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` as show in Figure 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[1]
Jensen[2]
Mycroft[3]

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`

Final paper for Comp150-05: Advanced Functional Programming

package to get the abstract syntax tree and print it back to readable source [5]. 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

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

data Genes = { strands :: [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

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 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 `Strand`s 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. Following this, we choose a random `Strand` from the `Genes` and generate a random bit vector. Then, we run through the `Strand`, flipping the state of strictness throughout the program 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 `Strand`s, 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. 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

3.2.3 Converging

4. Experiment

4.1 Hardware and Software

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.

4.2 binarytrees

4.3 fannkuchredux

5. Future Work

6. Conclusion

Acknowledgments

Acknowledgments, if needed.

References

[1] Peyton Jones ... simple strictness ...

[2] Jensen ... more complex analysis

[3] Mycroft ... effects and data flow

[4] benchmark games ...

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