Autobahn 2.0: Minimizing Bangs while Maintaining Performance

Marilyn Sun
Tufts University
marilyn.sun@tufts.edu

Kathleen Fisher
Tufts University
kfisher@cs.tufts.edu

Abstract

While lazy evaluation has many advantages, it can result in serious performance costs. To alleviate this problem, 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 labor intensive and difficult to reason about. The Autobahn 1.0 optimizer uses a genetic algorithm to automatically infer bang annotations that improve runtime 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 introduces non-termination or other semantic differences. We introduce Autobahn 2.0, which uses GHC profiling information to reduce the number of superfluous bangs. Specifically, Autobahn 2.0 adds a *pre-search phase* before Autobahn 1.0's genetic algorithm to focus the search space and a *post-search phase* to individually test and remove bangs that have minimal impact. 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 when compared with Autobahn 1.0.

1 Autobahn 1.0 Produces Too Many Bangs

Autobahn 1.0 [**?**] is a tool that helps Haskell programmers reduce their thunk usage by automatically inferring strictness annotations. Users provide Autobahn 1.0 with an unoptimized program, representative input, and an optional configuration file to obtain an optimized version of the program over the course of a couple of hours. If a program already contains strictness annotations, Autobahn 1.0 may remove existing annotations that do not improve program performance. Autobahn 1.0 uses a genetic algorithm to randomly search for beneficial locations to place bangs in the program. The genetic algorithm iteratively measures the runtime performances or memory consumptions of a series of candidate bang placements, depending on the performance metric specified by the user. Candidates that improve upon the original program's performance are preserved, and candidates that trigger non-termination or worsen program performance are eliminated. Autobahn 1.0 eventually returns a list of well-performing candidates, ranked by how much each candidate improves program performance. Because Autobahn 1.0 measures performance when the program is run on the supplied input, the resulting annotations optimize the program *for that specific input*, which is why it is important for users to provide representative 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 candidate bang placements before applying them to ensure the bangs maintain the desired semantics on all *relevant* inputs.

2 Autobahn 2.0

We present Autobahn 2.0, an improved version of Autobahn 1.0 that aims to reduce the number of generated bangs. Autobahn 2.0 introduces a *pre-search phase* and *post-search phase* that run before and after Autobahn 1.0, respectively. Both phases use information from GHC's profiler to locate and eliminate ineffective bangs. GHC profiles allow users to better understand which locations in a program consume the most resources. The profiling system adds annotations to the program and generates a report detailing the amount of time, memory allocations, or heap usage that is attributable to each location. Any program source location where a bang may be added is represented as a *gene* that can be turned *on*, corresponding to the presence of a bang, or *off*, corresponding to its absence. A *chromosome* comprises all of the genes within a file. We represent a chromosome as a fixed-length bit vector, in which each bit indicates the presence or absence of a bang at the corresponding location. Since a program is a collection of source files, it is represented as a collection of bit vectors, or chromosomes. Intuitively, a bang that appears in a location that uses many resources, which we call a *hot spot*, is more likely to be useful, while one in a location using few resources, which we call 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 when inferring bangs. By default, Autobahn 1.0 optimizes all source code files in the program's directory. The pre-search phase uses profiling information to adjust the set of files that Autobahn 1.0 considers during its optimization. Specifically, this phase instructs Autobahn 1.0 to optimize files that contain locations that require significant resources, skipping files that do not and potentially adding files not originally included by the users.

Just as manually reasoning about bang placement can be difficult, so too reasoning about which files to optimize can be challenging. The pre-search phase 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, except using a more targeted set of files/chromosomes. 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 runs, the post-search phase individually tests each candidate bang that falls within a costly location. Bangs that do not significantly impact program performance are eliminated. The system is parameterized, so users can 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, accepting some performance degradation over the level of optimization provided by Autobahn 1.0.

After Autobahn 1.0 explores the search space and suggests a winning set of chromosomes, Autobahn 2.0 uses GHC profiling information to eliminate any bangs on those chromosomes that don't significantly contribute to program performance. To that end, the post-search phase decides that any gene that is off in the winning chromosomes should not have a bang in the final recommendation. It removes these genes from further consideration. The post-search phase then maps each remaining gene, which must be turned on, to its corresponding cost center. It turns off all genes that do not fall within hot spots, deciding not to recommend bangs for the corresponding locations on the theory that such bangs are likely to be useless. It then removes these genes from further consideration.

The remaining genes are the interesting ones in that they both contain a bang and fall within a hot spot. These genes require further testing because they may still be useless, failing to improve program performance even though they fall within a hot spot. There is usually a sufficiently small number of remaining genes that are both turned on and within a hot spot that it is possible to measure their impact one by one. Specifically, the post-search phase tests each such gene in turn, turning it off while keeping all the remaining bangs on. It then runs the resulting program and compares its performance to that of the winning chromosomes as determined 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%. The post-search phase repeats this process for every bang that is both in the winning chromosomes and in a hot spot. The minimization result is the combination of all the surviving bangs.

It is possible that there are no surviving bangs at the end of testing, and that occurs when there are no hot spot bangs remaining after all the cold spot bangs are eliminated. Another possibility is that the combination of hot spot bangs prior to testing slows down the program runtime to equal to or more than the original runtime, in which case the pre-search phase will remove all bangs and return the original program.

3 Evaluation

To calculate the performance of a particular benchmark using a particular optimization system (*e.g.*, Autobahn 2.0 or a restricted version using only a subset Autobahn 2.0's phases), we ran the system under test on the benchmark 10 times to account for random fluctuations. To compute summary statistics across all the benchmarks, we calculate the average of the runtime performance ratios and the average of the total number of suggested bangs across all 60 benchmark programs.

In our evaluation results:

- We show that 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 runtime of the program optimized by Autobahn 1.0 alone. We refer to this performance change as a 15.7% *optimization degradation*.
- We demonstrate that the pre-search phase removed at least one file from consideration in 21 of the benchmarks in the NoFib suite, corresponding to 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.
- We use a microbenchmark to show that the pre-search phase's suggestions for additional files to consider can improve Autobahn 1.0's optimization results by 86.6%.
- We evaluate the post-search phase on the NoFib benchmarks, showing it can reduce the number of inferred bangs by 93.8% with a 33% optimization degradation.
- We use Autobahn 2.0 in a case study to optimize the performance of `gcSimulator` [**?**], a garbage collector simulator. The system reduced the number of inferred bangs by 81.8% with a 15.7% optimization degradation.

4 Demo

In our demo, we describe in detail the theory behind our approach. Specifically, we explain the significance of incorporating GHC profiling information with an otherwise random genetic search, and why it is able to automatically reduce the number of bangs inferred by Autobahn 1.0 while maintaining roughly the same level of optimization. This portion of the demo also describes how bangs are sorted into categories, filtered, and eliminated in each of the pre-search and post-search phases, and provides 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.