AUTOBAHN: Using Genetic Algorithms to Infer Strictness Annotations

Yisu Remy Wang
Tufts University, USA
remywang@protonmail.com

Diogenes Nunez
Tufts University, USA
dan@cs.tufts.edu

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

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 paper, we introduce AUTOBAHN, a tool that uses genetic algorithms to automatically infer strictness annotations that improve program performance on representative inputs. Users examine the suggested annotations for soundness and can instruct AUTOBAHN to automatically produce modified sources. 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% 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.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program editors; D.3.4 [*Programming Langauges*]: Processors—Code generation, optimization, profiling

Keywords Haskell, laziness, strictness annotations, genetic algorithms

1. Introduction

Meet Pat and Chris. They are intermediate Haskell programmers who coded their most recent project in Haskell. They are excited about how easy it was to write and confident that their code is correct, but they are dismayed to learn that the program 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. A little web research suggests the likely culprit (too much laziness) and a possible solution (adding strictness annotations). Unfortunately, figuring out where in a program to add strictness annotations is a black art, well understood only by expert Haskell programmers. For a while they 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 because the necessary annotations for the library code *depend upon how the library functions are used,* something the library writer cannot know in advance.

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 bad performance. In this paper, we describe AUTOBAHN[1], a tool we have built to help Haskell programmers like Pat and Chris by dynamically *inferring* strictness annotations that improve program performance. Pat and Chris still need to examine the suggested annotations and decide whether to apply them, but they don't have to produce the annotations themselves.

Stepping back, lazy functional programming languages such as Haskell offer the promise of only evaluating the expressions needed to compute the answer. Laziness makes for beautiful programs because it supports modularity [15]. It enables useful programming idioms and it lets users define first-class control constructs. Laziness is implemented using *thunks*. When a function is called, the system passes a heap-allocated thunk storing the unevaluated argument. If in the execution of the function it is determined that the value of the argument is actually needed, the thunk is *forced*, which causes the argument to be evaluated to weak head normal form. The thunk is then overwritten with the resulting value so future references don't need to re-evaluate it [23].

Lazy evaluation does not always improve performance. Allocating large numbers of thunks that eventually need to be forced can cause significant performance slow-downs in both time and space [9, 22, 24]. The `ghc` Haskell compiler uses a backward static analysis [31] to find program points where thunk creation can be avoided. Although this analysis provides consistent performance improvements, programs can still be too slow because the compiler must be conservative in its optimizations to preserve termination behavior on all possible inputs.

To address this deficiency, Haskell provides strictness annotations such as *bang patterns* [2] that allow programmers to instruct the compiler not to create a thunk but rather to evaluate the corresponding expression immediately.[2] Judicious use of strictness annotations can improve program performance in terms of speed and memory usage by significant amounts [20, 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 [17]

Haskell'16, September 22-23, 2016, Nara, Japan

http://dx.doi.org/10.1145/2976002.2976009

[1] Available at `https://genetic-strictness.github.io/Autobahn/`

[2] Available via the `-XBangPatterns` language pragma.

> 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.

A Stack Overflow post asking for help identifying a memory leak in a small program [19] illustrates the challenges. The program manipulates a list stored in an `MVar`, repeatedly reading it, transforming it, and storing it back into the `MVar`. A simplified version of the program (with the necessary annotation to eliminate the leak) is:

```
upgraderThread :: MVar [Int] -> Int -> IO ()
upgraderThread chanMVar 0 = do
  ns <- readMVar chanMVar
  print ns
upgraderThread chanMVar n = do job
  where
    job = do
      vlist <- takeMVar chanMVar
      let !reslist = transform vlist
      putMVar chanMVar reslist
      upgraderThread chanMVar (n - 1)
```

The function `transform` fully evaluates `vlist` in the process of transforming it. The original program suffered from a space leak caused by lazily evaluating the result of the transformation. One might imagine that annotating `chanMVar` with a bang in the case where `n` is not zero would fix the leak because `chanMVar` accumulates thunks at every recursive call. However, the "usual cure" [26] of annotating accumulating parameters does not work here because `chanMVar` is only reduced to weak head normal form (its outermost constructor), not fully evaluated. To completely eliminate the space leak, we need to instead add a bang before `reslist` (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.

In this paper, we explore using genetic algorithms to *automatically* infer strictness annotations, specifically bang patterns. In our approach, Pat and Chris write their Haskell program without worrying about strictness annotations. Once they are happy with the correctness of their code, they run AUTOBAHN, supplying the program and representative data. AUTOBAHN uses a genetic algorithm to search through the space of all possible bang patterns to find candidate annotations that reduce the value of a *fitness function* selected to improve program performance. AUTOBAHN can start with a program that already contains bang patterns or one that does not. It has the power to both add and remove annotations. AUTOBAHN returns a list of annotation sets, ranked by a measurement of how much each annotation set improved performance. Pat and Chris examine the proposed alternatives for soundness on relevant program inputs and decide whether to have AUTOBAHN produce modified sources corresponding to one of the generated annotation sets.

The 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, AUTOBAHN is guaranteed to only suggest alternative annotations that actually improve 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 to fail to terminate when given new input. For this reason, AUTOBAHN supplies a list of alternatives and asks the user to choose from among 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. We leave to future work the challenge of helping Pat and Chris reason about the soundness of the inferred annotations for their inputs of interest. The ability to make potentially unsound annotations is one reason why strictness annotations can lead to better performance than `ghc`'s optimizer; the compiler must be sound.

AUTOBAHN users can decide how much of the program's source AUTOBAHN 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.

The contributions of this paper are the following:

- We 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 different performance criteria: total runtime, garbage collection time, and live size (aka, peak allocation).
- We demonstrate the effectiveness of this approach on 60 programs from the NoFib [21] 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.
- We use AUTOBAHN in a case study to optimize the performance of a garbage collector simulator `gcSimulator` [27]. The annotations inferred on a small training set result in performance improvements on larger data sets: 23.6% decrease in running time and a reduction in live size to under 1% of the unoptimized program on the full dataset.
- We show in a second case study that AUTOBAHN can infer application-specific annotations for Aeson [5] library code to optimize driver programs `validate` and `convert` that require different annotations to produce optimal behavior.
- We conducted 10-fold cross-validation studies for `gcSimulator` and `convert`, showing that the inferred annotations are stable across different data sets. For `gcSimulator`, the study also shows that the inferred annotations generally outperform the annotations added by hand by the original author.

2. Genetic Algorithms

Our problem of finding a set of annotations to maximize program performance is a specific instance of a general search problem. We can cast the general problem as follows. Consider a function $\mathcal{F}$ that takes an argument x and returns some value $\mathcal{F}(x)$. We wish to find an argument that maximizes the value $\mathcal{F}(x)$. If the number of possible values for x is large, we cannot use exhaustive algorithms to search for x. Instead, we must turn to heuristic searches.

A *genetic algorithm* [11] uses ideas from natural selection to guide a heuristic search for a value of x that maximizes function $\mathcal{F}$. Each possible value of x is encoded as a sequence of *genes* that collectively form a *chromosome*. Function $\mathcal{F}$ 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 round 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 $\mathcal{F}$. It forms the population for the next round

by promoting the fittest individuals of the current population and adding the *offspring* of the current generation. The algorithm computes the offspring by randomly changing the genes in some members of the current population (*mutation*) and by splicing together the chromosomes of others (*crossover*). The result of the search is the "fittest" member of the population in the final generation.

Figure 1 shows pseudo-code for the genetic algorithm we use, while Table 1 lists the various parameters with which it can be configured. We 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 (numMutants). 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 (numChildren). 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.

3. AUTOBAHN

3.1 Genes and Chromosomes

What is AUTOBAHN's notion of a gene? Conceptually, a gene is a program source location where we may insert a bang pattern. A gene is *on* (represented by the bit `1`) if the corresponding source location has a bang; it is *off* (bit `0`) otherwise. Although syntactically legal, it is not sensible to put multiple bangs on the same pattern, so we disallow this possibility. For a given program `p`, we construct the related program `p'` that is just like `p` except `p'` has no bang annotations. We call `p'` the *bare* version of `p`. A gene for `p` is any program location in `p'` where a bang pattern is legal. We use `haskell-src-exts` [4] to identify the appropriate locations.

Because we disallow multiple bangs on the same pattern, any Haskell program has a fixed number of candidate bang pattern annotations and so a fixed number of genes. Consequently, we use a fixed-length bit vector to encode the space of all possible annotations. A chromosome is a particular value for the bit vector. For example, consider the following code:

```
module ABitLazy where
  foo !x ~y !z = x + z -- '~' denotes absent bang
```

This program has three genes, one for each parameter. The chromosome for the current annotations is the bit vector `'101'`, indicating the expressions bound to `x` and `z` should be evaluated eagerly, but `y` should be evaluated lazily. In some cases, adding or removing a bang at a program position does not have any effect. For example, a bang outside a tuple is superfluous because pattern matching the tuple forces its evaluation. We are exploring how to identify these program positions and remove them from the chromosome.

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, we have restricted this flexibility to the level of individual source files. AUTOBAHN users 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, AUTOBAHN users 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 [5] 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.

3.2 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, we produce the program to profile by parsing the sources using the `haskell-src-exts` library [4], 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 that users can choose between. Each of these functions works by parsing the output produced by `ghc` when invoked with the `+RTS -t` command-line option [10]. The first fitness 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. To implement this intuition, we timeout program runs that take longer than twice the running time of the original program. We allow programs that are slightly slower than the original because sometimes such programs lead to overall improvements when additional annotations are added in future generations. We assign very low scores to programs that trigger the timeout and to programs that terminate by throwing an exception, ensuring they die out.

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

```
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 *fitness* function starting from initial chromosome *seed*.

We also assign 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 [21] 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), AUTOBAHN asks the user to supply a smaller set of representative training data.

3.3 Algorithm Parameters

As discussed in Section 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 users 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.

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

We use the total time that the users 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, we find the "golden ratio" of the two parameters based on their effect on the possibility of discovering better annotation sets. In our experience, a $4/3$ ratio of *numGenerations*/*populationSize* works well as default. This ratio is somewhat unconventional for genetic algorithms, where the value of *numGenerations* is usually on the order of twenty times larger than *populationSize* [16]. We empirically adjusted these parameters to guarantee an affordable running time and a reasonable *populationSize*.

We use simple default values for four parameters: *archiveSize* (`7`), *mutateRate* (`0.2`), *mutateProb* (`0.2`), and *crossRate* (`0.8`). We chose these values because we found they worked well in practice.

In addition to the generic genetic algorithm parameters described in the previous section, we have an additional parameter *numFitnessRuns* that arises because our fitness function runs the program to measure its performance. To ensure the profiling information is accurate, we run each program on the training data *numFitnessRuns* times. We calculate an appropriate value for this parameter by iteratively profiling the unannotated program until the

```
{ 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

mean of the measured performance changes by less than 5%. We use that number of iterations as the value for *numFitnessRuns*. The record in Figure 2 shows a sample inferred configuration. We allow users to override default values if they wish.

3.4 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 *seed* and flips the value of that gene with probability *diversityRate*. Note that this process can both add and remove bang patterns.

3.5 Producing New Generations

We use the Haskell genetic algorithm library GA [14] to produce each successive generation, passing it our `mutation` and `crossover` functions, which we explain in turn.

Mutation. For each generation, the GA library calls our function `mutation` *mutateRate* $*$ *populationSize* times, each time passing in a randomly chosen chromosome `c` from the current population. Intuitively, our 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; we set this value according to the *mutateProb* parameter. Next, we use a `seed` parameter to generate randomness. Finally, we have 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, we compute and return the new chromosome `c'` by `xor`-ing `c` with the bit sequence.

Crossover. For each generation, the GA library calls our `crossover` function *crossRate* $*$ *populationSize* times, each time passing in a pair of randomly chosen chromosomes `c1` and `c2` from the current population. We chose to have our crossover function implement the Uniform Distribution [35] 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 the other parent. This strategy makes stronger genes more likely to survive: when a newly added annotation improves performance, its improvement is likely to persist regardless of other annotations. The function makes use of three parameters. As with mutation, we use a `seed` parameter to generate randomness. Next, we use two parameters, `c1` and `c2`, as the chromosomes that have been selected for crossover. Intuitively, our 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 we select the genes from one parent (`c1'`) using a bitwise-and operation (`.&.`). We use the off-bits to select the remaining genes from the other parent (`c2'`). We then use a bitwise-or operation (`.|.`) to generate a new chromosome with roughly half of its genes from each parent.

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

Figure 3. Inputs a user supplies to AUTOBAHN

3.6 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 users 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.

3.7 Putting it All Together

To summarize, we discuss how people like Pat and Chris use AUTOBAHN. First, they provide the inputs specified in Figure 3. 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 the users. 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 the users, AUTOBAHN will produce a new version of the input sources that match any of the chromosomes that survived to the final round.

3.8 Discussion

Haskell has four major kinds of strictness annotations: `seq` [30], `deepseq` [7], and related functions; strict application (`$!`) [32]; strict data type declarations [33]; and bang patterns [2]. Currently, AUTOBAHN searches only for bang patterns. We chose not to search for places to insert `seq` or related functions because the search space is too large. We are currently exploring using AUTOBAHN to insert strict applications and strict datatype annotations.

4. 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 force the evaluation of `length [1..]`, causing the program to run forever.

The `ghc` compiler uses a conservative strictness analyzer [31] to ensure that it won't introduce non-termination when it decides to eagerly evaluate an expression. As with all static analysis, this

analyzer is necessarily conservative 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 causes all control flow paths to be executed. 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 AUTOBAHN users to verify that the suggested annotations lead to appropriate termination behavior. For the same reason, it is important that AUTOBAHN users supply representative training data. If negative values for u are legal inputs, then the training data should include examples of this form. Note that users might pick an inferred annotation set *even if it introduces the possibility of non-termination*. In the example above, the users might know that the system will never in fact pass in a negative number and so introducing the bang pattern is fine.

We note that 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, we are currently developing an additional tool to explore the termination behavior of annotation sets.

5. Evaluation

We evaluate AUTOBAHN in a number of ways:

1. By running it on four small programs for which we can perform an exhaustive search, showing that AUTOBAHN computes the optimal annotation set.
2. By running it on 60 programs taken from the NoFib [21] 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 [34] 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 [5] 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. We compiled AUTOBAHN itself with ghc version 7.8.4 with the -O2 flag. We compiled the benchmarks with ghc version 7.10.3 with -O2 and -XBangPatterns along with NoFib's default flags. We 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, we forced ghc to perform frequent garbage collections via the flags +RTS -h -i0.01. For the Strict Haskell comparison, we used ghc version 8.0.1 because the relevant pragmas were not present in 7.10.3. We ran each benchmark 4 times and report our results using NoFib's geometric-mean reporting convention for uniformity with other studies.

5.1 Small Programs: A Sanity Check

First, we ran AUTOBAHN on a set of four small programs for which we were 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, we briefly describe the programs and their strictness properties.

The fib function below uses accumulating parameters (a, b1, and b2) to calculate the nth 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 [8] 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 the upgraderThread example discussed in Section 1. 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
```

5.2 NoFib Benchmarks

We ran AUTOBAHN on 60 benchmarks from the NoFib benchmark suite. We 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 [4] version `1.17.1`. The smallest of these benchmark programs (`rfib`) has 5 genes, while the largest (`anna`) has 7709. Table 2 lists the programs, giving the number of lines of code, the number of program files, and the number of genes in each.

Figure 4 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 one thus represent improvement. Because AUTOBAHN returns the program unchanged if it cannot find an improvement, we 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 NoFib conventions, we 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, we selected the twelve NoFib programs whose run times decreased the most and compared the heap profiles of the original and AUTOBAHN-annotated versions[3]. 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. We believe this reduction in memory usage translated to the runtime improvements.

5.3 Strict Haskell

Strict Haskell [34] 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 language extensions: `-XStrictData` and `-XStrict`. According to the `ghc` wiki page, 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, we compiled and ran the NoFib programs with `ghc` 8.0.1 using `-O2`, `-XBangPatterns`, `-funbox-strict-fields`, `-XStrict`, and `-XStrictData` flags. Figure 5 shows the results. Seventeen of the NoFib programs failed when using Strict Haskell. These programs failed for one of the following reasons:

[3] Available at `https://genetic-strictness.github.io/Autobahn/profiles`

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

Table 2. Statistics for the NoFib benchmarks

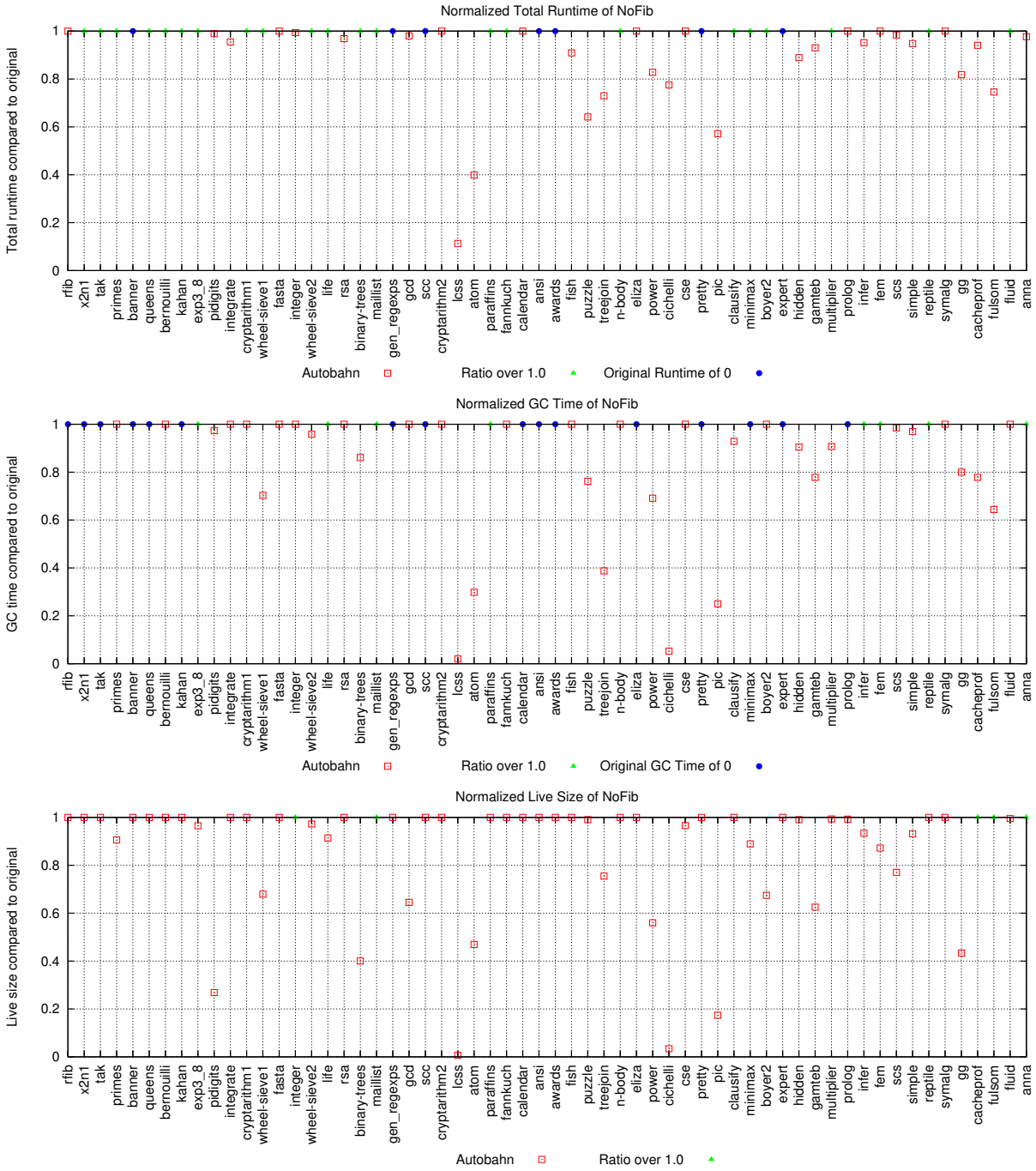

Figure 4. Performance of AUTOBAHN-optimized programs normalized by the original program's performance; lower values are better. The data show geometric mean improvements of 8.5%, 18%, and 7.2%, and maximum improvements of 89%, 98%, and 99.3% on the total runtime, garbage collection, and live size performance criteria, respectively.

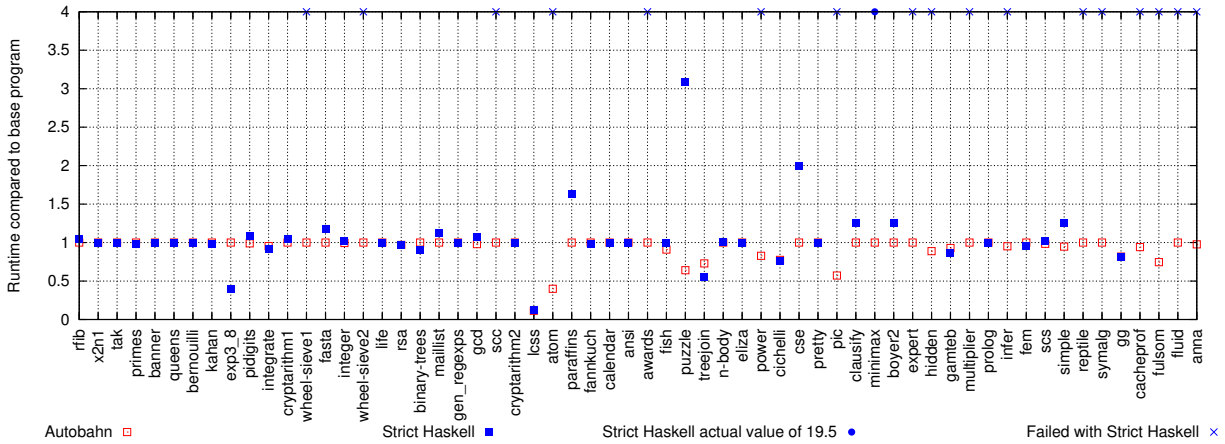

Figure 5. Runtime of AUTOBAHN-optimized programs and programs compiled with Strict Haskell normalized to base programs run in `ghc` 8.0.1. Lower on the y-axis means the AUTOBAHN version of the program ran *faster*.

- 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 aren't 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 [6].

5.4 Case Study: `gcSimulator`

As a case study, we used AUTOBAHN to optimize `gcSimulator`, a garbage collection simulator that uses trace files generated by the Elephant Tracks [27] 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 minutes. This performance makes running AUTOBAHN prohibitively time consuming. Consequently, we 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 [3]. We then evaluated the performance of the AUTOBAHN-optimized version of `gcSimulator` on increasing sizes of the same trace. For this study, we seeded the initial population with the bare program; we did not include the annotations the original author had added by hand. For these measurements, we compiled all versions of the `gcSimulator` with the same settings as NoFib along with `-rtsopts` to gather runtime information.

Input Data	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 3. 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 those for AUTOBAHN.

Table 3 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 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 6 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 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 3 and Figure 6 further show that the annotations inferred for a small training set provide benefits even when the program is given larger inputs.

5.5 Case Study: Aeson Library with Two Different Drivers

As a second case study, we developed two driver programs that both use the Aeson [5] parser library to manipulate a JSON file containing a list of records. The first driver program simply validates that the data file is well formed. Because it does not need

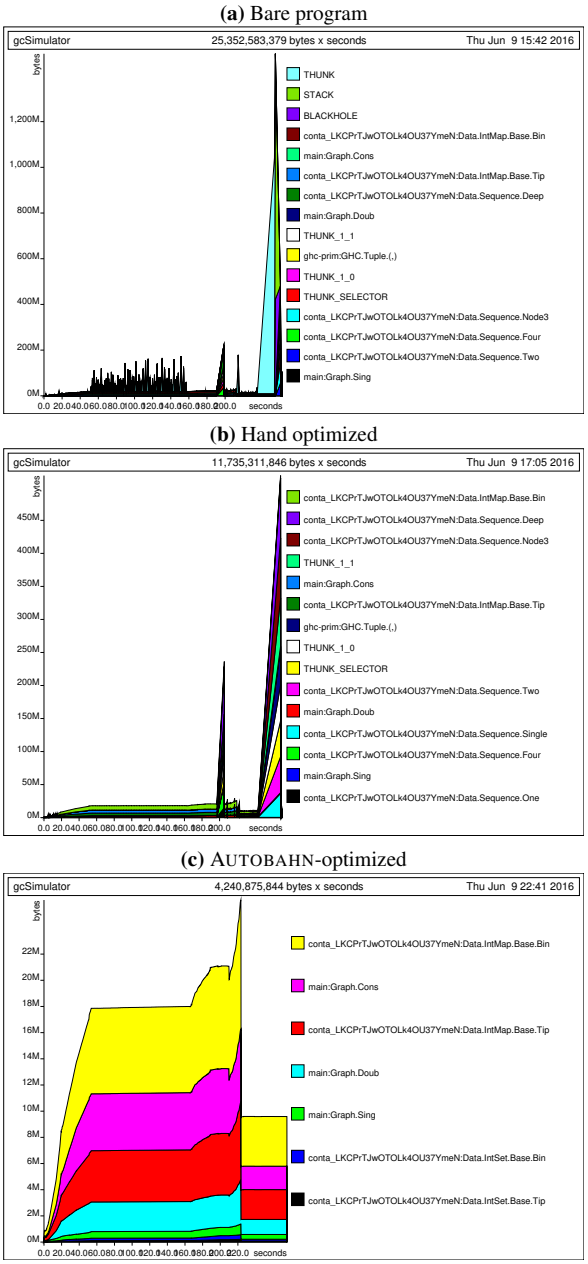

Figure 6. Heap profiles of `gcSimulator` on $\frac{1}{2}$ of the `batik` trace.

to deeply parse the data, using the library lazily provides the best performance. The second driver program converts the entire data file into a corresponding Haskell data structure, and so using the library eagerly provides the best performance. We call these two drivers `validate` and `convert`, respectively. To limit the search space to a reasonable size, we pick a subset of the Aeson library. As a result, each driver/library pair consists of two files totaling 320 lines of code. Each pair has a chromosome composed of 194 genes.

For each driver program, we ask AUTOBAHN to optimize a combination of the driver and the Aeson library to improve runtime performance. To construct a scenario with space for improvement, we have the driver programs use the "wrong" version of the Aeson parser and let AUTOBAHN try to correct the strictness annotations.

Input Data	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 4. 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`.

For data, we use JSON files published by the City of Chicago [1]: a 12MB file `objects.json` for training and four other data files of increasing size, which we call A, B, C, and D, for testing. For these measurements, we compiled the Aeson library and our drivers using the same settings as NoFib except we add `-rtsopts` to gather runtime information and remove `-funbox-strict-fields` since Aeson does not use strict datatypes. We used total runtime as the fitness function.

Table 4 shows that despite training the two drivers on the same data and optimizing both for runtime, AUTOBAHN's optimizations had different results. The `validate` driver ran in 63% of the time of its bare counterpart on data set C. The `convert` driver reached its fastest runtime on set A at 66%, but *also* reduced its peak memory usage to 45% of its bare version. The closest the `validate` driver gets is using 85% of the bare peak memory usage on set B. Looking at the geometric means, the annotations on `convert` lowered its peak memory usage to 71.7% of the bare version, indicating its annotations focused on reducing space usage. The annotations on `validate` do not change the peak memory usage, evidenced by the small reduction to 99.98% of the bare memory usage and 96.8% of the bare GC time.

To realize these performance improvements, AUTOBAHN infers *different* strictness annotations for the Aeson library. For the `convert` driver, AUTOBAHN adds a critical bang into the lazy parser before a pattern that stores the result of inserting values into a hashtable. Since the program eventually needs to fully evaluate the values in the hashtable, that bang helps avoid unnecessary thunks during updates. For the `validate` driver, AUTOBAHN removes the bang in the strict parser that triggers insertion into the hashtable so that the driver program does not waste effort updating a hashtable that is never evaluated. Both bangs appear before the result of a call to `H.insert` in the `objectValues` function.

5.6 10-fold Cross-validation

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

For `gcSimulator`, we chose as input ten different traces of programs from the DaCapo Benchmarks. Because the full traces lead to long training times, we selected the first 35 million lines from each trace. We tested the optimized programs, however, on

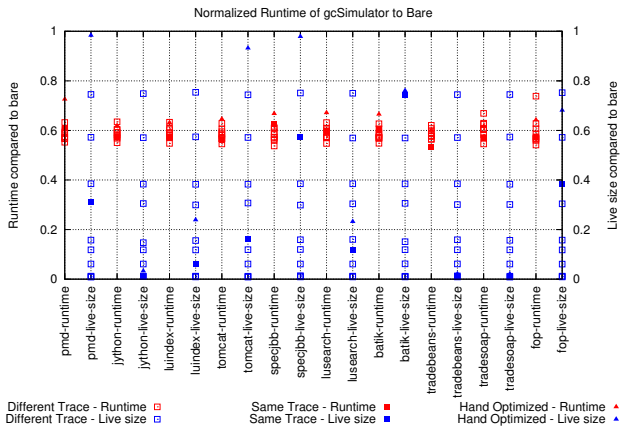

Figure 7. 10-fold evaluation for `gcSimulator`, showing runtime and live size performance improvements of AUTOBAHN versions of `gcSimulator` compared to the bare program. We highlight points where the AUTOBAHN-optimized program ran on its training trace. We also show how the hand-annotated program performed.

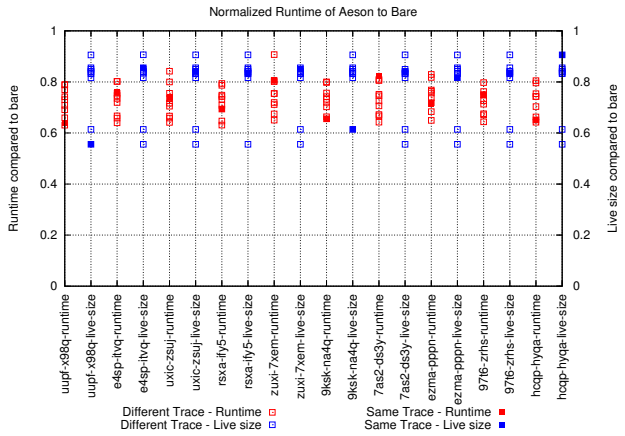

Figure 8. 10-fold evaluation for `convert`, showing runtime and live size performance improvements of AUTOBAHN versions of `convert` compared to the bare program. We highlight points where the AUTOBAHN-optimized program ran on its training trace.

the full traces. For `convert`, we chose ten different JSON data sets from the data made available by the City of Chicago, ranging in size from 32 to 68MB. We trained and tested the optimized programs on the full data files.

Figures 7 and 8 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`, we also show the performance of the hand-optimized version of the program.

Figure 7 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` running times 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 `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 9 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 (geomean). 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. We have yet to run this experiment to capture the runtime when optimizing for live size.

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, we 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, we see these running times as acceptable. We imagine Pat and Chris starting the tool before they go to bed and wake up with candidate annotations to consider. We note that a portion of AUTOBAHN's running time comes from parsing Haskell source files, using `haskell-src-exts` to add bang patterns, pretty printing the resulting program, and then calling `ghc` to compile and run the program. If we used the `ghc` compiler API to programmatically modify and compile various versions of the programs to be optimized, we could reduce AUTOBAHN's overhead at the cost of a tighter coupling to the fast-evolving `ghc` compiler.

6. Related Work

6.1 Static Analysis

Identifying opportunities to remove laziness has long been a key optimization in compilers for lazy functional languages [9, 22, 24]. Compilers traditionally use various forms of *strictness analysis* [18] to identify program fragments that can be evaluated eagerly. Many of the strictness analyses in the literature are based on applying forward abstract interpretation [21, 28, 39] to richer and richer languages. Other approaches are based on various flavors of resource-aware type systems [13, 37, 38]. The current strictness analyzer in `ghc` uses backward abstract interpretation, sacrificing some accuracy for compilation speed [25, 31]. 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 nontermination. AUTOBAHN is a dynamic analysis, so it does not have to approximate. It is not part of the compiler. Therefore, it does not have to guarantee termination on all inputs. Instead, it allows

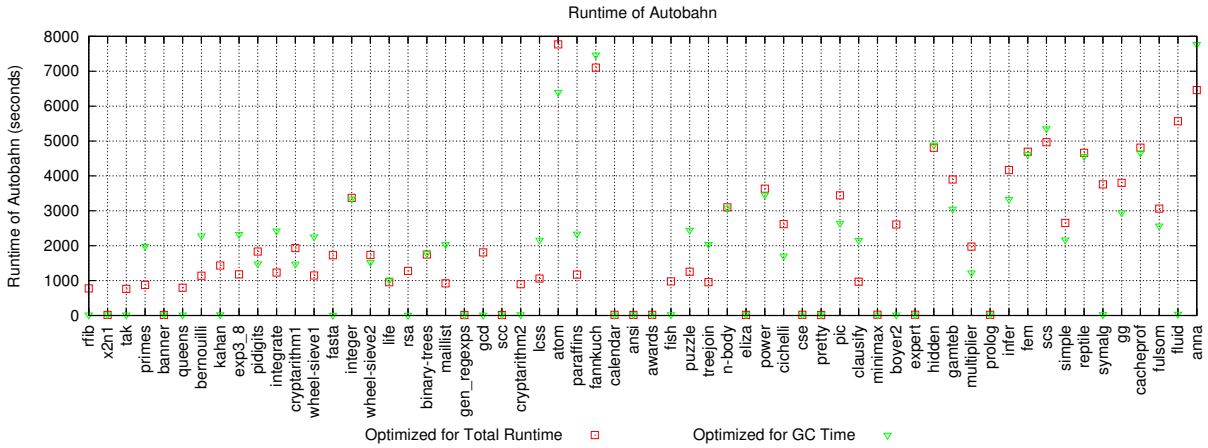

Figure 9. AUTOBAHN running time to optimize each program in the NoFib benchmark suite.

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.

6.2 Including Dynamic Information

There is extensive literature on using dynamic information to improve compiler performance. In the following, we 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 [9] 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 benchmark 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 [36] 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 users' code, so there is a runtime overhead that cannot be avoided. Earlier work [12] 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 [29] 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. We have not been able to compile the code and are not aware of any paper describing the algorithms or reporting on its performance.

6.3 Other Approaches

The recent Strict Haskell [34] 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 [6].

Chang and Felleisen's recent work [6] 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.

7. 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 correctly. AUTOBAHN is a tool we have designed and built 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%. In no case does AUTOBAHN degrade performance.

Acknowledgments

We thank Norman Ramsey for teaching the course that inspired this project. We thank Nathan Ricci for providing the `gcSimulator` use case. We thank John Launchbury for insights into strictness analysis and the `fact` example in Section 4. We thank Stephen Chang, Matthias Felleisen, and Simon Peyton Jones for insightful comments on an earlier draft. We thank Simon Marlow for his guidance on using `ghc` to obtain program statistics. This research was supported in part by DARPA contract FA8750-15-2-0033.

References

[1] City of Chicago Public Datasets. `https://www.opensciencedatacloud.org/publicdata/city-of-chicago-public-datasets/`, 2012.

[2] Bang Patterns. Bang Patterns. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/bang-patterns.html, 2016.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. OOPSLA '06. ACM, 2006.

[4] N. Broberg. The haskell-src-exts package. https://hackage.haskell.org/package/haskell-src-exts-1.17.1, 2015.

[5] Bryan O'Sullivan. The Aeson Package. https://hackage.haskell.org/package/aeson, 2016.

[6] S. Chang and M. Felleisen. Profiling for laziness. POPL '14, 2014.

[7] Deepseq. Deepseq. https://hackage.haskell.org/package/deepseq, 2015.

[8] Edward Z. Yang. Anatomy of a thunk leak. http://blog.ezyang.com/2011/05/anatomy-of-a-thunk-leak/, 2011.

[9] R. Ennals and S. Peyton Jones. Optimistic evaluation: An adaptive evaluation strategy for non-strict programs. ICFP '03, 2003.

[10] GHC Profling. GHC Profling. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html, 2016.

[11] D. E. Goldberg et al. *Genetic algorithms in search optimization and machine learning*, volume 412. Addison-Wesley Reading, 1989.

[12] T. Harris and S. Singh. Feedback directed implicit parallelism. ICFP '07, 2007.

[13] S. Holdermans and J. Hage. Making "stricterness" more relevant. PEPM '10, 2010.

[14] K. Hoste. The GA Package. https://hackage.haskell.org/package/GA, 2011.

[15] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2), Apr. 1989.

[16] K. A. Jong and W. M. Spears. *Parallel Problem Solving from Nature: 1st Workshop, PPSN I Dortmund, FRG, October 1–3, 1990 Proceedings*, chapter An analysis of the interacting roles of population size and crossover in genetic algorithms. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[17] N. Mitchell. Leaking space. *Queue*, 11(9), Sept. 2013.

[18] A. Mycroft. 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, 1980.

[19] Ondra. Thunk memory leak as a result of map function. http://stackoverflow.com/a/6631097/3694032, 2011.

[20] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2009. Available at http://book.realworldhaskell.org.

[21] W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1993.

[22] S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Functional Programming, Glasgow 1993*. Springer, 1994.

[23] S. Peyton Jones and J. Salkild. The spineless tagless G-machine. FPCA '89, 1989.

[24] S. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3), Sept. 1998.

[25] S. Peyton Jones, P. Sestoft, and J. Hughes. Demand analysis. Available from http://research.microsoft.com/en-us/um/people/simonpj/papers/demand-anal/demand.ps, July 2006.

[26] N. Ramsey. How do I write a constant-space length function in Haskell? http://stackoverflow.com/a/2777886/3694032, 2010.

[27] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Portable production of complete and precise GC traces. ISMM '13, 2013.

[28] T. Schrijvers and A. Mycroft. Strictness meets data flow. In *SAS'10*, 2010.

[29] A. Seniuk. Seqaid. http://hackage.haskell.org/package/seqaid, 2015.

[30] Seq. https://wiki.haskell.org/Seq, 2016.

[31] I. Sergey, D. Vytiniotis, and S. Peyton Jones. Modular, higher-order cardinality analysis in theory and practice. POPL '14, 2014.

[32] Strict Application. https://wiki.haskell.org/Performance/Strictness, 2016.

[33] Strict Fields. https://wiki.haskell.org/Performance/Data_types, 2016.

[34] Strict Haskell. https://ghc.haskell.org/trac/ghc/wiki/StrictPragma, 2016.

[35] G. Syswerda. Uniform crossover in genetic algorithms. In *3rd International Conference on Genetic Algorithms*, 1989.

[36] J. M. C. Trilla and C. Runciman. Improving implicit parallelism. Haskell '15, 2015.

[37] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. FPCA '95, 1995.

[38] H. Verstoep and J. Hage. Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. PEPM '15, 2015.

[39] P. Wadler. Strictness analysis on non-flat domains. In *Abstract interpretation of declarative languages*, 1987.