Towards Mutability as a Compiler Optimization for Pure Functional Languages

William Brandon
University of California, Berkeley, United States
williambrandon@berkeley.edu

Wilson Berkow
University of California, Berkeley, United States
wberkow@berkeley.edu

Frank Dai
University of California, Berkeley, United States
fydai@berkeley.edu

Benjamin Driscoll
University of California, Berkeley, United States
bdriscoll00@berkeley.edu

Abstract

Pure functional programming seeks to prohibit mutation of shared data structures, but performance requirements often demand the use of in-place mutation. Much prior work has explored how pure functional languages can *safely* expose mutation primitives to the programmer. However, enforcing correct use of these APIs typically either requires that all operations be performed inside a monadic context [4], or else requires the use of major type system extensions like linear types [2]. In contrast, we consider an approach to achieving the performance benefits of mutability without invasive source language extensions – such as linearity, ownership, or state-thread monads – by implementing mutability as a semantics-preserving compiler optimization. In this short paper, we propose a set of design principles for pure functional languages and compilers aiming to achieve safe in-place mutation via compiler optimizations, identifying multiple limitations of existing optimization techniques and suggesting concrete solutions.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Compilers, optimization, static analysis, alias analysis, functional programming, mutability, persistent data structures, defunctionalization

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements We wish to thank Sanjit Seshia, Jonathan Ragan-Kelley, Daniel Fremont, Alex Reinking, Ankush Desai, and Ethan Smith for their valuable feedback and guidance on this project.

1 Introduction

To motivate the problem of mutation optimization, consider the following functional pseudocode to construct a frequency histogram from a list of integer samples in the range $[0, \texttt{n\_bins})$:

```
build_histogram(n_bins: Int, samples: Array Int): Array Int =
  foldl(samples,
        repeat_zeros(n_bins), // Initial value for fold
        \(hist, sample) -> put(hist, sample, get(hist, sample) + 1))
```

Here, `put` is a function which must *behave* as if it produces a copy of an array with one element updated to a new value. A naïve implementation of `put` would do exactly this, creating a deep copy of its input array each time it is called, causing `build_histogram` to run in time $\mathcal{O}(\texttt{n\_bins} \cdot \texttt{len(samples)})$. This runtime complexity could be improved by implementing arrays as persistent data structures admitting logarithmic-time non-destructive updates (such as [8]), rather than as flat buffers. However, we can do even better: we never access any intermediate arrays in the fold after they are passed to `put`, so each call to `put` could be safely implemented as a destructive update. In this paper, we are interested in compiler optimizations capable of statically detecting situations like this one and exploiting them to improve performance.

Prior work has presented techniques which can successfully detect opportunities for safe mutation in many functional programs, but to the best of our knowledge all prior implementations suffer from two important limitations which stand in the way of their practical use. First of all, we are aware of no implementation of static mutation optimization which can usefully analyze *higher-order functions*. Second, we observe that prior implementations of mutation optimization suffer from *performance cliffs*, where the exact outcome of the optimization process can have large effects on a program's asymptotic runtime, making it difficult to reason about worst-case performance. To address these limitations, we propose the following novel techniques, of which we have developed a working prototype implementation:

- To analyze higher-order functions, we propose the use of a special form of *defunctionalization*, which we call **polyvariant defunctionalization**, to convert higher-order programs into a first-order representation before applying mutation optimization.
- To avoid performance cliffs, we propose using mutation optimization to make decisions about the low-level representations of entire **data structures**, rather than individual mutation points, to allow persistent data structures to be used where expensive deep copies would otherwise be necessary.

2 Prior Work

To the best of our knowledge, the state of the art in static mutation optimization can be found in Mazur's "Compile-Time Garbage Collection" [6], and Giuca's extension of that work in the language Mars [3]. Both of these approaches identify opportunities for safe mutation using interprocedural graph-based alias analyses, which statically over-approximate pairwise aliasing relationships between heap structures, and use that information to determine, at each potential mutation point, whether the structure being updated could ever be used again. These alias analyses perform well on a large class of first-order programs, and we believe they should serve as the foundation for future work on mutation optimization.

Regarding the analysis of higher-order functions, Giuca gives a detailed account of an algorithm to directly compute aliasing graphs for higher-order programs. However, Mars does not actually implement this algorithm, which we take as evidence that it may be prohibitively complex to implement in practice. By contrast, we elaborate on one of Mazur's suggestions for future work in proposing the use of polyvariant defunctionalization, which we claim offers a simple and practical approach to handling higher-order control flow.

3 Proposed Techniques

Polyvariant Defunctionalization

To perform mutation optimization on programs containing higher-order control flow, we propose a transformation we call *polyvariant defunctionalization* to convert higher-order programs into a first-order intermediate representation before performing alias analysis. Traditional defunctionalization [7] passes work by globally accumulating all functions in a program (or all functions with a given type signature) into a single enumeration type representing their union, and replacing dynamically-dispatched function calls with pattern matches on this enumeration type. Unfortunately, this strategy is too coarse to be useful for alias analysis, as it retains all the imprecision of ordinary dynamic dispatch at each call site. Our polyvariant defunctionalization, by contrast, uses a unification-based algorithm similar to [1] to approximate the flow of dynamic function values through a program, and then uses this information to produce copies of higher-order functions *specialized* to the actual set of functions they could operate on at each call site. For example, applying polyvariant defunctionalization to the `build_histogram` function described above would generate a first-order version of `foldl` specialized to use the particular lambda appearing in the example. We could then apply first-order alias analysis techniques to this specialized implementation.

While polyvariant defunctionalization does result in an increase in the number of functions generated by the compiler, the Rust programming language [5] applies similar techniques successfully on a large scale; in Rust, higher-order functions are typically *monomorphized* with respect to the functions they operate on, to avoid the need for dynamic dispatch. This practice suggests that polyvariant defunctionalization may in fact carry performance benefits of its own, distinct from its use in enabling alias analysis.

Avoiding Performance Cliffs by Choosing Appropriate Data Structures

A natural question which arises in mutation optimization is what to do when the compiler *cannot* prove that a given update operation can safely be made mutable. Prior work employs the simple fall-back strategy of performing a deep copy of the entire structure before updating it, but as deep copies can be expensive, this makes the performance of the resulting program extremely sensitive to the details of the compiler's mutation optimizer. Importantly, this can make it difficult to reason about the *asymptotic* runtime of a program; a missed optimization opportunity can cause an otherwise linear-time algorithm to balloon to quadratic time.

In contrast, we consider a form of mutation optimization which guarantees tight bounds on asymptotic runtime even when mutation optimization fails. We propose that whenever an update cannot be safely performed in-place, the compiler should adjust the type of the object being updated to some *persistent* data structure supporting efficient non-destructive updates. Through unification, we can determine for each collection-valued variable in the program whether it could ever be used in a context requiring non-destructive update. If so, the compiler must implement that collection via a persistent data structure such as an RRB tree [8]; otherwise, it can use a higher-performance mutable implementation. This guarantees that the performance of operations on these collections is always at least as good as would be achieved by using persistent data structures with efficient non-destructive updates.

4 Implementation Efforts and Future Work

We are developing an implementation of the techniques described in this paper in a compiler for a new functional language, *Morphic*. Our compiler already includes working implementations of polyvariant defunctionalization, alias analysis, and data-structure-level mutation optimization, demonstrating that these techniques are feasible to implement. Preliminary results suggest that our techniques successfully detect opportunities for safe mutations in nontrivial functional programs, but we have yet to conduct a systematic performance study.

We believe important areas for future research in mutability optimization include the design and implementation of diagnostic tools to report optimization decisions to the programmer, the design of foreign function interfaces compatible with interprocedural alias analysis, and an implementation of these techniques in the presence of incremental compilation.

References

1 Anindya Banerjee. A modular, polyvariant and type-based closure analysis. *ACM SIGPLAN Notices*, 32(8):1–10, 1997.

2 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

3 Matthew Giuca. *Mars: an imperative/declarative higher-order programming language with automatic destructive update*. PhD thesis, 2014.

4 John Launchbury and Simon L Peyton Jones. Lazy functional state threads. *ACM SIGPLAN Notices*, 29(6):24–35, 1994.

5 Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

6 Nancy Mazur. Compile-time garbage collection for the declarative language mercury. 2004.

7 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Software Practice and Experience*, 20(S2), 1972.

8 Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. Rrb vector: a practical general purpose immutable sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 342–354, 2015.