A dynamic tree data structure for the linearisation and tree-contraction cases in the functional programming setting

Juan Carlos Sáenz-Carrasco

Confirmation Review Report
The University of Sheffield
Department of Computer Science

November 2016
Abstract

Dynamic data structures have been extensively studied in the last three decades due to their wide applicability in many contexts. Recently, several implementations and experimental studies have been conducted investigating the practical merits of fundamental techniques and algorithms. However, most of the implementations and analyses have been done for the imperative programming paradigm. In this report, we survey several implementations for both programming paradigms, imperative and functional, along with their experimental studies for dynamic graphs (DynGs) and dynamic trees (DynTs). Potential applications and related data structures are also described. What has been lacking until now, to the best of our knowledge, is a clean way of describing such dynamic data structures and their algorithms in a functional language - especially a non-strict one without disposing the main features of functional languages: independence of order or evaluation, referential transparency, non-strict semantics, and so on.
Acknowledgements

I am extremely grateful to Dr Mike Stannett, who has been an exceptional supervisor. Also, thanks to Paolo Veronelli who gave me much valuable advise concerning functional programming. My gratitute goes as well to my colleagues in the Department of Computer Science at the University of Sheffield. This report would have not been possible without the financial support of El Consejo Nacional de Ciencia y Tecnología (CONACyT) of the Mexican government.
Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>Preface</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.1 Background and Motivation</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.2 Aims and Scope</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.3 Why functional setting?, Why Haskell programming language?</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1.4 Overview of this Report</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2.1 Computational complexity and amortised analysis</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2.2 Lazy vs Strict Lazy evaluation</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2.3 Stateful computation</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2.4 Data-structure operations and algorithms</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2.5 Tree data structure</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>2.5.1 Balanced trees</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>2.6 Ternarization</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>3 The dynamic tree problem</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.1.1 The dynamic tree problem</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.2 Related problems</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.2.1 Dynamic sets</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>3.2.2 Disjoint sets</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>3.2.3 Splay trees</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>3.2.4 Dynamic graphs</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>3.2.5 Database and file system data structures</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3.3 Dynamic tree approaches</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>Path decomposition</td>
<td>17</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Tree contraction</td>
<td>18</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Linearisation</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Dynamic tree applications</td>
<td>23</td>
</tr>
</tbody>
</table>

#### 4 Literature review

| 4.1 | Introduction | 24   |
| 4.2 | Previous research on dynamic trees with functional data structures | 24 |
| 4.2.1 | Purely functional data structures | 24 |
| 4.2.2 | Source code of Link-Cut trees | 26 |
| 4.3 | Other dynamic data structures and related problems | 27 |
| 4.3.1 | Persistent Union-Find data structure | 27 |
| 4.3.2 | Functional programming and graph algorithms | 27 |
| 4.3.3 | Inductive graphs | 27 |
| 4.4 | Review of functional finger trees | 29 |
| 4.4.1 | Applications of Finger trees | 30 |
| 4.4.2 | Developing the data structure | 30 |

#### 5 Implementation of Euler-tour trees in functional programming

| 5.1 | Introduction | 36   |
| 5.2 | Functional perspective for ET-trees | 37   |
| 5.3 | ET trees through Finger trees | 37   |

#### 6 Conclusions

| 6.1 | Problem formulation and future work | 44 |
| 6.2 | Problem formulation | 44 |
| 6.2 | Future work | 45 |

#### Bibliography

| 7.1 | Problem formulation and future work | 47 |

iv
Preface

Source code of the programs and functions regarding the contributions in this report can be found on the website https://github.com/jcsaenzcarrasco/ConfRev2016/

Font convention

We used the Times font in most part of this document, whereas the italic style is used to denote specific terms for which we are giving an explanation. Since we are describing Haskell source code, we use the default font in the \LaTeX\ package \texttt{minted} when referring to any function or component. Also, source code is coloured and shaded. In the mathematical context, we use calligraphic letters as in $\mathcal{A}, \mathcal{B}, \mathcal{C}$. 
Chapter 1

Introduction

1.1 Background and Motivation

Data structures impact (to a greater or lesser extent) in the solution of problems in Computer Science. Dealing with such structures demands a careful consideration at the time to store and to retrieve information. Since the available space (memory allocation) is finite and limited by other applications, an efficient functioning of these structures should be taken into consideration. Besides achieving an efficient utilisation of the space, the operations that manage the data should be also part of the design of the structures. In fact, data structures are implicitly defined through the set of operations used to manipulate them.

The way a data structure preserves its data is not a trivial task. We can classify them in two general approaches, the persistent and the mutable. The former, also called functional, implies that updating the data structure does not destroy the existing version, but rather creates a new version that coexist with the previous one. On the other hand, mutable data structure allows changes on the same version. Although perhaps most of the software written so far is for mutable data structures, the demand for functional data structures is growing prompted in part by the growth in software development in the large scale [HHW15].

Strictly speaking, all data structures are dynamic since they increase or decrease their size (i.e. insertions and deletions among other operations). The term dynamic in this context is to some extent vague, varying depending on the specific data structure. For instance, Italiano, Demetrescu and Finocchi [DF10:la] define dynamic graphs as data structures that allow the execution of updates and queries without the necessity of computing an algorithm over the whole structure. This is tremendously important, especially
for big graphs. Cormen et al. [CLRS09] define *dynamic sets* to the collection of data manipulated by algorithms that can grow, shrink, or otherwise change over time. Sleator and Tarjan [STS83] coined the term *dynamic trees* to the solution, in an efficient way, of maintaining a collection of vertex-disjoint trees under a sequence of two kinds of operations: a *link* operation that combines two trees into one by adding an edge, and a *cut* operation that divides one tree into two by deleting an edge. Although the latter has been extensively studied for mutable structures, we have found, to the best of our knowledge, an absence in the analysis, design and implementation for the functional programming setting.

This report describes an investigation, as a survey and proposal, into the development of a tree data structure for problems that require dynamic changes, such as deletions, insertions or simply queries. Such changes can be given individually or as a batch. This work has been motivated by an interest in developing such structures for the functional programming paradigm that tackle this problem in a more effective way than currently exists. In addition, given the relation of dynamic trees to other dynamic data structures such as dynamic graphs and big data, this investigation may also benefit the development of optimisation techniques that can be applied to other such problems.

### 1.2 Aims and Scope

Since persistent data structures and pure functional programming forbid assignments, it is very difficult to design and implement the structures together with their operations while pursuing the same temporal (and space) performance as in mutable data structures. Even if monadic computation is in place (where stateful programming is possible) there are other factors such as lazy evaluation, space leaks and so on, that difficult the analysis for the data structures. These are some arguments in favour of applying case studies, experiments and benchmarking, that is, a practical approach for the present investigation.

As expressed above, there is an open space for dynamic trees to be analysed and implemented in the functional setting. Two main objectives are pursued when constructing the data structure: performance running times the same order of magnitude as its counterpart (mutable structures) and maintaining purity as much as possible (avoiding stateful computations). Initially, this investigation considers finding one implementation solution and running experiments for the approaches *tree contraction* and *linearisation*. Then, we address the situation in which under the same data structure (or
one with minimal interface) both approaches are able to perform dynamic tree operations such as link and cut.

The main aim of this research (at the end of the doctoral programme) is to investigate extensions to the design, implementation and application of a data structures for dynamic trees in the functional programming setting. To the best of our knowledge, scarce work in this area has been published in the literature.

1.3 Why functional setting?, Why Haskell programming language?

Almost three decades ago, Hughes [Hug89] described the importance and benefits of developing software by designing, analysing and implementing programs in the functional programming setting. In particular, Hughes highlighted modularity as the key factor for reliable and easy to maintenance software. Recently, Hu, Hughes, and Wang [HHW15] reassure such assertions by showing current facts about the growth in the software industry such as the appearance of new functional languages and the incorporation of functional features for current tools. Also demanding best practices and a community asking and looking for more functional programming is part of the evidence.

In a recent study conducted by Nanz and Furia [NF15], the top eight most representative programming languages for the major programming paradigm were selected. Procedural: C and Go; Object-Oriented: C# and Java; Functional: F# and Haskell; and Scripting: Python and Ruby. In order to show impartiality, the study undertook 7,087 solution programs to 745 tasks.

On the other hand, by analysing the energy behaviour of a purely functional language, in particular Haskell, Lima et al. [LSNL+16] showed benchmarking and studies about sequential, concurrency, testing, profiling among others that have been reduced the energy footprint.

1.4 Overview of this Report

The basic definitions and terms along the document are detailed in Chapter 2. Definitions about the domain problem, the dynamic tree, as well as the data structures related are described in Chapter 3. Also in this chapter a formulation of the problem of research is exposed. The review of the literature regarding the functional programming setting is included in Chapter 4 altogether with the current functional data structure we consider, so far, as the one towards the solution of the problem. In Chapter 5 we attempt to
give a solution to a specific approach in the dynamic tree problem, that is, linearisation or the Euler-tour representation. Finally, in Chapters 6 and 7 we conclude the report as well as present a formulation of the research problem accompanied with work for future research within the following two academic years.
Chapter 2

Preliminaries

2.1 Computational complexity and amortised analysis

Complexity theory is concerned with the identification of problems that are computationally easy to solve and problems that are computationally hard to solve [MD79]. From a general point of view, the efficiency of an algorithm is assessed in terms of the computing resources that are needed to execute the algorithm and this includes execution time and space. The execution time is the number of steps that the algorithm takes to process the input and give an answer. The space is an indication of the amount of memory that is needed to run the algorithm. However, most of the algorithms in the literature are usually expressed in terms of its time complexity. As we are dealing with data structures rather than algorithms, the focus is then turn to the operations on such structures which also comprise a dynamic behaviour.

In an amortised analysis, introduced by Sleator and Tarjan in [ST85], the average time required to perform a set of data-structure operations over all the operations is computed. With amortised analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortised analysis differs from average-case analysis in that probability is not involved; an amortised analysis guarantees the average performance of each operation in the worst case.

In [CLRS09] three techniques to calculate amortised analysis are fully explained. For the rest to this document we will refer as amortised to the aggregate analysis which is the average cost per operation, i.e. for different operations on a data structure may exist different performance. It is worth to say that the perhaps this technique is not the best for every data structure
here analysed but for practical reasons and as initial investigation we will stick to this amortisation technique.

Example

Consider a queue splitted in two parts: the front of the queue and the rear of the queue. The front of the queue will be store elements in normal order (from the left), so remove elements (pop operation) is $O(1)$; the rear of the queue will be stored in reverse order, so inserting elements (push operation) at the end of the queue takes also $O(1)$.

In addition, we also keep track of the size of both lists. We will use this to enforce the following invariant: The front of the queue cannot be shorter than the rear. This is only illustrative and other invariants can take place.

When the invariant is violated, we restore it by moving the elements from the rear of the queue to the front; since the rear of the queue is stored in reverse order, but the front is not, the rear must be reversed, which takes $O(n)$ (where $n$ is the number of elements of the current list to be reversed). The invariant is violated when shrinking the front or growing the rear.

Now, consider what happens when we insert [1..7] into an empty queue in an eager evaluation:

<table>
<thead>
<tr>
<th>Queue 0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue 1</td>
<td>[1]</td>
<td>0</td>
</tr>
<tr>
<td>Queue 3</td>
<td>[1..3]</td>
<td>0</td>
</tr>
<tr>
<td>Queue 3</td>
<td>[1..3]</td>
<td>1</td>
</tr>
<tr>
<td>Queue 3</td>
<td>[1..3]</td>
<td>2</td>
</tr>
<tr>
<td>Queue 3</td>
<td>[1..3]</td>
<td>3</td>
</tr>
<tr>
<td>Queue 7</td>
<td>[1..7]</td>
<td>0</td>
</tr>
</tbody>
</table>

What happened here is that we only needed to reverse $n$ elements after having inserted $n$ elements; we therefore say that the amortised complexity (the complexity averaged over all operations) of the reverse operation is in fact $O(1)$. Details of the functional implementation for this example can be seen at the Well-Typed blog post[1].

2.2 Lazy vs Strict Lazy evaluation

Most functional programming languages can be classified as either strict (sometimes called eager) or lazy (sometimes called non-strict), according to

[1]https://www.well-typed.com/blog/2016/01/efficient-queues/
their order of evaluation. No one is superior to the other for all applicable cases (e.g. `foldr` vs `foldl` vs `foldl'` in Haskell). The difference between the two evaluation orders is most apparent in their treatment of arguments to functions.

In strict languages, function arguments are evaluated in advance. In lazy languages, arguments are evaluated in a demand-driven fashion; they are initially passed in unevaluated form and are evaluated only when the computation needs the results to continue. Furthermore, for an evaluated argument, the result or value of that argument is cached so that if it is ever needed again, it can be looked up rather than recomputed. This caching is known as memoisation. This is not the same as the memoisation in dynamic programming.

The above impacts straightforward in the design and analysis of data structures. Strict languages can describe worst-case data structures, but not amortised ones, and vice versa for lazy languages. To be able to describe both types of data structures, one needs a programming language supporting both evaluation orders. Fortunately, the combination of strict and lazy evaluations in Haskell [Jon03] (the language we use in the present document) is not difficult.

2.3 Stateful computation

Conventional imperative algorithms change the state of itself or other entities on which there is a relationship or call. Most of the time this change is done by the assignment, that is, the same identifier having different values (loops, data structure elements, etc). On the other hand, functional programming setting forbids this situation. In fact, many of the advantages of functional algorithms come from not having access to the state, however, in some cases this seems unavoidable.

2.4 Data-structure operations and algorithms

Sometimes is really difficult to distinguish between the operations that belong or define a data structure from those that are actually “independent” algorithms. On the other hand, some algorithms require specific data structures to be efficient or faster. For instance, the original Dijkstra’s shortest path algorithm, with an initial queue as the data structure, was improved, in terms of speed, thanks to a change of its data structure (i.e. holding the same data), in this case a Fibonacci heap. That is, the time complexity was
speed up from $O(v^2)$ to $O(e + v \log v)$ where $v$ is the number of vertices and $e$ is the number of edges in the graph. It is not difficult to prove that

$$(e + v \log v) \leq v^2$$

In general, the data structures described in the present document are actually abstract data types or simply ADTs. Following [Sta13], an ADT is defined by specifying its Types Also known as Sorts. What types have to be defined before the ADT is itself fully defined?

Syntax What operations are available, and what are their types?

Semantics How are the operations related to one another?

The goal is to provide a complete semantics; a set of rules that is powerful enough to let you prove anything about the data type that ought to be true. In general you do this by identifying which of the operations are constructors and which aren’t. You then provide one semantic rule for each non-constructor/constructor pair. That is, for each constructor $C$ and each non-construct $F$ you write down a rule explaining the overall effect of ‘first apply $C$, then apply $F$’.

2.5 Tree data structure

There is not a unique tree data structure. Depending whether the tree is labelled, rooted, directed, balanced, self-adjusted and so on, there is a specific ADT. In general a tree data structure can be defined as a connected graph with no cycles (directed or undirected). The following are examples of trees

Figure 2.1: Left: directed tree; center: unrooted labelled-vertex tree; right: rooted tree (reproduced from [Wer06])

9
whereas the following are examples of non trees.

Figure 2.2: Left: self loop; center: undirected cycle; right: non connected graph

In the present report we refer to a small portion of the tree data structures. In Chapter 5 we show an implementation with an unrooted tree. Next, balanced trees are covered.

2.5.1 Balanced trees

In the domain of tree data structures, the subset of balanced ones plays a key role when efficiency is on demand. One small example showing the importance of being balanced is the binary search tree with no self-balancing property or operation.

Figure 2.3: the input list (on the left) and its BST

One way to avoid the above undesirable structure is to feed it with random inputs, such
Now the problem relies on the input stream rather than the structure itself. Both problems can be avoided with a proper balanced tree structure: self-balanced trees or balanced-by-definition trees. The former with operations or invariants indicating the structure when to perform the balance of itself (or changing its structure). Examples are red-black trees \cite{Gottlieb1978}, AVL trees \cite{Adelson84} and splay trees (Section 3.2.3). The balanced-by-definition trees keep the balance by placing all the leaves at the same depth. Examples are 2-3 trees and 2-3-4 trees (discussed in \cite{CLRS09} and Section 4.4), and B-trees \cite{Bent02}.

### 2.6 Ternarization

The term *ternarization*, in the context of dynamic trees, is the process of reducing each vertex degree to a maximum of three.

The following example leaves a clearer idea of the term. Reproduced from \cite{Wer06}

![Figure 2.5: Example of ternarization. Every vertex with degree four or greater is replaced by a chain of vertices of degree three](image)

Figure 2.4: the input list (on the left) and its BST
The ternarization process is commonly used in computing topology, top and RC trees. In Chapter 3 we will see that ternarization is useful to compute dynamic trees.
Chapter 3

The dynamic tree problem

3.1 Introduction

In this chapter we present the definitions, and problems regarding the current solution of dynamic tree, as well as related problems, data structures and proposals for further work. At the end of the chapter we describe the potential formulation for the same or closer solution of the dynamic tree under the functional programming setting. All the operations described from now on can be grouped into two categories: queries or lookups, which simply return information from the structure, and modifying operations which change the structure, called updates.

3.1.1 The dynamic tree problem

According to Sleator and Tarjan [ST83], the dynamic tree problem is that of maintaining a collection of vertex-disjoint trees under a sequence of two kinds of operations: a link operation that combines two trees into one by adding an edge, and a cut operation that divides one tree into two by deleting an edge. Each operation should require no more than $O(\log n)$ time.

Initially, the name for these dynamic trees were called Link/Cut trees, but many authors refer them as ST-trees after the authors.

3.2 Related problems

The following data structures are not actual dynamic trees but, to some extent they support similar operations such as insertion (or linking) and deletion (or cutting). The major differences are explained in each section.
3.2.1 Dynamic sets

Mostly covered in [CLRS09], the term set here refers to any collection of data that is able to support queries and updates. So, a dynamic set can be a list, a queue, a stack, or a tree. The major difference between dynamic sets and dynamic trees is that of for any allowed operation, a dynamic set is closed under the same structure, that is, the target structure is single, there are not supersets or subsets.

3.2.2 Disjoint sets

Disjoint-sets are somewhat the continuation of dynamic sets. That is, elements in different sets are to be computed following some constraints to have a meaning. Similar to dynamic trees, disjoint sets create a superset under a link operation or two subsets under the cut operation. Following Cormen et al., [CLRS09, §21.1, p. 561]:

“A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. We identify each set by a representative, which is some member of the set. In some applications, it doesn’t matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.”

The last quote has a strong statement for our problem formulation, “...we care only ... if we ask ... of a dynamic set **twice without** modifying the set ... we get the **same answer** ...”. This is not only the definition for a persistent data structure, but the basis of a function, therefore our interest in the functional programming setting.

Testing connectivity is one of the application of disjoint-sets. It is the core of the Union-Find algorithm [Tar83] and the Kruskal’s algorithm [Kru56] for building the minimum spanning tree (MST).

Since the operations for disjoint sets are **union** and **find**, the lack of a **cut** operation highlights the difference respect to dynamic tree data structure.

3.2.3 Splay trees

Splay trees were introduced by Sleator and Tarjan [ST85]. This data structure was designed to be a binary-search-tree (BST) to maintain certain balance with low cost for both update and query operations. The main feature of splay trees is that after any operation, the vertex in question is left as the
root, i.e. recently accessed elements are retrieved again mostly in constant time.

Although splay trees are not fully balanced, it was shown by Pfaf [Pfa04] that in practice they perform as well as AVL, and red-black trees.

Splay trees are not considered dynamic trees since its operations for updates and queries are targeted to elements which are not trees, nevertheless splay trees are considered the best auxiliary tree for ST-trees according to Tarjan and Werneck [TW09]. The following depicts the three operations on splay trees. Circles are vertices and triangles are subtrees. In all cases, vertex $x$ is accessed (update or query) for the last time on the left, hence the right tree is re-balanced turning $x$ as the root.

Figure 3.1: The splaying step (adapted from [ST85]). (a) Zig: single rotation. (b) Zig-zig: two single rotations. (c) Zig-zag: double rotation.

3.2.4 Dynamic graphs

The study of dynamic graphs can be addressed according to its features: either the graph is directed, undirected, cyclic, acyclic, sparse, dense, labelled, unlabelled or any combination of those.
From the above, dynamic trees are called to build the solution for problems relying on dynamic undirected graphs. Mostly, the task for dynamic trees is graph decomposition, and partition either the vertices or the edges of the graph to be maintained. As stated by Demetrescu, Finocchi and Italiano [DFI04b], dynamic trees are really useful:

“Moreover, data structures that maintain properties of dynamically changing trees, . . . , are often used as building blocks by many dynamic graph algorithms.”

Another classification for dynamic graph problems is according to the types of updates allowed. In particular, a dynamic graph problem is said to be fully dynamic if the update operations include unrestricted insertions and deletions of edges or vertices. A dynamic graph problem is said to be partially dynamic if only one type of update, either insertions or deletions, is allowed. These terms are extensively referred in the literature when graphs, trees or algorithms support dynamic changes.

Among all the problems solved with the help of dynamic graphs, four are the most common: connectivity, the minimum spanning tree, the transitive closure, and the shortest path.

Zaroliagis [Zar02] gives a comprehensive survey of experimental implementations of dynamic graphs algorithms, stating among other insights, the discrepancies between the theoretical and the implementational issues. Specifically, in [Zar02, p. 237]:

“... the experimental results did not always comply with the theoretical analysis, implying that perhaps a different model of analysis is required for such a case”.

3.2.5 Database and file system data structures

In the database context, keeping up-to-date indices is a primary task. Kennedy and Ziarek [KZ15] introduce a generalization of adaptive indexes called just-in-time data structures or JITDs. A JITD is a class of indices that dynamically adapt to match offered workloads. It is an hybrid model between data structures managing updates of large blocks of data for external memory, such B-trees.

3.3 Dynamic tree approaches

There are three main approaches for studying the dynamic tree problem. The first, path decomposition represented by ST-trees, was originated in the
solution of problems in network flows (Tarjan [Tar83], and Sleator and Tarjan [ST83]). The second approach, tree contraction covered in Section 3.3.2 was initially devised to maintain a minimum spanning tree in the online fashion. Finally, linearisation described in Section 3.3.3 initially for finding the bi-connected components of an undirected graph by Tarjan and Vishkin [TV85] and later exploited by Hezinger and King [HK99] for testing connectivity on dynamic graphs.

From all literature describing the above mentioned approaches, perhaps the one given by Renato Werneck in [Wer06] is a depth compilation. In the general perspective, the data structures from the above approaches can be seen as having the same goal as the next function definition:

\[ f : t \rightarrow b \]

where \( t \) is an arbitrary tree, \( b \) is a balanced tree, and \( f \) is the dynamic tree computation.

### 3.3.1 Path decomposition

In order to compute faster solutions in Dinic’s algorithm [Din70] for the maximum flow problem, link-cut trees or simply ST-trees were designed to avoid repeated computations over paths. Instead of searching for an augmenting path edge-by-edge, ST-trees try to combine existing path segments to produce the desired path. This idea is illustrated in

![Figure 3.2: Left: original graph; right: potential paths](image)

The specific paths to be stored on the tree are determined by Dinic’s algorithm (not discussed here). Instead of managing an arbitrary tree, ST-trees hold the paths into a balanced binary tree, a splay tree.
The idea behind managing a self-adjusting binary tree is that every update or query takes $O(\log n)$. Original version of ST-trees stored values on vertices and then following computations such as mincost or addcost for vertices are possible in logarithmic time.

Figure 3.3 is an example between the original tree and its auxiliary binary balanced tree.

Figure 3.3: left: original tree; right: splay tree holding paths from original tree

### 3.3.2 Tree contraction

A contraction is a transformation of the original tree into a smaller one. Such transformation is done progressively. Contractions are based on two operations [MRS85], the rake operation eliminates vertices of degree one (Figure 3.4), while compress eliminates vertices of degree two (Figure 3.5). This is where ternarization takes place.

Information is stored on vertices, so when rake operation is performed the information of the eliminated vertex is passed to its unique neighbour. For the case of compress, the information is located at any of its neighbours or both. A contraction is a sequence of rake and compress operations that reduces the original tree into a single vertex which contains the information of the whole tree.

Figure 3.4: Left: original tree; right: after rake operation
Topology trees

Frederickson [Fre85] devised a data structure when a minimum spanning tree (MST) is to be maintained for an underlying dynamic graph (Figure 3.6). A topology tree is a representation of the hierarchy of clusters. A cluster is a grouping of contraction computations. The original tree is considered to be level zero of the contraction, with each vertex represented as an individual cluster. Level zero clusters are the leaves of the topology tree. Level 1 groups vertices either a single vertex or connected vertices (at most 2). Then, in level 3 the first contraction takes place. This process is repeated until only one vertex left, containing the information of the entire tree. Figure 3.7 visualizes this representation. The number of vertices in each level is reduced by at least $\frac{1}{6}$; which means that the height of the topology tree is $O(\log n)$. Figure 3.8 shows the balanced tree (topology) holding the MST in Figure 3.6. Whenever there is a link or a cut, the contraction must be updated.
Figure 3.7: Levels of the clusters formed during contraction and computing the topology tree

Figure 3.8: Topology tree: balanced tree holding the MST of Figure 3.6

**Top trees**

Alstrup et al. [AHLT05] designed a similar tree data structure than topology trees in order to maintain information in a dynamic forest. Unlike topology trees, top trees do not have constraints about degree, in other words, no ternarization is applied. The circular order to perform contractions matters, that is, the order in which compress and rake operations are computed. In the original paper counterclockwise order is assumed for each vertex.
Since there are not restrictions on vertex-degrees, dummy vertices are created in the computation of a round of contractions. A dummy vertex represents an edge that was not involved in any move in a previous round.

Werneck [Wer06] redesigned original top tree data structure in order to hold paths in the same fashion as ST-trees does. Also he determined the worst-case for this structure apart from the amortised given in the original paper which also $O(\log n)$ per operation.

Another feature of Werneck’s contribution is that top trees and ST trees share a similar self-adjusting data structure. Here, splay trees are called guarded splay. The details of splaying with guards are out of the scope of this report. However, it worth to mention that performance reached by this structure is also the same as the original splay trees, $O(\log n)$ per operation.

The following figure shows the rake (circles) and compress (shaded squares) operations. Dotted lines connect dummy vertices (the ones at the bottom are eliminated afterwards).

![Figure 3.9: The contraction process (to be read bottom-up). Left: the MST and its top tree on the right](image)

### 3.3.3 Linearisation

Formally introduced by Henzinger and King [HK99], Euler-tour tree (ET) is the simplest representation of dynamic trees so far [TW09].

The aim of the ET-tree data structure is to model the given tree (generally unrooted) as a tour and the tour as a binary tree.
The data structure can also be seen as a forest of Euler tours. By using splay trees Tarjan \cite{Tar97} showed that link and cut operations can be done in $O(\log n)$ amortised time. Similar bound applies for operations dealing with the traversal of the ET-tree (and the tour) such as finding the minimum element. Werneck \cite{Wer06} suggest that by using red-black trees as the binary search tree, the time is guaranteed $O(\log n)$ worst-case.

In order to represent the tour as binary search tree, the first step is to convert the tour into a linear list by breaking it at some arbitrary point, then the binary search tree is built by picking up those elements of the list that appear in symmetric order.

According to Tarjan and Werneck \cite{TW09}, ET-trees cannot handle path queries efficiently, as each edge in the forest appears as two vertices in the tour representation, and they may be arbitrarily far apart. This makes it difficult to aggregate information about a specific path.

Figure 3.10 shows the process of turning a tree into an Euler-tour representation and its binary tree afterwards. Circles represent the original vertices and squares the edges (forward and backwards) added during the process.
Dynamic tree applications

Dynamic trees have been included in several applications: solving problems on networks such as maximum flow and minimum cost path, Tarjan [Tar83]; as tools for solving algorithms on dynamic graphs [EGI98], [DFI04a]; merging dictionaries [Kar16]; combinatorial optimisation [Lee16] to name a few.

From the above, perhaps the more impact of dynamic trees is in the field of dynamic graphs. To this respect, four problems have been studied for the empirical and the theoretical points of view: connectivity, minimum spanning forest, transitive closure, and shortest path. Zariolagis [Zar02] and Tarjan and Werneck in [TW09] published a comprehensive list of results.
Chapter 4

Literature review

4.1 Introduction

In this chapter we review the literature of the dynamic tree problem solution from the perspective of functional programming setting.

4.2 Previous research on dynamic trees with functional data structures

4.2.1 Purely functional data structures

We mentioned earlier some differences between persistent and mutable data structures. Fiat and Kaplan [FK01] summarize the terms about the updating status of a data structure:

“We call the initial configuration of a data structure version zero. Every subsequent update operation creates a new version of the data structure. A data structure is called persistent if it supports access to multiple versions and it is called ephemeral otherwise. The data structure is partially persistent if all versions can be accessed but only the newest version can be modified. The structure is fully persistent if every version can be both accessed and modified.”

In general, the different configuration (mutability) for data structures can be seen as the following example. Assume $xs, ys, zs$ are identifiers for lists (linked lists). Let $++$ be the operation that appends two lists, that is,
a binary operator that given two lists, ++ returns the concatenation of the second list after the first one.

In Haskell, two consecutive colons :: means ‘is the type of’, and the rectangular brackets [ ] means ‘is a list of’, then

\[
\begin{align*}
1 & \quad \text{xs :: [Integer]} \\
2 & \quad \text{xs = [0,1,2]} \\
3 & \quad \text{ys :: [Integer]} \\
4 & \quad \text{ys = [3,4,5]} \\
5 & \quad \text{zs :: [Integer] \rightarrow [Integer] \rightarrow [Integer]} \\
6 & \quad \text{zs = xs ++ ys}
\end{align*}
\]

Now, in the imperative setting (and its corresponding pointer-based data structure) we can depict the above operation as (adapted from [Oka99a]):

Figure 4.1: Executing \( \text{zs} = \text{xs} \; ++ \; \text{ys} \). Arguments \( \text{xs} \) and \( \text{ys} \) are destroyed

In the functional setting, the above operation can be seen as (adapted from [Oka99a])

Figure 4.2: Executing \( \text{zs} = \text{xs} \; ++ \; \text{ys} \). Arguments \( \text{xs} \) and \( \text{ys} \) are unaffected
Fiat and Kaplan [FK01] investigated the transformations from a pointer-based data structure into a persistent one. In this report we focus only in persistent data structures except where otherwise stated. One seminal reading material for this topic is the book published by Chris Okasaki, [Oka99a].

Analogy to Section 3.2.1 Okasaki derives a functional implementation for common elementary, linear and non-linear data structures, such: lists, heaps, queues, binary search trees, red-black trees.

Respect to dynamic behaviour of data structures, Okasaki gives the underpinnings through concepts:

**Random-access lists** in [Oka95], describes the operations for lists through a tree data structure in order to speed up updates and look-ups.

**Splay trees** in [Oka99a], Okasaki replaces the trees for heaps in order to avoid the rebalancing of the structure for every query requested.

Among all the contributions by Okasaki [Oka96], perhaps the major ones in this subject, apart from his sixteen implementations, are the following:

**Persistent data structures** Showing that lazy evaluation and functional programming are not longer incompatible for amortisation as measurement of complexity in computations.

**Lazy and amortised analysis** Describing the benefits and drawbacks for laziness altogether amortised analysis in some data structures. Also, the analysis of *structural-decomposition* of some data structures, where lazy-rebuilding (lazy, then strict, and lazy again) allows improvements in the performance.

An instance of such lazy and amortised analysis, is the exercise given by de Vries[^1] relative to Efficient Amortised and Real-Time Queues in Haskell.

### 4.2.2 Source code of Link-Cut trees

In the Haskell package repository, or simply Hackage, Kmett uploaded the source code of a version of ST-trees (as Link-Cut[^2]). Unfortunately, the code does not show evidence supporting the claims for the complexity given neither contains references for experimental analysis.

[^1]: https://www.well-typed.com/blog/2016/01/efficient-queues/
4.3 Other dynamic data structures and related problems

4.3.1 Persistent Union-Find data structure

Regarding the structure described in Section 3.2.2, Launchbury and Peyton-Jones [LPJ94], described a way to encapsulate stateful behaviour safely (no side effects) in the functional programming setting. The applications motivating this work were at least the union-find algorithm and updates operations over hash tables.

Conchon and Filliâtre [CF07] devised an implementation of a persistent data structure for the union-find problem, claiming similar results than those in its counterpart imperative, although the programming language used in getting this results is not pure functional (OCaml).

4.3.2 Functional programming and graph algorithms

Similar to Okasaki, King presents in [Kin96] a comprehensive review of several graphs algorithms in the functional setting and points out the benefits and disadvantages from this paradigm. Although his research material does not show insight for dynamic data structures, it does analyse the static counterparts of shortest paths, minimum spanning trees and disjoint-sets from a calculational reasoning, pure and monadic perspective.

Another interesting contribution is the benchmarking between functional (Haskell) and imperative (C) implementations for the strongly connected components algorithm [Tar72].

One of the claims by King was that for some algorithms implemented in the functional setting, such the union-find and binary sort, it is necessary the incursion of monadic computations in order to achieve the same complexity as the imperative setting.

4.3.3 Inductive graphs

Graphs are particularly hard to reason with in the functional programming setting, specially in the pure one. One of the issues is the cyclic reference within the type of the data structure.

Erwig [Erw01] solves the cyclic type definition for a graph inductively. A graph can be either empty (base case) or defined by (inductive case)

- a list of edges pointing to a particular node,
- a node in question,
• a list of edges pointing from the particular node

Followed by the operator (&) to construct the graph.

There are other operators, making possible different settings for graphs such labelled, unlabelled, cyclic and acyclic. Erwig also provide a description for recursive functions to traverse such inductive graphs such, depth-search-first, shortest paths, minimum spanning trees among others. Another contribution by Erwig is a set of functions to deal with dynamic graphs in the functional graph library (FGL), accessible from the Hackage \[^{3}\] Unfortunately, Erwig’s material lack of experimental and complexity analysis.

Some authors have based their work about graphs algorithms in the Erwig’ proposal for inductive graphs.

**Structured graphs** The proposal by Oliveira and Cook, \[^{1}\] alike the type definition of Erwig’s inductive graphs, defines *structured graphs*. A structured graph is a directed graph where:

• the nodes are grouped into a hierarchy of regions;

• one or more designated named nodes in a region are the only possible targets for back-edges and cross-edges from nodes within that region (and its sub-regions).

The advantage of structured graphs over inductive ones is the potential application of more functional programming tools, such fixpoint \[^{4}\] computations and a more convenient and expressive interface. The proposal given by Oliveira and Cook is limited to the application of grammars, with no experimental or complexity analysis.

**RDF graphs** In the context of semantic web, Labra, Jeuring and Álvarez, \[^{2}\] and \[^{3}\], present an specialized (simplified) Erwig’s version called *inductive triple graphs*. The simplified representation of inductive graphs is due to three assumptions:

• each node and each edge have a label,

• labels are unique,

• the label of an edge can also be the label of a node

The implementation is presented in both pure functional language (Haskell) and in an impure one (Scala). A missing feature from inductive triple graphs is an experimental study. Similar to the above

[^1]: https://hackage.haskell.org/package/fgl

[^2]: A *fixpoint* or *fixed point* of a function $f$ is a value that doesn’t change under the application of the function $f$
functional insights, RDF graphs are presented with no experimental analysis. Also, details of complexity are scarce.

**Lazy functional graphs** Recently, Dexter et al., [DLC16], presented DeltaGraph, a data structure and a set of operations dealing with intentions, reactivity and propagation. The central idea, in order to compute dynamic graph operations in $O(1)$, DeltaGraph stores each intention of update in-graph (memoised) and then release (propagate) them gradually through the structure.

According to their experimental results, DeltaGraph under lazy evaluation is up to 50% faster than eager evaluation and up to 30% when parallel is performed to propagate updates.

This is the first material we have found, in the context of dynamic data structures for the functional setting, that includes experimental analysis for the different operations. However its complexity analysis is not complete.

### 4.4 Review of functional finger trees

Functional finger trees, or simply FT, were devised by Hinze and Paterson [HP06]. The term finger was coined by Guibas [GMPR77] referring to the pointers (in the imperative version) holding specific data within B-trees.

A finger tree is a functional representation of persistent sequences supporting access to the ends in amortised constant time, and concatenation and splitting in time logarithmic in the size of the smaller piece.

A comparison of FT operations against other related data structures and their performances is well described in the following table by Kmett.

<table>
<thead>
<tr>
<th>operation</th>
<th>Finger tree</th>
<th>Amortised bounds 2-3 tree</th>
<th>List</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons, snoc</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(1)/O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>viewl, viewr</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(1)/O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>length</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>append</td>
<td>$O(\log \min(l, n/2))$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n + m)$</td>
</tr>
<tr>
<td>split</td>
<td>$O(\log \min(l, n - l))$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>fromList,toList, reverse</td>
<td>$O(l)$</td>
<td>$O(l)$</td>
<td>$O(l)/O(l)/O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>index</td>
<td>$O(\log \min(l, n - l))$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Table 4.1: Common functional finger tree operations

4.4.1 Applications of Finger trees

The applications described in the original paper by Ross and Hinze ([HP06]), are the following:

**Sequences** An improving list data structure by offering random access.

**Max priority queues** Improving performance for current implementations in $\Theta(\log n)$.

**Ordered sequences** Holding and managing ordered sequences in the same data structure logarithmically.

**Interval trees** Supporting query operations on sets of intervals.

4.4.2 Developing the data structure

In this section we summarise the construction of finger trees in the functional programming Haskell [Jon03], following [HP06] and Gibiansky. Although it is a 2-3 tree originally, it allows from one to a maximum of four subtrees at each level in order to allow flexibility and perform queue operations efficiently.

```haskell
data FingerTree a = Empty
  | Single a
  | Deep (Digit a) (FingerTree (Node a)) (Digit a)

data Digit a = One a | Two a a | Three a a a | Four a a a a

data Node a = Node2 a a | Node3 a a a
```

The above data type defines a tree depicted in the following figure (reproduced from [HP06]), where $a$ is the type for the leaves in the tree, that is, Char.

Access to any of fourteen elements requires three steps, logarithmically in a general case. In applications such sequences, queues or deques elements are very often accessed at the front or the rear. In order to allow this, two

---

6 Augmented red-black trees [CLRS09, §3, chapter 14]

7 http://andrew.gibiansky.com/blog/haskell/finger-trees/

8 Double-ended queues
fingers are added. Also grabbing the first and last nodes, and pulling them up, letting the rest of the tree hang down, the following figure (reproduced from [HP06]) depicts the idea of such access.

This new data structure is known as a *finger tree*. The finger tree is composed of several layers (boxed in blue below) connected with a spine (the thick line), adapted from [HP06]:

Each layer of the finger tree has a prefix (left-most grey circles) and a suffix (right-most circles), as well as a link further down the spine. The prefix and suffix contain values in the finger tree on the first level, that is, contain values (2-3 trees of depth 0); on the second level, they contain 2-3 trees of depth 1; on the third level, they contain 2-3 trees of depth 2, and so on. The edges of the original 2-3 tree are now at the top of the spine. The root of the 2-3 tree is now the very bottom element of the spine. As we go
down the spine, we are traversing from the leaves to the root of the original 2-3 tree; as we go closer to the root, the prefix and suffixes contain deeper and deeper subtrees of the original 2-3 tree.

Adapting a little the definition of a finger tree we have:

```haskell
data FingerTree a = Empty
  | Single a
  | Deep {
      prefix :: Digit a,
      deeper :: FingerTree (Node a),
      suffix :: Digit a
  }
```

Our example so far is as follows. Auxiliary functions (layers) are involved for practicality.

```haskell
exampleFT :: FingerTree Char
exampleFT = layer1

layer1 :: FingerTree Char
layer1 = Deep prefix layer2 suffix
where
  prefix = Two 't' 'h'
  suffix = Three 'r' 'e' 'e'

layer2 :: FingerTree (Node Char)
layer2 = Deep prefix layer3 suffix
where
  prefix = Two (Node2 'i' 's') (Node2 'i' 's')
  suffix = Two (Node2 'n' 'o' 't') (Node2 'a' 't')

layer3 :: FingerTree a
layer3 = Empty
```

The main feature of finger trees devised by Hinze and Paterson, is the ability to deal with different applications (e.g. sequences, priority queues) under the same data structure. In order to do so, they add (annotate) a value in the internal nodes. This addition redefines the current definition:

```haskell
data Node v a = Node3 v a a a
  | Node2 v a a

data FingerTree v a = Empty
  | Single a
  | Deep {
      annotation :: v,
      prefix :: Digit a,
      deeper :: FingerTree v (Node v a),
      suffix :: Digit a
  }
```
Now, the annotated value $v$ is required to be monoidal. Recall that a monoid is the algebraic structure that has an associative operation ($\text{mappend}$) with an identity ($\text{empty}$). Also, the value $v$ should measure the elements of the tree. More precisely, there must be a measure function that accepts an element of the tree and outputs an annotation for that element. In order to do so, a new type class is implemented with the corresponding instances for the tree internals.

```haskell
class Monoid v ⇒ Measured a v where
  measure :: a → v

instance Measured a v ⇒ Measured (FingerTree v a) v where
  measure Empty = mempty
  measure (Single x) = measure x
  measure (Deep v _ _ _ ) = v

instance Measured a v ⇒ Measured (Node v a) v where
  measure (Node2 v _ _) = v
  measure (Node3 v _ _ _) = v
```

Finger trees in Hackage contain full details for the remaining code.

So far, the code and examples have shown how to build a finger tree and query for the left and right parts of such tree. In order to look for or update a specific element, Hinze and Paterson defined a search mechanism, called splitting. Splitting a finger tree provides a third finger besides the ones at the left and right ends of the tree.

The search mechanism uses a predicate $p$ of type $v \rightarrow \text{Bool}$ to operate on the monoidal annotation $v$. The search will find and return the location in the finger tree sequence where $p$ switches from being False to True; if no such place exists, the search will fail.

The uniqueness is maintained by a monotonic predicate. This is not the only way to do search, another technique is explained in the original paper.

**Example: random access**

The first application of Hinze and Paterson is computing an index over a sequence, that is, random access. The operator ($!$) is used to determine the value in the sequence at the index given, analogue to random access for arrays in Haskell.

[https://hackage.haskell.org/package/fingertree-0.1.1.0/docs/Data-FingerTree.html]
where \( \text{Seq} \ a \) is the type of the sequence, \( \text{Value} \) is the type of the element in sequence, \( \text{Size} \) is the type of the annotation which serves as reference to evaluate the predicate \( (> \text{Size} \ idx) \). \( \text{Size} \ 0 \) is an accumulator, and \( \text{tree} \) is the tree to search on.

The following is the monoid instance (specialized case) for \( \text{Size} \):

\[
\begin{align*}
\text{newtype} \ &\text{Size} \quad = \text{Size} \ \text{Int} \\
\text{newtype} \ &\text{Value} \ a \quad = \text{Value} \ a \\
\text{instance} \ &\text{Monoid} \ \text{Size} \ where \\
&\text{mempty} \quad = \text{Size} \ 0 \quad -- \text{identity value} \\
&\text{mappend} \ (\text{Size} \ x) \ (\text{Size} \ y) \quad = \text{Size} \ (x + y) \quad -- \text{binary operation} \\
\text{instance} \ &\text{Measured} \ (\text{Value} \ a) \ \text{Size} \ where \\
&\text{measure} \ _ \quad = \text{Size} \ 1
\end{align*}
\]

Following our example,

Below is the summary of the operations and their performance for constructing finger trees [HP06].
Table 4.2: Basic functional finger tree operations

The ASCII denotation for $\ll$ is $<\mid$ and for $\gg$ is $\mid>$
Chapter 5
Implementation of Euler-tour trees in functional programming

5.1 Introduction

The implementation presented in this chapter is not fully tested and the performance given is not formally proven. Nevertheless, to the best of our knowledge, this is the first attempt of a purely functional implementation for an Euler-tour representation of a dynamic tree. The current implementation comprises the following operations. A forest of trees (Euler-tours) is the universe.

\begin{itemize}
  \item **link** \(v\ \ w\)  Link the trees containing \(v\) and \(w\) by adding the edge \((v, w)\). Vertices \(v\) and \(w\) are supposed to be in different trees, otherwise an exception is triggered.
  \item **cut** \(v\ \ w\) Split the tree where \(v\) and \(w\) share an edge \((v, w)\), making two trees (and two tours). Vertices \(v\) and \(w\) are supposed to be in the same tree.
  \item **path** \(v\ \ w\) (Connectivity operation). If there is a path between \(v\) and \(w\) in the forest, returns such path, otherwise the reason about the lack of connection is given.
\end{itemize}

For practical purposes, let us limit our first implementation to unrooted and unlabelled trees.
5.2 Functional perspective for ET-trees

An Euler tour allows to model any tree as a sequence of its vertices. Our attempt is to model Euler tours and Euler-tour trees through a combination of types for linear (such as lists and sequences) and limited non-linear (such as trees). For instance, constructing an Euler tour given a list of elements, or given an Euler tour construct a tree.

Since finger trees (seen in Chapter 4) support logarithmic amortised operations and since we are aiming to perform similar bounds as the imperative counterpart, we implement our first version with finger trees as the underlying structure.

5.3 ET trees through Finger trees

We present the operations described in Section 5.1 following the notation of the Haskell programming language. Full source code is hosted in GitHub

Let us build up the implementation through an example, that is, a small non rooted tree and its Euler-tour or linear representation.

Figure 5.1: Original unrooted tree

(a) A tour of Figure 5.1: \textit{abcdcba}

(b) \textit{reversed} tour of Figure 5.1: \textit{abdcba}

Figure 5.2: Euler tour representations for trees

Notice that tours in Figure 5.2 were derived from vertex \textit{a} as the root. By picking a different vertex as the root, \textit{two} different tours will be generated. The number of different tours per tree also depend on the size of the tree and the number of branches per vertex.

For now, we keep these two tours per tree. The following is the definition for such pair of tours.

\begin{verbatim}
data Tour a = Tour (STour a) (STour a)
where the first \textit{(STour a)} represents the original tour and the second \textit{(STour a)} the reversed one.
Since each tour is a tree, we store it within a finger tree,

\begin{verbatim}
type STour a = FingerTree (TourMonoid a) (TourElem a)
\end{verbatim}

and its corresponding fields \textit{TourElem} and \textit{TourMonoid}:

\begin{verbatim}
newtype TourElem a = TourElem a deriving (Eq,Ord,Show)
newtype TourMonoid a = TourMonoid (Set a,Sum Int) deriving (Monoid,Show)
\end{verbatim}

Recalling the \textit{monoidal} annotation in finger trees, we do the same for our tours with the pair \textit{(Set a, Sum Int)}. With monoid \textit{Set} we can test membership of a \textit{TourElem a} in \textit{O(log n)} and with monoid \textit{Sum} we get its index also in \textit{O(log n)}.

The following function inserts elements into a tour.
The performance of this function is the length of the list. The tour (Tour) is constructed inductively, where the base case, line 2, the empty tour is built in $O(1)$ (Table 4.2). The inductive step, line 5, computes non empty cases also built up in $O(1)$ from both, original and the reversed tours, since the operators $\triangleleft$ and $\triangleright$ take $O(1)$.

For our example, in Figure 5.1 we have:

```
> fromList' "abcbdba"
 Tour [TourElem 'a',TourElem 'b',TourElem 'c',TourElem 'b',TourElem 'd',TourElem 'b',TourElem 'a'] (fromList' [TourElem 'a',TourElem 'b',TourElem 'd',TourElem 'b',TourElem 'c',TourElem 'b',TourElem 'a'])
```

Visually, the tour of the example looks like (just the original tour):

The annotation in the top vertex (Set Char, Sum Int) allows us to determine in a unique way which elements of the tour belong to this tree because its balanced-binary-tree (Set) as well as random access due to the index (Sum).

The operations per tree (tour) for updates and queries so far take $O(\log n)$ amortised as they follow the same operators as the finger tree (tables 4.1 and 4.2).

In order to complete the dynamic ET-trees, we define a new type to hold tours, that is, a forest of tours.
newtype TourForest a = TourForest (FingerTree (Set a) (Tour a))

For this type we annotate just the \texttt{Set} monoid since the index of an element is determined by the specific tour in the type \texttt{Tour}. Now the operations \texttt{link}, \texttt{cut}, and \texttt{path} are as follows.

\begin{verbatim}
link :: Ord a ⇒ a → a → TourForest a → TourForest a
link x y (TourForest f) = case select (member x) f of
  Nothing → error "vertex not found" -- vertex x
  Just (tour_x,f') → case member y $ measure tour_x of
    True → error "vertices already connected" -- x and y: same tour
    False → case select (member y) f' of
      Nothing → error "vertex not found" -- vertex y
      Just (tour_y,f'') → TourForest $ (splice (reroot x tour_x) y tour_y) $ f''
\end{verbatim}

The original forest (\texttt{f} in line 2) is preserved throughout the whole process. \texttt{f'} in line 4 is the forest not including the tour \texttt{tour_x} (the tour including the element \texttt{x}). If \texttt{y} is not a member of \texttt{tour_x}, then the \texttt{False} case in line 6 will return a new forest, \texttt{f''} altogether with \texttt{tour_y} and \texttt{tour_y} at position (given by the monoidal \texttt{Sum}) of vertex \texttt{y}. Finally, the link is performed in 9 by inserting (\texttt{$<$}) the tours to the forest \texttt{f''} which does not contain such tours.

The performance of \texttt{link} is \texttt{O(log n)} amortised and computed as

- line 2, testing membership of an element in the forest is \texttt{O(log n)}
- line 4, testing membership of an element in the tour is also \texttt{O(log n)}
- line 6, same as in line 4
- line 9, \texttt{reroot} uses \texttt{split}, so it takes \texttt{O(log min(n, n−l))}, and \texttt{\textless} is \texttt{O(1)}.

The case of \texttt{cut} operation is the following.

\begin{verbatim}
cut :: Ord a ⇒ a → a → TourForest a → TourForest a
cut x y (TourForest f) = case select (member y) f of
  Nothing → error "vertex not found" -- vertex y
  Just (tour_y,f') →
    case (do
      guard $ x 'member' measure tour_y
      let tour' = reroot y tour_y
      father x tour' >>= guard . (== y)
      return tour'
    ) of
    Nothing → error "vertices in different trees"
\end{verbatim}
Just tour_y →
let (tWith_x,tNO_x) = extract x tour_y
in TourForest <$(tWith_x ▷ tNO_x ▷ f')$

Same as with \textit{link}, we start testing membership of one of the vertices
given in line 2. $f'$ in line 4 is a new forest without the tour containing vertex
$y$. Unlike \textit{link}, we are interested in verifying that vertex $x$ is in the \textit{tour_y}
without generating a new forest (lines 5-10). In line 13, we \textit{extract} vertex $x$
from \textit{tour_y} having two new tours, the one hosting $x$ and its child ($tWith_x$)
and the remaining, that is, the one containing $y$ and its child ($tNO_x$). Finally,
in line 14 the new forest $f'$ increases the number of trees (tours) by one by
inserting the new sub tours.

The performance of \textit{cut} is $O(\log n)$ amortised and computed as

- line 2, testing membership of an element in the forest is $O(\log n)$
- lines (5-10), testing vertex $x$ to be in the same tour than $y$ without
generating a new forest takes $O(\log \min(l, n - l))$ because \textit{reroot} and
\textit{father} use \textit{split}.
- line 13, similar as previous lines is the case of \textit{extract}.
- line 14, $\lhd$ involves $O(1)$.

The final operation \textit{path} is described above.

\begin{verbatim}
  path :: Ord a ⇒ a → a → TourForest a → TourForest a
  path x y (TourForest f) = case select (member y) f of
    Nothing          → error "vertex not found"
    Just (tour_y,_)  → case member x $ measure tour_y of
    False            → error "vertex not found / NOT connection"
    True             → (cpath x y tour_y)
\end{verbatim}

In this operation we \textit{read only}, so no transformation over the forest is
done. Testing membership is done in lines 1 and 4 in $O(\log n)$ each. If
the requested vertices are found in the same tour then we call \textit{cpath} which
computes a list with path in $O(l)$, therefore \textit{path} complexity is $O(l)$.

Details of functions are given in the source code. Also, the error messages
here are replaced by proper exception management to avoid the application
to crash. Formal proofs and experimental study needs to be carried out to
claim the performance presented here. The potential worst case is when a
$O(\log n)$ in the forest leads to a different $O(\log n)$ in the tour, then $O(\log^2 n)$
is present. This is the case where refactoring is suitable, especially in the fashion of \textit{lazy-strict-lazy}, presented by Okasaki in \cite{Oka96} and exemplified by de Vries\cite{vries}.

\footnote{\url{https://www.well-typed.com/blog/2016/01/efficient-queues/}}
Chapter 6

Conclusions

The overall dynamic tree problem was well described in [ST83], [TW09] and [Wer06]. The related data structures and algorithms play an important role not only in the applicability of the dynamic problem but also as auxiliary constructs to improve its complexity for time and space such as balanced binary trees, specifically splay trees.

The present report focused on the literature review and suitability of implementing functional data structures (as claimed also by [HHW15]) in order to solve the dynamic tree problem.

The implementation of the linearisation approach for dynamic trees, i.e. ET tree, was coded into the functional programming language Haskell with successful performance with $O(\log n)$ amortised for the operations presented so far. However the results are incomplete and far to be considered as a formal evidence. The need for experimental studies is not limited to ET-trees under finger trees.
Chapter 7

Problem formulation and future work

7.1 Problem formulation

Definitions, applications and examples of the dynamic tree problem, seen in Chapter 3, are to some extent covered for the imperative programming paradigm. However, it is not the case for the functional programming setting to the best of our knowledge. We have seen that theory for dynamic data structures is scarce. Presented mostly for dynamic graphs as in [Erw01], [OC12], [GJR14a] and [DLC16]. On the dynamic trees side, specifically for the purely functional, an example is presented in the Hackage repository. In the experimental scene, so we have seen, the coverage to meet the theory falls short. We believe that the literature review presented in this report (Chapter 4) altogether our implementation attempt (Chapter 5) is sufficient to propose the following research projects.

Experimental Study Many reading material has been published in the pure functional data structures field concerning the practical performances of static algorithms for computing path problems, combinatorial problems, among other (see, e.g., [Oka99a], [Bir10]), but very little is known for the experimental evaluation of dynamic tree problem. In particular, to the best of our knowledge, there is no experimental study concerning the laziness, and persistence data structures of dynamic trees with Euler-tour representation and top trees. Although there is a case for the ST-trees defined in the Hackage repository, it does not

\[1\]urlhttps://hackage.haskell.org/package/structs-0/docs/src/Data-Struct-Internal-LinkCut.html\# LinkCut
cover empirical analysis or case study. In Chapter 7 of this report we make a step towards this direction discussing a potential experimental investigation which tries to fill this gap.

**Theoretical** Current persistent data structures have proven to be correct and efficient [HP06], and [Oka99b]. Unfortunately these are not targeted for dealing with dynamic contexts, as far as we aware. Since there is a recent instance for path-decomposition\(^2\), our second objective is to devise a purely data structure that covers the need specifically for the cases of linearisation and tree contraction.

The above projects are aimed to answer the following research questions, taking into account that functional data structures refer to the pure and the stateful (although safe) computations.

**RQ1** What are the bounds of the proposal data structures in the functional setting under empirical and theoretical analyses?

**RQ2** To what extent a functional data structure allows solutions for more than one dynamic tree approach?

**RQ3** What is the benchmarking among all potential implementations of functional data structures solving the dynamic tree problem?

**RQ4** Do functional data structures solve the dynamic tree problem stated in 3.1.1?

### 7.2 Future work

For the following potential contributions, we assume that three years of research have been concluded.

The contributions of the proposal thesis are summarised as follows:

- A description and formulation of the dynamic tree problem for the functional programming setting is presented. From the current theory provided by the imperative paradigm, two approaches have been prepared (linearisation and tree-contraction) in a proposed format and test instances have been made publicly available.

\(^2\)https://hackage.haskell.org/package/structs-0/docs/Data-Struct-Internal-LinkCut.html
• For the first time, an investigation on the suitability of applying functional data structures to solve the dynamic tree problem is presented. It is shown that these approaches can produce solutions of the same performance than those generated by current (imperative) setting, but in a much simpler and declarative denotation.

• Two functional data structures are presented, one persistent and one stateful-based, which produce the best known solutions for the test instances used in this thesis.

• For the first time, an investigation on the lazy evaluation of a data structure for the dynamic problem is provided. A form of relaxed computation is proposed and it is shown that using this form of evaluating solutions is beneficial in the optimisation of this problem.

• A new form of interface is proposed in which single-solution for update and query methods are extended to dynamic tree variants (approaches).
Bibliography


