DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE
AARHUS UNIVERSITY

Dean Jan Kratochvíl
Charles University in Prague
Faculty of Mathematics and Physics

Evaluation:
Doctoral Thesis by Milan Straka (version of 12 August 2013)

Department of Computer Science

Gerth Stølting Brodal

Associate Professor

Date: 31August 2013

Mobil: +45 5059 5432
E-mail: gerth@cs.au.dk

Web: www.cs.au.dk/

Sender's CVR no.: 31119103

This is an evaluation of the Doctoral Thesis submitted by Milan Straka on 12 August 2013.

The thesis bridges two core computer science research areas: programming languages and algorithmics. The topic of the thesis is the development of efficient functional data structures. Data structures in functional languages is a classic topic, but the algorithmic angle was cultivated by Chris Okasaki in his PhD thesis from 1996 at CMU on "Purely Functional Data Structures" – a thesis that was extended into a text book that now has become a must read for any person interested in data structures in functional languages. This is an important research area, but with quite limited new research since the work of Okasaki.

The thesis of Milan Straka tries to revoke the area. The thesis has a few minor contributions in various directions, but is far from reaching the level of Okasaki's outstanding thesis. The various contributions are discussed below. I would judge the work in Chapter 8 of the thesis to be the most important, since the developed tuned Haskel library code improves the performance of all programs relying on these library packages.

Part I of the thesis (Sections 2-5) addresses how to make data structures partially and fully persistent. These chapters mainly recall existing solutions for making data structures persistent, but Straka also contributes with a few new alternative details to the existing algorithms.

Section 2 covers persistent data structures. Sections 2.1-2.3 recall existing techniques for making data structures partially and fully persistent, both with an amortized and a worst-case overhead. Many of these techniques require the data structure to be pointer based where each node has bounded in-degree. In Section 2.4 new techniques are considered to make data structures with amortized update bounds fully persistent. Simple lower and upper bounds are proved under the condition that the data structures are black boxes with a restricted "rebuild" / "update" interface. I have to admit that I found it hard to follow the writing in Section 2.4, and was missing some intuition, in particular with respect to the different versions in the version tree.

Chapter 3 considers the maintenance of the version tree required for implementing fully persistent data structures. This is a technique by Driscoll et al. The contribution of this section is that the core problem of list labeling can be solved in worst-case logarithmic time by using the rebalancing scheme of the weight-balanced trees of Arge and Vitter to control the relabeling of the list. This is a nice

Department of Computer Science
Aarhus University
Åbogade 34
DK-8200 Aarhus N
Denmark

Tel.: +45 8715 4112
Fax: +45 8715 4113
E-mail: cs@cs.au.dk
http://www.cs.au.dk

new idea. The level of details in the proofs for this (Theorem 3.5) could have been slightly higher, since this is the main contribution of the section, but it is otherwise on the level of detail of Section 3.

Chapter 4 recalls van Emde Boas trees and discusses variations of van Emde Boas trees. In particular Strata gives a simple upper-bound trade-off between query time and space by adopting trie ideas to the van Emde Boas trees. This is quite a straightforward construction.

Chapter 5 describes how to make arrays fully persistent. The chapter covers the ideas by Dietz from 1989 for making arrays persistent with amortized expected bounds. The contributions of this chapter are worst-case bounds that can be obtained by combining the ideas of Dietz with existing worst-case predecessor structures (described in the earlier chapters). Furthermore the chapter considers how to garbage collect unused versions in fully persistent arrays.

Part II of the thesis (Chapters 6-8) considers the implementation and benchmarking of data structures in Haskel.

Chapter 6 considers simple Haskel implementations of fully persistent (functional) arrays. This section discards the theory of all the previous sections and just gives a very simple implementation using balanced trees using different branching factors. As a reader this is quite disappointing to realize. The implementation is compared experimentally with other approaches for supporting arrays in Haskel. The choice of implementation is not particular motivated and the set of experiments is not very throughout. E.g., is the chapter missing a comparison (both theoretically and experimentally) with the random-access lists of Okasaki (Chapter 8 actually does provide an experimental comparison, but I was missing this when reading Chapter 6). Parts I and II of the thesis are not very coherent with respect to the topic of functional arrays.

Chapter 7 considers the implementation of a class of binary search trees, known as BB-ω trees, a generalization of the weight balanced BB[α]-trees of Nievergelt and Reingold from 1972. The main contribution is a correctness analysis fixing errors in previous proofs, and an implementation hereof, fixing problems in Haskel's containers package.

Chapter 8 is a performance study of various data structure implementations from the Haskel standard library. Data structures considered are Sets and Maps, Intsets and Intmaps, and Sequences. Using several tricks and tips, the performance of many of the data structures have been engineered to achieve improved performance. Some aspects involve improving the compiler by the Haskell developers, other from insights in the details of how the evaluation of Haskel programs is performed. The results achieved in this section are not very deep, but are highly relevant to people interested in Haskel as an efficient programming language.

Detailed comments for the author:

- Page 2: "Because **a** purely functional language does" ("a" missing).
- Page 2, footnote: Reference missing for deforestation.
- Page 3: "**an** useful" → "**a** useful".
- Section 1.1.2: I was missing a discussion of Okasaki's result on "Purely Functional Random-Access Lists" with logarithmic updates.
- Section 1.2: Could have cited Tarjan's original paper introducing the concept of amortized analysis.

- Page 16: Okasaki had some nice results on the combination of lazy evaluation with data structures with amortized performance, where lazy evaluation was used to handle expensive computations that could be setup in advance and prepaid before actually before evaluated. I think this could have been discussed.
- Section 2: The section often mentions the constraint "bounded in-degree". Is this satisfied in practice? In particular, can this be assumed for functional programs. What if the condition gets violated? Can the solutions automatically adapt to the higher in-degree then?
- Section 2.4, lines 1-2: Please elaborate on "The complexity bounds of the discussed methods of making persistent structures do not hold for structures with amortized bounds". State more explicitly what you mean by "do not hold".
- Page 30, line 14: "in a **fully** persistent structure, we can repeatedly..." ("fully" missing).
- Page 32, Definition 2.3: "that **the** structure" ("the" missing).
- Page 34, "the cost...is amortized...happened before this rebuild" – I think it would be appropriate with a comparison with the work of Okasaki at this point (the reference is given just below this sentence, but the comparison is left to the reader).
- Page 34: "bound**ed** in-degree" ("ed" missing)
- Page 36: A more precise definition of "undo" would be appropriate.
- Page 37, Definition 2.8: "with potential at least b" - is this the total potential for a sequence of operations or the individual versions?
- Page 39, last line: A reference would have been appropriate for the lower bound (there are references later in Section 3.2.2).
- Page 40: The need for the extended version list is not described clearly.
- Page 54, proof of Theorem 3.4: "appropriate direction" could have been stated more precisely. Details for "easy to check" could have been included. How do "child subranges" relate to "appropriate directions"? There are some missing details here.
- Page 45: In the worst-case relabeling, it is not completely clear to me what invariants hold for the various incremental relabelings. This could have been stated more precisely, e.g. by a set of invariants.
- Page 46: "per an insertion", remove "an".
- Page 50: "polynomial space" should be "a range of polynomial size".
- Page 53: Last two paragraphs: It could be nice with a discussion of the role of being independent of the word size w=log U, i.e. a discussion of trans-dichotomous algorithms.
- Page 55: Space is here denoted S, while on page 59 space is denoted C. Please use consistent notation.
- Page 55, Theorem 4.3: The update bounds are "**expected** amortized" due to the application of dynamic perfect hashing.

- Page 58, Theorem 4.6: On page 55 space is denoted *S* whereas here it is search time.
- Page 59, line 5: Please explain why construction time implies a bound on the space.
- Page 65: The results by Demaine *et al.* [SWAT'08] (missing reference) about fully persistent arrays should also be stated in Section 5.
- Page 66: "Such **an** implementation has" ("an" missing).
- Page 70, Section 5.3.1: "amortized" should be "expected amortized".
- Page 87: You discard the theory of all previous chapters based on some calculations. It would have been interesting to see actual experimental comparison with some of these ideas – not necessarily the full theoretical constructions.
- Page 88: Any particular reason why you consider multi-way trees for the representation of the arrays? A motivation for this decision would be appropriate.
- Page 90: You only consider sequential access patterns in your experiments. A more comprehensive set of experiments with different access patterns would have been appropriate here. E.g., are there any caching of nodes?
- Page 90: Motivation for the `Tree_A` implementations is the overhead for pattern matching. This immediately raises the question if the pattern matching order influences the running time. Did you try to reorganize your code? Please give a forward reference to Chapter 8.
  Page 95: Would have been appropriate to mention BB[α] trees up front.
- Page 96: You have a reference to Chapter 8 for the experimental comparison of different search trees. There has been previous experimental work on comparing search trees (at least for the imperative setting). It would have been nice with a few references, and also a discussion of if results are expected to translate to the context of Haskel.
- Page 101: The experimental evaluation of the validity of the different combinations of parameters is a nice idea. Gives you an idea of what to prove. Could be nice with a more general theoretical proof, confirming the validity of parameter choices for a larger range.
- Page 106: You refer to the height if a red-black tree of size *n*, but don't state precisely what the height bound is.
- Page107: The data in Table 7.4 could have been complimented with the height of the generated trees.
- Page 110: You discuss the extension of the data constructors, and claim that it is not advantageous to add a fourth data constructor. In the imperative setting it is known that (2,4)-trees to have good performance because they adapt better to cache-lines. So I am not completely convinced by your argument – but you could be right.
- Page 117: You state that benchmarking a lazy evaluated language is tricky. Would be appropriate to discuss the aspects making the problem tricky.

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE
AARHUS UNIVERSITY

- Page 119: What is the quality of your red-black tree implementation. How tuned is it? How much does it e.g. relay on the work of Okasaki? Okasaki has shown that top-down rebalancing during insertions can be handled efficiently using the pattern matching abilities of functional languages. Matt Might described how to handle deletions.
- Page 121: Why is RBSet missing in the plot at the top, and AVL2 in the two bottom plots?
- Page 126: You mention Okasaki's Random-Access-List implementation here for the first time. Would have been appropriate to mention the existence of this structure much earlier.
- Page 129: The choice of only focusing on improving BB-ω trees appears arbitrary. Did you try to optimize the other data structures?
- Page 130: You discuss optimizing the repeated pattern matching. That the appropriate usage of pattern matching can speed up data structures was e.g. shown by Okasaki for red-black trees.
- Page 148: Is "performant" an English word?

Sincerely,

Gerth Stølting Brodal
Aarhus University