Dissertation Type: programming languages

DEPARTMENT OF COMPUTER SCIENCE

Variations on Normalisation by Evaluation in Haskell

Lucas O'Dowd-Jones

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 26th March, 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Lucas O'Dowd-Jones, Friday 26th March, 2021

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Executive Summary

This section should précis the project context, aims and objectives, and main contributions (e.g., deliverables) and achievements; the same section may be called an abstract elsewhere. The goal is to ensure the reader is clear about what the topic is, what you have done within this topic, *and* what your view of the outcome is.

The former aspects should be guided by your specification: essentially this section is a (very) short version of what is typically the first chapter. Note that for research-type projects, this **must** include a clear research hypothesis. This will obviously differ significantly for each project, but an example might be as follows:

> My research hypothesis is that a suitable genetic algorithm will yield more accurate results (when applied to the standard ACME data set) than the algorithm proposed by Jones and Smith, while also executing in less time.

The latter aspects should (ideally) be presented as a concise, factual bullet point list. Again the points will differ for each project, but an might be as follows:

- I spent 120 hours collecting material on and learning about the Java garbage-collection sub-system.
- I wrote a total of 5000 lines of source code, comprising a Linux device driver for a robot (in C) and a GUI (in Java) that is used to control it.
- I designed a new algorithm for computing the non-linear mapping from A-space to B-space using a genetic algorithm, see page 17.
- I implemented a version of the algorithm proposed by Jones and Smith in [6], see page 12, corrected a mistake in it, and compared the results with several alternatives.

Supporting Technologies

This section should present a detailed summary, in bullet point form, of any third-party resources (e.g., hardware and software components) used during the project. Use of such resources is always perfectly acceptable: the goal of this section is simply to be clear about how and where they are used, so that a clear assessment of your work can result. The content can focus on the project topic itself (rather, for example, than including "I used LaTeX to prepare my dissertation"); an example is as follows:

- I used the Java `BigInteger` class to support my implementation of RSA.
- I used a parts of the OpenCV computer vision library to capture images from a camera, and for various standard operations (e.g., threshold, edge detection).
- I used an FPGA device supplied by the Department, and altered it to support an open-source UART core obtained from `http://opencores.org/`.
- The web-interface component of my system was implemented by extending the open-source WordPress software available from `http://wordpress.org/`.

Notation and Acronyms

Any well written document will introduce notation and acronyms before their use, *even if* they are standard in some way: this ensures any reader can understand the resulting self-contained content.

Said introduction can exist within the dissertation itself, wherever that is appropriate. For an acronym, this is typically achieved at the first point of use via "Advanced Encryption Standard (AES)" or similar, noting the capitalisation of relevant letters. However, it can be useful to include an additional, dedicated list at the start of the dissertation; the advantage of doing so is that you cannot mistakenly use an acronym before defining it. A limited example is as follows:

AES	:	Advanced Encryption Standard
DES	:	Data Encryption Standard
	⋮	
$\mathcal{H}(x)$	:	the Hamming weight of x
$\mathbb{F}_q$	:	a finite field with q elements
x_i	:	the i-th bit of some binary sequence x, st. $x_i \in \{0, 1\}$

Acknowledgements

It is common practice (although totally optional) to acknowledge any third-party advice, contribution or influence you have found useful during your work. Examples include support from friends or family, the input of your Supervisor and/or Advisor, external organisations or persons who have supplied resources of some kind (e.g., funding, advice or time), and so on.

Chapter 1

Introduction

To implement a functional programming language, we need a normalisation function that maps each expression in the functional language to its normal form. [EXPLAIN WHAT NORMAL FORM IS?] In this dissertation we explore various implementations of normalisation by evaluation (NbE) for the lambda calculus.

NbE proceeds by interpreting each term as an element of a semantic set, where computation is easier to perform, before "reifying" the semantic value back into the set of normal terms. All β-equal terms "evaluate" to the same semantic value, so β-equal terms normalise to the same normal form.

NbE is a modern alternative to normalisation by reduction; a technique based on syntactic rewriting. Since the foundations of NbE are mathematical, we can prove that our implementation is correct and study its behaviour formally. [CITE]. Proving that the implementations are fully correct is beyond the scope of this project, but we do use types as machine-checked proof that our implementation satisfies certain correctness properties. Dependent type theories [NAMECHECK] use NbE to check for equality between dependent types by normalising type-level programs. NbE takes advantage of advanced features in the implementation language, so serves as a useful benchmark for the strength and expressiveness of functional languages. NbE can improve the speed of compilation of functional languages. [3]

First we present two approaches for NbE of the untyped lambda calculus. The first implementation generates fresh variables during reification, which introduces global state into the program. State can be difficult to reason about and introduces complexity when testing, so often leads to errors in programs. These issues motivate a second implementation of NbE which uses de Bruijn indices and levels to represent variable binding in terms instead of named lambda terms, eliminating the need for fresh variable generation and state. We present a simple method for translating between named-variable terms and de Bruijn terms, since named-variable terms are easier to read.

Next we explore NbE for the simply typed lambda calculus by porting an implementation written in Agda to Haskell, utilising cutting edge features available through compiler extensions. GADTs are used in conjunction with the DataKinds and PolyKinds extensions to define terms that are well-typed by construction. For the implementation of the normalisation function itself we need the RankNTypes extension, which gives finer control over quantification in polymorphic type signatures, and ScopedTypeVariables, to bind type variables within function bodies. To emulate reflection of dependent types from the type level to the value level at runtime we explore the singleton pattern.

Haskell does not support full dependent types, however the Haskell community are actively working to integrate dependent types through Dependent Haskell[1], following a proposal by Richard Eisenberg [2]. However, Haskell developers can already emulate the features of dependent types with GADTs and other cutting edge compiler extensions which strengthen the type system. Although these solutions are not as elegant as full dependent types, they allow developers to write complex-typed programs not possible in vanilla Haskell.

The aims of this project are:

1. To produce various implementations of NbE in Haskell
2. To explore how successful modern features of Haskell are in implementing an algorithm with complex types.

Chapter 2

The Structure of Normalisation by Evaluation

Chapter 3

Normalising the Untyped Lambda Calculus

Chapter 4

Representation of the Typed Lambda Calculus

Chapter 5

Normalising the Typed Lambda Calculus

Chapter 6

Critical Evaluation

This chapter is intended to evaluate what you did. The content is highly topic-specific, but for many projects will have flavours of the following:

1. functional testing, including analysis and explanation of failure cases,
2. behavioural testing, often including analysis of any results that draw some form of conclusion wrt. the aims and objectives, and
3. evaluation of options and decisions within the project, and/or a comparison with alternatives.

This chapter often acts to differentiate project quality: even if the work completed is of a high technical quality, critical yet objective evaluation and comparison of the outcomes is crucial. In essence, the reader wants to learn something, so the worst examples amount to simple statements of fact (e.g., "graph X shows the result is Y"); the best examples are analytical and exploratory (e.g., "graph X shows the result is Y, which means Z; this contradicts [1], which may be because I use a different assumption"). As such, both positive *and* negative outcomes are valid *if* presented in a suitable manner.

Chapter 7

Conclusion

The concluding chapter of a dissertation is often underutilised because it is too often left too close to the deadline: it is important to allocation enough attention. Ideally, the chapter will consist of three parts:

1. (Re)summarise the main contributions and achievements, in essence summing up the content.
2. Clearly state the current project status (e.g., "X is working, Y is not") and evaluate what has been achieved with respect to the initial aims and objectives (e.g., "I completed aim X outlined previously, the evidence for this is within Chapter Y"). There is no problem including aims which were not completed, but it is important to evaluate and/or justify why this is the case.
3. Outline any open problems or future plans. Rather than treat this only as an exercise in what you *could* have done given more time, try to focus on any unexplored options or interesting outcomes (e.g., "my experiment for X gave counter-intuitive results, this could be because Y and would form an interesting area for further study" or "users found feature Z of my software difficult to use, which is obvious in hindsight but not during at design stage; to resolve this, I could clearly apply the technique of Smith [7]").

Bibliography

[1] URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/dependent-haskell.

[2] Richard A. Eisenberg. "Dependent Types in Haskell: Theory and Practice". In: (). URL: https://www.cis.upenn.edu/~sweirich/papers/eisenberg-thesis.pdf.

[3] Sam Lindley. "Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages". In: (). URL: https://era.ed.ac.uk/bitstream/handle/1842/778/lindley_thesis.pdf.

Appendix A

An Example Appendix

Content which is not central to, but may enhance the dissertation can be included in one or more appendicies.
Note that in line with most research conferences, the marking panel is not obliged to read such appendices.