University of
BRISTOL

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.

Saturday 3$^{\text{rd}}$ April, 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, Saturday 3$^{\text{rd}}$ April, 2021

Contents

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 use types as machine-checked proof that our implementation satisfies certain 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. [7]

First we present two approaches for NbE of the untyped lambda calculus. The first implementation generates fresh variables during reification, which introduces 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 ScopedType-Variables, 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 [3]. 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

Normalising the Untyped Lambda Calculus
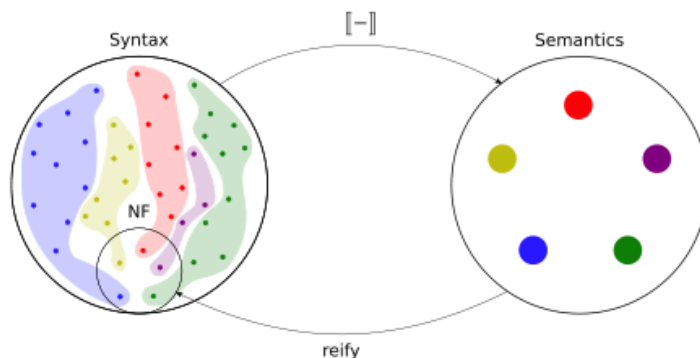
2.1 Overview of the NbE algorithm

Figure 2.1: A visual overview of the NbE algorithm from [6]

NbE proceeds in two steps. The first is to evaluate terms in the lambda calculus into a semantic set. In 2.1 terms are represented by dots in the syntax set and the evaluation function is denoted by $[\![-]\!]$, which we refer to as `eval`. The second step is to `reify` the semantic value back into the normal form of the original term. Thus, the `normalise` function which maps terms to their normal forms is the composition of `eval` and `reify`

2.1 illustrates why NbE works. The key property of the `eval` function is that β-equal terms (represented by dots of the same colour in the syntax set) evaluate to the same semantic value. This ensures that β-equal terms normalise to the same normal form. The key property of the `reify` function is that the codomain of `reify` is the subset of normal forms, so `normalise` is guaranteed to return a normal form.

The remainder of this chapter defines the data NbE operates on and functions to perform NbE in Haskell.

2.2 Syntax

```
type Name = String

data Expr = ExpVar Name
          | ExpLam Name Expr
          | ExpApp Expr Expr
```

Inhabitants of the inductively-defined datatype `Expr` are well-formed terms of the untyped lambda calculus with strings as variables. The first argument of the lambda case introduces a new variable name

bound in the function body defined by the second argument. For example, the identity function $\lambda x.x$ would be encoded as `ExpLam "x" (ExpVar "x")`.

```
data NormalForm = NfNeutralForm NeutralForm
                | NfLam Name NormalForm

data NeutralForm = NeVar Name
                 | NeApp NeutralForm NormalForm
```

We now define the target syntax of the `normalise` function, `NormalForm`. Note that `NormalForm` is inhabited by all the terms not containing β-redexes [6], since the definition of `NeApp` only permits application on non-lambda terms, which are encoded as values of type `NeutralForm`.

2.3 Semantics

```
data V = Neutral NeutralV
       | Function (V -> V)

data NeutralV = NeVVar Name
              | NeVApp NeutralV V
```

The semantic set `V` has a very similar structure to the set of normal forms, however lambda terms are replaced with Haskell functions of type $V \to V$. The similarity simplifies `reify` as for some terms there are obvious translations from semantics to syntax. The replacement of lambda terms will be useful in evaluating β-redexes at the semantic level instead of the syntactic level.

2.4 Evaluation

Evaluation proceeds by pattern matching on the given term, and evaluating its constituent subterms. However, the semantic meaning of subterms can differ depending on which variables are bound by which lambda term. For example, in the terms $\lambda x.\lambda y.xy$ and $\lambda y.xy$, the semantic meaning of the xy subterm is different. Thus, in addition to the term itself, `eval` needs information about the bound variables introduced by surrounding lambda terms.

To keep track of which variables have been bound by surrounding lambda terms, we construct an environment datatype.

```
type Env = Map Name V
```

Each key of the map corresponds to a bound variable name, and its associated value is the semantic value representing the variable. The environment can be thought of as the scope each subterm is evaluated in. It expands as more variables are bound in deeper subterms.

```
eval :: Expr -> Env -> V
```

`eval` takes an expression and the environment to evaluate it in, pattern matches on the expression, and returns the interpretation of the term in the semantic set. We now discuss the implementation of each case of the pattern match.

```
eval (ExpVar x) env = case lookup x env of
    Just y -> y
    Nothing -> Neutral (NeVVar x)
```

In the variable case, we lookup the variable in the environment. If the variable was bound by a surrounding lambda term, the variable will be present in the environment, and we can return the semantic value associated with it. Otherwise, the variables is free, so we return a semantic variable with the same name.

```
eval (ExpLam var m) env = Function f where
    f v = eval m env' where
        env' = insert var v env
```

The semantic interpretation of a lambda expression is a Haskell function of type $V \to V$. This function

takes an element `v` of the semantic set `V`, and returns the body of the lambda evaluated in an extended environment `env'`. In this extended environment, we bind the named variable `var` to `v`.

From the variable case, we see that whenever the variable `ExpVar var` is evaluated in the function body it will be interpreted as `v`. We can think of `v` as a semantic placeholder for the value `f` is applied to (if any). The use of this approach is demonstrated in the application case.

```
eval (ExpApp m n) env = app (eval m env) (eval n env)


app :: V -> V -> V
app (Function f) v0 = f v0
app (Neutral n)  v0 = Neutral (NeVApp n v0)
```

In the application case we evaluate the `m` and `n` terms in the same environment, before applying them to the `app` function, which handles the application of semantic values.

In the case where the first value is a function `f`, we evaluate `f` at the second argument `v0`. From the lambda case we see that this subcase corresponds to evaluating a redex term. Instead of contracting the redex by substituting at the syntactic level, we evaluate it using function application at the semantic level. The application instantiates the placeholder `v` at the semantic value `v0` in the lambda body.

In the case that the first value is a value `n` of type `NeutralV`, there is no redex to contract, so we return a placeholder Neutral application at the semantic level.

2.5 Gensym Reification

Reification proceeds by pattern matching on the semantic value.

Since evaluating a lambda yields a function, reifying a function should yield a lambda to ensure that `normalise` preserves terms already in normal form. This introduces a question: what variable should the returned lambda term bind? The new bound variable should be different to all other bound variables in scope, otherwise `reify` could produce invalid terms. It should also be different to all other free variables in the original term, to prevent free variable capture. Since terms are considered equal up to bound variable renaming by α-equivalence, we can choose any variable name that satisfies these requirements.

```
type FreshName = State [Name]


reify :: V -> FreshName NormalForm
reify (Neutral n)  = do
    reifiedN <- reifyNeutral n
    return (NfNeutralForm reifiedN)
reify (Function f) = do
    freshNames <- get
    -- Remove the first name from the freshNames stream
    let freshVar = head freshNames
    -- The first name is no longer fresh (we are abount to use it as a bound variable)
    -- Modify the state to remove the used variable name
    modify tail
    -- Reify the body of the semantic function when evaluated at the fresh bound variable
    normalForm <- reify (f (Neutral (NeVVar freshVar)))
    return (NfLam freshVar normalForm)

reifyNeutral :: NeutralV -> FreshName NeutralForm
reifyNeutral (NeVVar i)   = return (NeVar i)
reifyNeutral (NeVApp n m) = do
    reifiedNeutral <- reifyNeutral n
    reifiedNormal  <- reify m
    return (NeApp reifiedNeutral reifiedNormal)


normalise :: Expr -> NormalForm
normalise exp = evalState (reify (eval exp empty)) freshNames
    where
        freshNames = (getFreshVariableStream . getFreeVariables) exp
```

2.5.1 Fresh variables solution 1 - Gensym

approach based on [6]

    Implemented with State monad

    Issue with solution 1: Have to add monad everywhere (inescapable) - all functions dependent on state

    "Less functional" - carry around state (may as well use imperative)

2.6 de Bruijn Reification

2.6.1 Fresh variables solution 2 - Locally nameless terms

approach based on [2]

    Uses de Bruijn Indicies for syntax and deBruijn levels for semantics

    index n references nth abstraction, if m abstractions: if n ¡ m bound variables, otherwise free variable

    Shifting for abstractions

Chapter 3

Representation of the Typed Lambda Calculus

TypeOperators allows us to define types using infix operator :-¿. 'infixr :-¿ 9' sets strongest right assoicativity (matches usual convention that a -¿ b -¿ c   a -¿ (b -¿ c))

Problem: Only want to normalise well typed terms. Criteria: (1) Exclude $\lambda x.y$ - y free variable so not in typing context (2) Exclude $\lambda x.xx$ - untypable (types are finite)

Problem with standard terms - can't carry type information

3.0.1 Typed Term solution - GADTs

[3] Explicit type constructor arguments Language extensions: DataKinds, TypeOperators, PolyKinds, GADTs.

Example 1: Vec - indexed type advantage since total function

Fails for replicate without work [8]

Advantage over ADTs: type refinement by constructor

Justification of base type *extension work??: string/integer type* Example 2: IsElem - value as proof Example 3: Expr - all values well typed by construction

Difference between [] and '[] (kinds vs types) in terms of levels

Input/Ouput type as pre-condition/post-condition (eg empty context)

Chapter 4

Normalising the Typed Lambda Calculus

Investigation: Are GADTs in Haskell powerful enough? Types are erased at runtime so true dependent typing not part of Haskell (programs at type level)

Poissible to erase all type information, NbE on Untyped Issue: No proof that type preserved Solution: Track types as do evaluation - nbe program itself proof that types preserved (subject reduction parallel?)

Started by implementing same as untyped

Main difference in semantics (V := a -¿ b — Neutral) [6]

problem: Need to strengthen context evaluating body (eval Lam case)

4.0.1 Solution: Order Preserving Embeddings (OPEs)

Following implementation in Agda [4], agda has full dependent types (type system more powerful) - adapt for haskell, how nicely?

if a term well typed for one context, also well typed for any longer one

A value of type 'OPE strong weak' can derive weak from strong by dropping elements from context

OPE is a relation on typing contexts

4.0.2 Semantic set

Defintion of V using OPEs - Haskell vs agda

Need to quantify over 'strong' in function - OPE strong weak is guarentee that strong is a stronger context than weak (if quantified at start end up with values where weak stronger than strong) - need rank2 types extension for nested quantification

Helper functions (composition, strengthing relative to OPE) - explain derivations

4.0.3 implementing Eval

Defintion of environment (maps expressions in syntax context ctx to values in semantics with context ctxV)

problem: in app case how to we get identity OPE for semantic context?

But types erased at compile time to make Haskell efficient

How to generate a value at runtime dependent on type erased at compile time

dependent pattern match [5]

4.0.4 Solution: Singleton pattern

Method of Type to value known as reflection [5]

Idea: Create value-level tags for types - singleton types correspond type we're interested in, inhabited by only one value for each case

Examples: Reify case analysis, Ty reflection, Context reflection

Explicitly passing as value to pattern match on

Generate implictly using typeclass, use class constraint to imiplictly pass down ability to use contex methods through function calls. Is it a good idea to have class constraints in the GADTs/Syntax definitions?

Implementation in class vs full reflection - test this for speed?

problem : Inferring Any for ctxV (why?)

solution: scoped type variables - universally quantified variables used in type expressions bind over 'where' clause

(More usefully) can 'unpack' refined GADT types so that can create type definitions using refined types.

Analysis:

Have to specify type when normaling for correct eta-expansion (eta-long form)

Qs: How does locally nameless work in sematics? How does ctxV work in Env?

Chapter 5

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 6

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]   Andreas Abel. "Normalization by Evaluation Dependent Types and Impredicativity". In: (). URL: http://www2.tcs.ifi.lmu.de/~abel/habil.pdf.

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

[4]   András Kovács. "A Machine-Checked Correctness Proof of Normalization by Evaluation for Simply Typed Lambda Calculus". URL: https://github.com/AndrasKovacs/stlc-nbe/blob/separate-PSh/thesis.pdf.

[5]   Justin Le. *Introduction to Singletons*. URL: https://blog.jle.im/entry/introduction-to-singletons-1.html.

[6]   Sam Lindley. *Normalisation by evaluation*. URL: http://homepages.inf.ed.ac.uk/slindley/nbe/nbe-cambridge2016.pdf.

[7]   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.

[8]   Niki Vazou. *CMSC498V*. URL: https://nikivazou.github.io/CMSC498V/lectures/DependentHaskell.html.

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.