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 ??. 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`. In other words, `n` is evaluated eagerly. We say that `g` is strict in `n` and the process of forcing eager evaluation is adding strictness to the program.

2. Background

2.1 Strictness

2.2 Strictness Analysis

2.3 Related Work

3. Architecture

3.1 Overview

3.2 Rewrite

3.3 Genetic Algorithm

3.3.1 Genes

3.3.2 mutate

3.3.3 merge

3.3.4 fitness

3.3.5 Overall algorithm

3.3.6 Converging

4. Experiment

4.1 Hardware and Software

4.2 binarytrees

4.3 fannkuchredux

5. Future Work

6. Conclusion

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...