Tailored Termination for Improved Supercompilation

Introduction

Supercompilation is a powerful program specialization technique that can produce programs that are significantly more efficient than the original. This is done by performing as many reductions at compile time as possible that would usually be deffered to run-time. Through this, it can automatically perform many optimizations that are implemented manually in modern compilers such as deforestation/fusion, function specialization and constructor specialization. For example, here's what supercompilation can achieve:

```
map (+ 3)
  (map (- 2) xs))
```

→

```
let f xs = case xs of
             []     → []
             (y:ys) → y + 1 : f ys
```

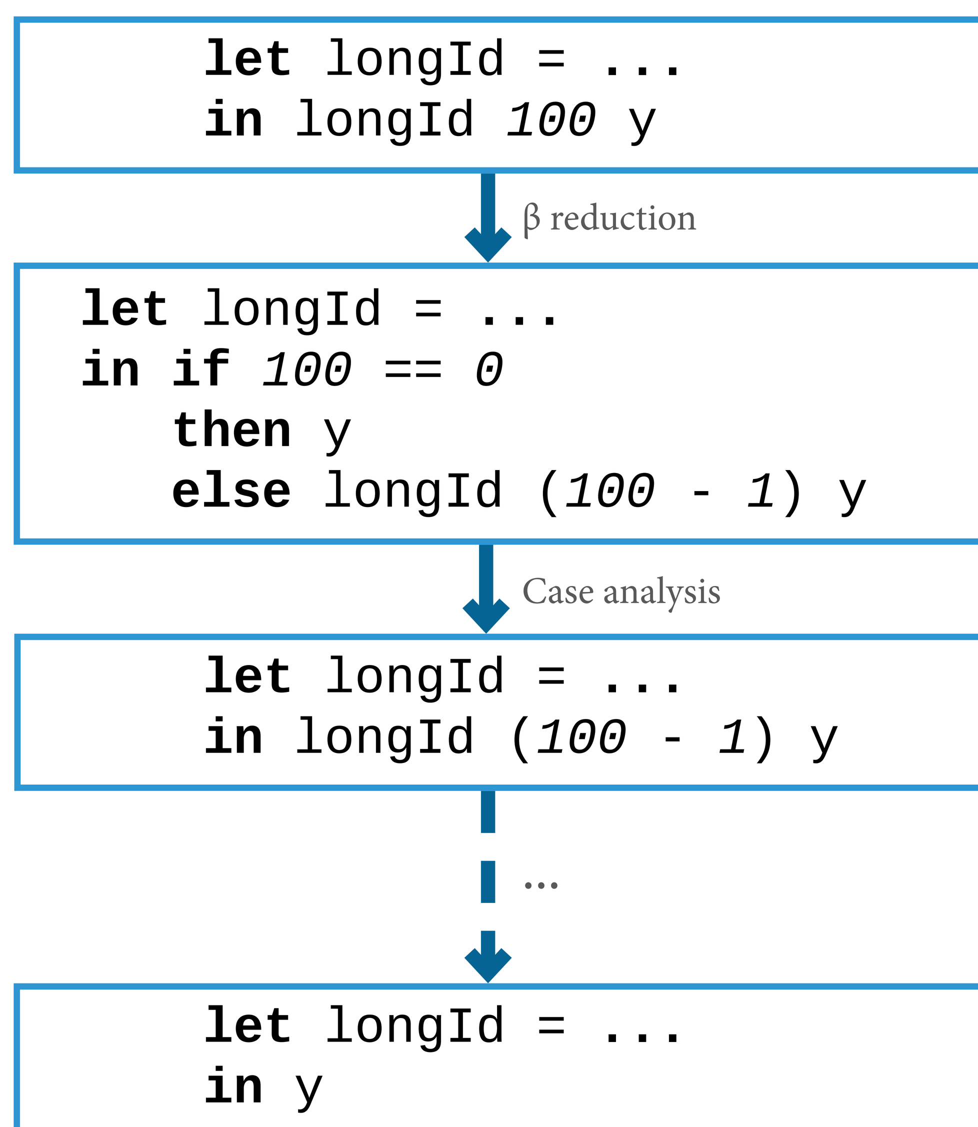
A key part of any supercompiler is the termination check: an algorithm that decides when the supercompilation process should stop. Strict termination crtieria that stops the process too soon can lead to missed optimization opprtunities. Conversly, a termination criterion that is overly permissive can result in a bloated and over-specialized program. In this project we investigated the effects of tailoring the termination criterion on a per program basis to measure its effects on supercompilation.

Underspecialization

Underspecialization is when the termination criterion stops the supercompilation process too soon. Consider this program for example:

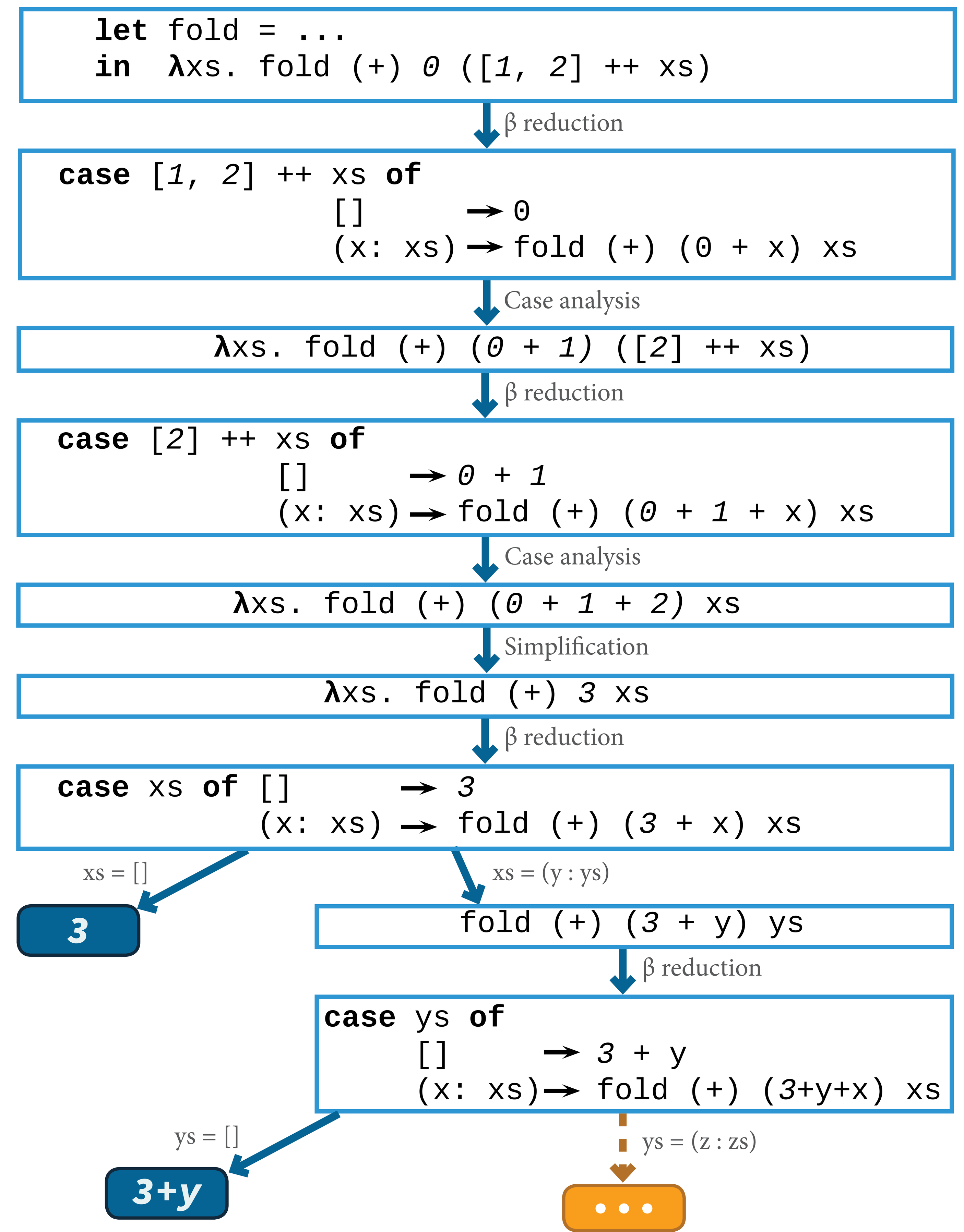
```
let longId = λn. λx. if n == 0
                     then x
                     else longId (n - 1) x
in longId 100 y
```

the optimal supercompiled version of this program would simply be "**y**" but if the termination algorithm detects that the supercompilation process is diverging and prematurely signals a stop, it could be left in an incomplete intermediate state as shown below.



Overspecialization

On the opposite end of the spectrum to underspecialization, overspecialization is when a program becomes bloated and specialized to no benefit. A small program can grow to become several megabytes, programs involving recursive list functions will generate specialized versions for every possible list length. This is demonstrated below using the supercompilation for a program involving **fold** called with some constant arguments.



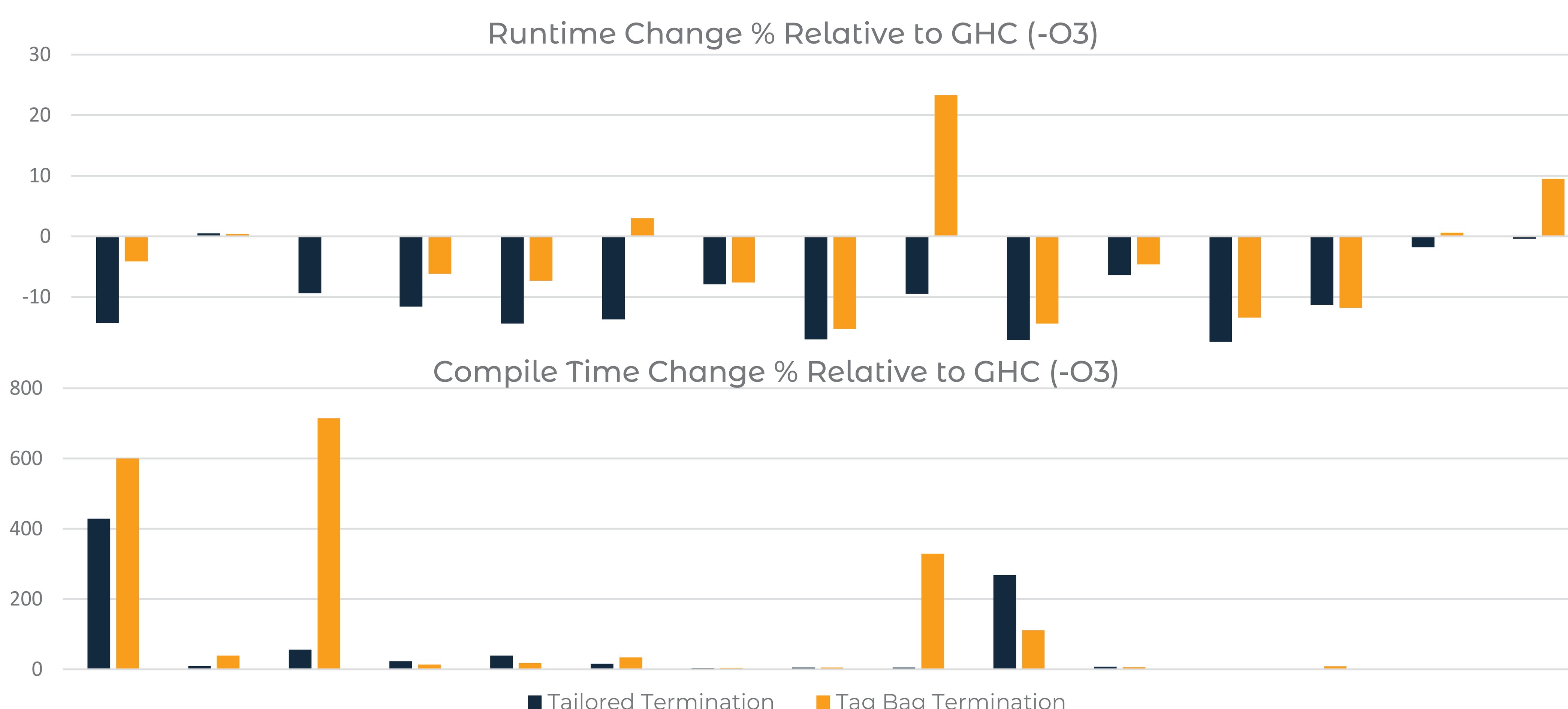
Methodology

Our work is based on the previous state of the art Haskell supercompiler created by Max Bolingbroke et al [1] which used a tag bag based termination criterion. Instead of using previous approaches of a one-size fits all termination criteria, we customized the termination criteria on a per-program basis.

This was achieved by tagging each state as shown in the previous supercompilation trees and using a termination criterion that would terminate the process given a set of tags. A wide variety of benchmarks were used to evaluate performance given the new termination criterion. Supercompilation trees as shown above were generated every 100 steps or so and then termination tags were selected to be fed back into the process.

[1] Maximilian Bolingbroke and Simon Peyton Jones. 2010. Supercompilation by evaluation. In ACM Sigplan Notices, Vol. 45.

Results



Initial results on a varying suite of benchmarks are fairly promising. We managed to obtain an average of 8% runtime reduction compared to GHC in -O3 (95% compared to -O0). Memory usage was reduced by 18% on average and compile times increased by 100%.

Future Work

By hand crafting the termination, we have shown there is potential improvements and gains to be made by using supercompilation. However, tweaking the termination by hand is quite laborsome and thus avenues for automation must be looked into. We believe that machine learning on the supercompilation trees or profile guided optimization is likely the best bet.

Acknowledgements

Many thanks are owed to my research adviser Benjamin Delaware for his guidance and reviews. Additionally, I'm thankful to Simon Peyton Jones for fruitful discussions.

Ammar Askar
aaskar@purdue.edu

PURDUE UNIVERSITY