Rufous

Automated comparison of implementations of purely functional data structures

Ben Simner

The University of York
Department of Computer Science

Submitted for the degree of MEng in Computer Science

Abstract

It is a common occurrence that a programmer is writing a program, and wishes to use an abstract datatype. Whether it's a set, queue or graph, they have many potential implementations available to them. Deciding which is the most appropriate for their given use case is difficult. Should they use a simple set implemented as lists? or is it better to use a more complex binary tree?

We present Rufous. A modern, easy to use tool for automatic data structure comparison. Centred around the idea of the datatype-usage-graph, or DUG, Rufous generates many usages and uses them to build a comprehensive picture of how implementations of an ADT perform. Multiple data structures are automatically profiled and compared, and a report is compiled for the user to study, so that they can choose the appropriate data structure for their program.

Contents

List of Figures

List of Tables

Statement of Ethics

This project has no ethical considerations. It does not use human test subjects, it deals with no personal or confidential data and there is no feasible way in which the project could be used to do harm to anything living or otherwise.

1 Introduction

1.1 Functional Programming

Today Functional Programming is a widely used paradigm. It is well understood why programmers might want to use functional programming languages for their projects [2]. People have argued for a long time [3] in support of functional programming.

But these languages have their challenges. Imperative data structures do not translate well to a functional language. Without mutation and with non-strict evaluation a, language like Haskell presents an entirely different landscape of data structures.

1.2 Abstract Data Types

Fundamentally, an abstract datatype does not change in a functional setting. Programmers still write and use sets, queues, lists, arrays and maps. However their implementations may be quite different in a functional language, without the mutable assignment found in their imperative counterparts.

This section will describe, briefly, a few implementations that will serve as examples throughout the rest of this report.

Functional Queues and amortization One key data structure, used as a running example throughout this report is the Queue. A First-In First-Out (FIFO) ordered collection. Functionally we would describe a Queue as having four main operations:

- empty. To construct new, empty queues.
- snoc. To append a new element to the end of the queue.
- head. To retrieve the first element of the queue.
- tail. To advance the queue forward to the next element.

A naively simple implementation of a Queue is just as a linked-list. This makes `snoc` very inefficient, a linear-time operation.

To do better, we can batch the operations up with two lists [4], (f, r) as a front-rear pair. Storing the rear as a reversed list means that `snoc` can become constant-time by pre-pending to r. Then only a single linear-time operation is needed to re-orientate the queue when the front is emptied.

It is even better is to use three lists and rotate elements between the rear and a working copy of the front. This explicitly spreads the cost of the rotation, the batched operation, over many subsequent operations [5].

Sets and balancing trees In modern languages and libraries, common set implementations are often hashmaps [6, 7]. A functional implementation is going to have to be more subtle.

As with queues, the simplest implementation is once again as a list. When lists are used, key operations are linear-time. Using ordered lists means that Set $\times$ Set operations can also be made linear-time.

Operations can be made logarithmic-time by using a balanced binary tree. Insertion and deletion can be done by simply traversing the tree. Set operations, like union and intersection, can be done with a divide-and-conquer approach that re-balances the tree efficiently [8].

1.3 Rufous – Data Structure Selection

If a functional programmer is presented with the choice of which data structure to use for an ADT they wish to use, there are a few options. They could use common wisdom, preferring structures with proven better worst-case asymptotic complexity. But this is not necessarily the best measure of performance. The programmer may have a case that performs better than the worst-case implies. Or maybe they just have a small data structure and the asymptotic complexity doesn't apply at all. Another option would be to create or use small programs (benchmarks) to measure the performance of each implementation empirically. But this option is also insufficient: these benchmarks may or may not represent the programmer's actual program, it can be very hard to tell how well a benchmark corresponds to the usage of a particular program, and creating benchmarks that do is a difficult task.

This report presents Rufous, a tool for automatic profiling and selection of data structures. Given an abstract datatype declaration, and a set of implementations, Rufous generates a sample set of benchmarks. These benchmarks represent a fair distribution of the possible usages of a data structure. Rufous then takes these benchmarks as test cases for the evaluation machinery, where they are each executed on a series of data structures, implementations of that ADT. The resulting values from operations, and the time taken to evaluate them is then collected. Each implementation is compared and the results of operations checked for consistency. This information is compiled into a report, which is then presented to the user for inspection.

1.4 Chapter Preview

The remainder of this report is split into five chapters.

Chapter 2 reviews existing literature related to data structure profiling and selection. Here, we identify the DUG from Auburn [1] as the data structure that will become central to Rufous' implementation.

Chapter 3 states, exactly, what the problem Rufous sets out to solve is. This includes a set of precise requirements that the implementation must meet.

Chapter 4 develops a high-level design of Rufous. Then uses that design to describe the implementation of Rufous. In doing so, the challenges that arise are discussed and their solutions presented.

Chapter 5 evaluates how well Rufous met those aims set out in Chapter 3. It does this by investigating the acceptability, performance and effectiveness of the solutions presented in this report.

Chapter 6 gives a high-level overview of the work presented in this report, and suggests potential future work.

2 Related Work

Abstract Data Types (ADTs) have been a much studied part of the Computer Science literature since at least the 1970's [9]. Since then, there has been much research into data structure implementations. Ranging from work in the late 1970's for SIMULA-like languages [10], to modern automated tools for Java [11]. This research has resulted in a handful of tools and techniques for analyzing, comparing, and selecting appropriate data type implementations.

This chapter will explore literature related to data structure selection and attempts at automating that process. Particular focus will be on different representations of ADTs and their usages, and on any aspects that can be translated into a functional setting.

2.1 Introduction to data structure selection tools

Terminology Each technique and tool has its own terminology for abstract data types and their implementations. To clarify this, we shall use *abstract data type* (or ADT) in the manner described by Liskov and Zilles. As a way of describing, abstractly, a set of objects without reference to underlying representations [9]. The actual representation is often called the *data structure* or simply an *implementation* of the ADT. These two terms for representations are used interchangeably throughout this report.

2.1.1 History of tools for ADT selection

Low's tool (1978, University of Rochester (New York)) Some of the earliest work done in the area of data structure selection is by Low [10]. His tool is designed to select the optimal data structure from a set of available representations, in a SIMULA-like language.

DAISTS (1981, University of Maryland) DAISTS [12] is a similar but different tool. It was not necessarily a selection tool but rather a testing tool, designed to test conformance to an ADT specification by a set of implementations. DAISTS would take as input user-supplied test cases and report whether all implementations agree with the specification by executing them.

Auburn (2001, University of York) Auburn [13] was primarily designed to benchmark *purely functional* data structures written in Haskell. When given an ADT and a set of implementations Auburn could automatically profile each implementation and produce a decision process to aide in selection of the most appropriate data structure.

Perflint (2009, Purdue University) Not all of the tools were designed to be stand-alone programs. Perflint [14] is a compiler aide. A *linter.* It takes the C++ program a user is editing and notifies them to inefficient usages of data structures from the C++ standard library.

Chameleon (2009) Chameleon [15] is tool for Java. It is the first tool to really automate the *selection* process for modern programs. Given a program that utilizes Java collections, Chameleon can automatically profile, measure and update the code, re-compiling with the selected implementation.

Brainy (2011) Brainy [16] is another data structure selection tool but with focus on hardware features, such as cache misses and branch prediction failures, when performing selection.

CoCo (2013) CoCo (2013) [11] is much like Chameleon. It takes programs that implement Java interfaces, profiles them and performs selection. However, it has one important difference, CoCo updates the code *at runtime* with the new selection. This allows the programmer to select the ADT and leave the representation up to the library to determine. This is much like work done earlier in SETL [17], where representations were chosen at compile time by the compiler and not by the programmer.

Common Processes Each of these tools follows a similar process, with many overlapping steps with respect to data structure *selection.* Namely, each of the tools, performs each of these steps:

1. Suitably define the ADT they are operating over (§2.2).

2. Characterize the target program's usage of the ADT (§2.3) .

3. Measure performance of available implementations (§2.4).

4. Select the optimal data structure for the target program (§2.5).

The rest of this chapter will examine each of these steps in detail and how each tool approaches them.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Figure 2.1: Example Queue interface in an object-orientated language (Java).

2.2 Abstract Datatype Definitions

Liskov and Zilles defines an abstract datatype as a class of abstract objects that are defined only by the operations available on those objects [9]. The tools described here use equivalent definitions, usually in a direct way as a structure with a set of operations. There are many ways of precisely defining an ADT for even simple types, broadly separated into 2 categories: *Imperative* datatypes and *functional* ones [5].

2.2.1 Imperative Datatypes

Imperative datatypes are by far the most common, generally represented as a class or interface. These datatypes typically rely heavily on mutation and assignment in their implementations [5], and their signatures reflect this by often being side-effectful like in Figure 2.1.

Typical object-orientated style classes are used by both Low's tool and DAISTS. Each class comes with a small number of visible primitives, such as push and pop for stacks. Multiple implementations of the ADT should be classes with the same primitives but these tools do not provide any first-class structure that represents the abstract datatype itself.

For representing the abstract datatype, later tools used interfaces. Chameleon, CoCo, Perflint and Brainy are all examples that use interfaces as their ADT representation. Representations are therefore classes that implement the interface. While Brainy and Perflint use interfaces to represent their ADTs, they do not take arbitrary interfaces as input, instead having a fixed set of abstract datatypes the tools operate over.

Example 2.2.1. Figure 2.1 gives an example ADT definition in Java. It is defined in a typical imperative way. It is an object-orientated structure, an interface. It defines impure methods for the operations: `remove` and `offer` do not return `Queue` type objects.

```
empty :: Queue a
snoc :: Queue a → a → Queue a
head :: Queue a → a
tail :: Queue a → Queue a
```

Figure 2.2: Example Queue ADT in Haskell [1].

2.2.2 Functional Datatypes

Functional data structures are generally quite different to their imperative counterparts. Representations of functional structures are void of mutation and assignment, so have the property that they are generally *persistent*. Updating a persistent structure does not alter the old version: it remains available for further processing. Imperative structures are generally not persistent but *ephemeral* [5].

Auburn's ADT representation is equivalent to the imperative definitions used by other tools, formally they are just a pair (C, M) of a type constructor, and a set of functions (also known as *methods*). M has the constraint that all signatures must be first-order. Auburn further characterizes the operations of the ADT into three groups: Generators, Mutators and Observers.

- Generator functions are those that create a version from non-version arguments.
- Mutators are those that modify a version argument to produce a version result.
- Observers take version arguments and produces a non-version result.

Example 2.2.2. Figure 2.2 gives an Queue ADT, equivalent to the one in Figure 2.1, as Haskell type signatures. It has no impure functions, they all return a new version of the data structure and no operation modifies a version in-place.

There is only one generator, `empty`. It takes no version arguments but produces a version result. There are two mutators: `snoc` and `tail`. There is only one observer, `head`.

2.3 Characterizing Application of ADTs in Programs

Not all programs use the same ADTs in the same way. Some may only use a subset of the available operations whereas some may use all of them but with different frequencies. Each tool has a way of capturing this information, and more, about the usage of a particular ADT in a program.

The most common way of doing this is with a set of usage statistics. These capture information about how each operation is used, how often, and any other information the technique or tool decides important. For example, Chameleon chooses to record heap information whereas Auburn only uses trace statistics.

Usually this is done with a dynamic profiling step, such as in Chameleon. This is achieved with library instrumentation, that records the calls to each ADT operation capturing information such as: the maximum size of objects; total number of calls to each operation; total number of objects in heap. CoCo and Perflint also profile at runtime to generate their usage statistics.

CoCo, being an "online" tool that makes decisions as the code is running must be careful to ensure that the profiling overhead is small. This is because CoCo must make the characterisation at runtime, and so must make any measurements of the program at runtime. It does this by sampling only a small selection of the calls, and by not sampling during the collections "start-up" phase [11], only recording once something is removed from the collection.

Dynamically collecting these statistics isn't the only way, Low's tool uses a dataflow anaylsis step to collect the required information about how the data structure is used in the application.

2.3.1 Auburn's approach: The DUG

Auburn takes a different approach. Instead of simply defining usage statistics, it instead defines a model of how the program uses the data structure. Auburn called this model a datatype-usage-graph, or DUG.

Definition 2.3.0. Auburn defines, a DUG as a 4-tuple $(\mathcal{G}, \eta, \sigma, \tau)$. Where $\mathcal{G}$ is a directed graph, η is the labelling function, σ is the evaluation ordering function and τ is the argument ordering function, for determining the order of input versions to the function.

Example 2.3.1. A simple DUG is given in Figure 2.3.

This DUG was extracted from the following Haskell program:

```
v0 = empty
v1 = add 1 v0
v2 = add 2 v0
v3 = union v1 v2

o1 = member 1 v2
o2 = member 2 v2

main = print o1 >> print o2
```

Nodes are labelled as $v_i : \eta(v_i)$. The component σ gives $v0 < v1 < v2 < v3 < o1 < o2$. Arguments ordered by τ are represented as numbered labels next to edges.

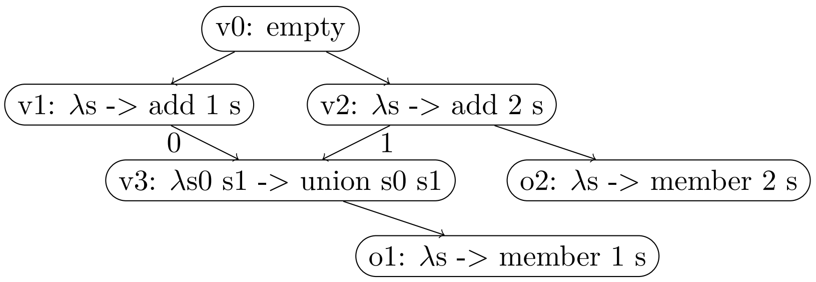


Figure 2.3: Example DUG of a small set program.

DUG Profile Since DUGs can grow very large, Auburn defines a condensed form of usage statistics called a *profile*. A profile captures the essence of the DUG.

Definition 2.3.1. Profiles are triples (G_w, MO_w, m, pmf, pof):

Where G_w and MO_w are mappings of the ratios of generator and observer operations in the DUG. The component m is the mortality, the ratio of nodes in the DUG that are never mutated. Persistence is captured by the pmf and pof components, which are the ratio of *applications* of mutations or observations that are persistent.

Example 2.3.2. Figure 2.3 has the profile:

- Only one generator is used, `empty`, so $G_w = \{\texttt{empty} \mapsto 1\}$.
- MO_w is also simple since only one mutator or observer is used, `cons`, so this property is just $\{\texttt{cons} \mapsto 1\}$.
- There are five versions, of which, two are not mutated. So $m = 2/5$.
- Of the three applications of `cons`, just one is persistent. So $pmf = 1/3$.
- There are no observers, so $pof = 0$.

2.4 Measuring Performance of ADT Implementations

Profiling representations of datatypes is not as simple as just timing the execution of each implementation. The performance may change drastically depending on how the datatype is used. It is therefore important to know not just how the implementation performs for a given input, but how the performance relates to the characteristics of the program that uses it.

Most tools cannot do this automatically and so require some amount of expert intervention. More generally there are two approaches: deriving a cost for each implementation abstractly or by generating a series of usages and profiling them empirically.

Representation	**Set non-empty**
Linked List	$23 + 13\lambda/2 + 27\pi$
AVL Tree	$32 + 88\pi + 20 * LOG2(\lambda)$
Array	42

Figure 2.4: A selection of cost functions for `set.remove` in Low's tool.

2.4.1 Static cost function derivation

The most common way is by defining some cost metric over the implementations. This is the approach Low and Perflint take: with a set of pre-determined cost functions for possible representations. These cost functions describe the performance overhead of using a particular representation for each operation.

Example 2.4.1. Figure 2.4 gives a cost function for the `set.remove` function, as given by Low's tool [10]. It says that the cost of performing a `set.remove` on a Linked List for example is `23 + 13*sizeOfSet/2 + 27*averageTimeItemInSet`. It is these parameters that Low's tool tries to discover when measuring performance.

2.4.2 Static Rule Engine

Another approach is by using rules. CoCo and Chameleon define selection rules between representations, rather than a cost for each individual representation. These rules encode the same information as cost functions, and are discussed more in §2.5.

2.4.3 Dynamic cost model derivation

Brainy also uses cost models, much like Perflint, but instead of having a fixed set of them, it generates them. This is done by generating many applications, and then profiling them. It does this with a simple function dispatch loop; a program that loops, selecting an operation and some data at random, and then calling it. Features in the cost model for Brainy are also quite different, with Brainy focusing on the architecture it collects not only the usage statistics as described in other tools but also hardware features [16].

Auburn also derives cost models in a similar way to Brainy, by generating many usages of the data structure and profiling them. Auburn does this by generating DUGs. Given a profile Auburn can generate many DUGs which conform to it, and then evaluate them to measure the execution time. Auburn would then take these times and feed them into a dynamic selection process, described in §2.5.

2.5 Selection of appropriate ADT representations

Finally, each tool must take the programs usage of the data structure (§2.3), a set of representations and their costs (§2.4), and decide which representation is the most appropriate for that program. This is where the tools vary the most. There are broadly two methods used by the tools discussed here to automate the selection process: replacement rules and optimization techniques.

2.5.1 Rule-based selection process

The simplest of the two methods is by using a rule engine to decide the appropriate representation based on usage statistics gathered beforehand. Chameleon and CoCo use rules as the basis of their selection mechanism, when determining whether to replace one structure with another. Chameleon uses rules in a straightforward manner: for each pair of possible replacements there is a boolean predicate that determines whether the replacement should be made or not.

Example 2.5.1. Figure 2.5 gives an example for such a set of rules for various set-like representations in Chameleon. These rules specify how the tool should decide a replacement data structure. For example, they specify that if the current data structure is a `HashSet`, and has a size of at most X, then suggest a replacement with an `ArraySet` instead.

Type	**Condition**	**Replacement**
ArrayList	`#contains > X` ∧ `maxSize > Y`	LinkedHashSet
LinkedList	`#get(int) > X`	ArrayList
HashSet	`maxSize < X`	ArraySet

Figure 2.5: A selection of rules from Chameleon.

Users can add their own replacement rules to Chameleon. This means it can be extended to work with many different interfaces, not just a pre-defined selection the tool is compiled with. After profiling these rules get applied by a rule-engine, and a replacement decision is entirely made by the tool.

Run-time replacement The rules do not need to be compile-time options. Instead of re-compiling the source with a different implementation, CoCo generates *glue code* that performs the replacement decisions at runtime. Although, it cannot generate the replacement code itself from the rules, the user must manually program them [11].

2.5.2 Optimisation techniques

The remaining tools use a variety of selection mechanisms, primarily variants of machine learning techniques to perform selection. The simplest optimisation is a hill-climbing algorithm, used in Low's tool, which seeks to minimize the total space-time product and choose the representation that yields the minimal value. More complex, Brainy uses an artificial neural network, taking the set of features described earlier in this chapter and the cost model they generated and chooses the data structure that yields the best performance, that is, the lowest cost.

More direct approaches exist, too. Auburn uses an inductive classification algorithm to generate decision trees that the user can follow to help select the appropriate data structure. It does this by taking the profiled DUGs generated in the performance gathering stage (as described in §2.4)

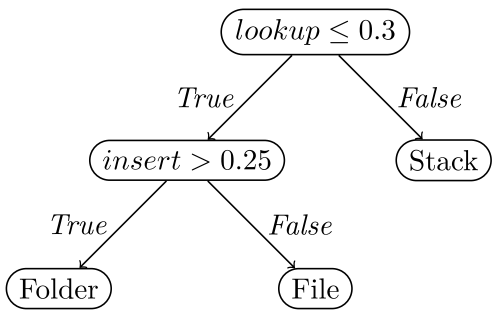


Figure 2.6: Example Decision Tree from Auburn.

	Auburn	Brainy	Chameleon	CoCo	DAISTS	Perflint	Low's tool
Imperative(I)/Functional(F) ADT	F	I	I	I	I	I	I
Auto Usage Collection	✓	✓	✓	✓		✓	✓
Auto Implementation Profiling	✓	✓		✓	N/A	✓	
Auto Selection	✓	✓	✓	✓	N/A	✓	✓
Correctness Check	✓				✓		
Auto Runtime Replacement				✓			

Figure 2.7: Comparison of tools and techniques for automatic data structure selection.

2.6 Summary of Key Concepts

Seven tools were discussed, relating to data structure selection. Figure 2.7 gives an overview of each of these tools. Auburn is the only tool that operates on *functional* data structures discussed here.

Very few of the tools consider the correctness of the implementations they perform replacement on.

Additionally, none of the tools are fully automated, all require a lot of effort from the user in at least one step. And only CoCo's selection step is automated to the point of being a runtime decision.

2.7 Chapter summary

Many tools exist for data structure selection, with a variety of techniques being used to achieve it. Auburn is the most pertinent to this report. It being a tool devoted to benchmarking purely functional data structures – and there is a distinct lack of literature studying automated empirical analysis of the performance of functional data structures making Auburn unique.

3 Rufous Requirements

3.1 Problem Analysis

Chapter 2 described many approaches to solving the problem of (automated) selection of data structures, but in doing so raised many potential problems that a new implementation would face. This chapter sets out to first describe, concretely, what it is Rufous will do (§3.1.1). Then, use that to build a set of formal descriptions of exactly how Rufous will function (§3.1.2).

3.1.1 Rufous - Description and Goals

Rufous shall be a simple, modern tool for automatic data structure comparison. It must allow functional programmers to easily, quickly and with little effort, compare implementations of a common datatype specification. Its implementation should use the core concept of a DUG, as defined in Auburn. Rufous should generate DUGs, then use those DUGs to profile many data structures. The timing information and operation results should be extracted and used to generate a report for the user to inspect.

3.1.2 Requirements Specification

The requirements set out in this section form the *minimal* set of behaviours Rufous must satisfy to meet the goals set out in the previous section.

Core Functional Requirements

R1.1 – Rufous must be able to generate a selection of DUG*s which conform to a given profile.*
As mentioned in the previous section, core to Rufous is the datatype usage graph. This data structure will form a major part of the API of Rufous. Any usage of Rufous will include generating DUGs. To ensure that Rufous can generate a fair distribution of DUGs it's important that Rufous can generate DUGs with a specific behaviour.

R1.2 – Rufous must be able to extract a Profile from a given input program.
Extracting Profiles is key to being able to use Rufous as a tool for optimisation: understanding how a particular program uses a datastructure and then selecting a more optimal structure.

R1.3 – Rufous must be able to take a DUG *and an implementation and output the time taken to evaluate the* DUG
The primary goal of Rufous will be to output some form of report to help guide the user to select the data structure based on the time taken to evaluate a selection of generated DUGs.

R1.4 – All structures output by Rufous should be in a format able to be read by the tool in subsequent stages.
In an effort to make Rufous as automated as possible (See R4.1), the key structures, DUGs, Profiles and ADTs should be first-class values. They should be able to be passed between functions exported by Rufous. This would allow the user to automatically feed the result from one phase into another, without manual intervention.

R1.5 – Any inconsistency in evaluating DUGs, where different implementations give differing results for observations, should be reported to the user.
If a data structure is incorrectly implemented, it may give optimistic timing information. The user should be informed if a DUG evaluates to different results on different implementations, as it may indicate a fault in the data structure.

Data structure representation

R2.1 – Profiles must be a concise summary of datatype usage.
This includes at least storing the count (or a ratio) of the operations in the DUG, the mortality and the ratio of persistent applications.

R2.2 – Rufous must be able to accept an ADT *representation as input.*
This includes at least storing the count (or a ratio) of the operations in the DUG, the mortality and the ratio of persistent applications.

R2.3 – ADT *representations must be able to encode pre-conditions over operations.*
Not all operations are valid, and Rufous needs some way to specify what the pre-conditions on certain operations are. For example, `head empty` is not a valid application for a Queue, and Rufous should not bother generating a DUG that contains it.

Timing Report

Finally, the intended output of Rufous is a collection of reports and a summary. These reports should contain, at least, some key pieces of information.

R3.1 – Rufous should output a timing report for the evaluated DUGs.
The intended output of Rufous is to be a report of the time taken to evaluate

a selection of DUGs generated by R1.1. It could be the raw timings or an aggregate of a selection of DUGs.

R3.2 – Timing reports should report times adjusted for overhead of the evaluation engine. Any times output by the tool should take into account any overhead in the evaluation machinery itself. Either removing it from the timing, or ensuring the overhead is minimal compared to the true evaluation time.

Non-Functional Aspects of Rufous

Rufous should be much more than just the set of inputs and outputs as described.

R4.1 – Rufous should be as automated as possible.
The key feature of Rufous is its level of automation. As a tool for programmers, it should at least be able to generate DUGs and evaluate them with a single function call and little effort on the part of the programmer.

R4.2 – Rufous should be implemented in Haskell.

R4.3 – Rufous should not inspect the source of the data structures.
In this sense Rufous should treat the data structures as a *black-box*. So whilst Rufous should be a tool for Haskell, it need not be useful for *only* Haskell programs. Instead, the user could supply a call to a `CFFI` function as a part of the data structure, and Rufous should work equally as well.

4 Rufous Implementation

This chapter will describe the design and implementation of Rufous. Going from the requirements defined in Chapter 3, to a high-level architecture and then the reification down to the low-level implementation details – describing and justifying each major data structure and algorithm.

The remainder of this chapter will be split into sections:

§4.1. High-level design, program flow and module layout.

§4.2. Technical details of key data structures.

§4.3. Technical details of key algorithms.

4.1 Design

Chapter 3 gives a precise set of requirements that any design should aim to meet. There are many choices for how to design and implement the solution. This section describes, abstractly, how Rufous operates. Following sections dive into the technical details and justify all aspects of the implementation.

4.1.1 High-level architecture

Rufous uses a fixed set of four *phases* through the program. Each designed to enable a key requirement to be fulfilled.

Program flow Figure 4.1 shows a diagrammatic view of the proposed program flow through Rufous. It starts with *profile extraction*. Here the programmer can supply a program, a Haskell function, and automatically extract the *profile* of a particular data structure during execution of that program. The extracted profile can be output to the user, for manual inspection to compare against the summary table, or can be automatically passed to the generator.

The next phase, generation, produces many DUGs conforming to a given profile. Generated DUGs can be output to the user, or passed directly to the evaluator.

During evaluation the time taken to evaluate each observer in the input DUG is measured and recorded. This is done for each implementation, resulting in an annotated DUG with timing information.

Finally the selection phase takes the DUGs and their timing information from the previous phase, then generates and outputs the final report for the user.

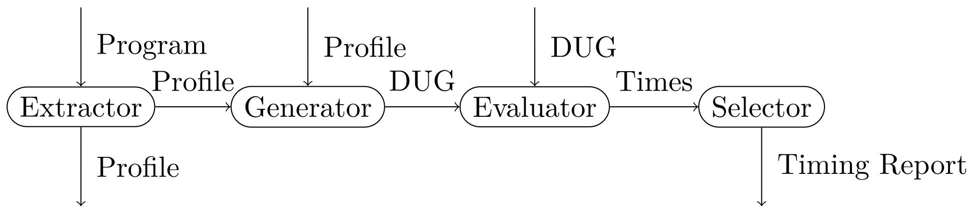


Figure 4.1: Control-flow through Rufous.

4.1.2 Module Structure

Each phase of Rufous is split into a Haskell module. These modules are all related in different ways. Not every module uses DUGs, and the four stages do not interact directly. Instead, there is a core `Test.Rufous` module that passes arguments from one module to the other.

Key datatypes and their operations are split into modules, DUGs, Profiles, and ADTs each have their own module. Phases are also dedicated a module each. And one for each phase:

These modules are laid out hierarchically in the Rufous namespace:

```
Rufous/
└─Test.Rufous
   ├─Test.Rufous.DUG
   ├─Test.Rufous.Profile
   ├─Test.Rufous.Signature
   └─Test.Rufous.Stages/
      ├─Test.Rufous.Stages.Extract
      ├─Test.Rufous.Stages.Generate
      ├─Test.Rufous.Stages.Run
      └─Test.Rufous.Stages.Select
```

Each module exposes key datatypes for use in other modules:

Test.Rufous.DUG The DUGs are stored in a simple module, with only a few key exports. The `DUG` type for the DUG itself, and the `Node` type which wraps a node in the DUG with information about the arguments and operation type.

The DUG module also exposes the `extractProfile :: DUG -> Profile` function, which returns the `Profile` the DUG corresponds to.

Test.Rufous.Profile Profiles are equally simple storing only the `Profile` type.

Test.Rufous.Signature The ADT is more complex, it is composed of many parts: The `Signature` object is the main ADT representation. It is composed of `Operations` and `Implementations`. Each `Operation` has a signature made of `Args`. These are described in more detail in Section 4.2.

4.1.3 Rufous' Phases

Extraction Extracting profiles is very useful for the user. It allows them to take a program they already have, run Rufous on the ADT signature and then compare the extracted profile with the set of generated profiles in the final report. However, extracting the profile from an arbitrary program is not easy. To do so, Rufous instruments an implementation with a wrapper function which logs applications to construct a DUG. The programmer would then use the instrumented version of the implementation, and it would automatically generate a DUG as the program ran.

This is done using a logging function, which given the operation and the set of arguments to the function constructs a new function which acts as an identity, with the side-effect of building up a DUG.

```
logOperation :: String -> [Arg] -> a -> a
```

The DUG itself can then be extracted through an extraction function, which wraps a given program and returns the DUG generated after evaluation.

```
extract :: ADTSignature -> IO a -> IO (a, DUG)
```

Example 4.1.1. Extraction of a given profile can then happen by taking the target program's `main` function, wrapping it in `extract` and retrieving the profile:

```
main = head (snoc 1 empty)

mainWithExtraction = do
    (_, dug) <- extract QueueADT main
    print ("ExtractedDUG: ", dug)
    let profile = extractProfile dug
    print ("ExtractedProfile: ", profile)
```

Generation Generation has a simple API. Given a Signature and a Profile, the `makeDUG` function generates a DUG of a given size:

```
makeDUG :: Signature -> Profile -> Int -> DUG
```

This function forms the core of Rufous: generating DUGs that conforms to given profiles. Section 4.3 describes the algorithm in more detail.

Evaluation For evaluation, the `runDUG` function takes a DUG and an implementation. It then evaluates the DUG, forcing evaluation of all observers and timing the execution time of the DUG. The DUG is then annotated with this new information and returned.

```
runDUG :: Implementation -> DUG -> IO TimingDUG
```

The only question here is what timing information should `TimingDUG` contain. At minimum, they must hold three pieces of information: (1) the value resulting from evaluation of each observer. (2) the time, with high granularity, of how long it took the observers DUG to evaluate. and (3) the implementations each result was run against.

Selection

Selection does not output a Rufous datatype but instead generates and prints a report to the user. `select` takes the annotated timing results from the Run stage and performs a selection step on them. This involves generating a report, which contains the profile for each DUG and execution time for each implementation. It then prints the report for the user to study.

```
select :: [TimingDUG] -> IO ()
```

4.1.4 Rufous API

Directly creating and manipulating the core data structures is not a user-friendly task. For this purpose Rufous has a top-level `Test.Rufous` module which contains the API.

Specifying ADTs Rufous uses Haskell's typeclass machinery to abstract over ADTs and allow the programmer to write implementations in a natural way using Haskell's own abstractions. Typeclasses are a useful Haskell tool for representing containers.

For specifying an ADT the programmer would write a (type)class. That class would define all the operations as simple source-level Haskell. Implementations are just instances of that typeclass.

To make use of the typeclasses, Rufous provides abstractions over typeclasses. Since Haskell itself has no abstractions over typeclasses, this is done with a templating engine, TemplateHaskell [18]. So, Rufous provides the `makeADTSignature` template function, which automatically generates the `Signature` datatype.

Example 4.1.2. Figure 4.2 is an example of such an ADT defined as a type-

```
class Sequence s where
    nil :: s a
    cons :: a -> s a -> s a
    head :: s a -> a

instance Sequence [] where
    nil = []
    cons = (:)
    head = Prelude.head

makeADTSignature ''Sequence
```

Figure 4.2: Example typeclass interface in Haskell, with Rufous templating.

class. The `makeADTSignature` call is the only Rufous-specific thing the programmer must do in order to create the ADT. They may already have the `Sequence` typeclass and its implementations, and need only insert the call to `makeADTSignature` to start using it with Rufous. This function just calls into the templating engine to build a `Signature` object, making it available as a top-level name.

Running Rufous Running Rufous is straight-forward: calling `mainWith` with the correct arguments. The arguments contain settings for each of the phases, such as the input profiles, the size of DUG to generate and the number of DUGs to generate.

```
    mainWith :: RufousArgs -> Signature -> IO ()

    data RufousArgs =
        RufousArgs
        { signature :: Signature  -- The ADT to operate over
        , profiles :: [Profile]   -- The profiles to use to generate (default: [])
        , dugs :: [DUG]           -- The DUGs to evaluate (default: [])
        , averageDugSize :: Int   -- The average DUG size (default: 1000 nodes)
        , numberOfTests :: Int    -- The number of DUGs to generate (default: 10)
        }
```

The user may wish to automatically pass the results of one phase onto another. For this purpose, the `extractProfile`, `makeDUG`, `runDUG` and `select` functions are also exposed in the top-level `Test.Rufous` module. For simplicity, the user needs only call `runRufous` with the correct argument object. If `profiles` or `dugs` are non-empty, the `mainWith` function will skip the generation or evaluation stages, using the arguments from those lists instead.

4.2 Key Data Structures

This section describes, in detail, the challenges and justification for alternative decisions that could be made for the key structures introduced in the previous section. Signatures (§4.2.1), Profiles (§4.2.2), and DUGs (§4.2.3).

4.2.1 ADT Signature implementation

ADT Datatypes The ADT is a core concept datatype in Rufous. The ADT signature itself is relatively straight-forward. It is a set of implementations, and a map of operations.

```
type Signature = ([Implementation], (String -> Operation))
```

The operations themselves are just a type signature, tagged with the operation kind. Type signatures are more complex. To simplify Rufous we restrict the signatures of operations to so-called *simple* types. The type needs to be expressive enough to represent the types of common operations over collections, but not totally unrestricted. Simple types are first-order types, with a single type-parameter.

```
data OperationKind = Mutator | Observer | Generator
type Operation = (OperationSignature, OperationKind)

type OperationSignature = [ArgType]
data ArgType = VersionType | NonVersion NonVersionArgType
type NonVersionArgType = VersionParam | IntArg | BoolArg
```

Example 4.2.1. Take the Queue ADT from earlier, the type of `snoc` is `a -> T a -> T a`, where `T a` is a version, `a` is a *version parameter*. This gives an encoding of `([NonVersion VersionParam, VersionType, VersionType], Mutator) :: Operation`.

Implementation types Rufous represents implementations as a map, associating a Haskell function with each operation. This representation allows Rufous to "run" an implementation directly: by simply forcing evaluation of the stored function on the correct arguments. Storing the operation in a type-safe way would be difficult without dependent types for the restricted (*simple*) types. Instead, the templating engine performs the typechecking and then a `Dynamic` cell containing the function is stored in the structure. Type-safety is ensured by tagging the operation with the expected return type, allowing Rufous to safely unwrap the Dynamic to force evaluation later.

```
data ImplResultType = forall t. ImplResultType t
data Implementation = Implementation (String -> (Dynamic, ImplResultType))
data Null a = NullImpl

instance Queue Null where
    snoc x0 x1 = NullImpl
    empty = NullImpl
    head x0 = throw NotImplemented
    tail x0 = NullImpl
```

Figure 4.3: Generated Null implementation for the Queue ADT.

Overheads and Null implementations For the final report, the times reported should have overheads of evaluation removed from them. Since the time taken for the evaluation machinery itself is dependent on the DUG, Rufous must compute this overhead for each DUG it evaluates. Rufous does this by generating a *Null* implementation. This special implementation has no behaviour: each operation simply ignores its arguments and returns Null, as shown in Figure 4.3. For observers, it cannot simply return `Null`. Here Rufous makes use of Haskell's exceptions, throwing a `NotImplemented` exception the evaluator knows to ignore.

For this to be valid the evaluation machinery must perform the same amount of work for each DUG, regardless of the strictness of the implementations (See Section 4.3.3).

Preconditions and shadow implementation Not all applications are valid. For Queues, `head empty` is undefined, and the generation machinery should not generate such applications. To encode this information, Rufous requires the programmer to provide a secondary structure, the *shadow* implementation. The programmer specifies that a particular implementation is the shadow by prefixing the type constructor's name with "Shadow". In all other ways, shadows are like normal implementations. They are just instances of the typeclass that specifies the ADT. However, they are only used when validating applications. Figure 4.4 gives the shadow implementation for a Queue ADT.

Invalid applications are signalled using exceptions. A `GuardFailed` exception indicates the precondition failed. Since Shadow structures are not real implementations, like null implementations they have no way of generating results for observers. For those situations, Rufous exposes a `ShadowUndefined` exception, which is ignored during generation. A undefined shadow represents a valid application, but one that generates no valid result when run on a shadow implementation.

Auburn also had shadow structures, with some minor differences. There was no unified representation for an implementation in Auburn, and so shadow

```
data ShadowQueue a = S Int

instance Queue ShadowQueue where
    snoc x (S n) = S (n + 1)
    empty = S 0
    head (S n) | n > 0  = shadowUndefined
    head (S n) | n <= 0 = guardFailed
    tail (S n) | n > 0  = S (n - 1)
    tail (S n) | n <= 0 = guardFailed
```

Figure 4.4: Shadow implementation for the Queue ADT.

structures were more difficult to write. This was compounded by the fact Auburn used a more complex pre-condition, instead of returning a boolean for all arguments instead the programmer had to specify the valid range of non-version arguments for the set of version arguments. This was noted to be difficult [1], so Auburn had a method of generating a simple guess at a trivial shadow.

4.2.2 Profile implementation

In order to meet requirement R2.1, the Profile type has 3 fields:

- `o : Operation -> Float`, the weights of proportion of each operation of the ADT.
- `p : Operation -> Float`, the weights of proportion of persistent applications of each operation of the ADT.
- `m : Float`, the mortality: the proportion of computed versions that are not mutated further.

Example 4.2.2. To understand each of these metrics, take a look at Figure 4.5. It has five versions: `[v0, v1, v2, v3, v6]`, of those, two are not mutated further: `v2` and `v6`. This gives a mortality of 2/5.

The DUG itself has seven nodes – five versions and two observations. The weights for those operations are:

- 1/7 `empty`
- 3/7 `snoc`
- 1/7 `tail`
- 2/7 `head`

For computing `p`, there are six applications, represented as edges between nodes. Of the three applications of `snoc`, just one of them, `v0 -> v2` is persistent: it occurs *after* another mutation (in this case, `v0 -> v1`). Here, “after” is in the sense of “is evaluated after”. Where the order of evaluation is defined by the node ordering `v0 < v1 < v2 < ... < vN`. Persistent applications of observers are defined similarly, as observations that occur after other observations. So, the second component of the profile, `p`, is as follows:

- `snoc` persistent: 1/3
- `head` persistent: 1/2
- `tail` persistent: 0/1

Generators, such as `empty`, does not appear here. In general, generators are not constructed from applications and so cannot be applied persistently.

Persistent factors Persistence is an important aspect of the profile for functional programs. Auburn [13] also used the idea of persistence in its profile. However, it used a combined "persistent mutation factor". This factor combined the persistent weights of each mutation operation into a single parameter. Rufous leaves this parameter as the full mapping for two reasons: (1) since the user does not manually input profiles in Rufous, it's okay to have the profile encode more information, making extracting profiles slightly easier and (2), to fit the usage better: capturing when some operations are used persistently and when others are not.

4.2.3 DUG Implementation

The datatype-usage-graph, or DUG, is a key part of Rufous. Auburn could perform many operations over DUGs and had various different implementations of them for different purposes.

DUGs as births and deaths Auburn didn't have a single definition of a DUG. Instead Auburn chose to use a list of *births or deaths* as the representation [1]. New nodes being created are births, and a node no longer having a future is *dead*. For evaluation, Auburn used a list of births, and for profiling it used a list of deaths. This made passing DUGs from one stage to another difficult without an intermediate conversion.

DUGs as Graphs Rufous takes a different approach. DUGs are, fundamentally, just graphs. So, most of the operations over a DUG are similar to those found over graphs. There should be ways of constructing dugs: creating empty DUGs, inserting new versions, and applying operations to arguments in the graph to construct new nodes. DUGs have one other key property: there is no requirement to ever delete versions from DUGs.

Rufous therefore chooses a more unified approach, with a single core `DUG` type. That type is parametrized by the annotations, and so as DUGs progress through the pipeline they can be annotated with extra information. This approach works for Rufous for two main reasons: (1) Rufous is more refined and smaller program than Auburn, and (2) modern GHC and libraries provide a richer choice of efficient and easy to use implementation options. As such, Rufous encodes a DUG quite directly: as an adjacency mapping. A good choice here as

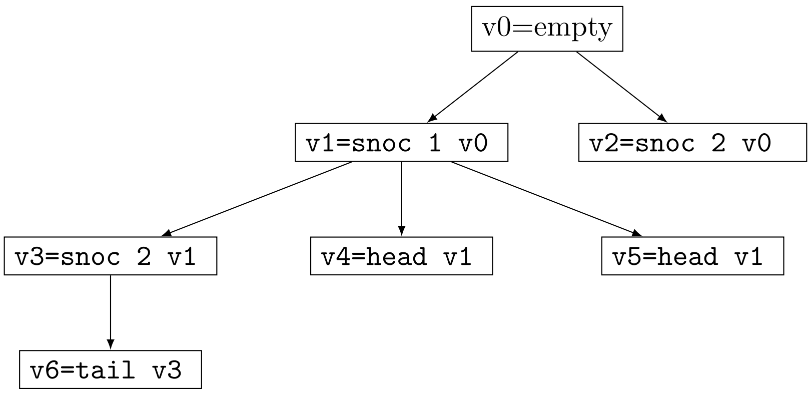


Figure 4.5: Example datatype usage graph for a small queue.

it is fast to insert and lookup from, two operations DUGs do a lot of. Evaluation order is then encoded implicitly as an ordering over the keys.

```
data Node n = Node Operation [Arg] n
data DUG n = DUG (Map Int (Node n))
```

Example 4.2.3. The DUG given in Figure 4.5 is an example DUG of a Queue ADT. Node `v1` could be defined `Node snoc [NonVersion (IntArg 1), Version v0] ()` If all nodes are defined similarly the whole DUG could be defined as as a simple map from $i \mapsto v_i$.

```
dug = DUG (fromList [(0, v0), (1, v1), (2, v2), (3, v3), (4, v4), (5, v5), (6, v6)])
```

4.3 Key Algorithms

This section will describe the challenges and justification over alternatives of the key algorithms used in Rufous: The extraction of profiles from source programs (§4.3.1), the generation of DUGs that conform to such profiles (§4.3.2), evaluation of DUGs to measure timing information (§4.3.3), and report generation (§4.3.4).

4.3.1 Extraction algorithm

Any algorithm to perform extraction must require little to no modification of the original source. It should not force evaluation of anything the original program did not evaluate, and it should not evaluate anything in a different order than the original program did.

DUG Extraction Rufous first extracts a DUG, and then extracts the profile from the DUG. To extract a DUG the algorithm is fairly straight-forward. Rufous will store a global set of nodes, the partially built DUG, and each operation of the ADT will be logged and stored in the set, alongside the list of arguments it was given.

1. Wrap each ADT operation with logging operation.
2. Run the program with the modified ADT.
3. Retrieve the set of nodes and build the DUG from it.

The use of a class for specifying ADTs means Rufous can use TemplateHaskell to generate an extraction implementation that is indistinguishable from the underlying implementation as far as the ADT operations are concerned, but is separate from the non-wrapped version [18]. Logging operations therefore operate over a wrapped version of the ADT, which stores the name of the operation, the arguments and the result.

```
type WrappedADT t a = WrappedADT (OperationName, [WrappedArg t a], WrappedArg t a
type WrappedArg t a = Arg (t a) a

extractVersion :: WrappedADT t a -> t a
extractNonVersion :: WrappedADT t a -> a
```

Then, for some ADT typeclass, the extractions logging methods look like:

```
logOperation :: ADT t => OperationName -> [WrappedArg t a] -> t a -> Wrapped t a
logObserver  :: ADT t => OperationName -> [WrappedArg t a] -> t a -> a
```

```
instance QueueADT (WrappedADT []) where
    empty =
        logOperation "empty" [] empty
    snoc xs x =
        logOperation "snoc" [Version xs, NonVersion x] (snoc (extractResult xs) x)
    head xs =
        logObserver "head" [Version xs] (head (extractVersion xs))
    tail xs =
        logOperation "tail" [Version xs] (tail (extractVersion xs))
```

Figure 4.6: Generated WrappedADT for a Queue implementation.

These log functions perform two operations. First, they construct a wrapped version of the result, and return it. Secondly, they must update the global extractor state to add the new wrapped version to the set of nodes. Because these functions are side-effectful, but should be used in pure code, they must perform this state update in an unsafe way. However, this global update is completely contained within Rufous' extractor module, and no other effects leak out.

```
logOperation name args v = unsafePerformIO $ do
    let wrapper = Wrapped name args (Version v)
    updateExtractorState wrapper
    return wrapper
```

Rufous then uses TemplateHaskell to generate wrapping instances of the ADT automatically. Figure 4.6 shows an example instance Rufous generated for the Queue ADT. It is a new implementation of the Queue ADT that wraps a list, separate from the unwrapped list. This implementation behaves exactly the same, any program that uses the list implementation should work exactly the same with this new implementation, with the side-effect of this implementation silently logging what operations are called.

Alternatives An obvious alternative is to use a wrapper which has no effect and returns the same type logging the name of the operation: `wrap :: String -> a -> a`. This is what Auburn did [1]. The difficulty with this approach is that the source program needs to be modified to call the new wrapper functions. Typeclasses mean that the wrapper implementations can be substituted by the compiler itself, in a way that does not affect the operation of the program or require any modification to the source.

4.3.2 Generation algorithm

The core generation algorithm is given by the sketch below:

1. Create an initial empty DUG.
2. Pick an operation.
3. Try to pick version arguments from the DUG.
4. Try to pick non-version arguments.
5. Try to commit to the DUG.
6. Repeat from step 2 until the DUG is large enough.

This algorithm is similar to the one used by Auburn [13]. We next identify the main issues and how Rufous approaches them and how the full algorithm given in Figure 4.7 solves them.

Undefined applications As described in in Section 4.2.1, Rufous uses *shadow datastructures* to encode preconditions over applications. During generation, Rufous builds a *shadow* DUG as well, containing shadow versions for each node. Then, before any node is committed to the DUG its shadow is first evaluated. If evaluation of the shadow fails because of a failing precondition then the application is invalid and discarded.

Fitting the profile Ideally, Rufous would be able to generate a DUG given any arbitrary profile which it matches as closely as possible. We could then generate profiles, and corresponding DUGs, uniformly. Simply generating DUGs directly at random would not uniformly match the profile space. There would be a relatively low proportion of persistent applications. For operation weights this is fairly easy: when picking an operation, choose one with probabilities equal to the desired weights of proportion. For persistence this is more challenging. Choosing the operation *before* the arguments is good for generating DUGs as the profiles are operation-centric. But, persistence is a metric on applications. Rufous solves this problem by tagging each node in the partially-built DUG with extra information about its persistence, for both mutations and observations:

Node persistence state:

- Infant - Not mutated
- Original - Mutated once
- Persisted - Mutated more than once

When generating an operation, Rufous decides whether it should be a persistent application or not, and chooses versions as arguments accordingly. For non-persistent applications that means disallowing non-Infant nodes as arguments. To obtain the correct mortality as specified by the profile, Rufous tracks whether a node should be mutated or not in future. A node that should no longer be mutated is referred to as a *dead* node.

Choosing non-version arguments Choosing non-version arguments is more subtle. For version arguments there is a limited pool to choose from. Non-version arguments, however, are not restricted. This leaves a tough decision: the user *could* specify the set of valid non-version arguments for a fixed set of version arguments, as in Auburn [1]. But this increases the effort of the programmer to an unreasonable degree. Additionally non-version arguments may be of a concrete type such as `Int` or as a version parameter, such as the first argument in `a -> T a`.

Rufous takes the simplest approach, unifying the two previous signatures by performing a uniform instantiation of `a` to `Int`, at least for the generator/evaluator. This does not hurt generality too much as `Int`s are a member of most common typeclasses, (and so satisfy any constraint a datatype might have over its inputs) and more complex types could be encoded into an integer in more extreme circumstances.

This decision also simplifies the generation machinery. A full set of arguments is passed to the shadow as one to be checked.

Choosing operations before arguments There is only a limited pool of version arguments to choose from. If Rufous decides to pick an operation *before* there are enough valid version arguments, it may not be able to fulfill and commit the operation. To solve this problem, alongside the partially-built DUG, Rufous also keeps a buffer of un-committed operations. At each step of the algorithm, after adding a new operation, Rufous attempts to commit any operations in this buffer that can now be fulfilled.

Using a buffer in this way leads to a process of rounds of inflation and deflation. Where the generator's buffers fill with new operations until a version becomes available. Then the buffer is emptied as operations are committed, each creating a new version which the next buffered operation may be able to use.

1. Initialise the generator state, s, to the empty DUG and with an empty buffer, b.
2. To *inflate* b:
    - 2.1. Pick an operation, o
    - 2.2. Pick a boolean p, which is whether o, should be a persistent application or not
    - 2.3. Insert (o, p) into b.
3. To *deflate* b:
    - 3.1. Remove an operation o from b.
    - 3.2. To fill (o, p), choose each version argument a_i:
        - If p: Pick a *living*, *infant* node n.
        - If not p: Pick any *living* node n.
        - If no nodes available: return (o, p) to b and continue with deflation loop.
    - 3.3. Pick non-version arguments for o.
    - 3.4. Evaluate o's shadow on as
    - 3.5. If fails: return (o, p) buffer and repeat loop, picking a new operation o'.
    - 3.6. If succeeds: commit o to s, by:
        - 3.6.1. Update the version argument nodes to the new node-persistent-state
        - 3.6.2. Create a new DUG node for the operation
        - 3.6.3. Create applications (edges) for each version argument
    - 3.7. Repeat from 3. until no more operations from buffer can be committed.
4. Repeat from 2. until DUG is large enough.

Figure 4.7: Rufous DUG generation algorithm.

Example 4.3.1. Let us walk through an example of generating a DUG, using the algorithm given in Figure 4.7. This will demonstrate the sequences of inflation and deflation that happen during a typical run.

First, we start with an empty DUG and buffer as the state `(s={}, b=[])`. Then, we begin *inflation*: we pick an operation o, at random, using the weights from the profile. For example, if we pick `o=snoc`, and decide `p=false`, we then insert (o, p) into the buffer: `(s={}, b=[(snoc, false)])`.

Next we try deflate: there are no version arguments, and so no work can be done. So, try inflation again, this time we pick a non-persistent application of `head`, now `(s={},b=[(snoc,false),(head,false)])`.

Next, we try deflation once more, but again there are no version arguments for each operation, so we skip and go straight to inflation: This time, we choose `empty`, and append it to the buffer:
`(s={},b=[(snoc,false),(head,false),(empty,false)])`.

Now there are nodes, deflation can do work. It first picks the buffered `empty` operation, and since it has no arguments to fill, it immediately gets committed to the DUG: `(s={v0=empty}, b=[(snoc,false),(head,false)])`. This insertion means the `snoc` application can now be committed too. For `snoc` there is only once choice of version argument: `v0`, and we arbitrarily pick a non-version argument: `7`. Next, try evaluate `snoc`'s shadow, it succeeds, and so it gets added to the DUG: `(s={v0=empty,v1=snoc 7 v0},b=[(head,false)]`

Finally, there's only `head` to try commit. We do the same process, pick a version argument: `v0`, and try evaluate the shadow. This time, the shadow fails, so we know the pre-condition failed and try again on a different version: `v1`. This time the shadow succeeds and we can commit this operation to the DUG, too.

In the end, the state is `(s={v0=empty,v1=snoc 7 v0,v2=head v1}, b=[])`. This process is repeated until the DUG s was of the desired size, but we will stop here for this example.

4.3.3 Evaluation

Evaluation of a DUG involves taking a DUG and an implementation, and running the DUG as if it were a standalone program. Forcing evaluation and collecting the results of each observer, and the time taken to evaluate them. An initial sketch of the evaluation algorithm is as follows:

1. Convert each node in the DUG into a `Dynamic` cell
2. Dynamically apply each argument to the `Dynamic` cell
3. Force evaluation of all nodes, recording the result and timing the process.
4. Construct the new timing DUG from the evaluation results.

But this simple algorithm suffers a couple of problems. Each problem is described below and how the full algorithm in Figure 4.8 solves them.

Recording results Rufous records the resulting value from evaluating observers. The observers are the functions that take a version and extract information from it. Forcing evaluation of any other nodes may make the implementation do more work, evaluate more of the DUG, than it would have otherwise.

By instantiating all version parameters to `Int`, Rufous also ensures that the results will always be comparable, even for observations that have version parameter results, such as head.

Strictness Many implementations may have different strictness in their arguments, and so forcing evaluation of all nodes may evaluate more than the original implementation did. Initially, just forcing evaluation of observers may seem like it fixes that problem. But, for the null implementation to be a valid metric of the overhead, the amount of work the evaluator itself does should not depend on the strictness of the implementation.

To solve this problem, Rufous ensures that while only the observation `Dynamics` are unwrapped and evaluation of them forced, every other node is also visited by the evaluator and unwrapped, but its evaluation is not forced any further.

```
1. For each node, $n$, in order of evaluation, in the input DUG $d$:
   1.1. For each argument in $n$:
        1.1.1. If the argument is a version argument: yield the Dynamic associated with it.
        1.1.2. Otherwise: cast to an Int then create and yield a Dynamic version.
   1.2. $d$'s operation is converted to a Dynamic, and each operation yielded in 1.1. is applied in order.
   1.3. Annotate $n$ with the new Dynamic.
2. For each observer, $o$ in $d$:
   2.1. Unwrap the annotated Dynamic, $v$, and force evaluation of $v$.
   2.2. Annotate $o$ with the evaluated value of $v$.
3. Annotate $d$ with the time taken to force the evaluation of the observers.
```

Figure 4.8: Rufous DUG evaluation algorithm.

	empty	head	snoc	tail	mortality	pmf	pof	BQueue	ListQueue	RQueue
A	14	9	49	28	0.220	0.069	0.222	1.362ms	2.555ms	1.853ms
B	17	28	33	22	0.340	0.121	0.058	1.134ms	2.348ms	1.544ms

Figure 4.9: Example summary report output for the alternative Queue implementations.

4.3.4 Selection

Selection involves generating reports for the user to read.

Summary report The summary table aggregates the details of the timing DUGs together into a single digest. It lists each DUG and its profile, along side the relative performances of each implementation.

This table is the final output of Rufous. The user should take the table, and use it to decide on the most appropriate implementation for their data structure based on the size and profiles of the reported DUGs to the performance of the listed implementations.

As mentioned in the technical discussion on profiles (§4.2.2), storing the persistence as a separate map was good for generation and extraction. However when displaying to the user, they are cumbersome and contain far too much information. So, in the summary report Rufous reports the persistence *factor*s. Like in Auburn, these factors describe the ratio of applications that are persistent. A high `pof`, for example, would indicate a large number of observations being applied persistently.

Example 4.3.2. Figure 4.9 shows an example summary for generating two DUGs for the Queue ADT. Both DUGs have 100 nodes, and both were evaluated over three separate implementations. DUG A has a low *pmf*, it uses Queues in a mostly single-threaded way. Generating many applications of `snoc` and `tail` on previous versions in a long chain. Whereas DUG B has a high *pmf*, indicating it has lots of branches and many of the versions are shared between multiple other versions.

5 Evaluation

This Chapter will explore how well Rufous met the aims set out in Chapter 3. We will do that in three sections. First, by exploring the key aim of Rufous, and evaluating its level of automation and ease-of-use (§5.1). Next, this Chapter identifies criticisms over the efficiency of the algorithms from Chapter 4 and the performance of the generated artefact (§5.2). Finally, the effectiveness of the tool itself, is investigated (§5.2)

5.1 Ease of use of Rufous

Rufous' primary goal was to be a simple, easy to use tool. Evaluating this quality will be difficult, it is notoriously hard to quantify "ease of use". There are many aspects to a qualitative measure of the usability of the tool: degree of automation, effort for the functional programmer, skill and difficulty of use, robustness to mistakes, and how widely applicable the tool really is. This section will explore all of these aspects, and suggest where Rufous works well, and where it could be improved.

5.1.1 Automation and Effort - A Step by Step guide to Rufous

The easiest way to evaluate the effort and level of automation is just to run through examples of typical usage, step-by-step, and evaluate how much work the programmer actually had to do at each step.

Example 5.1.1. Rufous as an *exploration* tool.

Say we have an abstract datatype and a list of implementations, and we wish to understand when to use one implementation over the other. Without a particular program in mind, we will have to use Rufous to generate many usages fairly over the profile space and then inspect the results.

Step 1 - Defining the ADT Many implementations may be located in various forms over multiple files and modules. The first step will be to narrow down the actual ADT. This process involves fully-specifying the ADT's operations and their signatures as a typeclass:

```
class Queue q where
    snoc :: a -> q a -> q a
    empty :: q a
    head :: q a -> a
    tail :: q a -> q a
```

This step is required for all ADTs. Rufous *needs* to know exactly what the ADT it is operating over is, and what operations it has. There is very little room for extra automation here. It may be possible to extract the ADT operations from source files, but it is not clear how a tool would exactly decide which operations were a part of the ADT itself, and which are not. Operations that take versions as arguments are not necessarily part of the ADT specification. For example, smart constructors are operations over versions, but are not part of the ADT specification itself.

Step 2 - Specifying Preconditions over Operations There may be pre-conditions on certain applications. For the Queue, the applications `head empty` and `tail empty` are undefined.

Rufous captures this information with the concept of a shadow implementation (See Section 4.2.1).

For shadows, the observers such as `head xs` can not give a sensible result; it is just a shadow and not a real implementation. For that reason, applications that *are* defined in a real implementation may be *undefined* in the shadow.

If there are no restrictions on the ADT operations, this step could be skipped entirely. However, here capturing just the size of the `Queue` in the shadow is sufficient:

```
newtype ShadowQueue x = S Int

instance Queue ShadowQueue where
    snoc x (S n) = S (n + 1)
    empty = S 0
    head (S n) | n > 0  = shadowUndefined
    head (S n) | n <= 0 = guardFailed
    tail (S n) | n > 0  = S (n - 1)
    tail (S n) | n <= 0 = guardFailed
```

This step is, potentially, a lot of work for the programmer. They must understand exactly when operations of the ADT are valid, and when they are not. Then encode this as a shadow implementation. For operations whose preconditions depend on the contents of the structure, this may be equivalent to defining a full *reference implementation*. There is some opportunity here for automation above what Rufous gives. Auburn for instance could “guess” a size-encoded shadow. Another option would be to use a reference implementation, and then operations can be partial functions. Invalid applications could then be detected directly by catching the exception.

Step 3 - Defining the Implementations The programmer then needs to "plug-in" the implementations as instances of the ADT typeclass. To do this, the programmer has to create an instance for each implementation and then call Rufous' `makeADTSignature` macro that will generate the *Signature* object automatically using TemplateHaskell.

```
instance Queue [] where
    snoc x xs = xs ++ [x]
    empty = []
    head xs = Prelude.head xs
    tail xs = Prelude.tail xs

makeADTSignature ''Queue
```

This step is straight-forward. It is safe to assume that operations already exist, with roughly the correct signature, for each implementation.

So, this step is simply just linking the implementation with the typeclass defined in *Step 1*. Because of that, all the same automation concerns arise here, too. If the ADT could be automatically discovered, then potentially the implementations could be too.

The instance is checked by the compiler's own typechecker, making it hard to connect the operations to the instance incorrectly.

Rufous does not inspect the source of the declarations in the instances at all. This means that the programmer is not restricted in how they write the implementations, or where they come from. Implementations could be anything from a pure Haskell function, to a `C` function called through *CFFI*. This versatility means that Rufous can be used for a variety of languages and data structures that are, at least somewhat, functional.

Step 4 - Running Rufous Finally, the programmer only needs to run Rufous, passing the *Signature* object to the `main` function.

```
main = mainWith args{signature=_Queue}
```

Running the compiled program will then generate tables similar to the one in Table 5.1. The programmer must then *manually* comb through the results, and build their own mental picture of the profile space. There is clearly room for improvement. Firstly, it is easy to misunderstand what the table is saying.

A lot of data gets displayed at once and there's no obvious sign for trends and patterns.

Example 5.1.2. Rufous as a tool for *optimisation.*

Alternatively, if we have a program that already exists, and wish to improve its performance, we can use Rufous to help select a more appropriate data structure. This example is different from the explorative scenario, since this time we do not want to generate a large distribution of profiles. Instead, we wish to generate many DUGs of similar profiles. Then use the timing results to carefully pick a more appropriate structure.

The first three steps are identical in this case, we still have to define the ADT, its implementations and any shadow implementation that's required.

Step 4 - Extracting the Target Program Profile Assuming we have a `main` function, or some other entry-point of the program, there are then two changes that need to made to the program to extract the profile. First, the ADT needs to be swapped for a wrapped version that logs calls to its operations. Then, they need to only wrap the `main` function with an `extract` function to retrieve the profile.

As an example, assume we have the following program for the `Queue` ADT:

```
main = print o1
    where v0 :: [Int]
          v0 = empty
          v1 = snoc v0 1
          v2 = snoc v1 2
          o1 = head v2
```

The two changes required would be to the second line, to change the type from `[Int]` to `WrappedADT [] Int`, and adding a wrapper function around main.

```
main = print o1
    where v0 :: WrappedADT [] Int
          v0 = empty
          v1 = snoc v0 1
          v2 = snoc v1 2
          o1 = head v2

mainExtract = do
    (_, dug) <- extract main
    let profile = extractProfile dug
    print ("The extracted profile is: ", profile)
```

This step is probably the step that requires the most effort from the programmer. Most obviously, they have had to modify their source program to use the generated logging variants of the ADT. If they already used the typeclasses in their program, then all they would need to change are the type signatures on the generators. However, if they do not use typeclasses already, then they would need to convert their program to use the typeclass ADT defined in *Step 1.* This could be a big task for large programs.

It is also easy to get this step wrong. Forgetting to change the real implementation for the wrapped variants, such as leaving the second line as `[Int]` in the above program, would go unnoticed. The modified program would continue to work, only partially extracting the DUG.

Step 5 - Running Rufous Now we've extracted the profile, all that's left to do is generate the results. This can be done by passing the extracted profile directly to Rufous' main function:

```
mainExtract = do
    (_, dug) <- extract main
    let profile = extractProfile dug
    mainWith args{signature=_Queue, profiles=[profile]}
```

This step generates tables just like the previous example. This time however, they are easier to understand. Since there is only one profile, the programmer only needs to concern themselves with picking the implementation that gives the best overall performance.

Example 5.1.3. Rufous as a *correctness* check.

Finally, let's explore the idea of using a *reference implementation* to automatically check the correctness of implementations.

If the user has a simple, correct reference implementation of their ADT, they can simply make it an instance of the ADT typeclass. Then, the programmer can just follow the steps of Example 5.2.1. This will not only time the reference implementation (for small DUGs it may be the most appropriate!), but it will also implicitly perform a correctness check. If any of the observations give inconsistent results, the tool will notify the user and print the related DUG.

However, Rufous was not designed to do this. Therefore the related DUG is large, and tracing the error is difficult. Tools like QuickCheck [19] solve this with a process of *shrinking*, where the large failing example is shrunk to a smaller failing example automatically.

Summary of Ease of Use The effort required by the user to perform most tasks in Rufous is mechanically quite minimal, with a few exceptions: Creating shadow structures is a task that is often needed, and could feasibly be automated. Extracting profiles from a program to compare against is a lot of effort, and in some cases a great task if the program is not already using a class-based API. The output tables are often long, and difficult to interpret; they could be turned into a more automated selection mechanism before being displayed to the user.

5.2 Performance

An important part of any tool's usability is its performance, in both time and space.

All experiments were conducted on the same machine, with a 2.3 GHz i3 CPU, under the GHC Haskell compiler (version 7.10.3).

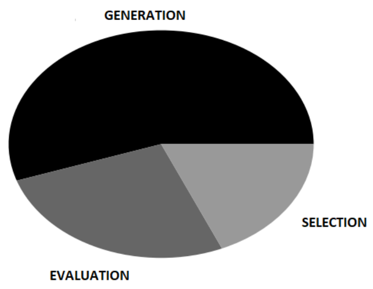
Execution time profiling The first experiment is simply to run Rufous, asking it to generate and evaluate a selection of large DUGs. For an average DUG size of $10,000$ nodes, generating 20 DUGs.

Figure 5.1a shows that Rufous spends approximately twice as much time in generation as it does in evaluation and selection combined. It is clear from the chart that Rufous spends the majority of its time on generating DUGs. Keeping the time taken for generation low is essential to make Rufous a viable exploratory tool. The implementation provided here can generate DUGs with many thousands of versions in under a minute.

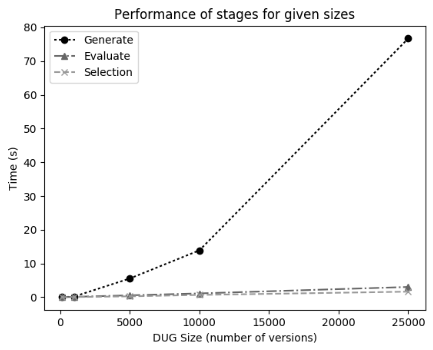
Time complexity and scalability Another measure of performance is how well Rufous scales. This experiment then generates many DUGs of different sizes. If the time grows linearly with the size of the DUG then this would make Rufous acceptable to scale for larger programs: the user could just leave Rufous running over a longer period of time.

However, Figure 5.1b shows that the amount of time Rufous spends on generation increases exponentially as the size of the desired DUGs increases. This drastically reduces scalability, making it difficult to generate DUGs with more than a few thousand nodes.

Memory Usage Rufous stores all of its intermediate structures in memory. Every DUG, and all the operation buffers. Typically the buffers are small. Phases of inflation during generation are short, and most of the operations get committed to the DUG after a short period. However, the generated DUGs only grow, they never shrink.



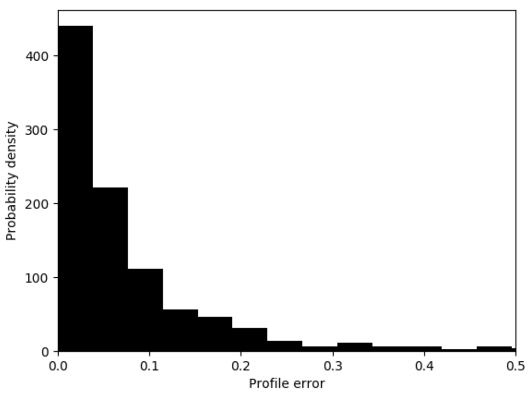
(a) Timing breakdown over the three stages.



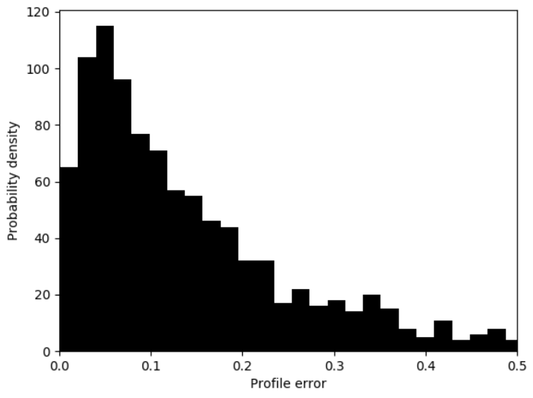
(b) Time taken for each stage as size of DUGs increase.

Figure 5.1: Performance analysis of Rufous.

In practice however, the compact DUG representation means that this is seldom an issue. In the performance experiments, many DUGs were generated at once each with tens thousand nodes and there was no noticeable impact on the available memory on the machine. After 50 DUGs the memory in use by Rufous was just under 500MiB. The programmer would need to generate hundreds of similar sized DUGs before they encountered problems.



(a) MSE of extracted profiles



(b) MSE of random profiles

Figure 5.2: Comparison of profile error in DUG generation algorithm

5.3 Effectiveness of Rufous as a Data Structure Selection Tool

Now, we will explore how effective Rufous actually is as a tool for data structure selection. This is done in two parts: First, an exploration of the accuracy in the DUG generation algorithm (§5.3.1). Then, we shall investigate what Rufous tells us about a selection of example data structures (§5.3.2).

5.3.1 Effectiveness of the DUG Generation Algorithm

The core of Rufous is the DUG generator. Ensuring it generates DUGs that match the input profile as accurately as possible is key to the usefulness of the tool. If Rufous does cannot generate a fair distribution of profiles then it would not be as useful a tool for the task of exploring the profile space.

One measure of effectiveness of this algorithm is the correlation between the input profile, and the extracted profile of the generated DUGs.

The Experiment To determine how well the DUG generation algorithm works, we calculate the mean-squared-error (MSE) between the input profile, and the profile corresponding to the generated DUG. This is then compared to the mean-squared-error of randomly generated pairs of profiles. Any significant improvement would indicate the DUG generation algorithm is generating DUGs with the correct profile.

Profile space sampling Generating profiles at random, uniformly, would be insufficient. This would lead to many profiles that were invalid. For example, profiles with 0 mortality are impossible – since DUGs are acyclic. Additionally,

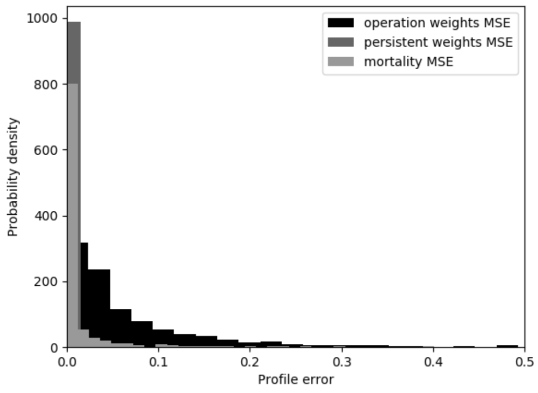


Figure 5.3: MSE of components of extracted profiles

there are many profiles that, while valid, are not useful. For example, profiles that contain zero weight for observers would force no work and is not a useful profile to generate.

Generating valid profiles by construction is difficult, whether a profile is valid or not is dependent on the ADT specification. For the Queue ADT, the mortality must be at least the sum of the generator weights (`m > o(empty)`); This is not true for ADTs with operations that have multiple version arguments such as sets.

So, instead of constructing valid profiles, we define a *sufficient* condition for profiles. If satisfied, the profile is certainly valid. Then only sample profiles that satisfy that condition. We generate profiles that are *reasonable* descriptions of real-world usage. This means, typically low mortality (`< 0.1`) and low persistence (`< 0.1`). Finally, we add the extra constraint from earlier that for the Queue ADT, `m > o(empty)`.

Results It's expected that the error between input and output profile would be very low. This result would be the ideal: it would indicate Rufous generates DUGs with exactly the desired profile where possible. For randomly generated profiles, it would be expected to be much higher, with a mean of approximately 0.2.

Figure 5.2 gives the results of the experiment. It is clear that Rufous performs very well, with a mean of 0.07. However, there are some data points with a high error. Figure 5.3 shows that this high error is caused mostly by a mismatch in the operation weights themselves, with high correlation in the mortality and persistence.

	empty	*head*	*snoc*	*tail*	*mortality*	*pmf*	*pof*	*BQueue*	*ListQueue*	*RQueue*
A	6375	1124	1540	955	0.774	0.219	0.427	0.014s	0.050s	0.014s
B	1590	1887	3825	2693	0.301	0.131	0.086	0.020s	0.022s	0.019s
C	3534	446	4749	1265	0.416	0.072	0.097	0.088s	0.015s	0.013s
D	3728	1643	3558	1065	0.510	0.119	0.124	0.018s	0.022s	0.017s
E	485	3879	4689	941	0.160	0.092	0.079	0.023s	0.027s	0.024s

Table 5.1: Summary table for three Queue implementations for 20 DUGs for 5 K-means chosen clusters.

5.3.2 Profiling Example Data Structures

To see how well Rufous captures the usage of datatypes, we will run the tool on a selection of programs To really understand how well Rufous works, it's best to just try run Rufous to profile some data structures.

Queues Take three implementations of Queues, a simple linked-list implementation, batched queues [4], and the rotating "Physicist's" queues [5]. Both batched and Physicist's methods can be constructed in Haskell with amortized $O(1)$ operations [5]. However, the Physicist's Queue's will yield slightly better performance in Haskell [5]. Rufous should be able to detect this.

Using Rufous to generate 20 DUGs we get a table of results. If we manually take these results and apply a K-Means clustering, with five means, we get the the summary given in Table 5.1.

This table shows that Rufous did manage to discover this. DUG Clusters B, and E have relatively high persistences. They both have the Physicist's Queues, the *RQueue*, outperforming the other implementation on average. Cluster C demonstrates that in the presence of a large number of `snoc` applications, where many are persistent, the Batched Queue implementation is very inefficient.

	add	delete	diff	empty	intersection	size	union	mortality	pmf	pof	BalTree	ListSet
A	744	645	359	1283	444	405	1110	0.030	0.132	0.073	7.90e-03s	9.49e-03s
B	781	1274	570	1236	304	556	268	0.121	0.093	0.040	7.78e-03s	9.17e-03s
C	177	204	92	194	73	170	79	0.088	0.139	0.093	1.22e-03s	1.28e-03s
D	937	791	364	416	189	2134	161	0.159	0.182	0.067	1.05e-02s	1.21e-02s

Table 5.2: Summary table for two Set implementations over 20 DUGs with 4 K-means chosen clusters.

Functional Sets Table 5.2 is example output from Rufous on the two functional set implementations given in Chaper 1, which is then run through a K-means clustering algorithm to find five clusters, and then takes the means of each cluster.

This example clearly shows the difference operation weights have on the appropriate structure. If the DUG is small, as in *C*, then the simple `ListSet` implementation beats the binary tree. As the DUG gets larger, the balanced binary tree starts to become more efficient. It also shows that for highly persistent usages, as in *D*, neither implementation is particularly efficient.

5.4 Chapter Summary

This chapter has investigated how well suited Rufous is to the task of data structure selection. It investigated the automation and ease of use (§5.1), concluding that Rufous performs well for the primary task of defining the ADT and running Rufous, but is substantially less user-friendly in the extraction phase. This chapter then ran experiments to explore the performance of Rufous (§5.2). The performance was good for small DUGs (less than $10,000$ nodes), but Rufous struggles with larger examples with poor memory efficiency and an exponential time generation algorithm. Finally, the effectiveness of Rufous as a data structure selection tool was tested more directly (§5.3). The experiments revealed that Rufous' generation algorithm performs very well, accurately capturing the desired properties in the generated DUG, and that Rufous was able to discover (with some work by the user) some interesting, non-trivial (and already known) performance characteristics of implementations of a few common data structures.

6 Conclusions and Future Work

This chapter will first draw conclusions on how effective Rufous is at the goal it was set out to achieve (§6.1), then suggests possible future extensions to Rufous (§6.2).

6.1 Conclusions

This report has detailed what Rufous is (Chapter 1); discussed related work on data structure selection (Chapter 2); laid out and justified requirements (Chapter 3); discussed challenges during implementation and proposed solutions (Chapter 4); and evaluated the solution for performance, effectiveness and acceptability (Chapter 5).

This section discusses Rufous as a tool, describing what parts worked well, and which parts did not. Conclusions will be drawn on four separate topics: (1) the typeclass API, designed in Section 4.1.4. (2) the timing reports Rufous generates, (3) the DUG representation and (4) the DUG generation algorithm presented in Section 4.3.2.

Typeclass API One core criticism of Auburn was the fact it was not user-friendly [1]. We solve this problem with a typeclass-based API for the specification and implementations of ADTs. Typeclasses for the ADT specification turn out to be both good in some circumstances, and bad in others. For writing the ADT specification, the typeclass API makes it much easier. Section 5.1 describes, in detail, how simple the ADT definition becomes with a typeclass. However, this very decision makes extraction incredibly difficult in some circumstances.

Using a typeclass to define the ADT specification is typically a very useful aspect of Rufous, it means that the compiler can typecheck both the specification and implementations. It also means that instances of the typeclass can automatically be promoted to implementations of the ADT, even if they already existed. It means that defining reference implementations, or precondition checking *shadow*s is all a uniform API, you just write an instance.

Rufous Output – Timing Reports Rufous' primary output is a summary table. This table contains all the timing data for each DUG. Whilst at first it would seem like all a programmer would need to begin using Rufous, Section 5.1.1 demonstrated that this is not enough.

When generating many DUGs, the user needs to then manually comb through the results trying to group the output by profile to get an effective measure of how an implementation performed. Section 6.2 discusses a few alternatives to just outputting the raw timing data.

DUG Representation Rufous uses a very different representation of DUGs to Auburn. This was mostly a beneficial change.

The unified DUG type meant that DUGs can be freely passed between the evaluator, generator and extractor without conversions between intermediate representations. Additionally, encoding the DUG as a functional graph directly made the representation very compact and efficient (See Section 5.2). The ability to use the DUG to annotate nodes abstractly also made evaluation very simple.

DUG Generation Algorithm Section 4.3.2 describes an algorithm for generating DUGs that conform to a given profile. Evaluation shows that this algorithm is very effective (§5.3.1). However, it also shows that the algorithm lacks optimisations to make it performant (§5.2).

With a few improvements and optimisations the algorithm could be made faster, such improvements are discussed in Section 6.2.

Final Remarks Building upon the work done in Auburn [1], we built Rufous. A tool for profiling purely functional data structures.

Using Rufous to guide optimisation of an existing program is very ineffective. Extracting the profile from an existing program can be difficult and easy to get wrong, and may require a lot of modification to the source program to use the typeclass ADT instead. Even if the user did manage to extract a profile, Rufous is very inefficient at generating very large DUGs.

However, Rufous performs very well as a tool for exploring the profile space. With only a small amount of user input, Rufous performs many otherwise laborious tasks. It can generate many, DUGs, each with tens of thousands of nodes, in only a few minutes. It quickly, and efficiently profiles each implementation, performing a correctness check as it does so. The user is then presented with the dataset and is able to see how the performance of each structure changes as the profile of the DUG does.

6.2 Future Work

Many areas for future work have been identified.

Extending the ADT signatures Rufous heavily restricts the signatures of ADT operations. There are two primary constraints that cause problems.

Rufous does not accept ADTs with higher-order signatures. These are common operations that programmers often want their data structures to do, such as:

```
map :: (a -> b) -> List a -> List b
fold :: (a -> b -> b) -> b -> [a] -> b
```

Rufous does not accept ADTs with non-version arguments other than `a`, `Int` or `Bool`. This could be relaxed to allow operations such as:

```
elements :: Set a -> [a]
```

Improved Selection Mechanism Simply printing timing data in tables is insufficient as a method of selection. Chapter 2 covered many other techniques for selection, including on-line ones and Auburn's decision-tree generation. Even a simple, automated, K-means clustering as was done manually in Section 5.3 would improve usability here greatly.

Shadow Guessing Defining shadow implementations may be a lot of redundant work for the user. Auburn could *guess* a shadow implementation based on how operations affect the size. Rufous could do something similar to reduce the burden on the programmer for defining simple shadows.

Correctness checking The approach of using a reference implementation to check correctness of the implementations was discussed in Section 5.1.1. Improvements could be made to make this a full feature of Rufous. Currently any inconsistency is simply printed to the user with the DUG it came from. To make the inconsistency easier to trace, Rufous could run a *shrinking* process similar to QuickCheck [19] to produce smaller, easier to understand DUGs.

Performance Improvements One key problem with Rufous is its inefficient generation of large DUGs. Rufous could improve performance by restricting the size of the buffers, or by only considering version arguments from a subset of the generated DUG. These would bound the amount of work Rufous does for each generated node.

Profile Extraction Using the typeclass specification for ADTs was great for usability for defining ADTs. However, when a program already exists and it does not use Rufous or a typeclass specification for its ADTs then extracting the usage from that program is very difficult. The programmer either has to manually extract the profile, or convert their entire program to use the new classes. Rufous needs some way of augmenting a pre-existing implementation with a logging side-effect to build a DUG, in a way that does not force the programmer to alter their existing program's source.

Bibliography

[1] Graeme E Moss. PhD thesis, University of York, 2000.

[2] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[3] John Backus. Acm turing award lectures. chapter Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. ACM, New York, NY, USA, 2007.

[4] Robert T. Hood and Robert C. Melville. Real time queue operations in pure lisp, 1980.

[5] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[6] Oracle. HashSet (Java Platform SE 7). `https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html`, 2018. [Online; accessed 24-Apr-2018].

[7] PSF. Python 3.6.5 documentation - Built-in Types - Sets . `https://docs.python.org/3/library/stdtypes.html#set`, 2018. [Online; accessed 24-Apr-2018].

[8] S. Adams, University of Southampton. Department of Electronics, and Computer Science. *Implementing Sets Efficiently in a Functional Language.* Technical report series / Department of Electronics and Computer Science, University of Southampton.

[9] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[10] James R. Low. Automatic data structure selection: An example and overview. *Commun. ACM*, 21(5):376–385, May 1978.

[11] Guoqing Xu. *CoCo: Sound and Adaptive Replacement of Java Collec-*

tions, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[12] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, July 1981.

[13] Graeme E. Moss and Colin Runciman. Inductive benchmarking for purely functional data structures. *Journal of Functional Programming*, 11(5):525–556, 2001.

[14] Lixia Liu and Silvius Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 265–274, Washington, DC, USA, 2009. IEEE Computer Society.

[15] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[16] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[17] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in setl programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, April 1981.

[18] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[19] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279, 2000.