Automated Sublinear Amortized Analysis for Lazy Evaluation

Armin Walch

Department of Computer Science, University of Innsbruck, Austria
armin.walch@student.uibk.ac.at

April 30, 2025

Abstract

This dissertation proposes an extension of automated amortized resource analysis techniques to support persistent, lazily evaluated data structures in functional programming languages like Haskell. Building on the ATLAS prototype, which automates non-linear amortized analysis via potential-based type inference, the project aims to adapt these methods to a call-by-need evaluation model. The core objective is to develop a sound and automated analysis for data structures such as lazy skew heaps.

1 Introduction

Amortization is a powerful analysis technique for the worst case costs of a data structure. In contrast to a simple analysis, that recognizes only the worst case cost of single operation, this method considers the average cost over a sequence of operations. As such, it allows to offset the cost of more expensive operations in the sequence with a series of cheaper operations. It was devised by Sleator and Tarjan [20, 22] to aid the design of *self-adjusting* data structures. Such data structures often allow for a much simpler implementation, while achieving the same worst case cost in the amortized sense, as their balanced counterparts.

Although a number of approaches have applied amortization techniques to enable automated resource analysis of programs [6, 7, 8, 1, 5], the prototypical example of *splay trees* had remained beyond the reach of automation—until the recent introduction of the prototype ATLAS [12]. Their approach provides a static analysis based on type inference combined with the potential method, which is the

same amortization technique that was originally employed by Sleator and Tarjan in the design of splay trees [20].

A well-known limitation of all amortization techniques, first pointed out by Okasaki [18], is that data structures need to be used ephemeral. This means that at any point in time there can only exist single version of a data structure. In imperative languages that allow destructive modifications this requirement is not an issue, but in functional languages we encounter persistent data structures. Such a data structure cannot be modified in-place but rather a new copy is created and in principle the previous versions can still be accessed. To demonstrate why this behavior breaks traditional amortized analysis, take as an example the *bankers method*. With this analysis technique, one assigns *credits* to data structure elements that can be used later in a sequence to pay expensive operations. Assume now we have a data structure S, that holds our saved up credits. We can then apply an expensive operation, payed by the credits in S, to obtain a new data structure S'. With persistence nothing stops us from reusing S and applying the expensive operation again, even though we already spend its credits.

Okasaki demonstrated that in call-by-need languages such as Haskell [15], amortized analysis can be adapted by replacing the traditional concept of credits with that of debits. Call-by-need semantics combine two key features: lazy evaluation, in which values are computed only when they are required, and memoization, which ensures that once a value is computed, it is stored and reused for subsequent references.

The main goal of this dissertation is to adapt the analysis techniques for non-linear resource bounds of self-adjusting data structures, presented by Leutgeb et al. [12] to the context of persistent data structures with lazy evaluation. This should allow to establish that the existing results for *splay trees*, *splay heaps* and *pairing heaps* also hold when they are used persistently, given a call-by-need evaluation strategy. In addition to the existing examples from this work, we consider lazy skew heaps [3] as the primary benchmark for our analysis.

Even though the analysis methods developed for ATLAS, specifically the logarithmic potential function template, are powerful enough to derive bounds on the operations of *splay trees*, *splay heaps* and *pairing heaps*, they do not seem to lend themselves to the analysis of *skew heaps*. For this reason, we anticipate that designing a template potential function for skew heaps will be a necessary prerequisite for achieving our main milestone.

Another challenge is that lazy evaluation is usually seen in the context of higher-order functions, whereas the analysis of [12] intentionally relies on a first order semantic, as it proofed sufficient for the analysis of data structure operations.

Finally, in addition to the main objective, the analysis of persistent data structures, we propose a generalization of the technique to cover other common ex-

amples from literature e.g. [14], such as lazy merge sort and possibly portions of Haskell's standard libraries, such a `containers`.

2 Related Work

There is a small but relevant body of literature [2, 4, 9, 14, 16] that provides various formalizations of Okasaki's debit method, and as such, is closely related to the proposed research. Most of these works offer strong formal foundations based on the propositions-as-types paradigm. For instance, Danielsson [2] employs dependent types in Agda [17, 23] to model a thunk monad that explicitly tracks debits. Similarly, Handley et al.[4] use refinement types via the Haskell extension Liquid Haskell [24].

In Liquid Haskell, the default resource model assumes eager evaluation—i.e., function arguments are fully evaluated—which does not account for Haskell's memoization semantics. This can lead to over-approximation of resource consumption and results in worst-case analyses. To incorporate laziness, the user must explicitly annotate parts of the program where call-by-need evaluation should apply, as in Danielsson's approach [2].

Mevel et al.[16] extend the Iris separation logic framework [10] with time credits and time receipts to reason about resource usage. However, all three approaches—[2, 24, 16]—require the user to manually provide resource bounds as proof obligations. In contrast, the goal of our work is to support automatic type inference of such bounds, aligning with the broader vision of fully automated resource analysis.

Jost et al. [9] present an approach much closer related to the proof techniques applied in ATLAS. In particular they present a type system, in which potential is encoded as an effect and for which type inference yields a linear constraint system. Their technique for encoding potential of data structures by annotating each constructor with a constant, resembles the bankers method, as it allows to place potential on each data structure node instead of only storing the overall potential. The main limitations of this system in regards to our research goals are on the one hand the restriction to linear potential functions and on the other hand the lack of sufficiently complex potential function templates to reason about self-adjusting data structures like splay trees.

In [14] thunks are modeled as lambdas with a parameter of unit type, where memoization is supported by a cache for function applications. Since the input language of this analysis is Scala, which has a strict evaluation model by default, the user has to explicitly mark data structure fields as lazy, as well as annotate functions to be memoized.

TODO not sure about the limitations here.

3 Methodology

In the following we describe the main project objectives, where each of them builds upon the foundation established by the preceding ones, to achieve the overarching milestone of analyzing lazy skew heaps.

3.1 WP1: Analysis of (Strict) Skew Heaps

As a first step toward analyzing lazy skew heaps, we plan to provide the currently missing analysis of strict skew heaps within the framework of ATLAS. This involves the design of a new template potential function that captures the unique characteristics of skew heaps, in a way that aligns with existing pen-and-paper proofs. The generic analysis framework developed in the author's thesis [25], should make it possible to incorporate such a function seamlessly as an additional instance. To achieve this, we anticipate to extract the essential properties of the potential function as observed in prior formalizations and encode them as expert knowledge—similar to the approach of [13] for their sum-of-logs potential function.

A crucial difference between the sum-of-logs potential and the weight-based potential used in the manual analysis of skew heaps [21] is that the latter is defined in a piecewise manner. In the original formulation, the weight $wt(x)$ of a binary tree node xx is the number of its descendants, including x itself. A node is defined as heavy if it has more descendants than its sibling, that is, $wt(x) > wt(p(x))/2$, where $p(x)$ denotes the parent of x; otherwise, it is light. The potential function for a skew heap is defined as the number of right-heavy nodes:

$$
\begin{aligned}
&rh(\,\text{leaf}\,) = 0 \\
&rh((l,a,r)) = \begin{cases} rh(l) + 1 + rh(r) & \text{if } wt(r) > wt((l,a,r)) \\ rh(l) + rh(r) & \text{otherwise} \end{cases}
\end{aligned}
$$

The challenge here is that determining whether the condition $wt(r) > wt((l,a,r))$ holds requires global reasoning. As such, it cannot, in principle, be resolved through a local typing judgment on data constructors. We propose to address this issue by introducing a dedicated sum type for binary trees that encodes the weight-based property directly in the type. This would allow the case distinction to be handled within the existing type rule for pattern matching. Additionally, we believe that the enriched type information could be leveraged in the subtyping rule to better integrate expert knowledge during the analysis.

3.2 WP2: Automated Sublinear Amortized Analysis for Lazy Evaluation

As the next step, we plan to adapt one of the approaches proposed in [2, 9], integrating the concept of template potential to support non-linear potential functions. Our goal is to obtain a formalization of Okasaki's debit method that enables fully automated inference of resource bounds for persistent, self-adjusting data structures. This new analysis will be formally proven sound with respect to a natural semantics that closely mirrors Haskell's call-by-need evaluation strategy. Existing works [2, 9, 14] suggest that Launchbury's semantics [11] present present a solid foundation, even though adaptation to our use case might be necessary. By restricting ourselves to a semantics faithful to call-by-need, we aim to ensure the practical relevance and applicability of our results in real-world scenarios.

To ground our work in first principles, we critically examine several assumptions made in the existing literature:

[9] develops a higher-order semantics to accommodate their examples. In contrast, ATLAS focuses on a first-order semantics, which is sufficient for expressing the benchmarks under consideration. It remains an open question whether supporting thunk types necessarily entails support for higher-order functions.

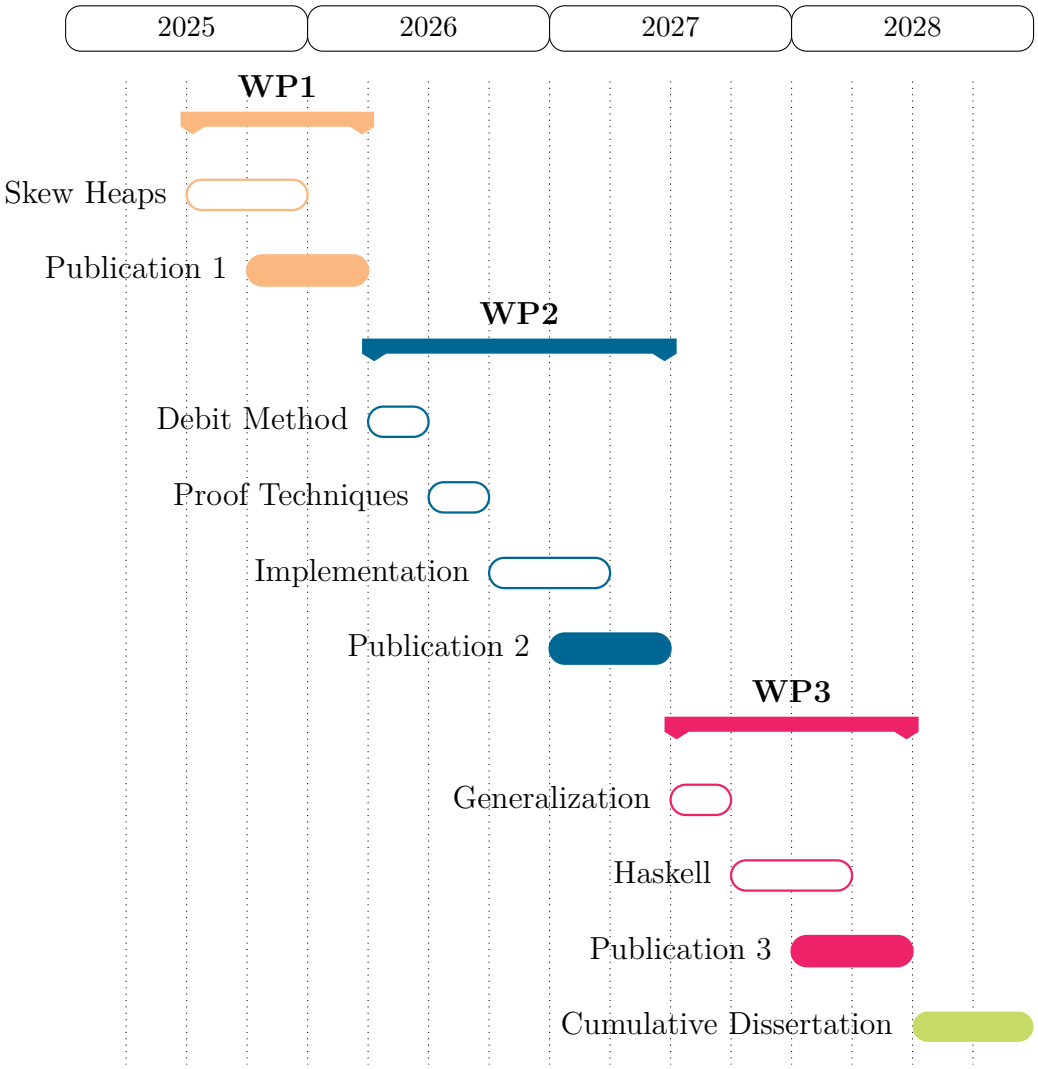
Okasaki observed that, in the transition from credit-based to debit-based reasoning, the potential method appears weaker than the banker's method—a discrepancy not seen in traditional settings. As part of our formalization, we intend to investigate whether this limitation also manifests in the type-based potential method employed in ATLAS.

3.3 WP3: Generalization to Haskell

Once appropriate proof techniques for a call-by-need semantics with potential templates have been established, we aim to generalize this approach to analyze the cost behavior of additional, well-known examples of lazy evaluation. This generalized method could be applied, for example, to Okasaki's lazy queues [18] or to algorithms such as lazy merge sort. Another promising direction for this analysis is the class of real-time data structures that use scheduling techniques to eliminate amortized costs [18], such as real-time queues.

As an optional objective, we also consider implementing the analysis as a GHC compiler plugin, in the style of [19], to enable cost analysis of real-world Haskell codebases. A potential application could be the analysis of commonly used libraries like Haskell's `containers` library.

4 Project Schedule



References

[1] Robert Atkey. "Amortised resource analysis with separation logic". In: *European Symposium on Programming*. Springer. 2010, pp. 85–103.

[2] Nils Anders Danielsson. "Lightweight semiformal time complexity analysis for purely functional data structures". In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 133–144.

[3] Jeremy Gibbons. *The fun of programming*. 2003.

[4] Martin AT Handley, Niki Vazou, and Graham Hutton. "Liquidate your assets: reasoning about resource usage in liquid haskell". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–27.

[5] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Multivariate amortized resource analysis". In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2011, pp. 357–370.

[6] Martin Hofmann and Steffen Jost. "Static prediction of heap space usage for first-order functional programs". In: *ACM SIGPLAN Notices* 38.1 (2003), pp. 185–197.

[7] Martin Hofmann and Steffen Jost. "Type-based amortised heap-space analysis". In: *Programming Languages and Systems: 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings 15*. Springer. 2006, pp. 22–37.

[8] Steffen Jost et al. "Static determination of quantitative resource usage for higher-order programs". In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2010, pp. 223–236.

[9] Steffen Jost et al. "Type-based cost analysis for lazy functional languages". In: *Journal of Automated Reasoning* 59 (2017), pp. 87–120.

[10] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20.

[11] John Launchbury. "A natural semantics for lazy evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1993, pp. 144–154.

[12] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. "ATLAS: automated amortised complexity analysis of self-adjusting data structures". In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer. 2021, pp. 99–122.

[13] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. "Automated expected amortised cost analysis of probabilistic data structures". In: *International Conference on Computer Aided Verification*. Springer. 2022, pp. 70–91.

[14] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. "Contract-based resource verification for higher-order functions with memoization". In: *Acm Sigplan Notices* 52.1 (2017), pp. 330–343.

[15] Simon Marlow et al. “Haskell 2010 language report”. In: (2010).

[16] Glen Mével, Jacques-Henri Jourdan, and François Pottier. “Time credits and time receipts in Iris”. In: *Programming Languages and Systems: 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 28*. Springer. 2019, pp. 3–29.

[17] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Chalmers University of Technology, 2007.

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

[19] Franz Siglmüller. “Type-based resource analysis on haskell”. In: *arXiv preprint arXiv:1908.06478* (2019).

[20] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting binary search trees”. In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

[21] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting heaps”. In: *SIAM Journal on Computing* 15.1 (1986), pp. 52–69.

[22] Robert Endre Tarjan. “Amortized computational complexity”. In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (1985), pp. 306–318.

[23] The Agda Team. *The Agda Wiki*. 2007. URL: `http://wiki.portal.chalmers.se/agda` (visited on 04/28/2025).

[24] Niki Vazou. *Liquid Haskell: Haskell as a theorem prover*. University of California, San Diego, 2016.

[25] Armin Walch. *Automated amortized multi-potential analysis*. `https://bibsearch.uibk.ac.at/AC17496175`. 2025.