

Malostranské nám. 25 • 118 00 Praha 1 • Czech Republic
tel.: (+420) 221 914 230 • fax: (+420) 257 531 014
e-mail: nana@kam.mff.cuni.cz • url: <http://iuuk.mff.cuni.cz>

August 26, 2013

Review of the doctoral thesis of Milan Straka “Functional Data Structures and Algorithms”

1 Thesis content

The thesis has two parts. The first part deals with design of persistent data structures that is data structures which allow access and modification of all historical versions of the structure, and with design of data structures that have good worst-case performance. The second part of the thesis contains analysis, implementation in the functional language Haskell and experimental comparison of several data structures.

The main contributions of the first part are in Chapters 2 and 5.

Chapter 2 explores the possibility of building persistent data structures using operations rebuild and undo. The rebuild operation rebuilds the data structure into a “minimal potential” state, the undo operation undoes previously performed operations on the data structure. The goal is to build a *persistent* data structure with good *amortized* performance from ordinary data structure with good amortized performance. This is not trivial as one can repeatedly invoke an operation with low amortized cost but high actual cost. This chapter shows that any general transformation of a data structure into a persistent one that relies solely on either the rebuild operation or on the undo operation has to incur large amortized cost. For the rebuild operation, the established cost increase is optimal as demonstrated in the thesis.

In Chapter 5, the author develops several variants of persistent array data structure that supports update and lookup operations with *worst-case* time bounds either $O(\log \log m)$ or $O((\log \log m)^2 / \log \log \log m)$. The only disadvantage of the faster one is that it uses non-linear space. This extends the previous results of Dietz (1989) who designed a persistent array with $O(\log \log m)$ time bounds for both operations but the time bound was only *amortized* for the update operation. It is known that $\Omega(\log \log m)$ time is the best possible time bound for operations on a persistent array. Persistent arrays could be thought of as the *ultimate* persistent data structures as any other data structure could be made persistent using them.

Beside the actual data structures the chapter also proposes new algorithms for identifying inaccessible data by garbage collection in the context of persistent arrays.

Chapter 5 builds on Chapters 3 and 4 which present solutions to the List ordering and the Dynamic dictionary problems. These are mainly known results but the thesis also presents several minor improvements and simplifications of known results and techniques.

The second part of the thesis contains theoretical and practical contributions.

The most important theoretical contribution is the analysis of BB- ω trees in Chapter 7. BB- ω trees are balanced search trees that have been analyzed previously most notably by Adams (1992, 1993). However, the analysis of Adams is flawed and it claims that BB- ω trees behave balanced for certain setting of parameters. There are various libraries in functional languages such as Haskell that rely on and implement BB- ω trees using the flawed parameters. This leads to their sub-optimal behavior which was independently observed by various researchers, the thesis' author being one of them. In Chapter 7, Straka provides a correct analysis of BB- ω trees and establishes which setting of parameters guarantees their balanced behavior. He also provides an implementation for them which became a part of the standard Haskell *Container* library where it is used to implement various data structures.

In Chapter 8, the author describes further improvements to the Haskell *Container* library, and provides experimental comparison of the performance of the library with other existing implementations. The new implementation leads to improved running time and smaller memory foot-print.

Chapter 6 provides implementation of persistent arrays in Haskell and experimental comparison with other solutions.

2 My opinion

In my view the most significant contribution of the thesis lies in the analysis and implementation of BB- ω trees in the later chapters. The analysis itself is a non-trivial multi-page analysis of various cases supported by computer exploration of trees up-to size 1 million. This analysis directly contributes to the new implementation of Haskell *Container* library on which at least one third of other Haskell libraries relies. Hence, the analysis provides a real world impact beyond what is typical for a theoretical work.

The results in Chapter 2 regarding the rebuild and undo operations constitute original results and an interesting research direction. I find them non-trivial and interesting. This part of the thesis was not published, yet.

The results in Chapters 3–5 taken together form a substantial contribution. The reason why I am less excited about them is that they don't bring many new ideas to the

table. They mostly rely on combination and modification of known techniques. Despite that combining them together is non-trivial so this part also has its merits.

To sum-up, the **author demonstrates the ability to carry out independent research**, and some of **his theoretical contributions have a significant practical impact**. The work meets the usual criteria for Ph.D. thesis.

3 Technical comments

The thesis is written using a fairly good English. There are only occasional small language issues, and overall the thesis reads well.

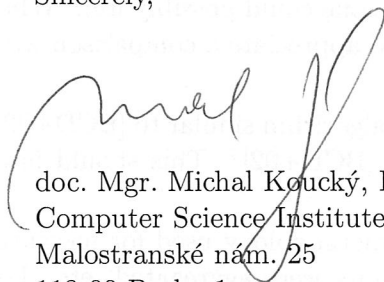
- My major complaint is about imprecision and sometimes missing details. For example the author frequently uses expressions like "Even if the update step has amortized complexity, the complexity of the whole operation [...] is worst-case." (page 25-26) and "[S]uppose that update has amortized complexity." (page 30). Like every body has a mass, every operation has its amortized complexity, and its worst-case complexity. Thus, the correct questions are: *what is* the worst-case complexity, *what is* the amortized complexity, and *what is the relationship* between them. This issue appears quite frequently throughout the thesis.
- Word "common" should be replaced by word "usual" in many instances.
- The example on page 30 for amortized complexity is not quite accurate.
- There are various things that are not completely explained. For example, I have no idea what is *fold* which is referred to in Chapters 7 and 8. I can only see that you can measure it (Table 7.6).
- I lack some deeper discussion why does the thesis focus only on rebuild and undo operations in Chapter 2. Are there some other operations one could possibly use? What about the combination of rebuild and undo? Also I would appreciate a comparison with what can be achieved using other than generic methods.
- In Chapter 3, on page 42, I wonder how much is *your* algorithm similar to [BCD+02]. Do you use techniques other than the one suggested in [BCD+02]? This should have been explained in more detail.
- In Chapters 6–8, I lack detailed explanation of the methodology used for measurements, like how many samples were taken, how the results were aggregated, etc. For example you consistently claim that you use "uniformly random samples". But you say at some point that in reality you use just *one* sequence generated by a pseudo-random generator with one *fixed specific* seed. This cannot be classified as measurements on uniformly random samples by any means. This can be exhibited in Table 7.4 which contains integral outcomes of measurements for random samples. I would expect averages

of several trials for randomly generated sequences that would lead to outcomes being reals.

Another issue is the presentation of the results and the scope of measurements. I would prefer much rather a plotted graph of the time versus instance size for different methods. That would allow to show much more data. For example Table 7.4 would be much more illustrative. I don't need to know the actual results to 8 digit precision but I want to understand the shape of the curve. This becomes significant in tables like 8.6 which exhibit results only for something like two or three input sizes. For these input sizes one gets very different relative results so one has to wonder when does one method overtakes the other, what is the trend, etc. Is there some unexpected behavior at some point?

- All the measurements use instance sizes at most 1 million. This happens to coincide with the bound of how far was the correctness of the parameters for $BB-\omega$ trees tested by computer. Is this a coincidence?
- Figure 7.3 would be easier to read if it were separated into four different figures, one for each value of δ .
- In Table 7.4, *Balance* calls either double rotation or single rotation, the former doing less pointer updates. What is the fraction of these calls. The number of *Balance* calls does not give the whole story, or does it?
- My understanding is that some of the measurements in Tables 8.6 and 8.12 are the same. Looking at the results for *RBSet* in *Set:union* the relative numbers are different in these two tables. What is the reason?

Sincerely,



doc. Mgr. Michal Koucký, Ph.D.
Computer Science Institute of Charles University
Malostranské nám. 25
118 00 Praha 1
Czech Republic