Random Strictness at Source Level

Diogenes Nunez
Tufts University
dan@cs.tufts.edu

Abstract

Haskell is a lazy language which can cause slowdown due to the creation of thunks. Strictness can force evaluation of expressions and avoid the cost of thunks. However, programmers adding it into source code for performance may not know where to start. This case worsens if it is code they did not write. We introduce a genetic algorithm with the purpose of developing a faster program through adding strictness to the source code.

1. Introduction

Haskell is lazy. Consider this function

```
-- Create infinite list
repeat :: a -> [a]

replicate :: Int -> a -> [a]
replicate n x = take n xs
                where xs = repeat x
```

In an eager language, `xs` would be evaluated immediately. This results in a nonterminating function. However, a lazy language would only evaulate `xs` when required. Even then, the expression only gets what it needs. Therefore, only `n` items are required and processed.

How does Haskell do this? To explain, consider this function

```
f :: Int -> [Int]
f n = replicate n n
```

Consider the application `f (x+1)`. Haskell creates a closure on the heap that contains the unevaluated expression `x+1`. This is a thunk. The function `f` then gets a pointer to that. When the value within is needed, Haskell traverses to the thunk and evaluates it, potentially writing something back to the heap. Note that an eager language would not have written the thunk in the first place, avoiding both the additional write and the read.

How well does this scale? Suppose we had a large group of thunks. The larger the group, the more memory pressure applied to the application. This leads to more frequent garbage collections and overall slowdown. In some cases, we may have programs that crash due to a lack of memory. We might save some overall time and space by evaluating some of those thunks early. Haskell has notation to force eager evaluation, the bang.

```
f :: Int -> [Int]
f !n = replicate n n
```

Now, we force Haskell to evaluate `n` and pass that value on to `f`. `n` is evaluated eagerly. Here, no thunk is created and if `n` is no longer used in the program, the garbage collector can throw it away. We have saved some time and space. We say that `f` is now strict in `n` and the process of forcing eager evaluation is adding strictness to the program.

Now consider a far more complicated program, spreading across multiple files. We must ask ourselves two questions. First, where do we start adding strictness? Here we only had one function, so the process was simple. With many functions, some recursive, adding strictness could cause problems, like nontermination. Second, when do we stop? Adding too much strictness can cause excess memory pressure. This would be no different, or worse, than the completely lazy version. We need to find a balance between lazy and eager in order to improve the program.

In this paper, we discuss a program that searches the space of strict programs using a genetic algorithm to find a faster version of a given program. Section 2 covers the background of strictness analysis and related work. Section 3 covers the program. Section 4 discusses some experiments. Section 6 concludes the paper.

2. Background

Strictness analysis looks at a function with n arguments and asks whether the i^{th} argument can be made strict. Programmers and compilers use the results of such an analysis to make their programs strict. The problem itself is hard. Part of strictness analysis is asking whether the program will terminate, also known as the halting problem. Work in the field focuses on approximating the analysis.

Peyton Jones and Partain created a simple strictness analyzer that would run on GHC Core.[1] The analysis was an approximation of finding fixpoints of functions. They used a "widening" operator on functions, removing some constraints and finding those fixpoints. As a static analysis, it will flag some functions as nonterminating when made strict on some arguments. Furthermore, they note their analyzer does not deal with non-flat structures and recursive functions. The work presented here does not ignore these two items. As long as the algorithm knows strictness can be annotated, it has a chance of doing so.

Schrijvers and Mycroft worked on a type inference algorithm that kept track of the effects of functions, both deterministic and non-deterministic [3, 4]. They use this to bring data flow to strictness analysis. Just like Peyton Jones and Partain, they consider flat data. Again, we do not ignore data so long as the algorithm knows it can make the data strict.

Jensen et. al. moved to an analyzer on GHC Core that handled higher-order functions and polymorphism [2]. They also approximate fixpoints. However, they make no note in their paper on using the results on functions. Their work incorporated the theoretic foundations from Wadler[4] to do this. Our work differentiates it-

Final paper for Comp150-05: Advanced Functional Programming

self from both Peyton Jones and Jensen by working closer to sugared source code than GHC Core. This allows us to output code for a Haskell user that they can learn from, rather than giving GHC Core back or just compiling straight to machine code.

3. Architecture

The program has three components, as shown in Figure 3. The Rewrite module deals with adding strictness to the source code itself. The Genetic module is the core of the program, deciding when and where to add the strictness. The rest of this section discusses the modules in more detail.

3.1 Rewrite

This module takes in the sugared Haskell source code, creates an abstract syntax tree, transforms it, and prints it back as sugared Haskell code without comments. We use the `haskell-src-exts` package to get the abstract syntax tree and print it back to readable source [6]. It exposes three functions, `flipBang`, `flipRandomBang`, `placesToStrict`.

`flipBang` takes in a `FilePath` to the file, a `String` containing the whole program, and an `Int` that denotes which declaration in the program to add or remove strictness. If the program was entered with strictness at that location, it will leave without it and vice-versa. `flipRandomBang` does the same, but in at a random location in the program. Finally, `placesToStrict` gets the file path and program as input and returns the number of places one can place strictness.

All three functions perform the same steps and traversal. To begin, they retrieve the module the program given describes. We use `parseFileContentsWithMode` to extract the abstract syntax tree of the module while enabling the compiler flag `-XBangPatterns` and the Haskell 2010 language.

From there, we can get a list of declarations. This includes all functions declared in the file. The package labels them as `FunBind`s. Each one contains a list of parameters, typed `Match`. Each of those contains contains `Pat`, or patterns. `Pat` contains multiple datatypes, but two concern us in this implementation. One is `PVar`, a parameter or variable declaration in a `where` clause. The other is `PBangPat`, something with strictness applied to it. `flipBang` and `flipRandomBang` will either replace a `PVar` with a `PBangPat` or vice-versa. `placesToStrict` increments its counter.

3.2 Genetic Algorithm

This module is the driving force behind the program. Its purpose is to run the genetic algorithm, exploring the space of programs, choosing some, and running them to determine the best. The rest of this subsection describes the module.

3.2.1 Datatypes and Operations

Listing 1 lists the important datatypes and operations used in the module. `Strand` is used to describe a single file. It contains the path to the file, the program in a `String`, a bit vector represented by an `Integer`, and an `Int` recording the size of that bit vector. Each bit in the vector, starting from the least significant, corresponds to a location in the program where the Rewrite module can place strictness. 0 means there is no strictness. 1 means there is. This allows us to keep track of the status of the file without digging into the program itself. For example the vector for the program in Listing 2 is 001. We record the size of the vector since `bitSize` of an `Integer` is undefined in `Data.Bits`.

The datatype `Genes` contains a list of these `Strands` and all our main operations are on these. We start with `mutate`. This function takes in a `Genes`. First, we write the `Genes` to a new unique directory on disk. For each `Strand` from the `Genes`, we generate a random bit vector. Then, we run through that `Strand` and flip the state of strictness throughout the file to match the vector. Finally, we write the changed program to disk in that unique directory.

`merge` takes two `Genes` as parents. We write one of them to disk in a new unique directory. Then we go through each pair of `Strands`, one from each parent, and compute the bitwise-or of their bit vectors. Just like `mutate`, we then change one of the programs to reflect this new vector and write that to disk in the same directory.

```
data Strand = {  path    :: FilePath
               , program :: String
               , vec     :: Integer
               , size    :: Int
              }

data Genes = { getStrands :: [Strand] }

mutate :: Genes -> Genes
merge :: Genes -> Genes -> Genes
fitness :: Int -> Float -> Genes ->
           IO Float
```

Listing 1: Datatypes and operations in Genetic module

```
replicate' :: Int -> a -> [a]
replicate' !n x = take n xs
                  where xs = repeat x
```

Listing 2: Example program

function MUTATESET(`Strand` s, `Integer` newBits, `Int` index)
    n ← `size` s
    bits ← `vec` s
    if index == n **then return** `writeToDisk` s
    else if bits `!!` index ! = newBits `!!` index **then**
        fp ← `path` s
        prog ← `program` s
        prog' ← FLIPBANG(fp prog index)
        return MUTATESET((`Strand` fp prog' bits n) newBits (index + 1))
    else
        return MUTATESET(s newBits (index + 1))
    end if
end function

function MUTATESTRAND(`Strand` s)
    n ← `size` s
    bits ← `random` $(0, 2^n)$
    return MUTATESET(s bits 0)
end function

function MUTATE(`Genes` g)
    g' ← `writeToDisk` g
    strands ← `getStrands` g'
    return `Genes` (`map` MUTATESTRAND strands)
end function

Figure 1: Mutation algorithm

```
function MERGESTRANDS(Strand s1, Strand s2)
    bits1 ← vec s1
    bits2 ← vec s2
    newBits ← bits1 — bits2
    return MUTATESET(s1 newBits 0)
end function

function MERGE(Genes g1, Genes g2)
    g1' ← writeToDisk g1
    strands ← zip (getStrands g1') (getStrands g2)
    return Genes (map (uncurry MERGE) strands)
end function
```

Figure 2: Merging algorithm

```
function FITNESS(Int reps, Float base, Genes g)
    exec ← path . head . getStrands g
    COMPILE(g)
    system "bash timer.sh " + exec + " " + base + "s " + reps
    times ← read "times.txt"
    return AVG(times)
end function
```

Figure 3: Fitness algorithm

```
function ALG([Genes] g, Int reps, Int runs, Float base)
    if runs == 0 then
        return g
    end if
    g' ← BUILDGENERATION(g)
    times ← map (FITNESS(reps, base)) g'
    results ← sort (zip times g')
    fastTime ← fst . head results
    nextGen ← take n (map snd results)
    return ALG(nextGen, reps, (runs - 1), fastTime)
end function
```

Figure 4: Genetic algorithm

This, combined with `mutate`, is how we explore the space of strict programs.

Finally, there is `fitness`, which must determine how well a `Genes` does. We do so by timing the executable. While the Rewrite module ensures the change is statically correct, we cannot confirm it terminates. As mentioned before, truly figuring this out is hard. However, we don't need the exact answer. We wish to consider programs that are faster than the original, which a non-terminating program is not. If we assume the original program does terminate, then we can time that to get a base. This base can then be used to timeout programs that run too long. It is in this way we can avoid non-termination.

The function `fitness` takes in the number of times to time the program, a base time in seconds for the timeout, and a `Genes` to time. We compile the program to an executable and use a bash script to time it and write those results to a file. Every run that times out or fails to run writes -1.0 to the file. We continue by reading that file, producing `[Float]`. We take the average of that after removing -1.0 from the list to produce an average. If the list is all -1.0, then the average is also -1.0.

3.2.2 Overall Algorithm

Figure 4 shows the genetic algorithm. We take a `Genes` and build a new generation based off them. This involves merging those in the original set and mutating the entire group. Then we measure the fitness of every gene using the given base time for timeouts. We sort the genes based off the fitness scores and record the fastest time. Finally, we take the n best and pass them through the algorithm again.

```
data GeneRecord = GR {  gene :: Genes
                     , t :: Float
                     }

type GeneDict = [GeneRecord]

createGeneRecord :: Genes -> Float -> GeneRecord

-- Functions on GeneDict
addGeneRecord :: GeneDict -> GeneRecord ->
                 GeneDict
findTimeForGene :: GeneDict -> Genes ->
                   Maybe Float

function FITNESS'(GeneDict dict, Int reps, Float base,
Genes g)
    cached ← findTimeForGene dict g
    if cached == Nothing then
        return FITNESS(reps, base, g)
    end if
    return cached
end function

Require: maxFailCount, threshold are built-in
function ALG'([Genes] g, Int reps, Int runs, Float base,
(GeneRecord, Int) (gr, failCount), GeneDict dict)
    if runs == 0 then
        return g
    else if failCount < maxFailCount then
        return [gene gr]
    end if
    g' ← BUILDGENERATION(g)
    times ← map (FITNESS'(reps, base, dict)) g'
    results ← sort (zip times g')
    fastTime ← fst . head results
    nextGen ← take n (map snd results)
    dict' ← foldl addGeneRecord dict records
    fastest ← t gr
    diff ← fastest − fastTime
    if diff < 0.0 then
        return ALG'(nextGen, reps, (runs - 1), base, (gr, fail-
Count + 1), dict')
    else if diff < threshold then
        return ALG'(nextGen, reps, (runs - 1), base, (gr, fail-
Count + 1), dict')
    else
        return ALG'(nextGen, reps, (runs - 1), fastTime, (gr, 0),
dict')
    end if
end function
```

Figure 5: Genetic algorithm with datatypes, caching, and failure count

3.2.3 Converging

The base algorithm will run a specified number of times before terminating, regardless of the results. We wish to terminate if we

	User CPU Time		Clock Time	
Program Name	Before	After	Before	After
binarytrees	116.358	76.61	75.667	63.782
fannkuchredux	35.015	33.601	10.514	10.155
gcGenSim	20.228	19.556	20.383	19.637

Table 1: Times for experiments in seconds, both before and after transformation

are not making any progress. This means that any of the following occurs

- Every program times out
- Every program is made slower
- The decrease in speed is small

We need to keep track of how many times any of these events occured. After a certain threshold, we terminate the program sooner than the specified number of runs. To measure our progress, we must also keep track of the current fastest program and its time.

Since `mutate` and `merge` are random, there are occasions we create programs that we previously encountered. Instead of running them again, we created a dictionary cache. All of these are reflected in Figure 5.

4. Experiment

We decided to run the algorithm on two programs from the benchmarks game [5] and an additional one created by Nathan Ricci. One is `binarytrees.ghc-1`, which allocates one large binary tree for memory pressure and allocates many small ones as it runs. The other is `fannkuchredux.ghc-5`, which takes in an integer n and calculates the maximum number of flips needed for any permutation of size n to get 1 to the front. The third, `gcGenSim`, simulates a generational garbage collector using a trace from a java program. We ran this one with a scaladoc trace, provided by Nathan.

These experiments were run on an Intel Xeon E31245, an 8 core CPU clocked at 3.30 GHz on 64-bit Ubuntu 12.04 LTS and 3.8 GB of memory. The code was compiled using the Glasgow Haskell Compiler version 7.4.1.

4.1 binarytrees

We removed all the strictness in the original program and sent it through the algorithm. We compiled all versions with `-threaded`, `-rtsopts`, `-XBangPatterns` options enabled and ran with `-N4 -K128M` RTS options enabled. Table 4 shows us the program sped up. We cut the CPU time by about 40 seconds while reducing the clock time by 12 seconds. Looking at the resulting code, we see that the algorithm added exactly one strictness annotation, listed below.

```
sumT :: Int -> Int -> Int -> Int
sumT d 0 t = t
sumT d i !t = sumT d (i-1) (t + a + b)
  where a = check (make i     d)
        b = check (make (-i) d)

-- traverse the tree, counting up the nodes
check :: Tree -> Int

...

-- build a tree
make :: Int -> Int -> Tree

...
```

Program Name	User CPU Time	Clock Time
binarytrees	124.397	71.335
edited binarytrees	67.426	55.853

Table 2: Times for orignal and edited `binarytrees` in seconds

The strictness is on the third integer `t`. How can this decrease the time? Suppose `i`> 1 and we run `sumT d i t`. Then we have

```
sumT d i t == sumT d (i-1) (t + a + b)
```

Since `i`> 1, we will evaluate the recursive case once more. We will say the call is `sumT d i' t'`. Since `sumT` is strict on `t'` in this call, we must evaluate it. However,

```
t' == (t + a + b)
```

This implies we must calculate `t`, `a`, and `b`. Note the definition of `a` and `b`. Both call `make` which creates trees. In the lazy version, these trees will be created only when some other function needs the value of a call to `sumT`. At that point, all trees will be created at once, leading to excess memory pressure and more thunks to evaluate. The strict version simply creates the trees, saving the operations for thunks and thusly time. Looking back at the original program, we found this annotation was not there.

We later realized that removing the strictness from the program removed the memory pressure needed for `binarytrees` to run as intended. Therefore, we timed two more versions of the program, the original program with the strictness for memory pressure, and another that also added the strictness in `sumT`. The times are reported in Table 4.1. We see that the additional strictness from the algorithm still gives us the decrease in runtime.

4.2 fannkuchredux

In this experiment, we did not change the program at all. Instead, we just gave it to the algorithm. Since the algorithm only assumes the program terminates and has the ability to add and remove strictness, this required no changes. We compiled with the same options as the `binarytrees` experiment and ran with the `-N4` RTS option. We see that we gained little time both in terms of CPU seconds and clock seconds. The only change we see is two strictness annotations

```
fannkuch :: Int -> Int -> (Int, Int)
fannkuch !n !i = ...
```

Now we must figure out where the program calls the function. The only call to the function is in `main`.

```
n <- fmap (read . head) getArgs
...
parMap rdeepSeq (fannkuch n) [0 .. (n - 1)]
```

Therefore, we save the writing and reading of $n + 1$ thunks. This explains the degree of the speedup. We expected a decrease in strictness annotations, not an increase. Looking at `merge` explains why. The bitwise-or actually introduces a bias towards the addition of strictness.

5. Future Work

Our algorithm does give us decreases in run time. However, there is much work to be done. The most glaring issue is speed. Currently, the fitness function is the bottleneck of the system because we run executables for timing. While we improved the run time with caching the run times of every mutation, `gcGenSim` still finished 10 generations in 13 hours and 46 minutes. We are looking into attempting to run more of these executables in parallel while

dealing with the variance that comes from scheduling multiple processes and threads. Another possibility is to allow the user to specify functions to enhance with strictness and improve those instead.

One suggestion we got was to use simulated annealing. Genetic algorithms are suited to breed a group of good options. Annealing can provide us with a singular best, or in our case fastest, program. We can also try changing the fitness function to measure maximum heap size rather than time to create more space-efficient programs.

Finally, our algorithm works on a single piece of test data. To provide better run times in general, we need to add in options for test suites, whether they be handcrafted by a user, or from `arbitrary` instances in the QuickCheck library to generate them on the fly.

6. Conclusion

We have written a genetic algorithm to explore the space of strict programs and obtain a faster program. We found the challenge lay in the genetic algorithm, not the rewriting and further work would definitely focus more on that side as well. However, we explored only the state of speedy programs. We need to look more into resource-aware programs, like space-efficient, and try to use strictness to optimize in that way.

Acknowledgments

Thank you Norman for bringing this class to this semester. Also would like to thank Nathan for bringing up the idea in a research group meeting as well as everyone in the workshop for the suggestions made to improve the algorithm.

References

[1] S. Peyton Jones and W. Partain. Measuring the Effectiveness of a Simple Stricntess Analyzer. 1993 In *Functional Programming 1993*, pages 201-220. 1993.

[2] K. D. Jensen, P. Hjaeresen, and M. Rosendahl. Efficient Strictness Analysis of Haskell. In *Static Analysis*, B. Le Charlier, Springer, Berlin, Heidelberg, 346-362. 1994.

[3] T. Schrijvers and A. Mycroft. Strictness Meets Data Flow. In *Proceedings of the 17th International Conference on Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer-Verlag, Berlin, Heidelberg, 439-454. 2010.

[4] P. L. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, editors, Ellis Horwood. 1987.

[5] benchmarksgame.alioth.debian.org

[6] https://github.com/haskell-suite/haskell-src-exts