Autobahn 2.0: Minimizing Bangs while Maintaining Performance (System Demonstration)

Marilyn Sun
Tufts University
USA
marilyn.sun@tufts.edu

Kathleen Fisher
Tufts University
USA
kfisher@cs.tufts.edu

Abstract

Lazy evaluation has many advantages, but it can cause bad performance. Consequently, Haskell allows users to force eager evaluation at certain program points by inserting strictness annotations, known and written as bangs (!). Unfortunately, manual bang placement is difficult. Autobahn 1.0 uses a genetic algorithm to infer bang annotations that improve performance. However, Autobahn 1.0 often generates large numbers of superfluous bangs, which is problematic because users must inspect each such bang to determine whether it is safe. We introduce Autobahn 2.0, which uses GHC profiling information to reduce the number of superfluous bangs. When evaluated on the NoFib benchmark suite, Autobahn 2.0 reduced the number of inferred bangs by 90.2% on average, while only degrading program performance by 15.7% compared with the performance produced by Autobahn 1.0. In a case study on a garbage collection simulator, Autobahn 2.0 eliminated 81.8% of the recommended bangs, with the same 15.7% optimization degradation.

CCS Concepts • **Software and its engineering** → *Preprocessors*; Control structures;

Keywords Program Optimization, Lazy Evaluation, Bang Patterns, Genetic Algorithms

ACM Reference Format:
Marilyn Sun and Kathleen Fisher. 2018. Autobahn 2.0: Minimizing Bangs while Maintaining Performance (System Demonstration). In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27–28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3242744.3264734

Haskell '18, September 27–28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3264734

1 Too Many Bangs

Autobahn 1.0 [2] is a tool that helps Haskell programmers reduce their thunk usage by inferring strictness annotations. Users provide Autobahn 1.0 with a program to optimize and representative input; it returns an optimized version after a couple of hours. It uses a genetic algorithm to randomly search for beneficial bang locations. It can both add and remove annotations. The genetic algorithm iteratively measures the runtime performance and memory consumption of a series of candidate bang placements. It preserves candidates that improve the original program's performance, discarding those that trigger worse performance (including non-termination). Because the system measures performance dynamically, the resulting annotations optimize the program *for the supplied input.* When run on different input, the annotations could change the termination behavior of the program, which may or may not be a problem. Users then face the time-consuming task of inspecting suggested bang placements to ensure the bangs maintain the desired semantics on all *relevant* inputs.

2 Autobahn 2.0

Autobahn 2.0 strives to reduce the number of generated bangs. To that end, it adds *pre-search* and *post-search* phases that run before and after Autobahn 1.0, respectively. Both phases use information from GHC's profiler to discover ineffective bangs. The profiling system reports the time, memory allocation, or heap usage attributable to each tracked location. Any program source location where a bang may be added is represented as a *gene* that can be turned *on* or *off*. A *chromosome* comprises all of the genes within a file. We represent a chromosome as a fixed-length bit vector, and a program as a collection of chromosomes. Intuitively, a bang that appears in a location that uses many resources, a *hot spot*, is more likely to be useful, while one in a location using few resources, a *cold spot*, is likely to be useless. Leveraging that intuition, Autobahn 2.0 preserves bangs that appear in hot spots while eliminating those in cold spots.

2.1 The Pre-search Phase

Autobahn 1.0 uses program source files as the unit of granularity for the set of program locations to consider. By default,

it optimizes all source code files in the program's directory. The pre-search phase uses profiling information to adjust the set of files under consideration. It begins by generating a GHC time and allocation profile for the unoptimized program by running it on user-provided representative input. Autobahn 2.0 then sets the optimization coverage for Autobahn 1.0 to be source files that contain at least one hot spot. In addition, Autobahn 2.0 identifies library files that contain hot spots and suggests to users that they add local copies so the library source files may be optimized as well. Autobahn 1.0 then optimizes the program as usual with the more targeted set of files. Note that this phase can both expand the search space, by identifying library files that contain hotspots, as well as reduce it, by identifying program source files with no hot spots.

Pre-search profiling offers three important benefits. First, it can greatly reduce the number of bangs Autobahn 1.0 suggests by limiting the search space to those program files that have a chance of significantly impacting performance. This focusing reduces the possibility of generating useless or dangerous bangs by eliminating them from consideration and increases the chances of generating useful ones by allowing more of the relevant search space to be explored within a given time budget. Second, if a hot spot is located in an external library file, the pre-search phase can identify the relevant files so they can be included in the optimization process. Third, it identifies programs that are potentially unsuitable for Autobahn 1.0 optimization. If a program contains a large number of cost centers that all contribute minimally to program runtime, there may not be any way to substantially improve program performance by adding bang annotations. If the pre-search phase concludes that a program only contains cold spots, it will alert the user and abort, saving the time and effort of running the remaining phases, which are unlikely to identify significant performance improvements.

2.2 The Post-search Phase

After Autobahn 1.0 suggests a set of chromosomes, Autobahn 2.0 uses GHC profiling information to eliminate bangs that do not significantly contribute to program performance. Specifically, the post-search phase eliminates any gene that did not either contain a bang in the recommendation or fall within a hot spot. The remaining genes require further testing because they may still be useless, failing to improve program performance despite falling within a hot spot. The number of such genes is sufficiently small that the post-search phase can test each such gene in turn, turning it off while keeping the remaining bangs on. It then runs the resulting program and compares its performance to that of the program recommended by Autobahn 1.0. If the absence of the bang slows down the program by the value of the *absenceImpact* threshold parameter, the post-search phase deems the bang useful and decides to keep it. Otherwise, the bang is deemed useless and is discarded. The *absenceImpact* threshold is adjustable; we set it to 6%. This parameter allows users to manage the tradeoff between aggressively reducing the number of bangs and preserving performance improvements. In our experiments, we chose to aggressively reduce the number bangs in exchance for some performance degradation. The post-search phase repeats this process for every bang that is both in the recommended chromosomes and in a hot spot. The minimization result is the combination of all the surviving bangs.

3 Evaluation

We report the average of the runtime performance ratios and the average of the total number of suggested bangs over 10 runs across all 60 benchmark programs.

- Autobahn 2.0 applied to the NoFib benchmark suite reduced the number of generated bangs by 90.2% on average, while increasing the runtime of the optimized program by 15.7% over the program optimized by Autobahn 1.0 alone. We refer to this performance change as a 15.7% *optimization degradation.*
- The pre-search phase removed at least one file from consideration in 21 NoFib benchmark programs, 35% of the programs we considered. For these programs, the pre-search phase eliminated 45 potential bang locations per 100 LOC, resulting in a mean bang reduction of 87.79% across the entire benchmark suite.
- Running the post-search phase alone on the NoFib benchmarks can reduce the number of inferred bangs by 93.8% with a 33% optimization degradation.
- When run on `gcSimulator` [1], Autobahn 2.0 reduced the number of inferred bangs by 81.8% with a 15.7% optimization degradation.

4 Demo

In our demo, we describe the intuitions behind our approach. Specifically, we explain the significance of incorporating GHC profiling information with an otherwise random genetic search, and why Autobahn 2.0 is able to automatically reduce the number of bangs inferred by Autobahn 1.0 while maintaining roughly the same level of optimization. We describe how bangs are sorted into categories, filtered, and eliminated in each of the pre-search and post-search phases, and provide an overview of Autobahn 2.0's optimization pipeline. Additionally, we discuss implementation specifics such as the program architecture and the representation of hot spots, cold spots, bangs, and source locations in our optimizer. We also take a closer look at the results obtained while evaluating the effectiveness of Autobahn 2.0. Finally, we demonstrate a typical use case of Autobahn 2.0 by running it on an example program and walking through the minimization process and outcome.

References

[1] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise GC Traces. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 109–118. https://doi.org/10.1145/2464157.2466484

[2] Yisu Remy Wang, Diogenes Nunez, and Kathleen Fisher. 2016. Autobahn: Using Genetic Algorithms to Infer Strictness Annotations. In *Proceedings of the 9th International Symposium on Haskell (Haskell '16)*. ACM, New York, NY, USA, 114–126. https://doi.org/10.1145/2976002.2976009