick}\ o\ w\ =\ \mathit{slowRev}\ \mathit{xs}\ \mathbf{in} \\
&\qquad\qquad\quad \mathbf{let}\ \mathit{Tick}\ p\ v\ =\ w\ +\!\!+\ [x]\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{let}\ \mathit{Tick}\ q\ u\ =\ v\ +\!\!+\ \mathit{ys}\ \mathbf{in}\ \mathit{Tick}\ (o\ +\ p\ +\ q)\ u)
\end{aligned}
$$

$$\Leftrightarrow.\ \ step\ 1\ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in}$$
$$\textbf{let}\ Tick\ p\ v\ =\ w\ \ \underline{+\!\!+}\ \ [x]\ \gg\!=\ (\underline{+\!\!+}\ ys)\ \textbf{in}\ Tick\ (o\ +\ p)\ v)$$

Folding the definition of $(\gg\!=)$ exposes the expression $w\ \underline{+\!\!+}\ [x] \gg\!= (\underline{+\!\!+}\ ys)$, which is two appends associated to the left. In order to continue with the calculation, these appends must be reassociated to the right. From example 5.6 of section 5.2, we know that this is an improvement that saves *length xs* resources:

$$\{-@\ appendAssoc_{QImp}\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow\ zs : [a]\ \rightarrow$$
$$\{\ (xs\ \ \underline{+\!\!+}\ \ ys\ \gg\!=\ (\underline{+\!\!+}\ zs))\ \succ\!=\!=\ length\ xs\ \Longrightarrow\ ((xs\ \underline{+\!\!+})\ =\!\!\ll\ ys\ \ \underline{+\!\!+}\ \ zs)\ \}\ @-\}$$

Therefore, we appeal to the $appendAssoc_{QImp}$ lemma with $xs = tval\ (\underline{slowRev}\ xs)$, $ys = [x]$, and $zs = ys$. Note that *slowRev*'s refinement type specification from above automatically entails that $length\ (tval\ (\underline{slowRev}\ xs)) = length\ xs$:

$$?\ \ \ appendAssoc_{QImp}\ (tval\ (\underline{slowRev}\ xs))\ [x]\ ys$$
$$.\!\succ\!=\!=\ length\ xs \Longrightarrow.\ \ step\ 1\ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in}$$
$$\textbf{let}\ Tick\ p\ v\ =\ [x]\ \ \underline{+\!\!+}\ \ ys\ \gg\!=\ (w\ \underline{+\!\!+})\ \textbf{in}\ Tick\ (o\ +\ p)\ v)$$

The proof continues by unfolding the definitions of $(\underline{+\!\!+})$, *pure*, and $(<_1>)$:

$$\Leftrightarrow.\ \ step\ 1\ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in}$$
$$\textbf{let}\ Tick\ p\ v\ =\ pure\ (x :)\ <_1>\ ([\,]\ \ \underline{+\!\!+}\ \ ys)\ \gg\!=\ (w\ \underline{+\!\!+})\ \textbf{in}$$
$$Tick\ (o\ +\ p)\ v)$$
$$\Leftrightarrow.\ \ step\ 1\ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in}$$
$$\textbf{let}\ Tick\ p\ v\ =\ Tick\ 1\ (x\ :\ ys)\ \gg\!=\ (w\ \underline{+\!\!+})\ \textbf{in}\ Tick\ (o\ +\ p)\ v)$$

Unfolding the definition of $(<_1>)$ has presented us with another opportunity to save resources. We take it, capturing the saving using quantified improvement:

$$.\!\succ\!=\!=\ 1 \Longrightarrow.\ \ step\ 1\ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in}$$
$$\textbf{let}\ Tick\ p\ v\ =\ Tick\ 0\ (x\ :\ ys)\ \gg\!=\ (w\ \underline{+\!\!+})\ \textbf{in}\ Tick\ (o\ +\ p)\ v)$$

Then, we unfold $(\gg\!=)$ and inline the resulting **let** binding:

$$
\begin{aligned}
&\Longleftrightarrow. \;\; \mathit{step}\ 1\ (\textbf{let}\ \mathit{Tick}\ o\ w\ =\ \underline{\mathit{slowRev}}\ \mathit{xs}\ \textbf{in} \\
&\qquad\qquad \textbf{let}\ \mathit{Tick}\ p\ v\ =\ \mathit{Tick}\ 0\ (x\ :\ \mathit{ys})\ \textbf{in} \\
&\qquad\qquad\quad \textbf{let}\ \mathit{Tick}\ q\ u\ =\ w\ \ \underline{+\!\!+}\ \ v\ \textbf{in}\ \mathit{Tick}\ (o\ +\ p\ +\ q)\ u) \\
&\Longleftrightarrow. \;\; \mathit{step}\ 1\ (\textbf{let}\ \mathit{Tick}\ o\ w\ =\ \underline{\mathit{slowRev}}\ \mathit{xs}\ \textbf{in} \\
&\qquad\qquad \textbf{let}\ \mathit{Tick}\ q\ u\ =\ (w\ \ \underline{+\!\!+}\ \ (x\ :\ \mathit{ys}))\ \textbf{in}\ \mathit{Tick}\ (o\ +\ q)\ u) \\
&\Longleftrightarrow. \;\; \mathit{step}\ 1\ (\underline{\mathit{slowRev}}\ \mathit{xs}\ \gg\!=\ (\underline{+\!\!+}\ \ (x\ :\ \mathit{ys})))
\end{aligned}
$$

The final step of the calculation is to rewrite the new definition of *revApp* to be self-contained. It should be clear that replacing the expression $\underline{\mathit{slowRev}}\ \mathit{xs} \gg\!= (\underline{+\!\!+}\ \ (x : \mathit{ys}))$ with $\underline{\mathit{revApp}}\ \mathit{xs}\ (x : \mathit{ys})$ saves resources. In particular:

$$
\begin{aligned}
&?\quad \underline{\mathit{revApp}}_{\mathit{Cost}_0}\ \mathit{xs}\ (x\ :\ \mathit{ys}) \\
&.\!>\!=\!= ((\mathit{length}\ \mathit{xs})^2\ `\mathit{div}`\ 2)\ +\ ((\mathit{length}\ \mathit{xs} + 1)\ `\mathit{div}`\ 2) \Longrightarrow.\ \ \mathit{step}\ 1\ (\underline{\mathit{revApp}}\ \mathit{xs}\ (x\ :\ \mathit{ys}))
\end{aligned}
$$

The above resource saving is calculated by subtracting *length xs* from the resource usage of $\mathit{slowRev}\ \mathit{xs} \gg\!= (\underline{+\!\!+}\ \ (x : \mathit{ys}))$, calculated previously using $\underline{\mathit{revApp}}_{\mathit{Cost}_0}$. Note that *length xs* must be subtracted because evaluating $\underline{\mathit{revApp}}\ \mathit{xs}\ (x : \mathit{ys})$ requires *length xs* recursive calls, whereby each recursive call is recorded by the *step* 1 that we banked earlier.

Resource saving

Having reached the end of the proof, we can now turn our attention to calculating *revApp*'s final resource usage. Below is a table listing the function's initial resource usage, u, calculated by $\underline{\mathit{revApp}}_{\mathit{Cost}_0}$; the total saving, s, calculated by summing up the individual savings throughout the proof; and the final usage, which is simply $u - s$.

Initial usage (u)	$\frac{\lvert(x\ :\ xs)\rvert^2}{2} + \frac{3\ *\ \lvert(x\ :\ xs)\rvert\ +\ 1}{2}$
Total saving (s)	$\lvert xs\rvert + 1 + \frac{\lvert xs\rvert^2}{2} + \frac{\lvert xs\rvert\ +\ 1}{2}$
Final usage $(u - s)$	$\lvert(x : xs)\rvert$

By way of a simple subtraction, we have calculated the final resource usage of *revApp* to be linear in the length of its first argument, as expected. Adding this bound to *revApp*'s

initial specification allows Liquid Haskell to verify that this property holds, and thus the derivation of an *optimised-by-construction* implementation for $\underline{revApp}$ is complete:

$$\begin{aligned}&\{-@\ \underline{revApp}\ ::\ xs : [a]\ \rightarrow\ ys : [a]\ \rightarrow \\ &\qquad \{\ t : (Tick\ [a])\ \mid\ \underline{slowRev}\ xs\ \gg\!=\ (\underline{+\!\!+}\ ys)\ >\!\simeq\!>\ t\ \wedge\ tcost\ t\ =\ length\ xs\ \}\ @-\}\end{aligned}$$

Similarly to the last case study on map fusion, quantified improvement makes the quantity and locality of each cost saving explicit throughout the above calculation. In particular, it shows a linear cost saving per recursive call. This corresponds precisely to evaluating $(\underline{+\!\!+})$ in order to fuse together the processes of reversing and appending, which was our primary goal. Furthermore, the proof combinators we used simply return their last arguments. As such, at compile time, GHC will remove all of the intermediate calculation steps, leading to the following concise definition of $\underline{revApp}$:

$$\begin{aligned}&\underline{revApp}\ ::\ [a]\ \rightarrow\ [a]\ \rightarrow\ Tick\ [a] \\ &\underline{revApp}\ [\,]\qquad\ ys\ =\ return\ ys \\ &\underline{revApp}\ (x\ :\ xs)\ ys\ =\ step\ 1\ (\underline{revApp}\ xs\ (x\ :\ ys))\end{aligned}$$

Optimising naive reverse

Finally, we can use $\underline{revApp}$ to improve the definition of $\underline{slowRev}$, as follows:

$$\begin{aligned}&\{-@\ \underline{fastRev}\ ::\ xs : [a]\ \rightarrow\ \{\ t : Tick\ [a]\ \mid\ \underline{slowRev}\ xs\ >\!\simeq\!>\ t\ \wedge\ tcost\ t\ =\ length\ xs\ \} \\ &\underline{fastRev}\ xs \\ &\quad =\ \ \underline{slowRev}\ xs \\ &\quad \Longleftrightarrow.\ \ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in let}\ Tick\ p\ v\ =\ pure\ w\ \textbf{in}\ Tick\ (o\ +\ p)\ v) \\ &\quad\ \ ?\ \ \ rightId_{QImp}\ (tval\ (\underline{slowRev}\ xs)) \\ &\ .\!\Longleftarrow length\ xs \Longrightarrow\!.\ \ (\textbf{let}\ Tick\ o\ w\ =\ \underline{slowRev}\ xs\ \textbf{in} \\ &\qquad\qquad \textbf{let}\ Tick\ p\ v\ =\ w\ \ \underline{+\!\!+}\ \ [\,]\ \textbf{in}\ Tick\ (o\ +\ p)\ v) \\ &\quad \Longleftrightarrow.\ \ \underline{slowRev}\ xs\ \gg\!=\ (\underline{+\!\!+}\ [\,]) \\ &\quad\ \ ?\ \ \ \underline{revApp}_{Cost_0}\ xs\ [\,] \\ &\ .\!>\!\Longrightarrow ((length\ xs)^2\ `div`\ 2)\ +\ ((length\ xs\ +\ 1)\ `div`\ 2) \Longrightarrow.\ \ \underline{revApp}\ xs\ [\,]\end{aligned}$$

Notice that by applying append's right identity law, $rightId_{QImp}$, in the right-to-left direction, the resource usage of the resulting expression is greater than or equal to that of the

initial expression, by a cost of *length xs*. This is captured using quantified *diminishment*. A simple subtraction (as above) reveals that the final resource usage of *fastRev xs* is *length xs*, which is verified by Liquid Haskell. Similarly to *revApp*, the intermediate calculation steps of *fastRev* will be removed at compile time, leading to the following concise definition:

```
fastRev :: [a] → Tick [a]
fastRev xs = revApp xs []
```

The familiar functions *revApp* and *fastRev* were the results of the derivations in (Hackett and Hutton 2014) and (Vazou et al. 2018). Each can be derived from *revApp* and *fastRev*, respectively, by simply removing the cost annotations:

```
revApp :: [a] → [a] → [a]
revApp []       ys = ys
revApp (x : xs) ys = revApp xs (x : ys)

fastRev :: [a] → [a]
fastRev xs = revApp xs []
```

In summary, the proof in this case study mirrors that of section 4.1 in (Vazou et al. 2018) *step-for-step*: we encourage readers to check. More concretely, we have replaced equational reasoning with inequational reasoning, whereby the resource saving of each rewrite is made explicit using the notion of quantified improvement. This allows us to calculate final resource usage on the fly as part of the derivation process. Thus, overall, we have taken a calculation aimed at deriving a correct-by-construction reverse function and transformed it into a calculation aimed at deriving an optimised-by-construction reverse function.

5.4.5 Summary of examples

To finalise our library's evaluation, we summarise all of the examples we have surveyed during its development. The corresponding source files for each example can be found on the library's GitHub page (Handley and Vazou 2019).

Overview

Table 5.1 provides a quantitative summary of each example and is split into five categories. The first three categories include examples from the existing literature, the fourth category consists of higher-order examples, and the final category includes complexity analyses of different sorting algorithms. An overview of the five categories is provided below.

– **Laziness** includes functions that manipulate lazy lists and lazy queues from (Danielsson 2008). For example, in section 5.4.2, we proved that non-strict insertion sort on lazy lists is linear. We also encoded lazy queues and proved that viewing a lazy queue and appending at the end are constant-time operations. Danielsson (2008) reifies cost using a type-level index, namely $\mathit{Thunk}\ n\ a$ where n is a type-level *Nat*, while we use a value-level integer field. Because of this distinction, Danielsson (2008) does not require ghost cost parameters (as per our $(=\ll\{\cdot\})$ of section 5.4.1). On the other hand, type-level costs cannot be abstracted, which is a requirement of our higher-order examples. Finally, as our analysis builds on top of Liquid Haskell's existing features, it incorporates additional (automated) correctness properties such as 'sortedness'.

– **Relational** comprises *all* cost analyses from (Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2018). These examples compare the resource usage of the same function on different inputs, for instance, the constant-time comparison example from section 5.2.2; or different functions on the same input, for instance, the memory allocation case study compares the space usage required by the standard and tail recursive implementations of Haskell's *length* function on lists.

  Overall, this set of examples highlights a number of distinctions between our approach and that of relational refinement type systems developed for resource analysis. First of all, our system is agnostic to the resource being analysed, which means that the user has the flexibility to define arbitrary resources but also the responsibility to manually annotate resource usage. In comparison, the approach taken by (Çiçek et al. 2017; Çiçek 2018) only analyses runtime complexity.

Table 5.1: Cost analysis using the *RTick* library

Code reports the lines of executable code, *Spec* reports the lines of specifications, and *Proof* reports the lines of proof terms.

	Property	Lines of code		
		Code	**Spec**	**Proof**
Laziness (Danielsson 2008)				
Insertion sort	$\textsc{Cost}(isort_{ns}\ xs) \leqslant \lvert xs \rvert$	11	9	0
Implicit queues	$\textsc{Cost}(snoc_{lz}\ q\ x) = 5$, $\textsc{Cost}(view_{lz}\ q) = 1$	47	22	0
Relational (Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2018)				
2D count	$\textsc{Cost}(count_{2D}\ find_1) \leqslant \textsc{Cost}(count_{2D}\ find_2)$	17	7	21
Binary counters	$\textsc{Cost}(decr\ k\ tt) = \textsc{Cost}(incr\ k\ ff)$	23	18	16
Boolean expressions	$\textsc{NoShort}(e) \Rightarrow \textsc{Cost}(eval_1\ e) = \textsc{Cost}(eval_2\ e)$	24	2	6
Constant-time comparison	$\textsc{Cost}(compare\ p\ u_1) = \textsc{Cost}(compare\ p\ u_2)$	6	5	2
Insertion sort	$\textsc{Sorted}(xs) \Rightarrow \textsc{Cost}(isort\ xs) \leqslant \textsc{Cost}(isort\ ys)$	11	19	53
Memory allocation of length	$\textsc{Cost}(length_{gr}\ xs) - \textsc{Cost}(length_{tr}\ xs) = \lvert xs \rvert$	13	6	5
Relational insertion sort	$\textsc{Cost}(isort\ xs) - \textsc{Cost}(isort\ ys) = unsorted_{diff}\ xs\ ys$	21	25	16
Relational merge sort	$\textsc{Cost}(msort\ xs) - \textsc{Cost}(msort\ ys) \leqslant \lvert xs \rvert * (1 + \log_2(diff\ xs\ ys))$	34	21	52
Square and multiply	$\textsc{Cost}(sam\ t\ x\ l_1) - \textsc{Cost}(sam\ t\ x\ l_2) \leqslant t * diff\ l_1\ l_2$	16	8	2
Datatypes (Vazou et al. 2018)				
Append's monoid laws	*see example 5.6 of section 5.2*	13	18	75
Appending	$\textsc{Cost}(xs\ +\!\!+\ ys) = \lvert xs \rvert$	9	6	0
Flattening	$\textsc{Perfect}(t) \Rightarrow \textsc{Cost}(fastFlatten\ t) = 2^{\lvert t \rvert} - 1$	21	17	56
Optimised-by-construction reverse	$slowRev\ xs \gg\!\!\sim\!\!> fastRev\ xs$	18	38	183
Reversing (naive)	$\textsc{Cost}(slowRev\ xs) = \frac{\lvert xs \rvert^2}{2} + \frac{\lvert xs \rvert + 1}{2}$	14	16	52
Reversing (optimised)	$\textsc{Cost}(fastRev\ xs) = \lvert xs \rvert$	8	7	0
Higher-order				
fold	$\textsc{Cost}(foldl\ xs) = \lvert xs \rvert$, $\textsc{Cost}(foldl'\ xs) = 0$	9	3	0
foldM	$\textsc{Cost}(foldlM\ xs) = (1 + n) * \lvert xs \rvert$, $\textsc{Cost}(foldlM'\ xs) = n * \lvert xs \rvert$	7	3	0
foldM relational	$foldlM\ xs \gg\!\!= \lvert xs \rvert \Longrightarrow foldlM'\ xs$	13	3	19
Map fusion	$(mapM\ f\ xs \gg\!\!= mapM\ g) \gg\!\!= \lvert xs \rvert \Longrightarrow (mapM\ (f \gg\!\!> g)\ xs)$	8	2	26
Sorting				
Data.List.sort	$\textsc{Cost}(sort\ xs) \leqslant 4 * \lvert xs \rvert * \log_2 \lvert xs \rvert + \lvert xs \rvert$	42	52	70
Insertion sort	$\textsc{Cost}(isort\ xs) \leqslant \lvert xs \rvert^2$	11	8	0
Merge sort	$\frac{\lvert xs \rvert}{2} \log_2 \lvert xs \rvert \leqslant \textsc{Cost}(msort\ xs) \leqslant \lvert xs \rvert * \log_2 \frac{\lvert xs \rvert}{2} + \lvert xs \rvert$	27	53	144
Quicksort	$\textsc{Cost}(qsort\ xs) \leqslant \frac{1}{2} * (\lvert xs \rvert + 1) * (\lvert xs \rvert + 2)$	11	3	25
Total		434	371	823

Secondly, unary cost analysis in our setting is automatically checked but users must specify appropriate cost bounds manually, which adds some degree of complexity to the analysis. In relational systems, such annotations are typically not required when the analysis is performed using 'synchronous' rules. Nonetheless, when synchronous rules fail, relational systems essentially replicate the unary analysis automatically performed by our system. Whether to use the synchronous or 'asynchronous' approach is dictated by heuristics in (Çiçek et al. 2019).

And finally, in many of our examples, we must manually prove extrinsic theorems that can be automatically inferred by relational type systems. This is to be expected, as such systems are specialised for resource tracking. On the other hand, these systems cannot encode the sophisticated correctness invariants, such as 'sortedness', that we frequently use to simplify our analyses or improve their precision.

– **Datatypes** includes properties concerning lists, trees, and optimisations whose Liquid Haskell correctness proofs initially appeared in (Vazou et al. 2018). We used the proof combinators in figure 5.2 to extend the correctness proofs with explicit resource tracking. Our experience, in accordance with the case studies of sections 5.4.3 and 5.4.4, is that because Liquid Haskell has SMT-automated integer arithmetic, reasoning about resource usage is as straightforward as reasoning about correctness. In fact, most of the proofs are very similar to their correctness counterparts.

– **Higher-order** includes three higher-order examples. As per example 5.2 in section 5.2, we tracked the number of thunks allocated by *foldl* and *foldl′*. We then extended the analysis, considering *foldM* and *foldM′* whose function arguments can also allocate thunks. For unary analysis, the cost of the function being folded is bounded above by a ghost cost parameter n. The relational comparison between *foldM* and *foldM′* does not require this bound and is greatly simplified using our proof combinators, similarly to the map fusion case study of section 5.4.3.

  This category illustrates two key features of our cost analysis. Firstly, tracked resources can have arbitrary, user-defined meanings, such as number of allocated thunks. And secondly, our analysis supports higher-order functions, whose resource analysis

is straightforward in a relational setting.

– **Sorting** includes cost analyses of well-known sorting algorithms: *Data.List*'s smooth merge sort, insertion sort, merge sort, and quicksort. Other than the known upper bounds of the algorithms, we proved a lower bound for merge sort (section 5.2.2) and that both insertion sort (section 5.4.1) and smooth merge sort require at most linear comparisons when applied to sorted lists.

  Two of the functions listed above have logarithmic bounds. We axiomatised logarithmic properties as Haskell functions using Liquid Haskell's *assume* feature. To prove these complexity bounds we used extrinsic reasoning, making explicit calls to the axioms when necessary. This showcases another feature of our analysis: despite Liquid Haskell only providing SMT automation for linear arithmetic, our analysis is still able to check arbitrarily expressive resource bounds.

Overall, we chose these examples because they: required both unary and relational cost analysis; often imposed constraints on the inputs/outputs of functions; were reasonably challenging to encode using our library; allowed us to draw comparisons against existing systems in the literature. Importantly, all of the examples demonstrate how correctness properties can be naturally integrated into our cost analysis.

Breakdown

Each line in table 5.1 describes an indicative property we proved. In some cases, we proved additional properties. In other cases, the desired property required proving a stronger theorem. For brevity, these additional properties are not included. However, the source files for all of the examples are available online (Handley and Vazou 2019).

Synopsis

In total, we wrote 434 lines of executable code, 371 lines of Liquid Haskell specifications, and 823 lines of proof terms. The total lines of code dedicated to specifications and proofs is approximately three times as much as executable code. Given the complexity of the

$$
\begin{array}{rrcl}
\textit{Constants} & c & ::= & 0, 1, -1, \ldots \mid \textit{true}, \textit{false} \mid \\
 & & & +, -, \ldots \mid =, <, \ldots \mid \textit{Crash} \\
\textit{Values} & v & ::= & c \mid \lambda x.e \mid D\ \vec{e} \\
\textit{Expressions} & e & ::= & v \mid x \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \\
 & & & \textbf{case}\ x = e\ \textbf{of}\ \{\ D\ \vec{x} \rightarrow e\ \} \\
\textit{Refinements} & r & ::= & e \\
\textit{Basic types} & B & ::= & \textit{Int}, \textit{Bool}, T \\
\textit{Types} & \tau & ::= & \{\ v : B \mid r\ \} \mid x : \tau \rightarrow \tau \\
\textit{Evaluation contexts} & \mathbb{C} & ::= & [-] \mid \mathbb{C}\ e \mid c\ \mathbb{C} \mid D\ \vec{e}\ \mathbb{C}\ \vec{e} \mid \\
 & & & \textbf{case}\ x = \mathbb{C}\ \textbf{of}\ \{\ D\ \vec{y} \rightarrow e\ \}
\end{array}
$$

Reduction

$$
\begin{array}{rcll}
\mathbb{C}[e] & \hookrightarrow & \mathbb{C}[e'] & \text{if } e \hookrightarrow e' \\
c\ v & \hookrightarrow & \delta(c, v) & \\
(\lambda x.e)\ e_x & \hookrightarrow & e[e_x/x] & \\
\textbf{let}\ x = e_x\ \textbf{in}\ e & \hookrightarrow & e[e_x/x] & \\
\textbf{case}\ x = D_j\ \vec{e}\ \textbf{of}\ \{\ D_i\ \vec{y_i} \rightarrow e_i\ \} & \hookrightarrow & e_j[D_j\ \vec{e}/x][\vec{e}/\vec{y_j}] &
\end{array}
$$

Figure 5.3: λ^U: syntax and operational semantics (Vazou et al. 2014).

properties we have proved, we consider this reasonable. Moreover, the sizes of many proof terms have been decreased by using Liquid Haskell's PLE feature (Vazou 2016).

5.5 Correctness of static cost analysis

In this section, we use the metatheory of Liquid Haskell (Vazou et al. 2014) to prove that our intrinsic and extrinsic methods of cost analysis are correct.

5.5.1 Metatheory of Liquid Haskell

Figure 5.3 summarises the syntax and operational semantics of 'Lambda-U', λ^U, which is the core language used to model Liquid Haskell (Vazou et al. 2014). The language λ^U includes constants, abstractions, applications, **let** and **case** statements, and datatypes. Its operational semantics is defined as a contextual, small-step, call-by-name relation $\hookrightarrow$ whose reflective, transitive closure is denoted by $\hookrightarrow^\star$.

Constants

Constants applied to values are reduced using the primitive constant operation $c\ v \hookrightarrow \delta(c, v)$. For example, consider $(=)$, the primitive equality operator on integers. In this instance, $\delta(=, n) = (=_n)$, where $\delta(=_n, m)$ equals *true* if and only if m is the same as n.

Types

The basic types in λ^U are integers, booleans, and type constructors. Types are either refinement types of the form $\{\ v : B \mid e\ \}$ where the basic type B, captured by the variable v, is refined by the boolean expression e; or dependent function types of the form $x : \tau_x \to \tau$, where the input x has the type τ_x and the result type, τ, may refer to the binder x.

Denotations

Each type τ denotes a *set* of expressions $[\![\tau]\!]$, defined by the dynamic semantics of Liquid Haskell, given in (Vazou et al. 2014). Let $\lfloor\tau\rfloor$ be the type obtained by erasing all refinements from τ and $e : \lfloor\tau\rfloor$ be the standard typing relation for the typed lambda-calculus. Then, the denotation of types is defined as follows:

$$
\begin{aligned}
[\![\{\ x : B \mid e_r\ \}]\!] &= \{\ e \mid e : B, \text{ if } e \hookrightarrow^\star v \text{ then } e_r[v/x] \hookrightarrow^\star \mathit{true}\ \} \\
[\![\ x : \tau_x \to \tau\ ]\!] &= \{\ e \mid e : \lfloor x : \tau_x \to \tau\rfloor, \forall e_x \in [\![\tau_x]\!]\,.\, e\ e_x \in [\![\,\tau[e_x/x]\,]\!]\ \}
\end{aligned}
$$

Syntactic typing

The typing judgement $\Gamma \vdash e :: \tau$ decides syntactically if e is a member of τ's denotation using the environment Γ that maps variables to their types:

$$\Gamma = x_1 : \tau_1,\ \ldots,\ x_n : \tau_n$$

To analyse resource usage in λ^U, we do not need to modify the typing rules of the language, which are given in (Vazou et al. 2014). Instead, we can use λ^U constants to encode *Tick*'s annotation functions. This approach corresponds to our implementation, as

we have defined a library for cost analysis that builds on top of Liquid Haskell, without changing the underlying behaviour of the system.

To type a λ^U constant c, we use the meta-function $\mathrm{Ty}(c)$ that returns c's type:

$$\frac{}{\Gamma \ \vdash \ c \ :: \ \mathrm{Ty}(c)} \ \text{T-Con}$$

To ensure soundness, $\mathrm{Ty}(c)$ must satisfy denotational inclusion, that is, for each c, $c \ \in [\![\mathrm{Ty}(c)]\!]$. For example, the following definitions ensure that this is true:

$$\begin{aligned} \mathrm{Ty}(3) &= \{ \ v : \mathit{Int} \ \mid \ v \ = \ 3 \ \} \\ \mathrm{Ty}(+) &= x : \mathit{Int} \ \rightarrow \ y : \mathit{Int} \ \rightarrow \ \{ \ v : \mathit{Int} \ \mid \ v \ = \ x \ + \ y \ \} \end{aligned}$$

Soundness of λ^U

The soundness of λ^U proves that if each constant belongs to the denotation of its assumed type, then syntactic typing implies denotational inclusion:

Theorem 1 (Soundness of λ^U)

If for all c, $c \ \in [\![\mathit{Ty}(c)]\!]$, then $\emptyset \ \vdash \ e :: \tau$ implies $e \ \in [\![\tau]\!]$.

5.5.2 Correctness of cost analysis

As λ^U contains type constructors, data constructors, and constants, but does not support type polymorphism, we formalise our approach by defining the *Tick* datatype and a number of its annotation functions as a type family, where each function is a λ^U constant. The correctness of our cost analysis is then simply a corollary of the soundness of λ^U.

The *Tick* datatype

For each type τ, we define a datatype Tick_τ with a single constructor: $\mathit{Tick}_\tau :: \mathit{Int} \rightarrow \tau \rightarrow \mathit{Tick}_\tau$. Tick_τ data constructors should *not* be used directly. Instead, each Tick_τ datatype should be accessed implicitly using the constants defined below.

Resource annotations

We define the following annotation functions from section 5.3.3 as constants in λ^U: $return_\tau$, $bind_{\tau_1,\tau_2}$, $step_\tau$, $tcost_\tau$, $tval_\tau$ for all types τ, τ_1, and τ_2. In turn, we use λ^U to define the types and (type-specific) bodies of each constant just as in section 5.3.3. Given that Liquid Haskell type-checks our previous definitions, it must be true that $c \in [\![\mathrm{Ty}(c)]\!]$ for each constant $return_\tau$, $bind_{\tau_1,\tau_2}$, $step_\tau$, and so on. Therefore, these constants can be used *safely* in λ^U while preserving soundness.

Safe expressions

Recall from section 5.3.4 the following restrictions on annotated expressions, required in order to correctly analyse their resource usage: firstly, expressions should not be defined using *tval* or *tcost*; secondly, expressions should not perform case analysis on the *Tick* data constructor. We formalise these restrictions by defining a safety predicate on λ^U expressions:

Definition 1 (Safety)

A λ^U expression e is safe if and only if:

- *$e : \tau$, that is, e is typeable;*
- *e's body is not defined in terms of any $tval_\tau$ or $tcost_\tau$ constants;*
- *e does not perform case analysis on any $Tick_\tau$ data constructors.*

Execution cost

Consider a safe, terminating function $f :: x : \tau_x \to Tick_\tau$. We define the execution cost of f on an input $e_x :: \tau_x$ to be the index of the returned value. In other words, the execution cost of $f\ e_x$ is i where $f\ e_x \hookrightarrow^\star Tick_\tau\ i\ v$. As f does not directly modify any $Tick_\tau$ datatypes, all resource consumptions or productions via applications of $step_\tau$ in f's definition accumulate in the cost i of the final value, $Tick_\tau\ i\ v$.

Static cost analysis

Finally, we use the soundness of λ^U to prove that the library's intrinsic and extrinsic approaches to analysing resource usage are both sound:

Theorem 2 (Soundness of cost analysis)

Let $p :: Int \to Bool$ *be a predicate over the integers and* $f :: x : \tau_x \to Tick_\tau$ *a safe, total, and terminating function.*

- ***Intrinsic cost analysis*** *If* $\emptyset \vdash e_f :: x : \tau_x \to \{\, t : Tick_\tau \mid p\ (tcost_\tau\ t) \,\}$*, then for all* $e_x \in [\![\tau_x]\!]$*,* $e_f\ e_x \hookrightarrow^\star Tick_\tau\ i\ v$ *and* $p\ i \hookrightarrow^\star true$*.*

- ***Extrinsic cost analysis*** *If* $\emptyset \vdash e :: x : \tau_x \to \{\, w : \tau \mid p\ (tcost_\tau\ (f\ x)) \,\}$*, then for all* $e_x \in [\![\tau_x]\!]$*,* $f\ e_x \hookrightarrow^\star Tick_\tau\ i\ v$ *and* $p\ i \hookrightarrow^\star true$*.*

The proof of this theorem follows immediately from the soundness of the core language λ^U, the denotations of dependent function types, and the definition of $tcost_\tau$.

Other annotations

Theorem 2 proves that the library's cost analysis is consistent for expressions defined using *return*, ($\gg=$), and *step*. However, the *RTick* module provides many more annotation functions, for example, *pure* and ($<*>$) introduced in section 5.3.3. Nonetheless, all such functions can be defined using *return*, ($\gg=$), and *step*: a proof of this fact can be found on the library's GitHub page (Handley and Vazou 2019). Thus, we tacitly extend theorem 2 to include expressions defined using any of the functions provided by the *RTick* module.

5.6 Discussion

The work in this chapter has been strongly influenced by Danielsson's (2008) lightweight framework for cost analysis in Agda. This library is based on the *Thunk* datatype (and shares the same name), which is indexed with a dependent type used to measure the time complexity of purely functional algorithms and data structures in the style of Okasaki

(1999). Our *Tick* datatype is comparable to *Thunk* but captures abstract resource usage, for example, recursive calls and thunk allocations, at the value-level.

A primary goal of this work was to formalise improvement proofs from chapter 4 within the language of Haskell. This led to our use of refinement types (Vazou 2016) and relational cost analysis (Çiçek et al. 2017). A notable distinction between our approach and that of Danielsson (2008) is that whereas our library supports both unary and relational cost analysis, Danielsson's library only supports the unary variant. As such, the improvement proofs underpinning sections 5.4.3 and 5.4.4 cannot be formalised using the *Thunk* library. Conversely, we have re-implemented all of the examples presented in (Danielsson 2008).

Finally, much of *Thunk*'s analysis requires basic equality proofs because Agda does not automatically prove arithmetic equalities. In contrast, our use of Liquid Types allows us to delegate all linear arithmetic necessary for our cost analysis to an SMT solver.

Another relevant Agda implementation is the *AoPA* library (Mu, Ko, and Jansson 2009), which encodes relational program derivations. As we noted in the previous chapter, this library does appear to support a form of inequational reasoning. Hence, we believe it could be used to formalise proofs of improvement, perhaps using relational cost analysis similarly to our work in this chapter. Given that Agda (like Haskell) has a call-by-need semantics, it may afford a more natural encoding for proofs of improvement à la Moran and Sands (see chapter 4) in contrast to our approach, whereby laziness must be modelled explicitly.

Indexed types have been widely used for resource analysis. Crary and Weirich (2000) index the type of functions to compute the number of recursive calls required by their executions. Sized types (Hughes, Pareto, and Sabry 1996; Vasconcelos and Hammond 2003), which index types with natural numbers that denote the size of their values, have also been used to analyse runtimes. However, none of these approaches can express correctness properties, which as we have seen, allow for a more precise analysis.

Recent work (McCarthy et al. 2017; Wang, Wang, and Chlipala 2017) combines indexed types with functional correctness. McCarthy et al. (2017) develop a Coq library that uses a monad indexed by a predicate to measure runtimes. The approach is comparable to Danielsson's (2008), however the predicate is used to express invariants of data structures. This enables more complex case studies (such as Okasaki's Braun Trees) to be examined.

Another distinction is that cost annotations can be automatically inserted, and then erased when code is extracted. A method for automated annotation is part of our future work. Similarly to (Danielsson 2008), relational cost analysis is not supported.

TiML (Wang, Wang, and Chlipala 2017) indexes the types of functions with their time bounds. A significant feature of this system is that it provides automated support for solving recurrence relations, by heuristically matching against cases of the Master Theorem. In comparison, we use extrinsic proofs to manually derive time complexity theorems. Similarly to our approach, TiML supports sophisticated invariants, however, they are only exploited for the purposes of cost analysis. Our library on the other hand uses invariants to simultaneously reason about correctness and resource usage.

More generally, existing cost analyses based on indexed types (Xu and Pfenning 1999) and those using dependent types in languages such as Coq (Bertot and Castéran 2013) and Agda (Norell 2008) are also capable of encoding correctness properties to aid their analysis.

Automatic Amortized Resource Analysis (AARA) (Hofmann and Jost 2003) aims to automatically derive amortised bounds on execution cost. This is achieved using a type system that generates resource-specific inequalities to be solved by a linear programming solver. The initial system (Hofmann and Jost 2003) supports linear bounds on monomorphic, first-order programs. This has since been generalised to incorporate polynomial bounds (Hoffmann, Aehlig, and Hofmann 2011; Hoffmann, Aehlig, and Hofmann 2012), higher-order functions (Jost et al. 2010), parallelism (Hoffmann and Shao 2015), and, most recently, a Haskell-like lazy semantics (Jost et al. 2017). As AARA focuses on automatically inferring bounds, its analysis is often less precise than ours. In particular, our library's extrinsic cost analysis can notionally compute resource bounds of any kind: examples of polynomial, logarithmic, and polylogarithmic bounds appear throughout this chapter. In comparisons, AARA is (at best) restricted to polynomial bounds, though such bounds can be automatically inferred. Correctness invariants are not supported by AARA.

RelCost (Çiçek et al. 2017; Çiçek et al. 2017) is a refinement type-and-effect system for both unary and relational cost analysis. The main idea is to reason about structurally related expressions as much as possible in order to calculate precise resource bounds via relational cost analysis. When programs or inputs are not structurally related, the system

reverts back to unary cost analysis, which is comparable to (Wang, Wang, and Chlipala 2017). This is achieved using two 'modes' of typing: one for similar expressions and one for unrelated expressions. Liquid Haskell only supports one mode of typing, nevertheless, our library fully supports relational cost analysis by way of extrinsic theorems. In fact, Liquid Haskell's refinement type system is more expressive than that of *RelCost*, allowing us to consider additional case studies, for example, those in (Radiček et al. 2018).

BiRelCost (Çiçek et al. 2019) is a bidirectional type checker for *RelCost*, implemented in OCaml. This system, which appears to be the first of its kind, is able to type check all of the examples presented in (Çiçek et al. 2017), and does so automatically while only requiring minimal annotations from the user. However, the implementation is incomplete and currently relies on example-driven heuristics to avoid nondeterminism in its type checking process. Nondeterminism (and completeness) is not a concern for our system, but we have seen throughout the article that users are often required to provide manual proofs of resource usage, specifically for our extrinsic approach. Fundamentally, we see this as a compromise between expressiveness and automation.

Radiček et al. (2018) develop theoretical frameworks for unary and relational cost analysis, implemented in relational higher-order logic (RHOL). Their operational model includes a monad used to encapsulate expressions with cost, much like our *Tick* datatype, which shares the same monadic implementation. Similarly to our approach, the frameworks can express correctness properties that allow for more precise analysis. In fact, Aguirre et al. (2017) show that relational logics are as expressive as HOL, which is in turn as expressive as Liquid Haskell (Vazou et al. 2017). The authors of (Radiček et al. 2018) note that the use of a cost monad "syntactically separates reasoning about costs from reasoning about functional properties, thus improving clarity in proofs". From our experience, reasoning independently about correctness and resource usage (using *tval* and *tcost*) can indeed simplify steps of (in)equational reasoning, especially in the latter case. On the other hand, we have also demonstrated that reasoning about both simultaneously can be useful, for example, when analysing higher-order optimisations such as map fusion.

Madhavan, Kulal, and Kuncak (2017) present a system that can verify resource bounds for a higher-order functional language with a call-by-need semantics, developed in Scala.

As with our library, users must specify desired resource bounds to be verified by the system. However, such bounds are so-called templates, which may contain 'numerical holes' that are automatically inferred. During this verification process, programs are transformed to make their resource usage explicit. In particular, the forcing of thunks is made explicit as per our *pay* function. It is worth noting that this work contains examples involving logarithmic bounds (as well as polynomial); this appears to be fairly uncommon in the literature.

Knoth et al. (2019) present a method for *synthesizing* recursive programs that satisfy refinement type specifications: both functional specifications and symbolic resource bounds. This approach is centred on an expressive type system that combines polymorphic refinement types with potential resource annotations in the style of automatic amortized resource analysis (AARA). Recent work (Knoth et al. 2020) on *Liquid Resource Types* extends the notion of Liquid Types (Rondon, Kawaguci, and Jhala 2008) in a similar fashion, in order to allow for automatic verification of resource usage. Synthesis of Haskell programs with bounded resource usage could be possible if Liquid Haskell is made compatible with Liquid Resource Types: currently it utilises Liquid Types.

Improvement theory (Moran and Sands 1999) inspired our notions of improvement and quantified improvement. As introduced in the previous chapter, Sands (1995) defined improvements as a semantic approach to relational cost analysis. However, improvements in this context only guarantee that one program uses no more resources than another. In this work, we have extended this notion to quantify such guarantees.

Our notion of quantified improvement is modest for two reasons. Firstly, as we mentioned in section 5.3.3, the relation $\succcurlyeq n \Longrightarrow$ is not contextually defined, unlike $\trianglerighteq$. This means that a proof showing that $M \succcurlyeq n \Longrightarrow N$ does not indicate that N is more performant than M in all program contexts, unlike a proof showing that $M \trianglerighteq N$. Secondly, as our library is implemented on top of Liquid Haskell, it does not, by default, account for lazy evaluation, that is, non-strictness plus sharing. On the contrary, our library's default analysis (that is, on standard Haskell datatypes) assumes that functions are monolithic and overlooks memoisation. In contrast, Sestoft's operatioal model (see figure 4.3) utilised by improvement theory provides a natural semantics for lazy evaluation.

For the reasons highlighted above, we describe our library's cost analysis as worst-case.

A more rigorous account of Haskell's on-demand evaluation *within Liquid Haskell* would likely require modifying the system itself. From our experience, it appears that Liquid Haskell would need to be instrumented to track reduction steps in its own operational semantics (Vazou et al. 2014). Given the effort required in ensuring refinement types are sound for proving correctness properties under lazy evaluation, this does not appear to be straightforward. If successful, such an implementation would remove the need for our safety predicate (see section 5.5), but would mean that only one resource could be analysed.

If Liquid Haskell's operational semantics could be instrumented, then it may be possible to devise a corresponding tick algebra (Moran and Sands 1999) for λ^U. Recall from section 5.5 that λ^U is both contextual and small-step. However, it is a call-by-name relation. Hence, this approach would still leave a gap between costs predicted by λ^U and actual cost, but given that Haskell does not yet have a formal call-by-need operational semantics, it is difficult to quantify how big a gap this would be. Nonetheless, it would certainly be smaller in comparison to our library's default analysis.

A disadvantage of instrumenting an operational semantics to track a particular resource is that it lacks generality. For example, recall from section 4.3 that improvement theory (1999) is tied to a specific language, semantics, and cost model. As such, it must be reworked for any new combination of these factors. This has been achieved, for example, in (Gustavsson and Sands 1999), however, the new theory was built from scratch, which makes it difficult to see how these fundamentally different approaches can be integrated into a common framework. Although our library's relational cost analysis is modest, it uniformly caters for different notions of resource usage.

The style of inequational reasoning implemented by our library is notably different from that which is supported by the Unie system of chapter 4. Program contexts aside, users of our library must construct steps of reasoning manually, by defining sequences of expressions—each of which is the result of a transformation applied to the previous expression. Hence, we see that users can freely choose which transformations to apply, and moreover, how to apply them. Proofs constructed in this manner can require additional user effort in comparison to those constructed using Unie. This is because the Unie system implements a specific set of transformation rules and governs when and where each can be

correctly applied. Furthermore, the system applies such transformations on the user's behalf and hence steps of reasoning are mechanised. On the other hand, inequational calculations in Liquid Haskell are formal, whereas those derived using Unie are only semi-formal. This begs us to question whether both systems can be combined.

To extend the work presented in this chapter, we consider four separate avenues. Firstly, in order to make intrinsic cost analysis fully automated, metaprogramming can be used to automatically add code annotations prior to analysis, and furthermore, remove cost annotations post analysis. While implementing the AutoBench system (chapter 3), we prototyped a system for automatically uncovering space leaks using the *GHC-Heap-View* package (Breitner and Felsing 2012). In doing so, we were able to add annotations to Haskell's Core language using a GHC source plugin. Pickering, Wu, and Németh (2019) have recently published a guide to writing source plugins, which can be used for guidance.

Secondly, our intrinsic cost analysis can be made more applicable by incorporating solutions to recurrence relations. As with the TiML language (Wang, Wang, and Chlipala 2017), this could initially be implemented by hard-coding specific cases of the Master Theorem. Heuristics could then be used to determine which case to match against. The TiML language also supports Big-O notation, which allows users to prove that a program has, for example, a quadratic worst-case time complexity, rather than by expressing a bound of the form $a_0 + a_1 x + a_2 x^2$. Our experience with the AutoBench system suggests that abstracting away the specific details (a_0, a_1, and a_2) in favour of higher-level analysis is more useful in practice. Hence, supporting Big-O notation in this setting would also be advantageous.

Thirdly, to cater for further real-world examples, cost analysis of impure Haskell code should be supported by the library. We have made some progress on this front, by reimplementing the *Tick* datatype as a monad transformer. However, as yet Liquid Haskell does not provide support for arbitrary monads or monad transformers.

Finally, for more accurate predictions of hardware costs, our analysis should account for garbage collection. This would require a separate 'points-to' analysis for Haskell, which does not currently exist. However, combining the results of such an analysis with our system would be feasible because it supports reasoning about any notion of resource.

5.7 Conclusion

In this chapter, we have demonstrated how refinement types can be used to reason in a precise manner about the execution cost of programs. More concretely, we have developed a Liquid Haskell library that can be used to analyse the resource usage of pure Haskell programs. Furthermore, by surveying a wide range of examples from the existing literature, we have shown how Liquid Haskell's existing support for correctness verification can be utilised to simplify our cost analysis as well as improve its accuracy.

At the start of this chapter, we set out to combine advantages of the AutoBench (chapter 3) and Unie (chapter 4) systems. Specifically, we sought to reason about the efficiency of Haskell programs within the language itself, and provide formal guarantees in the form of proofs as opposed to informal guarantees based on empirical analysis. Our resulting library achieves both of these aims, enabling users to formalise inequational reasoning in a style that is highly comparable to the pen-and-paper calculation used by ordinary Haskell programmers. In addition, our intrinsic cost analysis allows unary efficiency properties to be verified automatically, which is yet another advantage of the AutoBench system.

Previously, we highlighted that inequational reasoning in Liquid Haskell is more verbose than when reasoning with the aid of the Unie system. Despite this, our library supports more coarse-grained cost analysis than improvement theory, which may often be more useful in practice. For example, recall that the improvement property $slowRev \trianglerighteq fastRev$ does *not* hold, given than $fastRev\ xs$ requires more evaluation steps when xs is empty. In contrast, by capturing resource usage at a higher level of abstraction, namely recursive calls, we have proved that $slowRev\ xs >\simeq> fastRev\ xs$, which shows that the linear-time reverse function improves its quadratic counterpart in a more pragmatic sense.

Chapter 6

Conclusion

This final chapter is a reflection on the work presented in this thesis. In particular, we summarise the main contributions of each chapter to review our achievements, and conclude by discussing a number of possible avenues for further work.

6.1 Summary

In this thesis, we have studied three different methods for reasoning about the efficiency of Haskell programs at three different levels of formality. At each level, we were inspired by existing work on reasoning about program correctness, and aimed to bridge the gap to bring about a similar approach for addressing questions of efficiency. To ensure the practical applicability of our work, we implemented each approach in a new Haskell system, building upon tools used to reason about program correctness, and applied it to a range of case studies from the literature. More specifically, we have made the following contributions:

The AutoBench system

Chapter 3 was concerned with testing. In this chapter, we combined ideas from property-based testing, microbenchmarking, and statistical analysis to develop a simple means for comparing the time performance of Haskell programs. In doing so, we combined two popular systems, namely QuickCheck and Criterion, to give a lightweight, fully automated system that can be used by everyday programmers. Furthermore, we devised a custom

algorithm for approximating empirical time complexity based on linear regression analysis and demonstrated its effectiveness on a number of occasions. The latest version of AutoBench supports generic, sized, random data generation and polyvariadic benchmarking. Both notions were explored in detail: in the former case, the underlying theory—based on the solutions to Diophantine equations—was discussed. The applicability of AutoBench was exemplified in a number of case studies taken from the Haskell programming literature, including an erroneous implementation of the Sieve of Eratosthenes. In this instance, we demonstrated how the system can be used to uncover operational program errors.

The Unie system

Chapter 4 was concerned with semi-formal reasoning. In this chapter, we presented the design of an inequational reasoning assistant called Unie, which provides mechanical support for semi-formal proofs of program improvement. In doing so, we highlighted a number of difficulties in manually constructing such proofs and described how our system addresses these challenges. We illustrated the applicability of our system by verifying a range of results from the literature. In particular, we have mechanised all proofs in (Hackett and Hutton 2014), including the proof of the worker/wrapper improvement theorem, which relates to a general-purpose optimisation technique. We have also mechanically verified a number of proofs in Moran and Sands' original paper (1999) on improvement theory.

The RTick library

Chapter 5 was concerned with formal reasoning. In this chapter, we demonstrated how refinement types can be used to reason in a precise manner about execution cost. More concretely, we developed a Liquid Haskell library that can be used to formally analyse the resource usage of pure Haskell programs. By surveying a range of case studies from the literature, including all the examples presented in (Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2018), we demonstrated not only how refinement types can be used for the purpose of static cost analysis, but also gave evidence that this can be done easily in Liquid Haskell. In addition to cost analysis, we introduced a number of proof combinators that

support reasoning about correctness and efficiency properties in a combined, uniform manner. We exemplified such reasoning by proving that the well-known map fusion technique is an optimisation, and by deriving an optimised-by-construction list-reversing function. We proved the correctness of our analysis using the metatheory of Liquid Haskell.

Finally, we note that the source code for all of the systems and all of the examples discussed throughout this thesis is freely available online at the following addresses (Handley 2019; Handley 2018; Handley and Vazou 2019).

6.2 Further work

The AutoBench system

Throughout this thesis, we have focussed primarily on time performance. Nonetheless, space performance is often just as important. Thus, it seems fitting that the AutoBench system be extended to provide both time and space usage comparisons. As discussed previously in section 3.5.2, the Criterion library (O'Sullivan 2014a) used by AutoBench is capable of measuring total bytes allocated and total number of garbage collections. An alternative option is to use the Weigh library (Done 2016), which has a similar API to Criterion, and hence would fit well with AutoBench's existing implementation. Weigh is capable of measuring total bytes allocated, total number of garbage collections, total amount of live data on the heap, and maximum residency memory in use.

In order to integrate AutoBench's performance testing into existing build/deployment tools such as Travis (2019), the system would benefit from a domain-specific language (DSL) for specifying efficiency properties. This could be comparable to QuickCheck's (2000) language for specifying correctness properties, as discussed in section 3.3.1. We have made some initial progress on this, by designing a prototype DSL based on the notion of *orders* from Big-O complexity theory. Furthermore, we have described how the semantics of this language can be mapped to AutoBench's empirical complexity analysis. Despite our initial investigation demonstrating that this approach complements AutoBench's existing analysis, further work is required in order to determine how effective it would be in practice. The

source code for our prototype DSL can be found online (Handley 2019).

AutoBench's performance analysis currently only supports two-dimensional data. In practice, this amounts to comparing the size of a program's first input against a given performance indicator. Despite this, the latest version of the system allows for testing programs with multiple inputs (section 3.3.3). Hence, there is somewhat of a mismatch between AutoBench's 'front end' and 'back end'. To address this, the analysis and visualisation components of the system could be extended. In the former case, we previously discussed multiple linear regression analysis (section 3.5.3), but this requires further consideration. In the latter case, alternative ways of visualising performance results could be incorporated, including heat maps and/or three-dimensional graphs. Useful methods for visualising data with three or more degrees of freedom are less obvious.

Other avenues for further work include supporting real-world test data and implementing alternative regression algorithms, for example, the LASSO method (section 3.5.1). This could involve using R's standard LASSO implementation via the HaskellR library (Boespflug et al. 2014), which enables Haskell and R code to interoperate via quasiquotation.

The Unie system

To improve the Unie system, we would like to investigate higher-level support for navigating through terms, and for applying transformations at specific locations during improvement proofs. Profunctor optics (Pickering, Gibbons, and Wu 2017) provides a framework for accessing, modifying, and traversing elements of data structures in a manner that is both modular and composable, and has recently been the subject of many new implementations, for example, the Generic-Lens package (Kiss 2017). Hence, we recommend this approach. We note that this addition would be best implemented while simultaneously replacing the underlying Kure system for a modern data-generic programming library, for example, (Kiss, Pickering, and Wu 2018). See section 4.7 for more details.

It would be interesting to see if Coq-style proof tactics could be integrated into Unie, to capture and express common recipes for improvement such as the one discussed at the end of section 4.2. In this instance, we are encouraged by the fact that, on many occasions,

rules from Moran and Sands' tick algebra (section 4.3.4) can be correctly applied to a term in at most one way. In consequence, a 'proof search' of some kind may prove to be computationally viable, but such an approach requires a thorough investigation.

A current limitation of the Unie system is that its reasoning is semi-formal. To address this, the system's output could be translated into a proof object to be independently verified by a proof assistant such as Coq or Agda, to provide formal guarantees of correctness. This may require interfacing with, for example, the AoPA library (Mu, Ko, and Jansson 2009). An alternative approach would be to define a corresponding tick algebra for the operational semantics of Liquid Haskell (Vazou et al. 2014), and then combine Unie's mechanised support for inequational reasoning with Liquid Haskell's refinement types for formal verification. An advantage of this approach is that it could be fully Haskell-based.

The RTick library

In order to make the RTick library's unary cost analysis fully automated, metaprogramming could be used to automatically add code annotation prior to analysis (and possibly remove cost annotations post analysis). While implementing the AutoBench system (chapter 3), we prototyped a system for automatically uncovering space leaks using the GHC-Heap-View package (Breitner and Felsing 2012). In doing so, we were able to add annotations to Haskell's Core language using a GHC source plugin. Pickering, Wu, and Németh (2019) have recently published a guide to writing GHC source plugins.

The library's intrinsic cost analysis can be made more applicable by incorporating solutions to recurrence relations. As with the TiML language (Wang, Wang, and Chlipala 2017), this could be implemented by hard-coding specific cases of the Master Theorem. Heuristics could then be used to determine which case to match against. The TiML language also supports Big-O notation, which allows users to prove that a program has, for example, quadratic time complexity, rather than by expressing a bound of the form $a_0 + a_1x + a_2x^2$. Our experience with the AutoBench system suggests that abstracting away the specific details (a_0, a_1, and a_2) in favour of a higher level analysis is more useful in practice. Hence, supporting Big-O notation would be advantageous.

To cater for further real-world examples, cost analysis of impure code should be supported by the library. We have made some preliminary progress on this front by reimplementing the library's *Tick* datatype as a monad transformer. However, as yet Liquid Haskell does not provide support for arbitrary monads. In fact, *Tick*'s applicative and monad methods (defined in section 5.3) are currently implemented as stand-alone functions. The initial focus of this extension thus might be on improving Liquid Haskell's support for type classes.

Bibliography

Abramsky, Samson (1990). "The Lazy λ-Calculus". *Proceedings of Research Topics in Functional Programming.*

Adams, Michael D., Andrew Farmer, and José P. Magalhães (2014). "Optimizing SYB is Easy!" *Proceedings of Workshop on Partial Evaluation and Program Manipulation.*

Adams, Michael D., Andrew Farmer, and José P. Magalhães (2015). "Optimizing SYB Traversals is Easy!" *Science of Computer Programming.*

Aguirre, Alejandro et al. (2017). "A Relational Logic for Higher-Order Programs". *Proceedings of International Conference on Functional Programming.*

Antoy, Sergio, Rachid Echahed, and Michael Hanus (2000). "A Needed Narrowing Strategy". *Journal of the ACM.*

Ariola, Zena M. et al. (1995). "A Call-By-Need Lambda Calculus". *Proceedings of Symposium on Principles of Programming Languages.*

Arts, Thomas et al. (2006). "Testing Telecoms Software with Quviq QuickCheck". *Proceedings of Erlang Workshop.*

Aspinall, David et al. (2007). "A Program Logic for Resources". *Theoretical Computer Science.*

Atkey, Robert (2010). "Amortised Resource Analysis with Separation Logic". *Proceedings of European Symposium on Programming.*

Barr, Earl T. et al. (2014). "The Oracle Problem in Software Testing: A Survey". *IEEE Transactions on Software Engineering.*

Bernardy, Jean-Philippe, Patrik Jansson, and Koen Claessen (2010). "Testing Polymorphic Properties". *Proceedings of European Symposium on Programming.*

Bershad, Brian N., Richard P. Draves, and Alessandro Forin (1992). "Using Microbenchmarks to Evaluate System Performance". *Proceedings of Workshop on Workstation Operating Systems.*

Bertolino, Antonia (2007). "Software Testing Research: Achievements, Challenges, Dreams". *Proceedings of 2007 Future of Software Engineering*.

Bertot, Yves and Pierre Castéran (2013). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer.

Bird, Richard (1988). *Lectures on Constructive Functional Programming*. Oxford University Computing Laboratory.

Bjerner, Bror and Sören Holmström (1989). "A Composition Approach to Time Analysis of First Order Lazy Functional Programs". *Proceedings of Conference on Functional Programming Languages and Computer Architecture*.

Boespflug, Mathieu et al. (2014). "Project H: Programming R in Haskell". Unpublished draft.

Bornat, Richard and Bernard Sufrin (1997). "Jape: A Calculator for Animating Proof-On-Paper". *Proceedings of International Conference on Automated Deduction*.

Bornat, Richard and Bernard Sufrin (1999). "Animating Formal Proof at the Surface: The Jape Proof Calculator". *The Computer Journal*.

Bowen, Jonathan and Victoria Stavridou (1993). "Safety-Critical Systems, Formal Methods and Standards". *Software Engineering Journal*.

Brady, Edwin (2013). "Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation". *Journal of Functional Programming*.

Brady, Edwin and Kevin Hammond (2005). "A Dependently Typed Framework for Static Analysis of Program Execution Costs". *Proceedings of Symposium on Implementation and Application of Functional Languages*.

Breitner, Joachim and Dennis Felsing (2012). *The GHC-Heap-View Package*. Available online at: `https://github.com/chrisnc/ghc-heap-view`.

Brown, Neil (2010). *The Progression Package*. Available online at: `https://hackage.haskell.org/package/progression`.

Burstall, Rod M. (1969). "Proving Properties of Programs by Structural Induction". *The Computer Journal*.

Burstall, Rod M. and John Darlington (1977). "A Transformation System for Developing Recursive Programs". *Journal of the ACM*.

Campbell, Brian (2009). "Amortised Memory Analysis Using the Depth of Data Structures". *Proceedings of European Symposium on Programming*.

Cardelli, Luca and Peter Wegner (1985). "On Understanding Types, Data Abstraction, and Polymorphism". *ACM Computing Surveys*.

Christiansen, Jan and Sebastian Fischer (2008). "EasyCheck – Test Data for Free". *Proceedings of International Symposium on Functional and Logic Programming*.

Çiçek, Ezgi (2018). "Relational Cost Analysis". PhD thesis. Saarland University, Saarbrücken, Germany.

Çiçek, Ezgi et al. (2017). "Relational Cost Analysis". *Proceedings of Symposium on Principles of Programming Languages*.

Çiçek, Ezgi et al. (2019). "Bidirectional Type Checking for Relational Properties". *Proceedings of Conference on Programming Language Design and Implementation*.

Claeskens, Gerda and Nils L. Hjort (2008). *Model Selection and Model Averaging*. Tech. rep. Cambridge University Press.

Claessen, Koen (2000). *The QuickCheck Package*. Available online at: `https://hackage.haskell.org/package/QuickCheck`.

Claessen, Koen (2012). "Shrinking and Showing Functions". *Proceedings of International Symposium on Haskell*.

Claessen, Koen, Jonas Duregård, and Michał H. Pałka (2015). "Generating Constrained Random Data with Uniform Distribution". *Journal of Functional Programming*.

Claessen, Koen and John Hughes (2000a). "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". *Proceedings of International Conference on Functional Programming*.

Claessen, Koen and John Hughes (2000b). *The QuickCheck Online Manual*. Available online at: `http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html`.

Claessen, Koen and John Hughes (2002). "Testing Monadic Code with QuickCheck". *Proceedings of Haskell Workshop*.

Claessen, Koen, Nicholas Smallbone, and John Hughes (2010). "QuickSpec: Guessing Formal Specifications Using Testing". *Proceedings of International Conference on Tests and Proofs*.

Copeland, Chris (2017). *The HVX Package*. Available online at: `https://github.com/chrisnc/hvx`.

Coppa, Emilio, Camil Demetrescu, and Irene Finocchi (2012). "Input-Sensitive Profiling". *Proceedings of Conference on Programming Language Design and Implementation*.

Coppa, Emilio et al. (2014). "Estimating the Empirical Cost Function of Routines with Dynamic Workloads". *Proceedings of International Symposium on Code Generation and Optimization*.

Crary, Karl and Stephnie Weirich (2000). "Resource Bound Certification". *Proceedings of Symposium on Principles of Programming Languages*.

Curry, Haskell B. (1934). "Functionality in Combinatory Logic". *Proceedings of National Academy of Sciences of the United States of America*.

Daka, Ermira and Gordon Fraser (2014). "A Survey on Unit Testing Practices and Problems". *Proceedings of International Symposium on Software Reliability Engineering*.

Danielsson, Nils A. (2008). "Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures". *Proceedings of Symposium on Principles of Programming Languages*.

Danielsson, Nils A. et al. (2006). "Fast and Loose Reasoning is Morally Correct". *ACM SIGPLAN Notices*.

Docker, Tim (2006). *The Chart Package*. Available online at: `http://hackage.haskell.org/package/Chart`.

Dolan, Elizabeth D. and Jorge J. Moré (2002). "Benchmarking Optimization Software with Performance Profiles". *Mathematical Programming*.

Done, Chris (2016). *The Weigh Package*. Available online at: `https://hackage.haskell.org/package/weigh`.

Duregård, Jonas, Patrik Jansson, and Meng Wang (2013). "Feat: Functional Enumeration of Algebraic Types". *Proceedings of International Symposium on Haskell*.

Estivill-Castro, Vladmir and Derick Wood (1992). "A Survey of Adaptive Sorting Algorithms". *Proceedings of ACM Computing Surveys*.

Farmer, Andrew (2015). "HERMIT: Mechanized Reasoning During Compilation in the Glasgow Haskell Compiler". PhD thesis. University of Kansas, Kansas, USA.

Farmer, Andrew, Neil Sculthorpe, and Andrew Gill (2015). "Reasoning with the HERMIT: Tool Support for Equational Reasoning on GHC Core Programs". *Proceedings of International Symposium on Haskell*.

Farmer, Andrew, Christian H. zu Siederdissen, and Andrew Gill (2014). "The HERMIT in the Stream". *Proceedings of Workshop on Partial Evaluation and Program Manipulation*.

Farmer, Andrew et al. (2012). "The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs". *Proceedings of International Symposium on Haskell*.

Felleisen, Matthias (1987). "The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages". PhD thesis. Indiana University Bloomington, Indiana, USA.

Felleisen, Matthias and Matthew Flatt (1989). "Programming Languages and Lambda Calculi". Unpublished lecture notes.

Felleisen, Matthias and Robert Hieb (1992). "The Revised Report on the Syntactic Theories of Sequential Control and State". *Theoretical Computer Science*.

Fetscher, Burke et al. (2015). "Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System". *Proceedings of European Symposium on Programming Languages and Systems*.

Foner, Kenneth, Hengchu Zhang, and Leonidas Lampropoulos (2018). "Keep Your Laziness in Check". *Proceedings of International Conference on Functional Programming*.

Fox, John (1997). *Applied Regression Analysis, Linear Models, and Related Methods*. Sage Publications.

GHC Team (2001). *The List Package*. Available online at: `http://hackage.haskell.org/package/base-4.11.1.0/docs/src/Data.OldList.html#sort`.

GHC Team (2007). *The Process Package*. Available online at: `https://hackage.haskell.org/package/process`.

GHC Team (2017). *The GHC API*. Available online at: `https://hackage.haskell.org/package/ghc`.

GHC Team (2019). *The Glasgow Haskell Compiler's User Guide*. Available online at: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/`.

Gibbons, Jeremy (2006). "Datatype-Generic Programming". *Proceedings of International Spring School on Datatype-Generic Programming*.

Gill, Andrew (2006). "Introducing the Haskell Equational Reasoning Assistant". *Proceedings of Haskell Workshop*.

Gill, Andrew and Graham Hutton (2009). "The Worker/Wrapper Transformation". *Journal of Functional Programming*.

Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin (2008). "Grammar-Based Whitebox Fuzzing". *ACM SIGPLAN Notices*.

Godefroid, Patrice, Michael Y. Levin, and David Molnar (2012). "SAGE: Whitebox Fuzzing for Security Testing". *Communications of the ACM*.

Goldsmith, Simon F., Alex S. Aiken, and Daniel S. Wilkerson (2007). "Measuring Empirical Computational Complexity". *Proceedings of European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Grieco, Gustavo, Martín Ceresa, and Pablo Buiras (2016). "QuickFuzz: An Automatic Random Fuzzer for Common File Formats". *ACM SIGPLAN Notices*.

Grieco, Gustavo et al. (2017). "QuickFuzz Testing for Fun and Profit". *Journal of Systems and Software*.

Groote, Jan F. and Frits Vaandrager (1992). "Structured Operational Semantics and Bisimulation as a Congruence". *Information and Computation*.

Gustavsson, Jörgen and David Sands (1999). "A Foundation for Space-Safe Transformations of Call-By-Need Programs". *Electronic Notes in Theoretical Computer Science*.

Guttmann, Walter et al. (2003). "Tool Support for the Interactive Derivation of Formally Correct Functional Programs". *Journal of Universal Computer Science*.

Hackett, Jennifer and Graham Hutton (2014). "Worker/Wrapper/Makes It/Faster". *Proceedings of International Conference on Functional Programming*.

Hackett, Jennifer and Graham Hutton (2018). "Parametric Polymorphism and Operational Improvement". *Proceedings of International Conference on Functional Programming*.

Hackett, Jennifer and Graham Hutton (2019). "Pre-ordered Metric Spaces for Program Improvement". Unpublished draft.

Handley, Martin A.T. (2018). *GitHub Repository for the University of Nottingham Improvement Engine (Unie)*. Available online at: `https://github.com/mathandley/Unie`.

Handley, Martin A.T. (2019). *GitHub Repository for the AutoBench System*. Available online at: `https://github.com/mathandley/AutoBench`.

Handley, Martin A.T. and Graham Hutton (2018a). "AutoBench: Comparing the Time Performance of Haskell Programs". *Proceedings of International Symposium on Haskell*.

Handley, Martin A.T. and Graham Hutton (2018b). "Improving Haskell". *Proceedings of International Symposium on Trends in Functional Programming*.

Handley, Martin A.T. and Niki Vazou (2019). *GitHub Repository for the RTick Library*. Available online at: `https://github.com/mathandley/RTick`.

Handley, Martin A.T., Niki Vazou, and Graham Hutton (2020). "Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell". *Proceedings of Symposium on Principles of Programming Languages*.

Hans, Chris (2009). "Bayesian Lasso Regression". *Biometrika*.

Harper, Robert (2014). "The Structure and Efficiency of Computer Programs". Unpublished draft.

Hennessy, Matthew and Robin Milner (1980). "On Observing Nondeterminism and Concurrency". *Proceedings of Automata, Languages and Programming.*

Hinze, Ralf and Johan Jeuring (2003). "Generic Haskell: Practice and Theory". *Proceedings of Generic Programming.*

Hoerl, Arthur E. and Robert W. Kennard (1970). "Ridge Regression: Biased Estimation for Nonorthogonal Problems". *Proceedings of Technometrics.*

Hoffmann, Jan, Klaus Aehlig, and Martin Hofmann (2011). "Multivariate Amortized Resource Analysis". *ACM SIGPLAN Notices.*

Hoffmann, Jan, Klaus Aehlig, and Martin Hofmann (2012). "Resource Aware ML". *Proceedings of International Conference on Computer Aided Verification.*

Hoffmann, Jan and Zhong Shao (2015). "Automatic Static Cost Analysis for Parallel Programs". *Proceedings of European Symposium on Programming.*

Hofmann, Martin and Steffen Jost (2003). "Static Prediction of Heap Space Usage for First-Order Functional Programs". *ACM SIGPLAN Notices.*

Hughes, John (2007). "QuickCheck Testing for Fun and Profit". *Proceedings of International Symposium on Practical Aspects of Declarative Languages.*

Hughes, John (2016). "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane". *A List of Successes That Can Change the World.*

Hughes, John, Lars Pareto, and Amr Sabry (1996). "Proving the Correctness of Reactive Systems Using Sized Types". *Proceedings of Symposium on Principles of Programming Languages.*

Hutcheson, Graeme D. (2011). "Ordinary Least-Squares Regression". *The SAGE Dictionary of Quantitative Management Research.*

Hutton, Graham (1998). "Fold and Unfold for Program Semantics". *ACM SIGPLAN Notices.*

Hutton, Graham (2016). *Programming in Haskell (Second Edition).* Cambridge University Press.

Hyams, Daniel G. (2019). *CurveExpert.* Available online at: `http://www.curveexpert.net`.

Jansson, Patrik et al. (2006). "Testing Properties of Generic Functions". *Proceedings of Symposium on Implementation and Application of Functional Languages.*

Jay, Barry C. and J.R.B. Cockett (1994). "Shapely Types and Shape Polymorphism". *Proceedings of European Symposium on Programming*.

Jin, Guoliang et al. (2012). "Understanding and Detecting Real-World Performance Bugs". *ACM SIGPLAN Notices*.

Jones, Mark P. (2003). *The Hugs Interpreter*. Available online at: `https://www.haskell.org/hugs`.

Jost, Steffen et al. (2010). "Static Determination of Quantitative Resource Usage for Higher-Order Programs". *ACM SIGPLAN Notices*.

Jost, Steffen et al. (2017). "Type-Based Cost Analysis for Lazy Functional Languages". *Journal of Automated Reasoning*.

Kahn, Gilles (1987). "Natural Semantics". *Proceedings of Symposium on Theoretical Aspects of Computer Science*.

Kiss, Csongor (2017). *The Generic-Lens Package*. Available online at: `https://github.com/chrisnc/generic-lens`.

Kiss, Csongor, Matthew Pickering, and Nicolas Wu (2018). "Generic Deriving of Generic Traversals". *Proceedings of International Conference on Functional Programming*.

Kleene, Stephen C. (1964). *Introduction to Metamathematics*. North Holland Publishing Company.

Klein, Casey et al. (2012). "Run Your Research: On the Effectiveness of Lightweight Mechanization". *Proceedings of Symposium on Principles of Programming Languages*.

Knoth, Tristan et al. (2019). "Resource-Guided Program Synthesis". *Proceedings of Conference on Programming Language Design and Implementation*.

Knoth, Tristan et al. (2020). "Liquid Resource Types". *Proceedings of the ACM on Programming Languages*.

Lämmel, Ralf, Eelco Visser, and Joost Visser (2003). "The Essence of Strategic Programming". Unpublished draft.

Lampropoulos, Leonidas et al. (2017). "Beginner's Luck: a Language for Property-Based Generators". *Proceedings of Symposium on Principles of Programming Languages*.

Lapets, Andrei (2010). "Machine Involvement in Formal Reasoning: Simulated Contexts and an Interface Layer for Formal Verification". Unpublished draft.

Li, Huiqing, Claus Reinke, and Simon Thompson (2003). "Tool Support for Refactoring Functional Programs". *Proceedings of Haskell Workshop*.

Lima, Luís G. et al. (2016). "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language". *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering*.

Löh, Andres (2015). "Applying Type-Level and Generic Programming in Haskell". Unpublished lectures notes.

Madhavan, Ravichandhran, Sumith Kulal, and Viktor Kuncak (2017). "Contract-Based Resource Verification for Higher-Order Functions with Memoization". *ACM SIGPLAN Notices*.

Martì, Daniel (2007). *The Hint Package*. Available online at: `https://hackage.haskell.org/package/hint`.

Martin-Löf, Per (1984). *Intuitionistic Type Theory*. Bibliopolis.

McBride, Conor and James McKinna (2004). "The View from the Left". *Journal of Functional Programming*.

McCarthy, Jay et al. (2017). "A Coq Library for Internal Verification of Running-Times". *Science of Computer Programming*.

McGeoch, Catherine et al. (2002). "Using Finite Experiments to Study Asymptotic Performance". *Experimental Algorithmics*. Springer.

Meertens, Lambert (2004). "Calculating the Sieve of Eratosthenes". *Journal of Functional Programming*.

Mista, Agustín, Alejandro Russo, and John Hughes (2018). "Branching Processes for QuickCheck Generators". *Proceedings of International Symposium on Haskell*.

Mitchell, Neil (2007). "Deriving Generic Functions by Example". *Proceedings of York Doctoral Symposium on Computing*.

Moran, Andrew and David Sands (1999). "Improvement in a Lazy Context: An Operational Theory for Call-By-Need". *Proceedings of Symposium on Principles of Programming Languages*.

Moss, Graeme E. (2000). "Benchmarking Purely Functional Data Structures". PhD thesis. University of York, York, UK.

Moura, Leonardo de and Nikolaj Bjørner (2008). "Z3: An Efficient SMT Solver". *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

Mu, Shin-Cheng, Hsiang-Shang Ko, and Patrik Jansson (2009). "Algebra of Programming in Agda: Dependent Types for Relational Program Derivation". *Journal of Functional Programming*.

Norell, Ulf (2007). "Towards a Practical Programming Language Based on Dependent Type Theory". PhD thesis. Chalmers University of Technology, Göteborg, Sweden.

Norell, Ulf (2008). "Dependently Typed Programming in Agda". *Proceedings of International School on Advanced Functional Programming*.

Okasaki, Chris (1999). *Purely Functional Data Structures*. Cambridge University Press.

O'Keefe, Richard A. (1982). *A Smooth Applicative Merge Sort*. Department of Artificial Intelligence, University of Edinburgh.

O'Neill, Melissa E. (2009). "The Genuine Sieve of Eratosthenes". *Journal of Functional Programming*.

OriginLab (2019). *Origin*. Available online at: `https://www.originlab.com`.

O'Sullivan, Bryan (2014a). *Criterion: Robust Reliable Performance Measurement and Analysis*. Available online at: `http://www.serpentine.com/criterion`.

O'Sullivan, Bryan (2014b). *The Aeson Package*. Available online at: `http://hackage.haskell.org/package/aeson`.

O'Donnell, John and Cordelia Hall (2006). *Discrete Mathematics Using a Computer*. Springer Science and Business Media.

Pałka, Michał H. et al. (2011). "Testing an Optimising Compiler by Generating Random Lambda Terms". *Proceedings of International Workshop on Automation of Software Test*.

Park, David (1981). "Concurrency and Automata on Infinite Sequences". *Proceedings of GI-Conference on Theoretical Computer Science*.

Pickering, Matthew, Jeremy Gibbons, and Nicolas Wu (2017). "Profunctor Optics: Modular Data Accessors". *Art, Science, and Engineering of Programming*.

Pickering, Matthew, Nicolas Wu, and Boldizsár Németh (2019). "Working with Source Plugins". *Proceedings of International Symposium on Haskell*.

Plotkin, Gordon D. (1975). "Call-by-Name, Call-by-Value and the λ-Calculus". *Theoretical Computer Science*.

Plotkin, Gordon D. (1981). "A Structural Approach to Operational Semantics". *Aarhus University Computer Science Department*.

Radiček, Ivan et al. (2018). "Monadic Refinements for Relational Cost Analysis". *Proceedings of Symposium on Principles of Programming Languages*.

Rondon, Patrick M., Ming Kawaguci, and Ranjit Jhala (2008). "Liquid Types". *ACM SIGPLAN Notices*.

Ruehr, Karl F. (1992). "Analytical and Structural Polymorphism Expressed Using Patterns Over Types". PhD thesis. University of Michigan, Michigan, USA.

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad (2008). "Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values". *Proceedings of International Symposium on Haskell*.

Sands, David (1995). "Total Correctness by Local Improvement in Program Transformation". *Proceedings of Symposium on Principles of Programming Languages*.

Sands, David (1997). "Improvement Theory and Its Applications". *Proceedings of Higher Order Operational Techniques in Semantics*.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers.

Schmidt-Schauß, Manfred and David Sabel (2015). "Improvements in a Functional Core Language with Call-By-Need Operational Semantics". *Proceedings of International Symposium on Principles and Practice of Declarative Programming*.

Scott, Dana (1982). "Domains for Denotational Semantics". *Proceedings of International Colloquium on Automata, Languages and Programming*.

Scott, Dana and Christopher Strachey (1971). "Toward a Mathematical Semantics for Computer Languages". *Proceedings of Symposium on Computers and Automata*.

Sculthorpe, Neil, Andrew Farmer, and Andrew Gill (2013). "The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language". *Proceedings of Symposium on Implementation and Application of Functional Languages*.

Sculthorpe, Neil, Nicolas Frisby, and Andrew Gill (2014). "The Kansas University Rewrite Engine". *Journal of Functional Programming*.

Sculthorpe, Neil and Graham Hutton (2014). "Work It, Wrap It, Fix It, Fold It". *Journal of Functional Programming*.

Sestoft, Peter (1997). "Deriving a Lazy Abstract Machine". *Journal of Functional Programming*.

Stump, Aaron (2016). *Verified Functional Programming in Agda*. Morgan & Claypool.

Sutton, Michal, Adam Greene, and Pedram Amini (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education.

Systat Software (2019). *TableCurve 2D*. Available online at: `http://www.sigmaplot.co.uk`.

Thompson, Simon and Huiqing Li (2013). "Refactoring Tools for Functional Languages". *Journal of Functional Programming*.

Travis-CI.org (2019). *Travis CI*. Available online at: `https://travis-ci.org`.

Tullsen, Mark A. (2002). "Path, A Program Transformation System for Haskell". PhD thesis.

Turner, David A. (2004). "Total Functional Programming". *Journal of Universal Computer Science*.

Vasconcelos, Pedro B. (2008). "Space Cost Analysis Using Sized Types". PhD thesis. University of St. Andrews, Fife, Scotland.

Vasconcelos, Pedro B. and Kevin Hammond (2003). "Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs". *Proceedings of Symposium on Implementation and Application of Functional Languages*.

Vazou, Niki (2016). "Liquid Haskell: Haskell as a Theorem Prover". PhD thesis. University of California San Diego, California, USA.

Vazou, Niki, Patrick M. Rondon, and Ranjit Jhala (2013). "Abstract Refinement Types". *Proceedings of European Symposium on Programming*.

Vazou, Niki et al. (2014). "Refinement Types for Haskell". *Proceedings of International Conference on Functional Programming*.

Vazou, Niki et al. (2017). "Refinement Reflection: Complete Verification with SMT". *Proceedings of Symposium on Principles of Programming Languages*.

Vazou, Niki et al. (2018). "Theorem Proving For All: Equational Reasoning in Liquid Haskell". *Proceedings of International Symposium on Haskell*.

Vries, Edsko de and Andres Löh (2014). *The Generics-Sop Package*. Available online at: `http://hackage.haskell.org/package/generics-sop`.

Wadler, Philip (1987). "The Concatenate Vanishes". Appeared as a note on an electronic mailing list.

Wadler, Philip (1988). "Strictness Analysis Aids Time Analysis". *Proceedings of Symposium on Principles of Programming Languages*.

Wadler, Philip (2015). "Propositions As Types". *Communications of the ACM*.

Wang, Peng, Di Wang, and Adam Chlipala (2017). "TiML: A Functional Language for Practical Complexity Analysis with Invariants". *Proceedings of the ACM on Programming Languages*.

Watson, Henry W. and Francis Galton (1875). "On the Probability of the Extinction of Families". *Journal of the Anthropological Institute of Great Britain and Ireland*.

Weyuker, Elaine J. and Filippos I. Vokolos (2000). "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study". *IEEE Transactions on Software Engineering*.

Woodside, Murray, Greg Franks, and Dorina C. Petriu (2007). "The Future of Software Performance Engineering". *Proceedings of Future of Software Engineering*.

Xu, Hongwei and Frank Pfenning (1999). "Dependent Types in Practical Programming". *Proceedings of Symposium on Principles of Programming Languages*.

Xu, Qingsong and Yi-Zeng Liang (2001). "Monte Carlo Cross Validation". *Proceedings of Chemometrics and Intelligent Laboratory Systems*.

Appendices

Appendix A

Additional background

A.1 Program semantics

For completeness, we overview program semantics, which is the field of study that lays the foundations for both equational (section 2.3.1) and inequational reasoning (section 2.3.1). In particular, denotational semantics (section A.1.1) relates expressions to mathematical objects, typically for verifying correctness properties; and operational semantics (section A.1.2) relates expressions to steps of execution, which can be used to verify efficiency properties.

A.1.1 Denotational semantics

Denotational semantics (Schmidt 1986) is concerned with giving mathematical models for programming languages. A denotational semantics gives meaning to a programming language by assigning a mathematical meaning to each of its terms, known as a *denotation*. Each term in the language is thus said to denote a particular mathematical object. A denotational semantics is defined by first selecting a set of mathematical values collectively known as a *semantic domain* (or just a domain), which contains all the possible meanings. Terms are then interpreted by mappings from the language's syntax to this domain.

Formally, a denotational semantics for a language T of syntactic terms comprises a set V of semantic values (that is, V is a semantic domain) and a valuation function $[\![\cdot]\!]$ of type $T \to V$ that maps terms to their meanings as values. As a concrete example, we may consider Hutton's (1998) language of simple arithmetic expressions:

```
data Expr = Val Int | Add Expr Expr
```

This language is built up from integers and addition and has implicit bracketing. For example, the expression $0 + (1 + 2)$ is represented by $Add\ (Val\ 0)\ (Add\ (Val\ 1)\ (Val\ 2))$ of type *Expr*. A denotational semantics for this expression language can be given by taking the set V of semantic values as the Haskell type *Int* of integers, and the valuation function as a recursive evaluation function for expressions, defined as follows:

```
eval :: Expr → Int
eval (Val n)     = n
eval (Add x y) = eval x + eval y
```

The semantics thus assigns the meaning of the term $Add\ (Val\ 0)\ (Add\ (Val\ 1)\ (Val\ 2))$ to the result of $eval\ (Add\ (Val\ 0)\ (Add\ (Val\ 1)\ (Val\ 2)))$, which is 3.

The above valuation function demonstrates a fundamental principle of denotational semantics, which is that denotations must be *compositional*. Compositionality means that the denotation of a compound term is defined purely in terms of the denotation of its subterms. In the case of *eval*, we can see that the meaning of an addition $Add\ x\ y$ is defined in terms of the meaning of its operands x and y, which is precisely the requirement.

The concept of denotational semantics originated in the work of Scott and Strachey, who defined the denotation of a program as a function taking an initial state as input and producing a final state as output (Scott and Strachey 1971). To give denotations to recursively defined programs, Scott proposed working with continuous functions between domains, specifically between complete pointed partial orders (Scott 1982), or CPPOs, which are partial orders with a completeness property that ensures functions always have least fixed-points. Moreover, this algebraic structure also provides an algorithm to calculate such fixed points, in the form of the Kleene fixed-point theorem (Kleene 1964).

Complete pointed partial orders form the basis of most modern denotational semantics for functional languages. For lazy functional languages, such as Haskell, types are represented by CPPOs and functions between types are continuous functions between their corresponding CPPOs. Non-termination is represented by the least element of every CPPO, denoted $\bot$ and pronounced 'bottom', and strictness for continuous functions corresponds

precisely to strictness in the functional programming sense.

Denotational equality

A fundamental requirement for any notion of equality between programs is for it to be a congruence relation. More specifically, an equivalence relation that is compatible with the syntactical constructs of the respective programming language. Such a relation is desirable for two reasons. Firstly, the transitivity property affords the well-known technique of equational reasoning (section 2.3.1), where equalities of the form $t = t'$ can be derived from sequences of intermediate equalities $t = t_0 = t_1 = \cdots = t_n = t'$. Secondly, the compatibility property enables equalities between compound terms of the form $t[s] = t[s']$ to be deduced from equalities between corresponding subterms, that is, $s = s'$. In practice, this is perhaps an even more useful technique of equational reasoning.

In the denotational setting, two terms are considered equal if they are mapped to the same value in the domain. This definition is clearly transitive (and reflexive). Furthermore, assuming compositionality, this definition allows a subterm s of a compound term t to be replaced with another subterm s' of the same denotation without changing the overall meaning of t. As such, a compositional denotational semantics supports the substitution of denotationally equal subterms and is indeed a congruence relation.

Universally being able to substitute 'equals for equals' in this manner is often called *referential transparency*, and is responsible for much interest in functional programming precisely because it admits the techniques of equational reasoning discussed above, which are akin to everyday mathematics (O'Donnell and Hall 2006).

Denotational approaches to program equivalence also lend themselves to *calculational programming* (Bird 1988), in which algebraic techniques are used to derive programs that satisfy high-level specifications. These techniques are frequently used when applying program transformations, as a resultant program can often be calculated from one or more preconditions that ensure the transformation's correctness. A notable example appearing in chapter 4 is the worker/wrapper transformation (Gill and Hutton 2009).

A.1.2 Operational semantics

In contrast to denotational semantics, the purpose of operational semantics is to describe not what a program *is* in a mathematical sense but to describe *how* a computation is performed. To this end, an operational semantics gives a meaning to a program as the sequence of steps that must be performed to calculate its result. There are two main approaches to operational semantics: *structural operational semantics*, initially developed by Plotkin (1981); and *natural semantics*, first introduced by Kahn (1987). We focus on the former approach as it is most relevant to the work of this thesis.

Structural operational semantics

Structural operational semantics (SOS), a form of small-step semantics, describes how a program is executed in individual steps. Usually, it is expressed as a transition relation that captures the execution steps taken by an appropriate transition system (Plotkin 1981).

Formally, an SOS for a language T of syntactic terms comprises a set S of states and a transition relation $\rightarrow \subseteq S \times S$ that relates states to all other states reachable by performing a single execution step. We write $s \rightarrow s'$ to mean $\langle s, s' \rangle \in \rightarrow$ and say there is a *transition* from state s to state s'. Instead of enumerating every possible transition between states, the transition relation is typically defined using rules of inference.

To make this concrete, we return to the example of basic arithmetic expressions from section A.1.1. In doing so, we can give arithmetic expressions a simple operational semantics by taking S as the Haskell type *Expr* of expressions and by specifying the transition relation $\rightarrow \subseteq \mathit{Expr} \times \mathit{Expr}$ using the following inference rules:

$$\frac{}{\mathit{Add}\ (\mathit{Val}\ m)\ (\mathit{Val}\ n)\ \rightarrow\ \mathit{Val}\ (m\ +\ n)}$$

$$\frac{x\ \rightarrow\ x'}{\mathit{Add}\ x\ y\ \rightarrow\ \mathit{Add}\ x'\ y} \qquad \frac{y\ \rightarrow\ y'}{\mathit{Add}\ x\ y\ \rightarrow\ \mathit{Add}\ x\ y'}$$

The first rule states that two values can be added together to give a single value and is known as an *evaluation* rule. The last two rules are known as *structural congruence* rules.

Informally, they say that the first rule can be applied to either argument of an addition. These rules of inference can be translated into Haskell in the form of a function that maps expressions to lists of expressions reachable by a single step (Hutton 1998):

$$
\begin{array}{lcl}
trans\ ::\ Expr\ \rightarrow\ [Expr] & & \\
trans\ (Val\ n) & = & [\,] \\
trans\ (Add\ (Val\ m)\ (Val\ n)) & = & [\,Val\ (m\ +\ n)] \\
trans\ (Add\ x\ y) & = & [\,Add\ x'\ y\ \mid\ x'\ \leftarrow\ trans\ x] \\
 & +\!\!+ & [\,Add\ x\ y'\ \mid\ y'\ \leftarrow\ trans\ y]
\end{array}
$$

In this simple example there are only two structural congruence rules. However, typically many more arise in specifications of real-world programming languages. As such, a more compact method of expressing where evaluation rules can be applied is often required. *Evaluation contexts*, introduced by Felleisen (1987), provide a mechanism to do just that.

Reduction semantics

An *evaluation context* $\mathbb{E}$ is a meta-term representing a family of terms that contain a single *hole*, written $[-]$. If $\mathbb{E}$ is an evaluation context, then $\mathbb{E}[t]$ represents $\mathbb{E}$ with the term t substituted for the hole. Every evaluation context $\mathbb{E}$ induces a context rule

$$\frac{t\ \rightarrow\ t'}{\mathbb{E}[t]\ \rightarrow\ \mathbb{E}[t']}$$

which says that we may apply the reduction $t \rightarrow t'$ in the context $\mathbb{E}$ that has been substituted with t. Making use of evaluation contexts for arithmetic expressions

$$\mathbb{E}\ ::=\ [-]\ \mid\ Add\ Expr\ \mathbb{E}\ \mid\ Add\ \mathbb{E}\ Expr$$

we can redefine our previous operational semantics in a more concise form, as follows:

$$\frac{}{Add\ (Val\ m)\ (Val\ n)\ \rightarrow\ Val\ (m\ +\ n)}$$

Now we have a single evaluation rule, which, just as before, states that two values can be added together to give a single value. In addition, the simple grammar of $\mathbb{E}$ defines the

corresponding set of evaluation contexts for the type *Expr*, which indicate precisely where in a given term the evaluation rule can be applied.

As a concrete example, given the term $Add\ (Val\ 0)\ (Add\ (Val\ 1)\ (Val\ 2))$, we can apply the evaluation rule to the right subterm using the context $Add\ (Val\ 0)\ [-]$. The evaluation rule can then be applied directly to the result, $Add\ (Val\ 0)\ (Val\ 3)$, of this reduction to give $Val\ 3$. In general, each time we use an evaluation context $\mathbb{E}$ to apply an evaluation rule $t \rightarrow t'$, we say that "t reduces to t' in context $\mathbb{E}$".

Overall, this contextual approach, known as *reduction semantics* (Felleisen and Hieb 1992), is a much more convenient presentation of operational semantics, as we need only define inference rules that evaluate terms. In turn, evaluation contexts describe how such rules can be applied in a way that naturally corresponds to the recursive definitions of terms. By making the evaluation context explicit and modifiable, reduction semantics with evaluation contexts is considered by many to be a significant improvement over the more conventional forms of small-step semantics (Felleisen and Flatt 1989).

Operational equivalence

Recall that a denotational semantics identifies terms that are mapped to the same value in the domain. This notion of equality does not carry over to the operational setting, where the meaning of a term incorporates *all* of the individual steps that constitute its evaluation over time. For example, a direct translation of denotational equality to the operational setting would merely identify terms with the same end state. However, this overlooks all prior steps—and intermediate states—that make up the term's evaluation process, of which the result is simply a distinguished state. In consequence, an operational definition of equivalence is typically more intricate than its denotational counterpart. Nonetheless, a significant number of such definitions are centred around a simple principle, which is known as *The Identity of Indiscernibles*, or *Leibniz's Law*

$$\forall x \forall y\ [\ \forall P\ [\ P(x) \iff P(y)\ ]\ \implies x = y\ ]$$

and informally states that two objects are identical if they satisfy the same properties.

Contextual equivalence (Plotkin 1975) is a natural notion of program equivalence that

follows Leibniz's Law. In this definition, two terms are said to be equal if and only if their observable behaviour is indistinguishable, even when used as subterms of any other term.

One method of defining such a relation is *bisimulation*, a notion of equivalence between abstract transition systems that was first developed by Park (1981). Bisimulation intuitively captures what it means for two transition system to have the same behaviour. In particular, two systems are said to be *bisimilar* if there exists a relation between the states of the systems such that related states have related successor states.

A well-cited example of bisimulation is Abramsky's *applicative bisimulation* (Abramsky 1990). In the context of lazy lambda calculus, where termination is defined as reduction to a lambda abstraction, Abramsky states that two closed lambda terms are *applicative bisimilar* if, in all program contexts, they have the same termination behaviour. This notion closely resembles another technique for defining operational equivalence between terms, known as *observational equivalence*, introduced by Hennessy and Milner (Hennessy and Milner 1980).

The idea behind observational equivalence also echoes Leibniz's Law: if an external observer is not able to distinguish between two programs, then they are equivalent. More formally, if, for all program contexts $\mathbb{C}$, the programs $\mathbb{C}[p]$ and $\mathbb{C}[q]$ have indistinguishable *observable* results, then p and q are equivalent. The precise nature of what constitutes an observation varies, but typically it is sufficient to consider only termination. This is because should $\mathbb{C}[p]$ and $\mathbb{C}[q]$ both terminate with different results, then it is often possible to construct a $\mathbb{C}'$ such that $\mathbb{C}'[p]$ terminates and $\mathbb{C}'[q]$ diverges, or vice-versa.

When discussing denotational semantics, we stated that any notion of equality between programs must fundamentally satisfy the transitivity and compatibility properties of a congruence relation. This is also true in the operational setting. In fact, many notions of operational equivalence are congruence relations by definition: one such example being observational equivalence (as outlined above). Bisimilarity relations are not necessarily congruences by definition. However, certain syntactic restrictions can be placed on the form of transition rules in order to enforce this condition (Groote and Vaandrager 1992).

I think I'll have myself a beer

—Reel Big Fish, *Beer*