AUTOBAHN: USING GENETIC ALGORITHMS TO INFER STRICTNESS ANNOTATIONS

YISU REMY WANG

Bachelor of Science
Computer Science
Tufts University

Committee: Kathleen Fisher, Norman Ramsey

May 2017

[May 14, 2017 at 18:42]

Yisu Remy Wang: Autobahn*: using genetic algorithms to infer strictness annotations,* Bachelor of Science, © May 2017

Wir fahren fahren fahren auf der Autobahn.

— Kraftwerk

I dedicate this thesis to my loving family.

NOTE

This thesis is based on joint work with Diogenes Nunez and Kathleen Fisher published in the 2016 Haskell Symposium under the same title (Wang, Nunez, and Fisher, 2016).

ABSTRACT

Although laziness enables beautiful code, it comes with non-trivial performance costs. The GHC compiler for Haskell has optimizations to reduce those costs, but the optimizations are not sufficient. As a result, Haskell also provides a variety of strictness annotations so that users can indicate program points where an expression should be evaluated eagerly. Skillful use of those annotations is a black art, known only to expert Haskell programmers. In this thesis, I demonstrate that automated heuristic search can find strictness annotations that consistently and significantly improve program performance. I introduce Autobahn, a tool that uses genetic algorithms to automatically infer strictness annotations that improve program performance on representative inputs. Experiments on 60 programs from the NoFib benchmark suite show that Autobahn can infer annotation sets that improve runtime performance by a geometric mean of 8.5%. Case studies show Autobahn can reduce the live size of a GC simulator by 99.3% and infer application-specific annotations for Aeson library code. A 10-fold cross-validation study shows the Autobahn-optimized GC simulator generally outperforms a version optimized by an expert. To achieve those improvements, Autobahn typically runs about 100 times longer than the running time of the program it optimizes. That can sometimes become several hours, and then the user can run Autobahn over night. Autobahn also adds an average of 24 annotations per 100 LOC. The user needs to reason about the soundness of each annotation, either with a termination proof or with unit tests. A second pass of genetic algorithms can reduce the average number of bangs to 16 per 100 LOC while retaining at least 85% of the performance improvement from the first pass. A demand analysis marks 10% of the remaining bangs as sound.

ACKNOWLEDGMENTS

Right now it is summer of 2017. two years ago, I walked into Professor Kathleen Fisher's office and said to her: "I couldn't get any internship this summer. Do you have anything for me to do?" Thus started my journey with AUTOBAHN. Looking back, I would not have traded that summer with any internship. Two years earlier still, I walked in a crowded conference room full of first year students. A tall, energetic man with a full head of Luxuriant Flowing Hair[1] was on stage. His passionate speech convinced me to enroll in his introductory programming course even though I planned to major in architecture. Two weeks later, I dropped the architecture class. Two weeks before graduation, I was still in a class with Professor Norman Ramsey, this time as a TA. I want to thank Professor Kathleen Fisher and Professor Norman Ramsey for teaching me almost everything I know about programming and computer science research.

Most of the pictures in this thesis were made in Halligan Hall, our computer science lab at Tufts University. For many nights, together with me late in Halligan was Diogenes Nunez. Without his support and encouragement, I could not have tamed the dozens of benchmark programs and survive the anxious hours of experiments that might result in a single error exit code.

Many others in the Computer Science department and the programming languages group at Tufts supported me with their friendship and expertise. They helped me find the right words in Norman's Technical Writing course as well as take me out to lunch after a whole morning of typing.

Nathan Ricci, an alumnus of PL at Tufts, provided the `gcSimulator` program. Our anonymous referees at the 2016 Haskell Symposium as well as Stephen Chang, Matthias Felleisen, and Simon Peyton Jones commented on an earlier version of this project. John Launchbury guided us into the strictness analysis literature and provided the `fact` example in Chapter 4. Simon Marlow helped us use ghc to obtain program statistics.

Finally, I could not have met any of these wonderful mentors and friends without my family's endless support. They have sacrificed a much more comfortable life for my education. I will always cherish their love and remember their dream to contribute to science and teaching.

1 See http://www.improbable.com/hair/gallery1/

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 MY PROGRAM IS TOO LAZY!

Meet Pat and Chris. They coded their most recent project in Haskell, a programming language they have been learning for the past few months. They are excited about how easy the project was to write and are confident that it is correct, but they are dismayed to learn that it is too slow. They chose algorithms and data structures that should lead to an efficient implementation, and so they are at a loss as to why the program is inefficient. After a little web search, they found a code segment very similar to theirs on Stack Overflow (Ondra, 2011). The code suffers from a performance bug called a *thunk leak*, which results from Haskell's *lazy evaluation strategy*. We will take a deeper look into the code in the next section.

To fix a thunk leak, the usual strategy is to insert program annotations called *strictness annotations*. A strictness annotation reverses the effect of lazy evaluation, or *laziness*, on the part of program it appears, thereby fixing the thunk leak. However, each annotation comes with risks. In a minor case, it can make a program run slower. In a serious case, it can make an originally terminating program run forever. These risks make adding strictness annotations a black art, well understood only by expert Haskell programmers.

For a while Pat and Chris thought they could solve the problem by relying on Haskell libraries that experts had already optimized. But then they realized that approach could not work: the necessary annotations for the library code depend upon how the library functions are used, but the library writer cannot always predict that. The GHC Haskell compiler also optimizes laziness, but the optimizations are too conservative and did not help. At this point, Pat and Chris are faced with unpalatable choices: spend a long time learning how to use strictness annotations, find an expert to help them, rewrite the code in a different language, or cope with the bad performance.

1.2 MY THESIS

In this thesis, I demonstrate that useful strictness annotations can be found by automated heuristic search. I have implemented such a search in a tool called Autobahn, which finds strictness annotations that consistently and significantly improve program performance. In my experiments, Autobahn made 60 programs run 8.5% faster on average. For some programs, Autobahn can produce spectacular im-

provements. The program in one of my case studies had 99% less live size after Autobahn optimization.

To achieve the above performance improvements, AUTOBAHN imposes unusual costs. It typically runs for about 100 times longer than the running time of the program it optimizes, and that can sometimes become several hours. After running AUTOBAHN, Pat and Chris still need to examine the annotations it suggests and decide whether to apply them. For each annotation, they need to see if it can introduce non-termination. They can either prove termination by hand or rely on unit tests. To make the job easier for Pat and Chris, AUTOBAHN runs the heuristic search a second time to reduce the number of bangs in the program. It also runs a static analysis in the GHC compiler to mark which of the bangs are safe. In my experiments, the second search reduced 45% bangs from the first search; the demand analysis marked 10% of the remaining bangs as safe.

Given AUTOBAHN's running time and the manual labor it requires, Pat and Chris should deploy it at the end of their development cycle when they only need to make the program run faster. Starting AUTOBAHN when they go to bed, they can expect to find a faster program when they wake up. Then they can spend the morning inspecting the annotations.

1.3 WHAT IS LAZINESS AND WHY IS IT GOOD?

Stepping back, lazy functional programming languages like Haskell can produce elegant and efficient programs. They only evaluate the expressions needed to compute the answer. For example, in a simplified version of the Stack Overflow code Pat and Chris found, lazy evaluation saves a lot of work by avoiding uneccessary function calls:

```
upgraderThread :: [Int] -> Int -> Int
upgraderThread ns 0 = length ns
upgraderThread ns n =
  let ns' = map (+ 1) ns
   in upgraderThread ns' (n - 1)
```

Function `upgraderThread` recursively updates the integer list `ns` in its first argument `n` times. Each time it increments every number in the list by one. At the end, it returns the length of the updated list. Under non-lazy, viz., *eager* evaluation, `upgraderThread` would always increment all the numbers in the list for every recursive call, whether their values are needed or not.

Under lazy evaluation, an expression is only evaluated when its value is needed. Formally, every new expression is stored unevaluated on the heap as a *thunk*. A thunk can be thought of as a lambda without arguments which saves all information needed for evaluation in its closure. Then, whenever a future evaluation *demands* the value of the expression, the evaluation *forces* the thunk. At this time, the

evaluation still may not complete. Instead, it forces the thunk to its outer-most data constructor into *weak head normal form*. For example, the list `[1..]` becomes `1:[2..]`, where `:` is "cons" in Haskell, and `[n..]` represent the thunk of an infinite integer list starting from n.

Thanks to lazy evaluation, the program Pat and Chris found above never wastes time in updating the integer list. Since `length` never demands the individual values in the list, it only forces the list into a sequence of cons's holding thunks. The call to `map` remains unevaluated.

Incidentally, laziness is one of the reasons why Pat and Chris chose Haskell. They believed laziness can make programs run faster. Besides performance, laziness also makes for beautiful programs because it supports modularity: it supports useful programming idioms and first-class control constructs. With laziness, researchers have developed robust algorithms for a variety of applications (Shan and Ramsey, 2017; Mangal et al., 2015; Cheung, Madden, and Solar-Lezama, 2016).

1.4 WHY CAN LAZINESS BE BAD SOMETIME?

Although laziness promises efficient and beautiful code, it can sometimes make programs unpredictable and slow. Because not all expressions are evaluated, the asymptotic behavior is not obvious from the source code; because a large number of thunks are eventually forced, they end up wasting time and space (Peyton Jones and Partain, 1994; Peyton Jones and Santos, 1998; Ennals and Peyton Jones, 2003). In this thesis, I focus on laziness' performance costs. To optimize laziness, the GHC Haskell compiler uses a static analysis (Sergey, Vytiniotis, and Peyton Jones, 2014) to find program points where it can avoid creating thunks. Although this analysis provides consistent performance improvements, programs can still be too slow. Because forcing a thunk, for example the length of an infinite list, may not terminate, the compiler must only force those it can prove to terminate. The compiler optimization is too conservative.

To address this deficiency, Haskell provides strictness annotations such as bang patterns[1] (*Bang Patterns* 2016). These annotations allow programmers to instruct the compiler to evaluate the corresponding expression immediately, without creating thunks. Judicious use of strictness annotations can improve program performance in terms of speed and memory usage by significant amounts (O'Sullivan, Stewart, and Goerzen, 2009, Chapter 25).

Unfortunately, as Pat and Chris discovered, non-expert programmers often struggle with how to add strictness annotations to improve performance. As Mitchell points out in his 2013 ACM Queue article (Mitchell, 2013)

1 Available through the -XBangPatterns compiler argument

> Compilers for lazy functional languages have been dealing with space leaks for more than 30 years and have developed a number of strategies to help. There have been changes to compilation techniques and modifications to the garbage collector and profilers to pinpoint space leaks when they do occur. . . . Despite all the improvements, space leaks remain a thorn in the side of lazy evaluation, producing a significant disadvantage to weigh against the benefits.

The full version[2] of the Stack Overflow code Pat and Chris found illustrates the challenges. Instead of directly updating a list of integers, the original `upgraderThread` wraps the list in a `Maybe` monad.

```
upgraderThread :: Maybe [Int] -> Int -> Maybe Int
upgraderThread nsM 0 = sum <$> nsM
upgraderThread nsM n = do
  ns <- nsM
  let !ns' = transform ns
  upgraderThread (return ns') (n - 1)

{- transform fully evaluates its argument -}
transform :: [Int] -> [Int]
...
```

The original program without the underlined annotation suffered from a space leak. The leak was caused by lazily evaluating the result of the call to `transform`. One "usual cure" (Ramsey, 2010) to fix thunk leaks is to add a bang at the accumulating parameter of recursive functions. Accordingly, one might expect that annotating `nsM` with a bang in the case where `n` is not zero would fix the leak because `nsM` accumulates thunks at every recursive call. However, that does not work here because `nsM` is only reduced to weak head normal form (its outermost constructor, either `Just` or `Nothing`), not fully evaluated. To completely eliminate the space leak, one needs to instead add a bang before `nsM'` (as underlined in the code fragment above) to trigger the call to `transform`. As program size grows, it is hard to spot the bindings where thunks build up.

1.5 AUTOBAHN COMES TO RESCUE

Pat and Chris can automatically infer strictness annotations with AUTOBAHN. First, they write their Haskell program without worrying about strictness annotations. Once they confirm their code is correct, they supply the program and representative data to AUTOBAHN. Autobahn then runs a genetic algorithm to search through all possible sets of annotations. Within these sets, it finds several candidate annotations that significantly improve program performance. AUTOBAHN

2 I still make minor modifications to the program here to simplify it

can start with a program with no annotations, or one that Pat and Chris have already started to optimize with bangs. It can also both add and remove annotations. At the end, AUTOBAHN returns a list of candidate annotation sets, ranked by how much each candidate improves performance. Pat and Chris then examine if the candidates are sound on relevant program inputs. Finally, if they are happy with an annotation set, they can instruct AUTOBAHN to apply it to the program.

AUTOBAHN's genetic algorithm iteratively considers a collection of candidate annotations. In each round, it preserves those annotations that demonstrate the best performance on the supplied data. Since AUTOBAHN starts with the original program, it is guaranteed to only suggest alternative annotations that actually improve upon the original performance on the supplied dataset.

As with any dynamic approach, it is important that the training data be representative of the data of interest. In the worst case, AUTOBAHN could introduce annotations that cause the program fail to terminate when given new input. For this reason, AUTOBAHN supplies a list of alternatives and asks Pat and Chris to choose from them. Pat and Chris may decide to adopt an annotation set that could lead to non-termination because they know that the triggering input values will never occur in practice. AUTOBAHN goes a step further to help Pat and Chris reason about the soundness of the annotations it infers. With a second pass of genetic algorithm, AUTOBAHN tries to reduce the number of bangs from the best performing candidates. With a demand analysis from the GHC compiler, AUTOBAHN marks bangs that the analysis can prove to be sound. In the future, I plan to extend AUTOBAHN to synthesize example inputs that trigger non-termination given a set of annotations. If those example inputs should never arise in practice, Pat and Chris can safely accept the annotations.

Pat and Chris can decide how much of the program's source it should infer annotations for. At one extreme, AUTOBAHN can analyze a single annotation point; at the other, it can analyze the entire source code for a program, including libraries. This expansive mode can be useful because in general, the appropriate strictness annotations for libraries is a property of how they are used, information not available to the library writer. Note, though, that AUTOBAHN will not duplicate code to allow for different annotations in different contexts, an important limitation particularly for larger programs.

In this thesis, I

- show how to use genetic algorithms to automatically infer strictness annotations that enable non-expert Haskell programmers to improve the performance of their programs on a variety of performance criteria: total runtime, garbage collection time, and live size (a.k.a. peak allocation).

- demonstrate the effectiveness of this approach on 60 programs from the NoFib (Partain, 1993) benchmark suite, showing geometric mean improvements of 8.5%, 18%, and 7.2%, and maximum improvements of 89%, 98%, and 99.3% on the total runtime, garbage collection time, and live size performance criteria, respectively. To achieve those performance improvements, AUTOBAHN inserts 24 annotations per 100 LOC on average.

- use AUTOBAHN in a case study to optimize the performance of a garbage collector simulator `gcSimulator` (Ricci, Guyer, and Moss, 2013). The annotations inferred on a small training set result in performance improvements on larger data sets: with 11 annotations per 100 LOC the running time decreased by 23.6% and the live size reduced to under 1% of the unoptimized program on the full dataset.

- show in a second case study that AUTOBAHN can infer application-specific annotations for Aeson (Bryan O'Sullivan, 2016) library code to optimize driver programs `validate` and `convert` that require different annotations to produce optimal behavior.

- show that the inferred annotations are stable across different data sets through a 10-fold cross-validation on `gcSimulator` and `convert`. For `gcSimulator`, the study also shows that the inferred annotations generally outperform the annotations added by hand by the original author.

- show that a second pass of the generic algorithm can reduce the number of bangs AUTOBAHN infers. For NoFib programs, the second pass reduced the geo-mean of 24 bangs per 100 LOC to 16. With fewer bangs, the programs retain at least 85% of the performance improvements from the first pass.

- show that the demand analysis from the GHC compiler can further reduce the number of bangs an AUTOBAHN needs to reason about. For NoFib programs, the analysis marks an average of 10% of the bangs to be safe. Thus Pat and Chris only need to inspect an average of 14.4 bangs per 100 LOC.

2 GENETIC ALGORITHMS

2.1 WHY A GENETIC ALGORITHM?

Intuitively, AUTOBAHN models how Pat and Chris insert bangs by trial and error. Unable to infer the bangs analytically, they would add bangs at a few arbitrary places and then benchmark the program. They would repeat this process until a certain set of bangs produce satisfactory performance.

This problem of finding a set of annotations to maximize program performance can be formalized as a search problem. Consider a function F that takes an argument x and returns $F(x)$. We wish to find an x that maximizes the value $F(x)$. If the number of possible values for x is large, we cannot exhaustively search for the x that optimizes F. Instead, we must turn to heuristic searches.

A genetic algorithm seems a natural choice. Just like the natural evolution it is modeled after, it can combine two strong entities to produce a stronger one. In the case of bang patterns, it is likely that if two sets of bangs both improve performance, together they will combine the improvements. For example, if functions `f` and `g` both benefit from bangs at their arguments, and they do not interfere with each other's evaluation, then adding the bangs for both will combine the improvements from adding the bangs for either. Even if that's not the case, genetic algorithm can generate new programs to escape from the local optimum.

AUTOBAHN also allows Pat and Chris plug in other search algorithms. Future work can compare the performance of other algorithms with that of genetic algorithms.

2.2 HOW DOES IT WORK?

A genetic algorithm (Goldberg, 1989) uses ideas from natural evolution to guide a heuristic search for a value of x that maximizes a function F. Each possible value of x is encoded as a sequence of *genes* that collectively form a *chromosome*. Function F is called a *fitness function* because it measures how fit each chromosome is to survive. The algorithm runs for a number of rounds, each of which is called a *generation*. Each generation starts with a group of chromosomes called a *population*. The algorithm computes the fitness of each chromosome in the current population by calculating the corresponding value of F. It forms the population for the next generation by promoting the fittest individuals of the current population and adding the offspring of the

current generation. The algorithm computes the offspring in two was: first, it randomly changes the genes in some members of the current population to perform *mutation*s; second, it splices together the chromosomes of others to perform *crossover*s. The result of the search is the "fittest" member of the population in the final generation.

Figure 1 shows the pseudo-code for the genetic algorithm AUTOBAHN uses, and Table 1 lists the various parameters with which the algorithm can be configured. I use *italics* to indicate the names of parameters. The algorithm creates an initial population using a *seed* chromosome. The *diversityRate* parameter determines how much the chromosomes generated for the initial population differ from the seed. When constructing a new chromosome for the initial population, each gene in the seed is mutated with probability *diversityRate*.

For each of *numGenerations* generations, the algorithm evolves a population of *populationSize* chromosomes. For each generation, the algorithm uses the *fitness* function to score each individual. To form the next generation, it first selects the *archiveSize* fittest chromosomes from the current generation in an operation called archiving. It then uses *mutateRate* to calculate the number of chromosomes for the next generation that should be created via mutation. To generate each such chromosome, it randomly picks a chromosome from the previous generation and modifies each of its genes with probability *mutateProb*. Next, the algorithm computes the number of chromosomes for the next generation that should be created via crossover. To generate each such chromosome, it randomly picks two chromosomes from the previous generation and splices them together. The algorithm returns either the highest scoring chromosome in the final generation (as shown in Figure 1) or a list of all the chromosomes in the final population along with their fitness scores.

Genetic algorithms differ from other heuristic search algorithms in the randomness introduced when creating each generation. Specifically, mutation and crossover introduce chromosomes that archiving alone would not. This randomness helps prevent the algorithm from getting stuck at local maxima. High values for *mutateProb* and *diversityRate* cause bigger chromosomal changes. Bigger changes lead to faster convergence, but also increase the odds of missing a good "nearby" chromosome.

```
procedure GENETICALG(diversityRate, numGenerations, populationSize, archiveSize, mutateRate, mutateProb, crossRate)
    population ←GENPOPULATION(seed, diversityRate, populationSize)   ▷ Generate initial population from seed using diversityRate
    scores ←MAP(fitness, population)                                 ▷ Calculate fitness of individuals

    for i = 1→ numGenerations do
        fittest ←SELECT(archiveSize, scores, population)             ▷ Get the archiveSize fittest chromosomes
        numMutants ← (populationSize - archiveSize) * mutateRate     ▷ Calculate number of mutants from mutateRate
        mutants ←MUTATE(population, numMutants, mutateProb)          ▷ Mutate numMutants chromosomes
        numChildren ← (populationSize - archiveSize) * crossRate     ▷ Calculate number of children from crossRate
        children ←CROSSOVER(population, numChildren)                ▷ Use crossRate to generate numChildren chromosomes
        population ← fittest ++ mutants ++ children                  ▷ Ready the population for the next generation
        scores ←MAP(fitness, population)                             ▷ Calculate fitness of each new individual
    end for
    best ← SELECTBEST(scores, population)
    return best
end procedure
```

Figure 1: Pseudo-code of a genetic algorithm to maximize the value of the *fitness* function starting from initial chromosome *seed*.

3 AUTOBAHN

3.1 GENES AND CHROMOSOMES

What is a gene in AUTOBAHN? Conceptually, a gene is a program source location where a bang may appear. A gene is *on* (represented by the bit `1`) if a bang appears at the corresponding source location; it is *off* (bit `0`) otherwise. Although multiple bangs can appear at the same location according to the Haskell syntax, AUTOBAHN inserts at most one bang per location. That is because bangs are idempotent: any value can be reduced to weak head normal form once and for all. Formally, for a given program `p`, consider the related program `p'` that is just like `p` except `p'` has no bangs. I call `p'` the *bare* version of `p`. A gene for `p` is any program location in `p'` where a bang pattern is legal. AUTOBAHN uses the `haskell-src-exts` library (Broberg, 2015) to identify the appropriate locations.

Because AUTOBAHN adds at most one bang per location, any Haskell program has a fixed number of candidate bangs and so a fixed number of genes. Consequently, AUTOBAHN encodes the space of all possible annotations with a fixed-length bit vector . A chromosome is a particular value for the bit vector. For our favorite `upgraderThread` function:

```
upgraderThread :: Maybe [Int] -> Int -> Maybe Int
upgraderThread _nsM _0 = sum <$> nsM
upgraderThread _nsM _n =
  let !nsM' = transform nsM
   in upgraderThread nsM' (n - 1)

transform :: Maybe [Int] -> Maybe [Int]
```

The program has 5 genes, one for each parameter and one for the `let` binding. The chromosome for the current bangs is the bit vector `00001`. The corresponding bangs indicate the expressions bound to `nsM'` should be evaluated eagerly, but all function arguments should be evaluated lazily. In some cases, adding or removing a bang at a program location does not have any effect. For example, a bang like `!(x, y)` is superfluous because pattern matching the tuple forces its evaluation. I am exploring how to identify these program locations and remove them from the chromosome.

3.2 HOW MANY GENES?

Another key question is deciding the extent of the program to allow AUTOBAHN to consider. The approach works at the level of source code, so it cannot explore changing annotations within pre-compiled portions of the program. For the portion for which source code is available however, there is complete flexibility. Conceptually, the tool can consider any subset of the program source: everything from the entire program down to a single bang pattern location. For simplicity, I have restricted this flexibility to the level of individual source files. Pat and Chris specify which source files they want AUTOBAHN to consider. The possible bang pattern locations in these files form the chromosome that AUTOBAHN will optimize over. By specifying which source code files to consider, Pat and Chris can limit the size of the search space by not including libraries or their own source code whose performance is irrelevant.

This approach means that for any libraries for which source code is available, AUTOBAHN can search for application-specific annotations. Currently, authors of high-performance libraries provide multiple versions of some functions to accommodate different use patterns. For example, the Aeson (Bryan O'Sullivan, 2016) library for parsing JSON provides strict and lazy versions of the key parsing function. To the extent that AUTOBAHN is successful, Pat and Chris won't have to worry about choosing the appropriate versions of such functions. Note, however, that AUTOBAHN will not copy library functions to infer different annotations for copies called in different contexts.

3.3 FITNESS FUNCTIONS

Genetic algorithms can search for chromosomes that optimize any measurable fitness function. Our approach for measuring the fitness of a particular chromosome is to run the corresponding program on user-supplied training data and measure the resulting performance using statistics provided by the GHC runtime. Given a chromosome and the associated Haskell sources, AUTOBAHN produces the program to profile by parsing the sources using the `haskell-src-exts` library (Broberg, 2015), modifying the resulting data structure representation of the program to reflect the bang pattern annotations described by the chromosome, and then pretty-printing the modified sources so they can be compiled, linked with binaries, and profiled with GHC.

There are a variety of performance metrics that programmers like Pat and Chris might care about. AUTOBAHN provides three different fitness functions they can choose among. Each of these functions works by parsing the output produced by GHC when invoked with the `+RTS -t` command-line option (*GHC Profling* 2016). The first fit-

ness function uses the total running time as the measure, rewarding faster genes. The second uses the reported garbage collection (GC) time: shorter GC times mean less GC work, which in turn implies less allocation; a reduction in GC time is also directly reflected in the total runtime of the program. The third uses the peak allocation statistic, corresponding to the live size of the program.

When evaluating programs to measure the fitness of the corresponding chromosome, we must keep in mind that introducing bang patterns may cause non-termination. Intuitively, chromosomes that cause non-termination are not fit and should be given poor fitness scores so that they die off. Therefore AUTOBAHN timeouts program runs that take longer than twice the running time of the original program. It allows programs that are slightly slower than the original because sometimes such programs lead to overall improvements when additional annotations are added in future generations. AUTOBAHN assigns very low scores to programs that trigger the timeout and to programs that terminate by throwing an exception, ensuring they die out.

AUTOBAHN also assigns fatally low scores to programs with invalid bang pattern annotations. AUTOBAHN generates such programs because the `haskell-src-exts` library permits bang pattern annotations in two kinds of places that trigger GHC errors. An example of the first kind comes from the NoFib (Partain, 1993) benchmark :

```
copy n x = take (max 0 n) xs
   where !xs = x : xs
```

The parser in ghc flags this use of a bang pattern on a recursively used variable as an error. The second kind of error arises when variables within typeclass instance declarations are annotated with bangs. For instance,

```
instance Monad Foo where
    !c1 >>= f = ...
```

raises a parse error on the bind operator. In both cases, ghc returns an error exit code. AUTOBAHN catches the error and assigns a low score to kill off the chromosome.

Another challenge for AUTOBAHN is when the original program takes a long time to run on the training data. A fundamental limit on the number of chromosomes AUTOBAHN can explore is how long it takes to run the original program on the training data. Shorter running times enable searching a larger portion of the annotation space. When a profiling iteration takes so long to finish that AUTOBAHN determines it cannot run 10 generations with a population size of 10 chromosomes (its defaults), it asks the user to supply a smaller set of representative training data.

3.4 ALGORITHM PARAMETERS - HOW DO I USE IT?

As discussed in Chapter 2 and shown in Table 1 , genetic algorithms can be configured in a number of ways. Choosing a good set of parameters can be confusing, and so AUTOBAHN attempts to determine reasonable default values, asking Pat and Chris to supply only the amount of time they are willing to let AUTOBAHN run and a measure of their confidence that a good set of annotations is "close" to the annotations in the program they supply. The goal is to maximize the possibility of performance improvement while guaranteeing the optimizer runs for a reasonable time.

TERM	TYPE	DESCRIPTION
diversityRate	float	probability with which each gene in seed is mutated to form initial population
numGenerations	int	number of generations to run algorithm
populationSize	int	number of chromosomes in each population
archiveSize	int	number of chromosomes to promote to next generation unchanged
mutateRate	float	percentage of the new population generated by mutation
mutateProb	float	probability with which each gene in a chromosome selected for mutation is changed
crossRate	float	percentage of the new population generated by crossover

Table 1: Genetic algorithm parameters

AUTOBAHN uses the supplied confidence level to set the *diversityRate* parameter. If Pat and Chris believe only slight changes to the original bang patterns are necessary, AUTOBAHN assigns a low value to *diversityRate* so that AUTOBAHN will focus on chromosomes that resemble the original annotation set. If Pat and Chris are less confident, AUTOBAHN uses a higher value to explore the search space more widely.

AUTOBAHN uses the total time that Pat and Chris are willing to run AUTOBAHN to calculate the highest possible values for the parameters *numGenerations* and *populationSize*, allowing us to explore as large a portion of the annotation space as possible in the allocated time. Since both parameters prolong AUTOBAHN's runtime, I find the "golden ratio" of the two parameters based on their effect on the possibility of discovering better annotation sets. In practice, a 4/3 ratio of *numGenerations / populationSize* works well as default. This ratio is somewhat

```
{ diversityRate  = 0.4 -- 1st generation diversity
, numGenerations = 20  -- Evolve for 20 generations
, populationSize = 15  -- 15 chromosomes/generation
, archiveSize    = 7   -- 7 best chromosomes survive
, mutateRate     = 0.2 -- 20% from mutation
, mutateProb     = 0.2 -- 20% chance a bang flips
, crossRate      = 0.8 -- 80% from crossover
, numFitnessRuns = 4   -- Profiling iterations }
```

Figure 2: Sample inferred configuration

unconventional for genetic algorithms, where the value of *numGenerations* is usually on the order of twenty times larger than *populationSize* [16]. I empirically adjusted these parameters to guarantee an affordable running time and a reasonable *populationSize*.

AUTOBAHN] uses simple default values for four parameters: *archiveSize* (7), *mutateRate* (0.2), *mutateProb* (0.2), and *crossRate* (0.8). I chose these values because they worked well in practice.

In addition to the generic genetic algorithm parameters described in the previous section, AUTOBAHN has an additional parameter *numFitnessRuns* that arises because the fitness function runs the program to measure its performance. To ensure the profiling information is accurate, AUTOBAHN runs each program on the training data *numFitnessRuns* times. It calculates an appropriate value for this parameter by iteratively profiling the unannotated program until the mean of the measured performance changes by less than 5%. It uses that number of iterations as the value for *numFitnessRuns*. The record in Figure 2 shows a sample inferred configuration. AUTOBAHN allows Pat and Chris to override default values if they wish.

3.5 THE FIRST GENERATION

After generating the algorithm parameters, AUTOBAHN populates the first generation. It seeds this generation with the chromosome that encodes the bang patterns in the user-provided Haskell source program. Starting from this seed chromosome, AUTOBAHN uses the *diversityRate* parameter to generate the required number of chromosomes to comprise a full generation (specified by *populationSize*). To produce each new chromosome, AUTOBAHN considers each gene in the seed and flips the value of that gene with probability *diversityRate*. Note that this process can both add and remove bang patterns.

3.6 PRODUCING NEW GENERATIONS

AUTOBAHN uses the Haskell genetic algorithm library GA (Hoste, 2011) to produce each successive generation, passing it the mutation and crossover functions, which I explain in turn.

MUTATION. For each generation, the GA library calls the function `mutation` *mutateRate* * *populationSize* times, each time passing in a randomly chosen chromosome `c` from the current population. Intuitively, the `mutation` function independently flips each gene in `c` whenever a randomly chosen floating point number between `0` and `1` exceeds the *mutateProb* threshold. The function makes use of three parameters. The first is a parameter `p` that represents the probability that a given gene should be flipped; AUTOBAHN sets this value according to the *mutateProb* parameter. Next, AUTOBAHN uses a `seed` parameter to generate randomness. Finally, `mutation` takes a parameter `c` that is the chromosome selected for mutation. The function works by calculating the number of genes in the chromosome (`len`), generating a random sequence (`fs`) of `len` floats, converting `fs` into a sequence (`bs`) of bits where a given bit is set whenever the float has a value smaller than (`p`). Finally, AUTOBAHN computes and returns the new chromosome `c'` by xor-ing `c` with the bit sequence.

CROSSOVER. For each generation, the GA library calls the `crossover` function *crossRate* * *populationSize* times, each time passing in a pair of randomly chosen chromosomes `c1` and `c2` from the current population. The `crossover` function implements the *Uniform Distribution* (Syswerda, 1989) strategy to ensure that each gene has an equal opportunity to change through evolution. Intuitively, the `crossover` function randomly picks half of the genes for the new chromosome from the corresponding positions in one parent, and the rest from another. This strategy makes stronger genes more likely to survive: when a new bang improves performance, its improvement is likely to persist regardless of other annotations. The function makes use of three parameters. As with `mutation`, AUTOBAHN uses a seed parameter to generate randomness. Next, `crossover` takes two parameters, `c1` and `c2`, as the chromosomes that have been selected as parents. Intuitively, the `crossover` function first generates a random sieve (`s`) whose length (`len`) matches that of a chromosome and that statistically will have half of its bits on (`map (< 0.5) fs`). For all the on-bits AUTOBAHN selects the genes from one parent (`c1'`) using a *bitwise-and* operation (`.&.`). It uses the off-bits to select the remaining genes from the other parent (`c2'`). AUTOBAHN then use a *bitwise-or* operation (`.|.`) to generate a new chromosome with roughly half of its genes from each parent.

3.7 DETERMINING A WINNER

When AUTOBAHN has evolved the population through the number of generations viable in the user-specified time window, AUTOBAHN returns a list of the surviving chromosomes ranked by their fitness score. AUTOBAHN provides a web interface that allows Pat and Chris to see the program with the suggested annotations. Upon request,

AUTOBAHN will produce copies of the program sources with the annotations specified by a particular chromosome.

3.8 PUTTING IT ALL TOGETHER

To summarize, I discuss how people like Pat and Chris use AUTOBAHN. First, they provide the inputs specified in Figure 1. AUTOBAHN uses this information to compute parameter values for the genetic algorithm (unless Pat and Chris have provided explicit parameter values instead). AUTOBAHN populates the first generation from the chromosome corresponding to the source program supplied by Pat and Chris. It uses the Haskell GA library for genetic algorithms to produce each successive generation, during which bang patterns can be both added and removed. At the end, it generates a web page showing a list of candidate program annotations ranked by the fitness score. If directed to do so by Pat and Chris, AUTOBAHN will produce a new version of the input sources that match any of the chromosomes that survived to the final round.

Complete program sources
Cabal file with compilation instructions
Subset of program sources to analyze
Level of confidence on current annotations
Metric to be optimized
Representative input data
A cap on the amount of time available to search

Table 2: User inputs to AUTOBAHN

3.9 SOUNDNESS

One challenge with introducing bang patterns is the possibility of changing the termination behavior of the program being optimized. For example, consider the following silly program:

```
let x = length [1..] in 10
```

This program terminates because there is no need to evaluate the expression bound to `x`. If, however, we annotate `x` with a bang pattern, we will force the evaluation of `length [1..]`, causing the program to run forever.

The GHC compiler uses a conservative strictness analyzer (Sergey, Vytiniotis, and Peyton Jones, 2014) to ensure that the compiler won't introduce non-termination when it decides to eagerly evaluate an expression. As with all static analysis, this analyzer is necessarily con-

servative and so GHC will consequently miss some optimization opportunities.

AUTOBAHN does not limit its search to annotations that do not change termination behavior. It will kill off any chromosomes whose corresponding program runs more than twice as long as the original program on the training data. As a result, AUTOBAHN will rule out any annotations that introduce non-termination on the training data. It is, however, possible to have a program that terminates on all training data but that fails to terminate on other input, even if the training data exercises all control flow paths. Consider, for example, the following program:

```
{- Note that fact diverges on negative numbers. -}
fact 0 = 1
fact z = z * fact (z - 1)

g x b = if b then x else 5
print (g (fact u) b)
```

Assume that `b` is almost always true, variable `u` is read from input, and the result of the call to `g` is needed (suggested by the call to `print`). The program will terminate for all values of `u` as long as `b` is `false`. Now, suppose the training data only has positive values for `u`. In this scenario, AUTOBAHN might well decide to add a bang pattern on the `x` variable in the definition of `g` because most of the time `b` is true and eagerly evaluating the call to fact is a performance win. Note that every line of code in the program is executed during the training, even though the user only passes in positive values for `u`. Now suppose `b` is true and the user passes in a negative value for `u` after AUTOBAHN has introduced the bang pattern. At this point, the program will diverge when the original would have terminated.

This scenario leads us to not automatically apply the best performing annotation set that AUTOBAHN finds: this annotation set might not preserve termination. It is up to Pat and Chris to verify that the suggested annotations lead to appropriate termination behavior. For the same reason, it is important that Pat and Chris supply representative training data. If negative values for `u` are legal inputs, then the training data should include examples of this form. Note that Pat and Chris might pick an inferred annotation set even if it introduces the possibility of non-termination. In the example above, Pat and Chris might know that the system will never in fact pass in a negative number and so introducing the bang pattern is fine.

Verifying soundness may not be easy. Even with this limitation, however, AUTOBAHN can help programmers like Pat and Chris, who currently have to first produce an annotation set and then reason about its soundness. With AUTOBAHN, the problem is reduced to reasoning about the soundness of annotation sets that achieve the desired performance. To help with this task, I extend AUTOBAHN with

a second pass of genetic algorithms which focuses on reducing the number of bangs in the progra. Finally, I run the GHC demand analyzer and mark the bangs it can prove to be safe.

3.10 FEWER BANGS!

After I implemented the original AUTOBAHN and ran it on a set of benchmark programs (see Chapter 4), I discovered one problem: it inserts too many bangs! On average, AUTOBAHN added 24 bangs per 100 LOC. This many bangs make it hard for Pat and Chris to reason about soundness. They would have to inspect the bangs one by one. In practice, only a few "critical bangs" can some time improve the program performance drastically. For example, program `convert` in the case study in Section 4.5 runs 25% faster with only 3 bangs, whereas AUTOBAHN inserted 124 to achieve the same performance.

The bangs AUTOBAHN infers usually include all the critical ones as well as some "non-opt" ones that barely affect performance. These non-opt bangs survive because of the nature of genetic algorithms: if a gene does not affect the fitness score, it is equally likely to have any value. In the case of bang patterns, a non-opt bang has equal chance to be on or off. As a result, half of them appear in the final candidates. Some of the non-opt bangs are easy to eliminate. If they appear in functions that are never called, i.e. dead code, AUTOBAHN can simply remove them. Others do not affect performance because they appear before terms already evaluated. For example, a bang like `!(x, y)` can appear because the pair is a valid pattern. However, the bang is redundant because the pattern match on the tuple already evaluates the pair constructor.

I extend AUTOBAHN in two ways to reduce the number of bangs it adds in order to help Pat and Chris reason about the final candidates. First, AUTOBAHN runs a second pass of the genetic algorithm but focuses on reducing the number of bangs. I call this second pass *minimizing phase*. This minimizing phase takes over the candidates at the end of the first pass as the initial population. It uses the number of bangs as the fitness score but assigns a bad score to any chromosome performing 15% worse than the best candidate. In this way, the minimizing phase can reduce the number of bangs without undoing the performance improvement. Besides the new fitness function, the minimizing phase uses the same configuration in the first pass. That means it will still explore chromosomes that did not appear in the first pass, and potentially discover those with even better performance.

Second, AUTOBAHN tries to locate dead code and eliminate bangs at already-forced terms. On outcome of the minimizing phase, AUTOBAHN runs the GHC compiler's internal analysis. GHC implements a static analysis that labels each expression with how it is used. This

includes two levels of information: 1. whether the expression is *used* at all to compute the final result, and 2. whether the expression needs to be *evaluated* for future computation. To understand the distinction between usage and evaluation, consider the following function:

```
f :: [Int] -> [Int] -> [Int] -> (Int, [Int])
f xs ys zs = (sum xs, ys)
```

Function `f` completely evaluates its first argument `xs` to compute the sum of the integers in the list. But it does not need to be evaluated `ys` at all, since it is returned as-is at the end of the function - it is *used* but not evaluated. `f` can simply ignore the last argument `zs` since it does not appear anywhere in the body - it is *unused*. The demand analysis of GHC therefore tags `xs` as evaluated and used, `ys` as unevaluated but used, and `zs` as unevaluated and unused. If we feed this function to AUTOBAHN, it can safely add a bang at `xs` which needs to be evaluated anyways; it may or may not introduce non-termination if it adds a bang at `ys`, because `f` does not evaluate `ys`; it can always avoid adding a bang at `zs` which will never evaluate.

The augmented AUTOBAHN reduced the number of bangs inferred by 54 % down to 11 per 100 LOC in NoFib programs. Within the remaining bangs, it marks 7% as safe. Therefore, Pat and Chris only needs to reason about 10 bangs per 100 LOC. Section 4.8 evaluates the reduction in more detail.

3.11 DISCUSSION

Haskell has four major kinds of strictness annotations: `seq` (*Seq* 2016), `deepseq` (*Deepseq* 2015), and related functions; strict application ($!) (*Strict Application* 2016); strict data type declarations (*Strict Fields* 2016); and bang patterns (*Bang Patterns* 2016). Currently, AUTOBAHN searches only for bang patterns. I chose not to search for places to insert seq or related functions because the search space is too large. I am currently exploring using AUTOBAHN to insert strict applications and strict datatype annotations.

4 EVALUATION

I evaluate Autobahn in a number of ways:

1. By running it on four small programs for which I can perform an exhaustive search, showing that Autobahn computes the optimal annotation set.
2. By running it on 60 programs taken from the NoFib (Partain, 1993) benchmark suite and measuring the performance gains when optimized for total runtime, garbage collection time, and live size.
3. By comparing the NoFib performance produced by Autobahn with that produced by Strict Haskell (*Strict Haskell* 2016) via the pragmas `-XStrict` and `-XStrictData` that force eager evaluation.
4. By running it on a garbage collection simulator whose poor performance was the original motivation for Autobahn and showing that (1) performance gains inferred from a small training set carry over to larger datasets and (2) Autobahn annotations compare favorably to those introduced by hand.
5. By running it on two different driver programs that use the Aeson (Bryan O'Sullivan, 2016) library for parsing JSON and showing that Autobahn infers application-specific annotations for the Aeson code that improve the overall performance of the programs.
6. By measuring the stability of Autobahn's optimization with 10-fold cross-validation on the garbage collection simulator and one of the Aeson driver programs.
7. By measuring the time it takes Autobahn to infer the annotations for the NoFib benchmark and for the two case studies.

Experimental Setup. All programs were compiled and run on a computer with four 16-core AMD Opteron 6380 processors clocked at 2.5 GHz and 128 GB of RAM. I compiled Autobahn itself with GHC version `7.8.4` with the `-O2` flag. I compiled the benchmarks with ghc version `7.10.3` with `-O2` and `-XBangPatterns` along with NoFib's default flags. I add `-funbox-strict-fields` for those benchmarks that already have strict fields to ensure they still compile. Profiling was not enabled. To obtain more accurate information about live sizes, I

forced ghc to perform frequent garbage collections via the flags `+RTS -h -i0.01`. For the `StrictHaskell` comparison, I used GHC version `8.0.1` because the relevant pragmas were not present in `7.10.3`. I ran each benchmark 4 times and report our results using NoFib's geometric-mean reporting convention for uniformity with other studies.

4.1 SMALL PROGRAMS: A SANITY CHECK

First, I ran AUTOBAHN on a set of four small programs for which I was able to exhaustively explore all possible bang pattern annotations to calculate the "right answer". For all four programs, AUTOBAHN infers the bang patterns that produce the best performance on all of our performance criteria. In the following, I briefly describe the programs and their strictness properties.

The `fib` function below uses accumulating parameters `a`, `b1`, and `b2` to calculate the `n`th Fibonacci number. The function has six genes, one before each parameter to fib (including `0` and `_`). Adding bangs to either `a` or `b2` completely eliminates the thunk leak. Other bangs have no effect.

```
fib :: Int -> Integer -> Integer -> Integer
fib 0 _ b1 = b1
fib n !a !b2 = fib (n - 1) b2 (a + b2)
```

The second program, taken from Edward Yang's blog on thunk leaks (Edward Z. Yang, 2011) needs three bangs to achieve top performance. Each of the three bangs individually improves performance slightly, but all three together produce the best performance.

```
f [] c = c
f (x : xs) !c = f xs (uncurry (tick x) c)
tick x !c0 !c1
  | even x = (c0, c1 + 1)
  | otherwise = (c0 + 1, c1)
```

The third program is our favorite `upgraderThread` program. The fourth program, unlike the previous three, introduces the possibility of non-termination. In particular, adding strictness before variable `as` causes the program to diverge. The best annotation set is to only annotate variable `a`.

```
u = 0 : go (head u) (tail u)
go !a as = a + 1 : go (head as) (tail as)
main = do print $ u !! 1999999
```

4.2 NOFIB BENCHMARKS

I ran AUTOBAHN on 60 benchmarks from the NoFib benchmark suite. I optimized each program three separate times: once on runtime, once

on GC time, and finally on live size. These benchmarks comprise all of the NoFib programs that are can be compiled by GHC version `7.10.3` and can be processed without errors by the `haskell-src-exts` parser library (Broberg, 2015) version `1.17.1`. The smallest of these benchmark programs (`rfib`) has 5 genes, while the largest (`anna`) has 7709. Table 3 lists the programs, giving the number of lines of code, the number of program files, and the number of genes in each.

Figure 3 shows the performance of each of the 60 NoFib benchmark programs when trained and measured on total runtime, GC time, and live size, respectively. In each graph, the horizontal axis lists the benchmark programs in order of increasing number of genes. The vertical axis shows the normalized performance of each benchmark, reporting the ratio of AUTOBAHN-optimized performance to the original program's performance. Values less than 1.0 thus represent improvement. Because AUTOBAHN returns the program unchanged if it cannot find an improvement, I expect values to be less than or equal to one. In the graphs, programs whose data point is a triangle represent programs for which the AUTOBAHN-optimized version of the program performs slightly worse than the original. A manual review revealed the degradation was caused by noise. Circles represent programs whose original performance took so close to zero time that no measurable improvement was possible.

The results are encouraging. Following Fleming and Wallace, 1986, I report results as geometric means. Overall, AUTOBAHN decreased the runtime by 8.5%, reduced time spent in GC by 18%, and reduced the live size by 7.2%. The `lcss` program saw the best improvement on all metrics, with deltas of 89%, 98%, and 99.3% respectively. The comments for `lcss` state there are many opportunities for optimization, which helps explain why this particular program improved so much.

To explore the reason for the performance improvements, I selected the twelve NoFib programs whose run times decreased the most and compared the heap profiles of the original and AUTOBAHN-annotated versions [1] . The heap profiles of the improved programs all reported decreases in peak memory use. However, the shapes of the graphs remained largely unchanged. Some programs, like `simple` and `fulsom`, reduce the amount of time the program spent at peak memory. I believe this reduction in memory usage translated to the runtime improvements.

4.3 STRICT HASKELL

Strict Haskell (*Strict Haskell* 2016) provides language pragmas to make Haskell modules strict rather than lazy by default to improve performance. Specifically, as of version `8.0.1`, ghc has two additional

1 Available at https://genetic-strictness.github.io/Autobahn/profiles

PROGRAM	LOC	FILES	GENES	...			
rfib	12	1	5	awards	115	2	99
x2n1	35	1	6	fish	128	1	102
tak	16	1	9	puzzle	170	1	103
primes	18	1	12	treejoin	121	1	119
banner	108	1	17	n-body	188	1	122
queens	19	1	18	eliza	267	1	138
bernouilli	40	1	19	power	142	1	180
kahan	58	1	23	cichelli	195	4	205
exp3_8	93	1	24	cse	464	2	222
pidigits	22	1	27	pretty	265	3	229
integrate	43	1	28	pic	527	9	235
cryptarithm1	164	1	33	clausify	184	1	246
wheel-sieve1	41	1	38	minimax	238	6	299
fasta	58	1	44	boyer2	723	5	302
integer	68	1	47	expert	525	6	424
wheel-sieve2	47	1	47	hidden	507	14	430
life	53	1	48	gamteb	701	13	458
rsa	74	2	50	multiplier	501	1	468
binary-trees	74	1	51	prolog	643	9	514
maillist	178	1	52	infer	590	16	586
gen_regexps	39	1	54	fem	1286	17	655
gcd	60	1	57	scs	585	7	770
scc	100	2	59	simple	1129	1	845
cryptarithm2	128	1	60	reptile	1522	13	895
lcss	60	1	71	symalg	1146	11	1148
atom	188	1	74	gg	812	9	1192
paraffins	91	1	75	cacheprof	2151	3	1228
fannkuch	103	1	88	fulsom	1392	13	1433
calendar	140	1	92	fluid	2401	18	1688
ansi	128	1	96	anna	9561	32	7709

Table 3: Statistics for the NoFib benchmarks

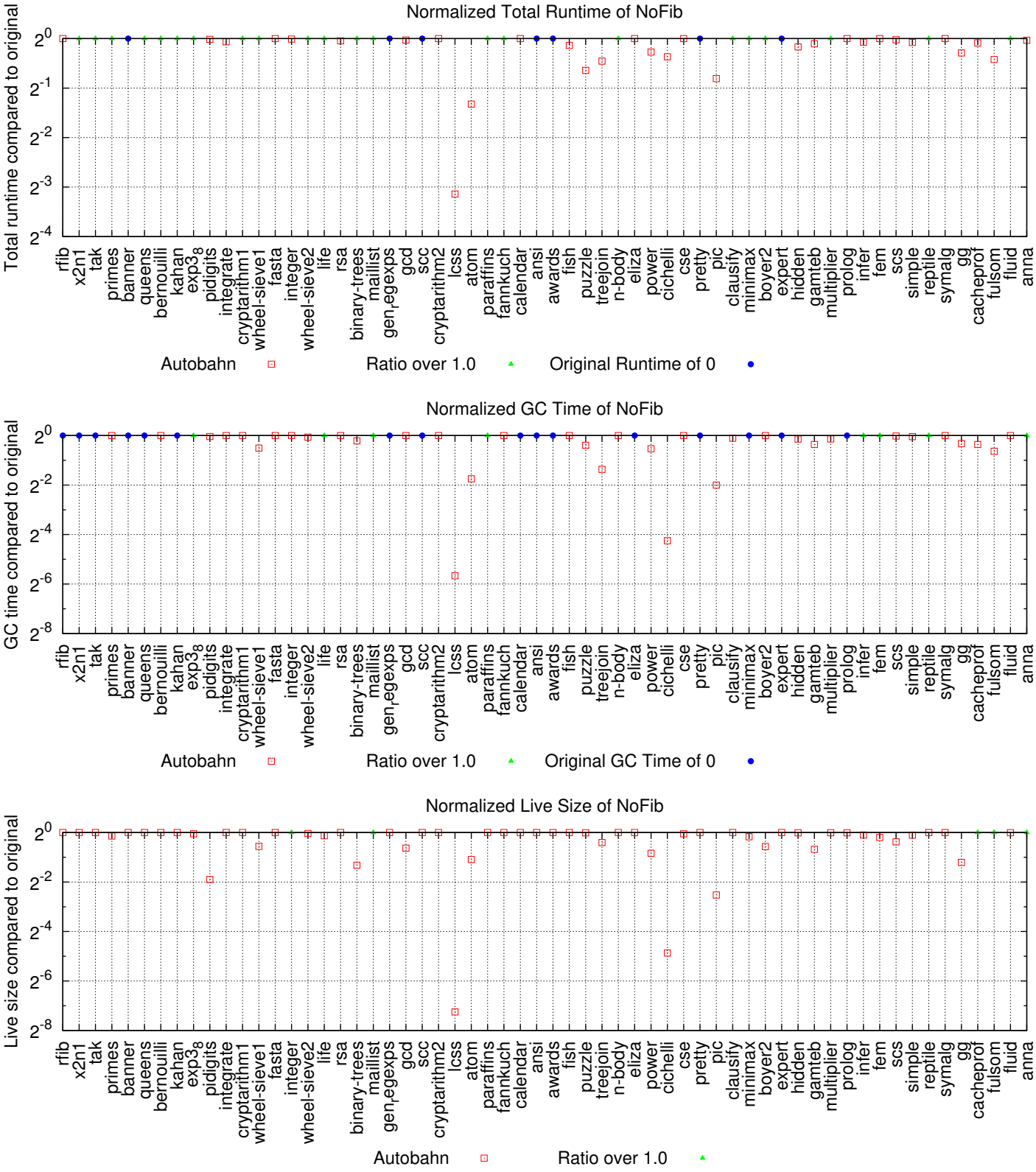


Figure 3: Performance of AUTOBAHN-optimized programs normalized by the original program's performance; lower values are better. The data show geometric mean improvements of 8.5%, 18%, and 7.2%, and maximum improvements of 89%, 98%, and 99.3% on the total runtime, garbage collection, and live size performance criteria, respectively.

language extensions: `-XStrictData` and `-XStrict`. According to the GHC wiki page (*Strict Fields* 2016), when someone compiles a module with the pragma `-XStrictData`, datatypes declared in that module become strict by default. When compiled with `-XStrict`, the compiler makes functions, data types, and bindings in the module strict by default.

How does the performance of programs compiled with Strict Haskell compare to those optimized with AUTOBAHN? To answer this question, I compiled and ran the NoFib programs with GHC `8.0.1` using `-O2`, `-XBangPatterns`, `-funbox-strict-fields`, `-XStrict`, and `-XStrictData` flags. Figure 4 shows the results. Seventeen of the NoFib programs failed when using Strict Haskell. These programs failed for one of the following reasons:

- The program uses an infinite list. For example, wheel-sieve1 and wheel-sieve2 specify an infinite list of primes but demand only the first few. With Strict Haskell, the programs try to evaluate the infinite lists.
- The program depends on a lazy evaluation of error to detect a specific problem. For example, infer puts error at the end of a list; reaching this value signals an error. With Strict Haskell, the error is always triggered.
- The program contains a latent dynamic error. For example, reptile crashes when nil is passed to the tiletrans function, which does not occur when the program is evaluated lazily, but does occur when using Strict Haskell.

Nine programs performed worse; some significantly so, because Strict Haskell forces the evaluation of expressions that are not needed. AUTOBAHN did better than Strict Haskell on all of these programs. Of the programs that improved under Strict Haskell, two did better than AUTOBAHN: `exp3_8` and `treejoin`. In all other cases, AUTOBAHN did as well as or better than Strict Haskell. It is possible to add laziness annotations to the programs whose performance degrades under Strict Haskell, but that requires determining where to insert the annotations, another hard problem (Chang and Felleisen, 2014).

4.4 CASE STUDY: GCSIMULATOR

As a case study, I used AUTOBAHN to optimize gcSimulator, a garbage collection simulator that uses trace files generated by the Elephant Tracks (Ricci, Guyer, and Moss, 2013) tool to understand the performance of garbage collectors. The simulator consists of 20 files and 2026 lines of code. A chromosome for this program consists of 132 genes. The trace files are very large, on the order of gigabytes, resulting in gcSimulator execution times on the order of several min-

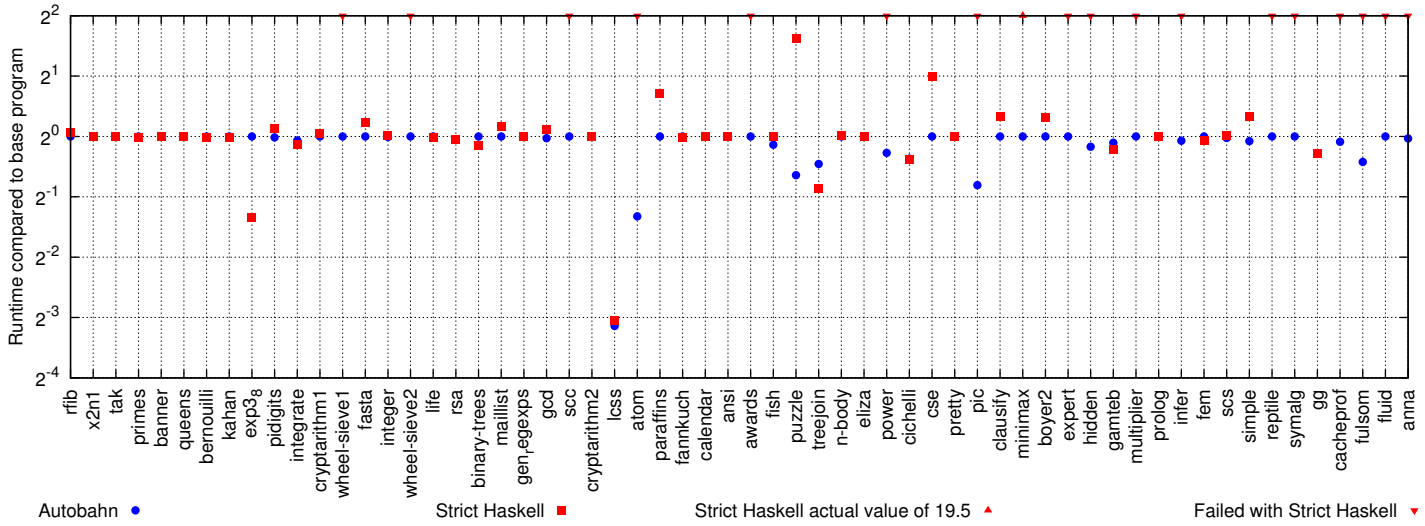


Figure 4: Runtime of AUTOBAHN-optimized programs and programs compiled with Strict Haskell normalized to base programs run in GHC 8.0.1. Lower on the y-axis means the AUTOBAHN version of the program ran *faster*.

utes. This performance makes running AUTOBAHN prohibitively time consuming. Consequently, I used only the first 512KB of one of the traces as the input data source, specifically, a prefix of the trace for the batik program of the DaCapo benchmarks (Blackburn et al., 2006). I then evaluated the performance of the AUTOBAHN-optimized version of gcSimulator on increasing sizes of the same trace. For this study, I seeded the initial population with the bare program; I did not include the annotations the original author had added by hand. For these measurements, I compiled all versions of the gcSimulator with the same settings as NoFib along with `-rtsopts` to gather runtime information.

Table 4 shows the performance of the hand- and AUTOBAHN- optimized programs normalized to the bare version. AUTOBAHN strove to decrease peak allocation, a choice consistent with that the goal the original author used when manually inserting bang patterns. Each colored band in the table represents a run with trace inputs of increasing size. When run against the training data, both the hand- and the AUTOBAHN-optimized version use 80% of the peak memory of the unannotated version. As the input data increased in size, we see AUTOBAHN's version pull ahead in reducing peak memory usage and, as a result, time spent in garbage collection. When given the entire batik trace, we see AUTOBAHN has reduced memory usage to less than 1% of the unannotated version, compared to the original author's 7%. This change results in an overall reduction of runtime to 76.4% of the bare program. Figure 5 shows the heap profile for three versions of gcSimulator when run on half of the `batik` trace. We see the expert's annotations reduced the live size of the program and eliminated some

	PEAK ALLOC(MB)	TOTAL RUNTIME	GC TIME
training data	0.8	0.898	0.556
	0.8	0.905	0.417
$\frac{1}{3}$ of trace	0.140	0.616	0.094
	0.137	0.586	0.050
$\frac{1}{2}$ of trace	0.318	0.612	0.136
	0.021	0.812	0.069
full trace	0.072	0.914	0.444
	0.005	0.764	0.272

Table 4: Peak memory allocation, total runtime, and GC time for hand- and AUTOBAHN-optimized version of `gcSimulator`, normalized to the bare program. For each band, the first row shows the by-hand results and the second AUTOBAHN results.

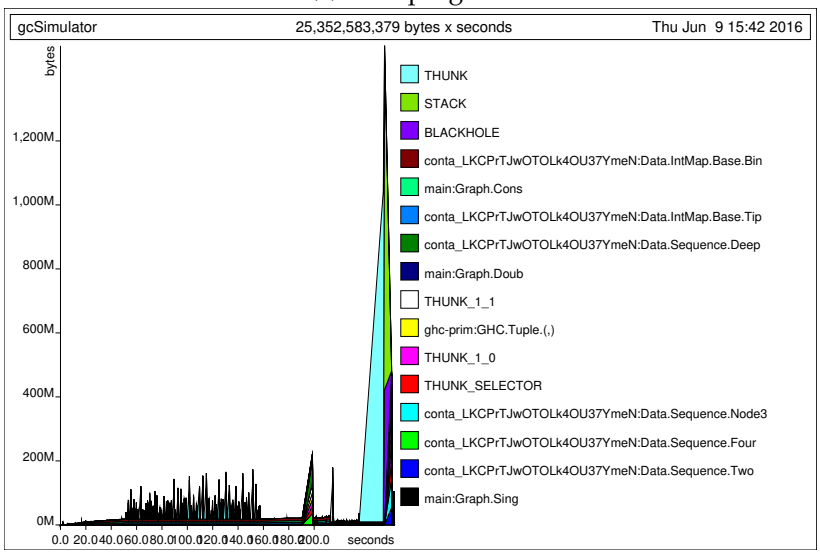
thunk leaks. The AUTOBAHN- optimized version further reduced the live size and also changed its allocation behavior. In particular, we do not see a sharp rise in memory usage at the end of the program's lifetime. Table 4 and Figure 5 further show that the annotations inferred for a small training set provide benefits even when the program is given larger inputs.

4.5 CASE STUDY: AESON LIBRARY WITH TWO DIFFERENT DRIVERS

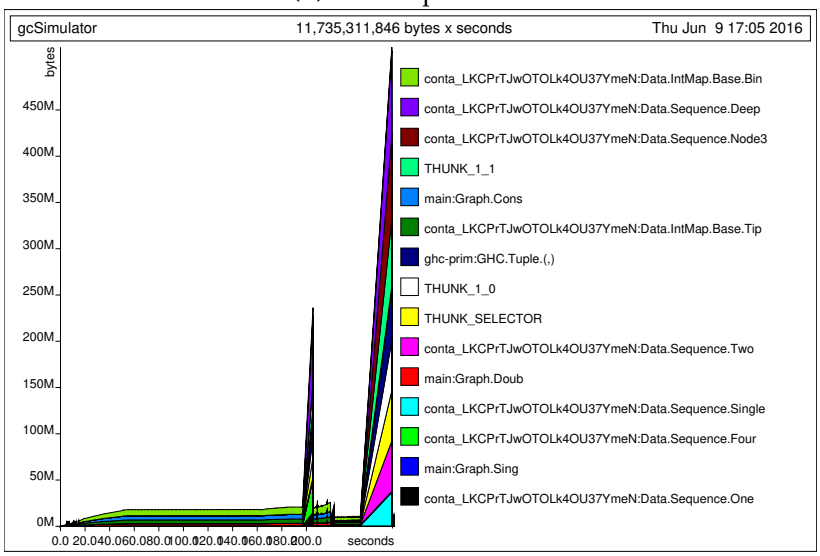
A second case study shows AUTOBAHN can infer different annotations for the same library code based on how the library is used. The Aeson parser library pairs with two different diver programs called `validate` and `convert`. Each driver program takes in a JSON file containing a list of records. Driver `validate` checks if its input respects the JSON syntax. It does not need to completely evaluate each JSON value. Therefore, `validate` achieves the best performance if it evaluates some parsing functions lazily. Driver `convert` converts its input to a Haskell data structure. It needs to completely evaluate each JSON value. Therefore, `convert` achieves the best performance if it evaluates all parsing functions eagerly.

Each driver program takes in one of 5 JSON files published by the City of Chicago (*City of Chicago Public Datasets* 2012). The files are `objects.json`, `A`, `B`, `C`, and `D`, by increasing size. The smallest file `objects.json` is passed to each driver during AUTOBAHN optimization so that AUTOBAHNfinishes in reasonable time. The larger 4 files are passed to each driver before and after AUTOBAHN optimization so that performance improvements can be measured on realistic inputs. The drivers and the Aeson library are compiled with the same set-

(a) Bare program



(b) Hand optimized



(c) AUTOBAHN-optimized

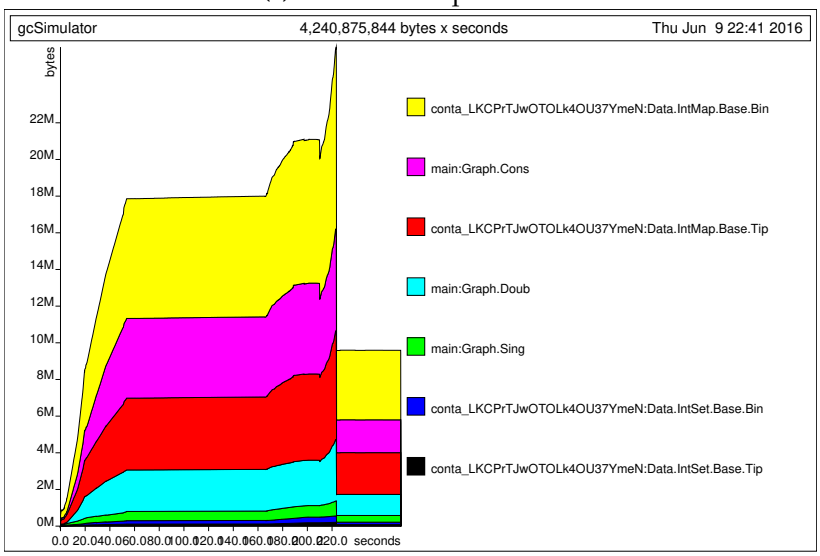


Figure 5: Heap profiles of `gcSimulator` on $\frac{1}{2}$ of the `batik` trace.

tings as NoFib except for 2 compiler flags. The `-rtsopts` flag is added to gather runtime information, and the `-funbox-strict-fields` flag is removed since Aeson does not use strict fields.

AUTOBAHN takes in the source of each driver program as well as the relevant part of the Aeson library source. Each driver/library pair consists of two files totaling 320 lines of code. Each pair has a chromosome with 194 genes. The original driver programs use the "wrong" version of the parsing functions so that there is room for improvement. The original `validate` uses strict parsing functions with too many annotations, and the original `convert` uses lazy ones with too few annotations. AUTOBAHN optimizes each driver/library pair for shorter runtime.

Table 4 shows the result of AUTOBAHN's optimizations. Both programs ran significantly faster than their original versions: `validate` ran a maximum of 37% faster and `convert` ran a maximum of 34% faster. `convert` also used an average of 28.3% less memory than its original version, whereas `validate` used the same amount of memory as its original version. Because `convert` used less memory, it also spent less time in garbage collection, therefore also ran faster overall.

To achieve these performance improvements, AUTOBAHN inferred different annotations in the Aeson library for each driver. For `convert`, AUTOBAHN adds a bang in a function [2] that inserts values into a hash table. For `validate`, AUTOBAHN removes the same annotation in an eager version of that function. Because `convert` eventually needs to fully evaluate the values in the hash table, it will force all hash table insertions. Therefore an eager insertion that does not create thunks is more efficient than a lazy one that creates thunks. AUTOBAHN adds the annotation to make the insertion eager and eliminates unnecessary thunks. Because `validate` does not need to fully evaluate the values in the hash table, it will not force all hash table insertions. Therefore a lazy insertion that does not insert the value is more efficient than an eager one that inserts the value. AUTOBAHN removes the annotation to make the insertion lazy and avoids unnecessary computation.

4.6 10-FOLD CROSS-VALIDATION

To assess the applicability of AUTOBAHN's optimizations to non-training data, I perform 10-fold cross-validation on `gcSimulator` and the convert Aeson driver. To apply this methodology to the optimization of `gcSimulator`, I first select ten different input data sets. Next, I use AUTOBAHN to optimize `gcSimulator` using each input file in turn to obtain ten different optimized programs. Finally, I evaluate each optimized program on all ten inputs. For these experiments, AUTOBAHN optimized for runtime. I measured the performance of the optimized

2 the function calls `H.insert` in the `objectValues` function in the Aeson library source.

	PEAK ALLOC(MB)	TOTAL RUNTIME	GC TIME
A (46MB)	1.0	0.869	0.979
	0.457	0.661	0.434
B (50MB)	0.999	0.907	1.062
	0.853	0.830	0.808
C (51MB)	1.0	0.638	0.855
	0.834	0.754	0.671
D (68MB)	1.0	0.900	0.988
	0.810	0.787	0.694

Table 5: Peak memory allocation, total runtime, and GC time for AUTOBAHN-optimized version of two Aeson driver programs, normalized to the bare program. For each band, the first row shows the results for `validate` and the second for `convert`.

programs on both runtime and live size. The methodology for convert is analogous.

For `gcSimulator`, I chose as input ten different traces of programs from the `DaCapo` Benchmarks. Because the full traces lead to long training times, I selected the first 35 million lines from each trace. I tested the optimized programs, however, on the full traces. For convert, I chose ten different JSON datasets from the data made available by the City of Chicago, ranging in size from 32 to 68MB. I trained and tested the optimized programs on the full data files.

Figures 6 and 7 show the performance of each program/input pair compared to the bare program for `gcSimulator` and convert, respectively. Each label `training-opt` on the horizontal axis corresponds to a program trained on input `training` and evaluated with performance criteria `opt`. For `gcSimulator`, I also show the performance of the hand-optimized version of the program.

Figure 6 shows that AUTOBAHN produces consistent runtime improvements of roughly 60% for `gcSimulator`. The live size measurements suggest the runtime improvement likely comes from fixing thunk leaks. The data also shows that AUTOBAHN annotations outperform the hand annotations for `gcSimulator`: the geometric mean of `gcSimulator` runtime when optimized by AUTOBAHN is 58.6% of the bare runtime, compared to a geometric mean of 64.8% of bare for the hand-optimized version. AUTOBAHN-optimized versions used a geometric mean of 9.6% of the bare live size whereas the hand optimized ones used 22.8%.

The AUTOBAHN version of convert reduced its total runtime to a geometric mean of 65.2% of the bare runtime. AUTOBAHN also reduced the live size to a geometric mean of 78.6% of the bare live size, which contributed to the improved runtime. Since the bare version of the

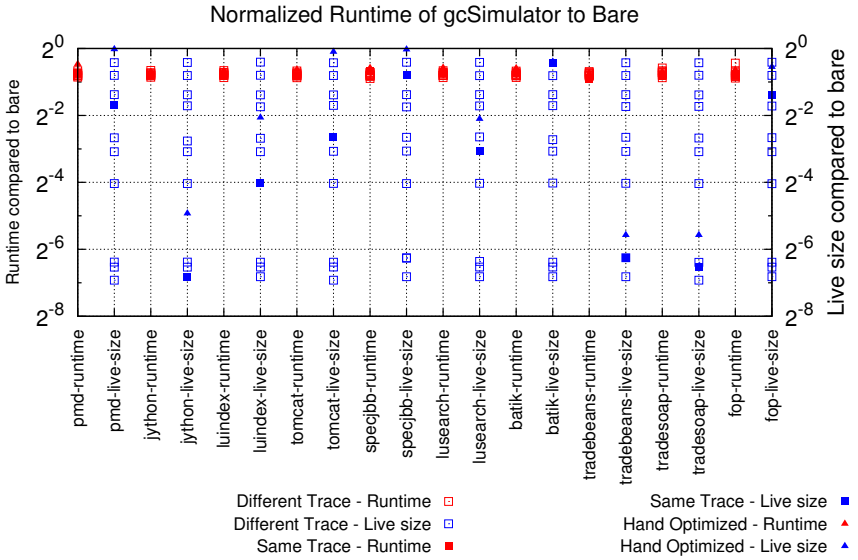


Figure 6: 10-fold evaluation for `gcSimulator`, showing runtime and live size performance improvements of AUTOBAHN versions of `gcSimulator` compared to the bare program. I highlight points where the AUTOBAHN-optimized program ran on its training trace. I also show how the hand-annotated program performed.

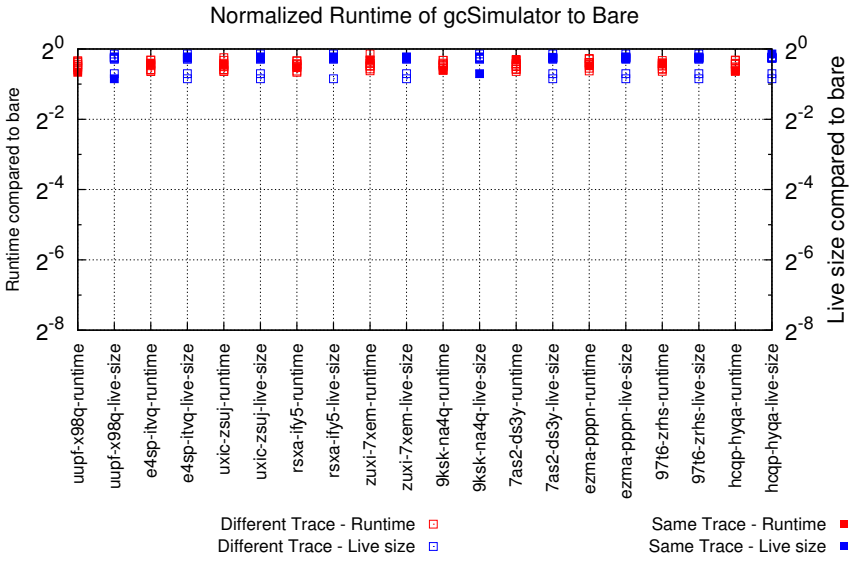


Figure 7: 10-fold evaluation for `convert`, showing runtime and live size performance improvements of AUTOBAHN versions of `convert` compared to the bare program. I highlight points where the AUTOBAHN-optimized program ran on its training trace.

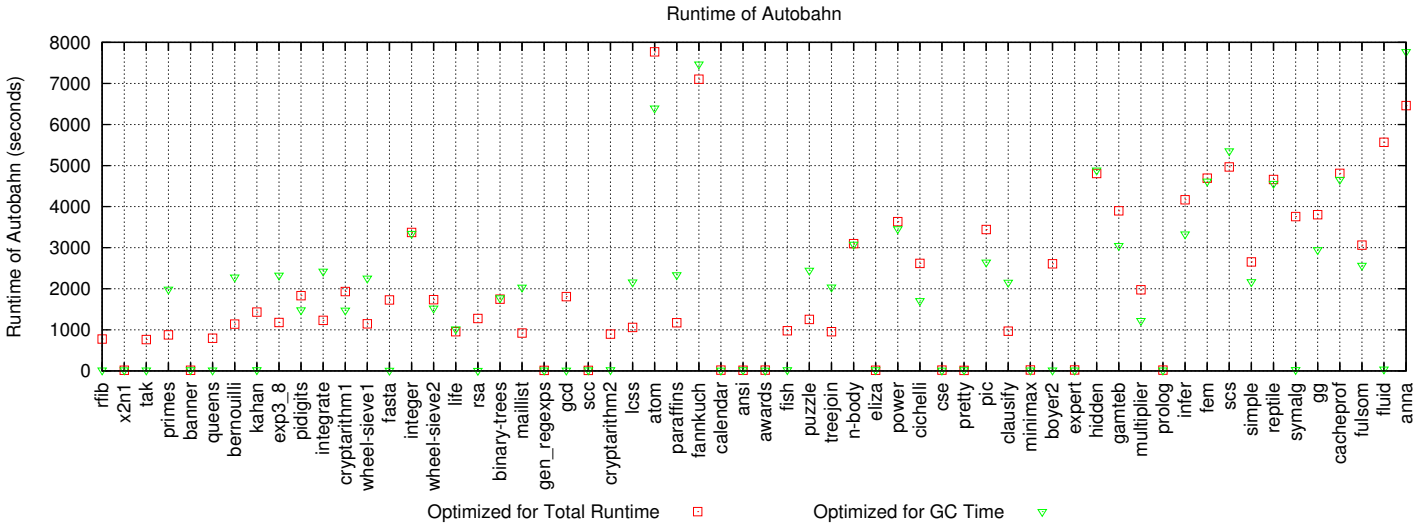


Figure 8: AUTOBAHN running time to optimize each program in the NoFib benchmark suite.

convert driver evaluates as much as it can to weak head normal form, the live size and runtime are related.

5.7 AUTOBAHN Performance

Finally, Figure 8 shows how long it took AUTOBAHN to analyze each of the programs in the NoFib benchmark suite for the total runtime and GC time performance criteria. When optimizing the benchmarks for runtime, AUTOBAHN took 861.166 seconds, or 14 minutes (geometric mean). When AUTOBAHN is run optimizing for GC time, it took 732.451 seconds, or 12 minutes. We see that there are some benchmarks where AUTOBAHN runs for close to no time at all when optimizing for runtime and GC time. In those cases, AUTOBAHN found that the bare program had a runtime of 0 CPU seconds or spent no time in the collector. Since it cannot optimize below that value, AUTOBAHN terminated and reported the bare program as the optimal chromosome. I have yet to run this experiment to capture the runtime when optimizing for live size.

4.7 AUTOBAHN PERFORMANCE

As for our case studies, it took AUTOBAHN 5 hours and 34 minutes to optimize the `gcSimulator` and 2 hours and 12 minutes to optimize the Aeson convert driver. For each program, I ran AUTOBAHN once, optimizing for runtime. These two programs in particular have much longer running times than those in the benchmark suite. For instance, `gcSimulator` emulates an entire program run, complete with garbage collections, on top of a garbage collected language. This results in a long runtime before AUTOBAHN adds any strictness.

Since the use case for AUTOBAHN is that programmers like Pat and Chris use the tool once they have a working program they want to

optimize, I see these running times as acceptable. I imagine Pat and Chris starting the tool before they go to bed and wake up with candidate annotations to consider.

4.8 SOUNDNESS

AUTOBAHN gives good performance improvement if we do not limit how many bangs it should add. But too many bangs make it difficult for Pat and Chris to decide if they may introduce non-termination. Therefore, as described in Section 3.10, AUTOBAHN's second phase aims to reduce the number of bangs without sacrificing performance. After the second phase, the resulting programs run at least 85% as fast as those after the first phase, with a few programs running even faster. The final programs from the NoFib benchmark suit have an average of 11 bangs per 100 lines of code, reduced from the 24 bangs per 100 LOC after phase 1 by 54 %.

The algorithm in phase 2 guarantees the final program runs at least 85% as fast as those after phase 1. Because AUTOBAHN assigns very bad score for chromosomes with more than 15% worse performance, they rarely survive through the next generation. It still allows for the small slowdown to trade off between a fast program and a readable one. Figure 9 compares the performance improvement from the original program after phase 2 with that after phase 1. In most cases, phase 2 did not affect performance. For a few programs, like `gcd`, `atom` and `pic`, phase 2 produced slightly slower program than phase 1. Programs `banner`, `gen_regexp`, `scc` and `ansi` ran too fast for AUTOBAHN to improve. For all other programs phase 2 improved upon the original performance. The same figure compares the number of bangs after each phase. As expected, phase 2 consistently produces fewer bangs than phase 1. This is also guaranteed by the algorithm, because phase 2 starts with the candidate programs from phase 1. Just as phase 1 never produces programs slower than the original one, phase 2 never produces programs with more bangs than phase 1's. Finally, for programs like `scs`, `puzzle`, and `binary-trees`, phase 2 allows for slightly slower programs in favor of fewer bangs.

After the second phase of genetic algorithm, AUTOBAHN goes a step further to help Pat and Chris reason about the remaining bangs. As described in section 3.10, AUTOBAHN runs GHC's demand analysis on the original program to find out which locations are safe to have bangs. Figure 10 shows the ratio of bangs marked safe over all bangs AUTOBAHN produced after phase 2. The unmarked bangs are the ones Pat and Chris need to reason about. This figure also explains AUTOBAHN's advantage over the compiler optimization. Since a good portion of the bangs AUTOBAHN added cannot be proven safe in the compiler, the compiler must never force the annotated binder. The same bangs must also be responsible for the performance improve-

ment in our experiments, since the baseline of the measurements are programs already optimized by GHC.

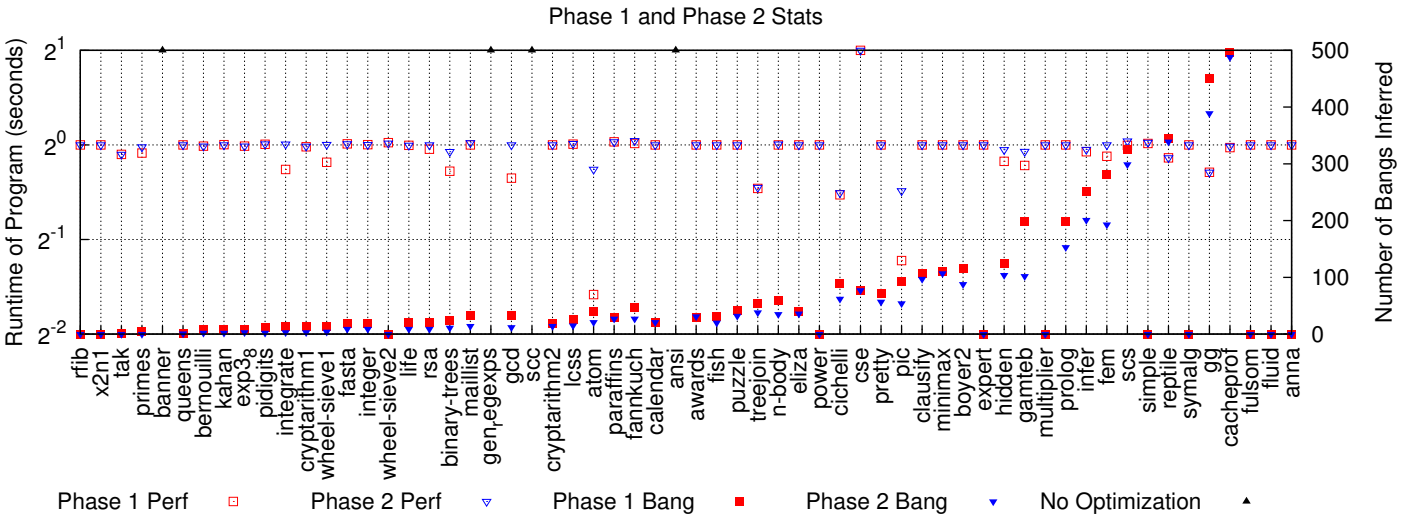


Figure 9: Minimizer performance and bang reduction

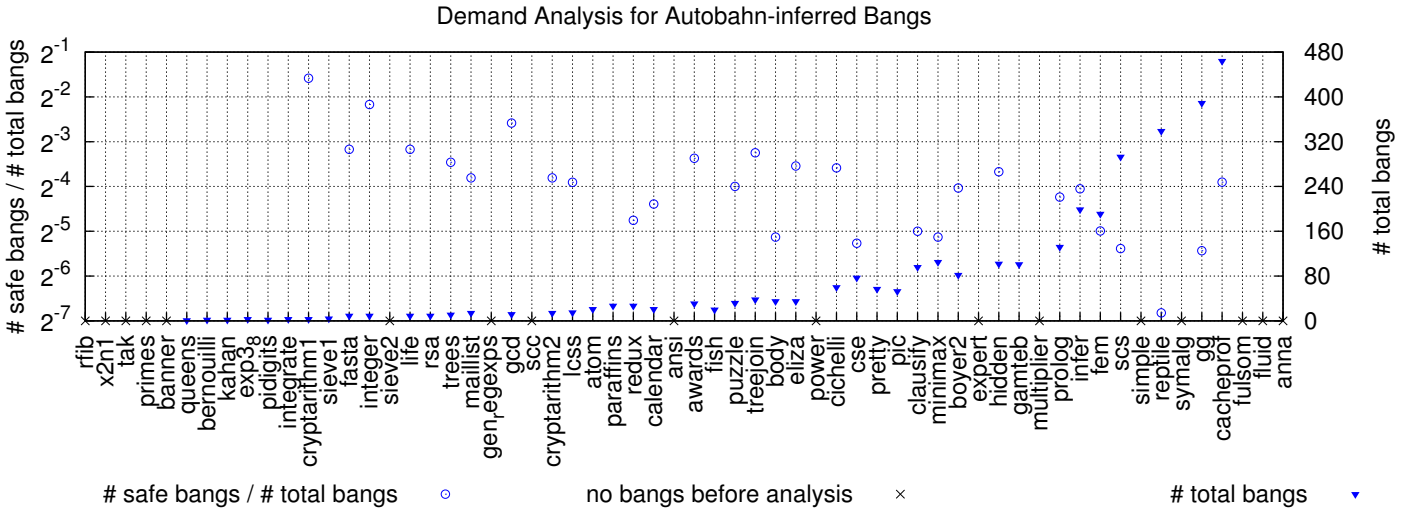


Figure 10: Ratio of bangs marked safe / total bangs in program

5 RELATED WORK / FUTURE WORK

5.1 STATIC ANALYSIS

Identifying opportunities to remove laziness has long been a key optimization in compilers for lazy functional languages (Peyton Jones and Partain, 1994; Peyton Jones and Santos, 1998; Ennals and Peyton Jones, 2003). Compilers traditionally use various forms of strictness analysis (Mycroft, 1980) to identify program fragments that can be evaluated eagerly. Many of the strictness analyses in the literature are based on applying forward abstract interpretation (Wadler, 1987; Partain, 1993; Schrijvers and Mycroft, 2010) to richer and richer languages. Other approaches are based on various flavors of resource-aware type systems (Turner, Wadler, and Mossin, 1995; Holdermans and Hage, 2010; Verstoep and Hage, 2015). The current strictness analyzer in GHC uses backward abstract interpretation, sacrificing some accuracy for compilation speed (Peyton Jones, Sestoft, and Hughes, 2006; Sergey, Vytiniotis, and Peyton Jones, 2014). Because such analyses are static, they are necessarily approximate. Since they are part of the compiler, they are necessarily conservative, identifying binding locations as strict only if they can guarantee that eagerly evaluating the corresponding expression can never cause non- termination. AUTOBAHN implements a dynamic analysis, so it does not have to approximate. It is not part of the compiler, so it does not have to guarantee termination on all inputs. Instead, it allows programmers to decide whether a given annotation set has the necessary termination behavior on an application-specific basis. AUTOBAHN does require programmers to reason about the soundness of the inferred annotation sets, which may not be easy.

5.2 INCLUDING DYNAMIC INFORMATION

There is extensive literature on using dynamic information to improve compiler performance. In the following, I focus only on work that has used dynamic information to improve the performance of Haskell programs by changing when expressions are evaluated. Ennals and Peyton Jones (Ennals and Peyton Jones, 2003) extended GHC's strictness analysis to incorporate dynamic information, exploiting the observation that most thunks are either always evaluated or are cheap to evaluate. They speculatively evaluated thunks, aborting if the evaluation took too long. This approach produced significant speedups (5-25%) on programs from the NoFib benchmarks over purely static

approaches. In contrast to AUTOBAHN, the profiling overheads are necessarily part of the execution time of the user program. In practice, the complexity of the analysis outweighed its performance benefits, and it never became part of the official GHC release.

Recent work (Trilla and Runciman, 2015) explores using runtime profiling in conjunction with static analysis to enable embedded `seq` calls, dynamically adjusting the amount of parallelism in the program. As in Ennals' work, this approach instruments the user's code, so there is a runtime overhead that cannot be avoided. Earlier work (Harris and Singh, 2007) explored using dynamic profiling to identify program points that could be profitably executed in parallel, essentially finding places to insert `par`. Unlike AUTOBAHN, the focus of this work is on adding parallelism, rather than improving performance by reducing laziness.

The Seqaid (Seniuk, 2015) project on hackage seems closely related to AUTOBAHN. It is a Haskell compiler plug-in that uses dynamic profiling to selectively force thunks via `deepseq-bounded`. As with AUTOBAHN, Seqaid is not guaranteed to be sound. Comments on the project webpage indicate the optimizer is under development. I have not been able to compile the code and are not aware of any paper describing the algorithms or reporting on its performance.

5.3 MULTI-OBJECTIVE OPTIMIZATION FOR PROGRAM SYNTHESIS

I made great effort to reduce the number of bangs without affecting performance. In essence, I face a two-objective optimization problem where the relation between the objectives (performance and readability) is not clear. I am not alone.

One recent work(Darulova et al., 2013) synthesizing fixed-point programs uses genetic algorithms to search for a program with the least error. However a more accurate program sometime take longer to run. Therefore, performance constitutes the second objective. Without a clear way to combine the objectives, the authors directs the genetic algorithm to optimize over accuracy and selects the fastest among the most accurate candidates.

In yet another domain, researchers use machine learning algorithms to look for minimal binary abstractions for static analysis (Liang, Tripp, and Naik, 2011). A binary abstraction is a bit vector that represents a proof. Only certain vectors represent valid proofs, and the researchers wish to find the vector with the fewest bit on. They construct efficient algorithms and guarantee to find the correct minimal abstraction. Their success is based on two properties of the correlation between the binary abstraction and its validity - bivalent and monotonicity. That is, an abstraction can only be either valid or invalid, and if an abstraction *is* valid, then it stays valid with any more bits on.

It would be interesting to explore extending the abstraction algorithms into continuous, non-monotonic functions like that mapping bang vectors to performance. Furthermore, a more general algorithm that can be parameterized on the objectives to be optimized can greatly benefit work in program synthesis.

5.4 OTHER APPROACHES

The recent Strict Haskell (*Strict Haskell* 2016) effort avoids the laziness problem by allowing programmers to make specific modules strict-by-default rather than lazy-by-default by using the `-XStrict` and `-XStrictData` language pragmas. This approach is complementary to AUTOBAHN's, offering performance benefits at the cost of eliminating laziness. Adding such pragmas to existing code can be problematic, triggering non-termination or reducing performance. Of course, laziness can be recovered by inserting explicit delays, which is another known hard problem (Chang and Felleisen, 2014).

Chang and Felleisen's recent work (Chang and Felleisen, 2014) is the complement of AUTOBAHN. It uses dynamic profiling to compute a laziness potential that guides the insertion of laziness annotations into programs written in a strict language. It would be interesting to see whether their approach could be adapted to inferring performance-enhancing strictness annotations for Haskell programs. As with AUTOBAHN, this adapted approach would face the soundness problem, which arises from trying to eliminate rather than introduce laziness. Another possibility would be to use laziness potential to add laziness to Strict Haskell programs.

6 CONCLUSION

Excessive laziness has been a performance problem for lazy functional languages since their inception. Despite decades of work on optimizing compilers for lazy languages and associated strictness analyses, poor performance remains a problem. Strictness annotations allow programmers to control the laziness of their programs, but they require high levels of expertise to use effectively and correctly. I have designed and built Autobahn, a tool that uses a genetic algorithm to automatically infer annotations that optimize program performance. Users inspect the suggested annotation sets for soundness and can ask Autobahn to automatically patch their program sources. Experiments show that Autobahn improves runtime performance on NoFib benchmark programs an average of 8.5% and up to 89%, with an average of 24 annotations per 100 LOC. In no case does Autobahn degrade performance. With a second pass of genetic algorithms, Autobahn can reduce the number of bangs to 11 per 100 LOC while still retaining at least 85% of the performance improvement from the first pass. It also marks an average of 7% of the bangs to be safe with GHC's demand analysis.

BIBLIOGRAPHY

Bang Patterns (2016). https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bang-patterns.html (cit. on pp. 3, 20).

Blackburn, Stephen M. et al. (2006). "The DaCapo Benchmarks: Java Benchmarking Development and Analysis." In: OOPSLA '06. Portland, Oregon, USA: ACM (cit. on p. 27).

Broberg, Niklas (2015). *The haskell-src-exts package*. https://hackage.haskell.org/package/haskell-src-exts-1.17.1 (cit. on pp. 11, 12, 23).

Bryan O'Sullivan (2016). *The Aeson Package*. https://hackage.haskell.org/package/aeson (cit. on pp. 6, 12, 21).

Chang, Stephen and Matthias Felleisen (2014). "Profiling for Laziness." In: POPL '14 (cit. on pp. 26, 39).

Cheung, Alvin, Samuel Madden, and Armando Solar-Lezama (2016). "Sloth: Being Lazy Is a Virtue (When Issuing Database Queries)." In: *ACM Trans. Database Syst.* 41.2, 8:1–8:42. ISSN: 0362-5915. DOI: 10.1145/2894749. URL: http://doi.acm.org/10.1145/2894749 (cit. on p. 3).

City of Chicago Public Datasets (2012). https://www.opensciencedatacloud.org/publicdata/city-of-chicago-public-datasets/ (cit. on p. 28).

Darulova, Eva, Viktor Kuncak, Rupak Majumdar, and Indranil Saha (2013). "Synthesis of Fixed-point Programs." In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. EMSOFT '13. Montreal, Quebec, Canada: IEEE Press, 22:1–22:10. ISBN: 978-1-4799-1443-2. URL: http://dl.acm.org/citation.cfm?id=2555754.2555776 (cit. on p. 38).

Deepseq (2015). https://hackage.haskell.org/package/deepseq (cit. on p. 20).

Edward Z. Yang (2011). *Anatomy of a thunk leak*. http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/ (cit. on p. 22).

Ennals, Robert and Simon Peyton Jones (2003). "Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-strict Programs." In: ICFP '03. Uppsala, Sweden (cit. on pp. 3, 37).

Fleming, Philip J. and John J. Wallace (1986). "How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results." In: *Commun. ACM* 29.3, pp. 218–221. DOI: 10.1145/5666.5673. URL: http://doi.acm.org/10.1145/5666.5673 (cit. on p. 23).

GHC Profling (2016). https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html (cit. on p. 12).

Goldberg, David E et al. (1989). *Genetic algorithms in search optimization and machine learning*. Vol. 412. Addison-Wesley Reading (cit. on p. 7).

Harris, Tim and Satnam Singh (2007). "Feedback Directed Implicit Parallelism." In: ICFP '07. Freiburg, Germany (cit. on p. 38).

Holdermans, Stefan and Jurriaan Hage (2010). "Making "Stricterness" More Relevant." In: PEPM '10. Madrid, Spain (cit. on p. 37).

Hoste, Kenneth (2011). *The GA Package*. https://hackage.haskell.org/package/GA (cit. on p. 15).

Liang, Percy, Omer Tripp, and Mayur Naik (2011). "Learning Minimal Abstractions." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, pp. 31–42. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926391. URL: http://doi.acm.org/10.1145/1926385.1926391 (cit. on p. 38).

Mangal, Ravi, Xin Zhang, Aditya V. Nori, and Mayur Naik (2015). "Volt: A Lazy Grounding Framework for Solving Very Large MaxSAT Instances." In: *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing*. Vol. 9340. Lecture Notes in Computer Science. Springer International Publishing, pp. 299–306. DOI: 10.1007/978-3-319-24318-4_22 (cit. on p. 3).

Mitchell, Neil (2013). "Leaking Space." In: *Queue* 11.9 (cit. on p. 3).

Mycroft, Alan (1980). "The Theory and Practice of Transforming Call-by-need into Call-by-value." In: *Proceedings of the Fourth 'Colloque International Sur La Programmation' on International Symposium on Programming*. Springer-Verlag (cit. on p. 37).

O'Sullivan, Bryan, Don Stewart, and John Goerzen (2009). *Real World Haskell*. Available at http://book.realworldhaskell.org. O'Reilly Media (cit. on p. 3).

Ondra (2011). *Thunk memory leak as a result of map function*. http://stackoverflow.com/a/6631097/3694032 (cit. on p. 1).

Partain, Will (1993). "The `nofib` Benchmark Suite of Haskell Programs." In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer-Verlag (cit. on pp. 6, 13, 21, 37).

Peyton Jones, Simon and Will Partain (1994). "Measuring the effectiveness of a simple strictness analyser." In: *Functional Programming, Glasgow 1993*. Springer (cit. on pp. 3, 37).

Peyton Jones, Simon and André L. M. Santos (1998). "A Transformation-based Optimiser for Haskell." In: *Sci. Comput. Program.* 32.1-3 (cit. on pp. 3, 37).

Peyton Jones, Simon, Peter Sestoft, and John Hughes (2006). "Demand analysis." Available from http://research.microsoft.com/en-us/um/people/simonpj/papers/demand-anal/demand.ps (cit. on p. 37).

Ramsey, Norman (2010). *How do I write a constant-space length function in Haskell?* http://stackoverflow.com/a/2777886/3694032 (cit. on p. 4).

Ricci, Nathan P., Samuel Z. Guyer, and J. Eliot B. Moss (2013). "Elephant Tracks: Portable Production of Complete and Precise GC Traces." In: ISMM '13. Seattle, Washington, USA (cit. on pp. 6, 26).

Schrijvers, Tom and Alan Mycroft (2010). "Strictness Meets Data Flow." In: *SAS'10* (cit. on p. 37).

Seniuk, Andrew (2015). *Seqaid*. http://hackage.haskell.org/package/seqaid (cit. on p. 38).

Seq (2016). https://wiki.haskell.org/Seq (cit. on p. 20).

Sergey, Ilya, Dimitrios Vytiniotis, and Simon Peyton Jones (2014). "Modular, Higher-order Cardinality Analysis in Theory and Practice." In: POPL '14. San Diego, California, USA (cit. on pp. 3, 17, 37).

Shan, Chung-chieh and Norman Ramsey (2017). "Exact Bayesian Inference by Symbolic Disintegration." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, pp. 130–144. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009852. URL: http://doi.acm.org/10.1145/3009837.3009852 (cit. on p. 3).

Strict Application (2016). https://wiki.haskell.org/Performance/Strictness (cit. on p. 20).

Strict Fields (2016). https://wiki.haskell.org/Performance/Data_types (cit. on pp. 20, 26).

Strict Haskell (2016). https://ghc.haskell.org/trac/ghc/wiki/StrictPragma (cit. on pp. 21, 23, 39).

Syswerda, Gilbert (1989). "Uniform Crossover in Genetic Algorithms." In: *Proceedings of the 3rd International Conference on Genetic Algorithms* (cit. on p. 16).

Trilla, José Manuel Calderón and Colin Runciman (2015). "Improving Implicit Parallelism." In: Haskell '15. Vancouver, BC, Canada (cit. on p. 38).

Turner, David N., Philip Wadler, and Christian Mossin (1995). "Once Upon a Type." In: FPCA '95. La Jolla, California, USA (cit. on p. 37).

Verstoep, Hidde and Jurriaan Hage (2015). "Polyvariant Cardinality Analysis for Non-strict Higher-order Functional Languages: Brief Announcement." In: PEPM '15. Mumbai, India (cit. on p. 37).

Wadler, Philip (1987). "Strictness analysis on non-flat domains." In: *Abstract interpretation of declarative languages* (cit. on p. 37).

Wang, Yisu Remy, Diogenes Nunez, and Kathleen Fisher (2016). "Autobahn: Using Genetic Algorithms to Infer Strictness Annotations." In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. Nara, Japan: ACM, pp. 114–126. ISBN: 978-1-4503-4434-0.

DOI: 10.1145/2976002.2976009. URL: http://doi.acm.org/10.1145/2976002.2976009 (cit. on p. v).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/