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.

---

Sunday 18th 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, Sunday $18^{\text{th}}$ 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 |
| | $\vdots$ | |
| $\mathcal{H}(x)$ | : | 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. 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 $\beta$-equal terms "evaluate" to the same semantic value, so $\beta$-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. [8]

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 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 [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

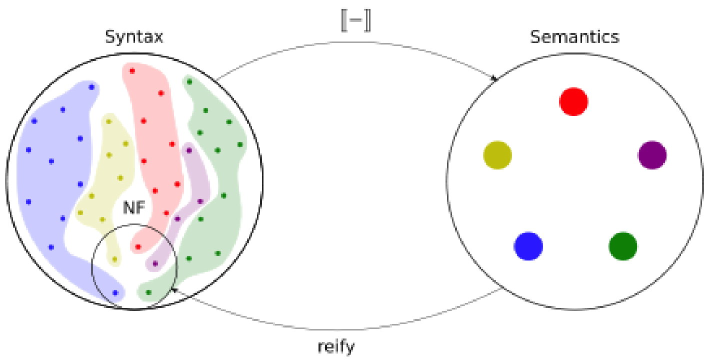


Figure 2.1: A visual overview of the NbE algorithm from [7]

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 $\beta$-equal terms (represented by dots of the same colour in the syntax set) evaluate to the same semantic value. This ensures that $\beta$-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 $\beta$-redexes [7], 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 $\beta$-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 -> V
    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` → `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)
    where
        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 `app`, 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, and recursively reifying its constituent values.

Since evaluating a lambda yields a function, reifying a function should yield a lambda to ensure that `normalise` preserves terms already in normal form. But 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 $\alpha$-equivalence, we can choose any variable name that satisfies these requirements.

One approach to resolve this issue suggested by [7] is to generate a fresh variable name during reification whenever a new bound variable is needed. However, generating fresh variables during the execution of `reify` would require the function to track which variables have already been bound between calls to `reify`, which suggests the use of state. This could be modelled using the `State` monad, in which case `reify` would have the type signature `reify :: V -> State [Name] NormalForm`, where `[Name]` corresponds to a list of variable names that have already been bound. This would allow us to implicitly pass the list of bound variable names between calls to reify. However state introduces additional complexity and makes testing more difficult, which can lead to flawed implementations.

Instead, we opt for a more functional approach. Before the execution of `reify`, we generate a suitable stream of fresh variable names of type `[Name]`. By making `reify` a function of the semantic element and the stream of fresh variables, we can simply pop a fresh variable name from the stream whenever a new variable is bound. In recursive calls, `reify` only passes the tail of the stream to ensure bound variable names are never reused.

```
reify :: V -> [Name] -> NormalForm
```

`reify` proceeds by pattern matching on the first argument of type `V`.

```
reify (Neutral n)  freshVars = NfNeutralForm (reifyNeutral n freshVars)
    where
        reifyNeutral :: NeutralV -> [Name] -> NeutralForm
        reifyNeutral (NeVVar i)   freshVars = NeVar i
        reifyNeutral (NeVApp n m) freshVars = NeApp reifiedN reifiedM
            where
                reifiedN = reifyNeutral n freshVars
                reifiedM = reify m freshVars
```

In the neutral case, we use `reifyNeutral` to convert the value `n` of type `NeutralV` to a `NeutralForm`, which is promoted to a `NormalForm` by the `NfNeutralForm` constructor. The `reifyNeutral` function also proceeds by patten matching.

In the variable case we extract a neutral syntactic variable of the same name as the semantic variable.

In the application case we reify the semantic values `n` and `m`, and return a neutral syntactic application of the resulting terms. We are able to reify both semantic values using the same fresh variable stream since the returned value is a `NeutralForm`, which by definition contains no redexes. This means `reifiedM` will not be substituted into `reifiedN`, so there will not be any variable name clashes. For example, in the neutral term $(y(\lambda x.x))(\lambda x.x)$ there is no issue in the left and right terms reusing $x$ as a bound variable since there is no ambiguity about which $x$ is being referred to in any part of the term. Since there are no redexes to contract, there is no need to rename the bound variables.

```
reify (Function f) (v:vs)    = NfLam v body
where
    body = reify (f (Neutral (NeVVar v))) vs
```

The reification of an abstract semantic function `f` produces a concrete syntactic description for `f`. `reify` abstracts out the argument of `f` into a syntactic variable `v` with a lambda expression. In the body of `f`, we replace the abstract argument with the variable `v` by applying `f` to `Neutral (NeVVar v)`. This value has type `V`, so we can `reify` it to produce a syntactic representation for the body of `f`, which completes the term. We `reify` the term with the stream of fresh variables `vs`, since the variable name `v` is bound in the body of the lambda, so is no longer fresh.

Using the implementations of `eval` and `reify`, we can define the `normalise` function as follows.

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

`normalise` takes an expression, and returns its normal form. Since `normalise` returns a value of type `NormalForm` it is guaranteed that the returned expression is normal. First we evaluate the given expression in the empty environment, since no variables are bound to begin with. Then we reify the returned semantic value of type `V` back into a concrete `NormalForm` term.

We produce a stream of valid fresh variable names `freshVars` of type `[Name]` for the given expression using the following functions.

```
getFreeVariables :: Expr -> Set Name
getFreeVariables (ExpVar x)   = singleton x
getFreeVariables (ExpLam x m) = delete x (getFreeVariables m)
getFreeVariables (ExpApp m n) = getFreeVariables m `union` getFreeVariables n

getFreshVariableStream :: Set Name -> [Name]
getFreshVariableStream freeVars = [freshVariable i | i <- [0..],
                                   notMember (freshVariable i) freeVars]
    where
        freshVariable i = "v" ++ show i
```

`getFreeVariables` takes an expression and returns the set of free variable names. `getFreshVariableStream` takes the set of free variable names and returns a stream of fresh variable names, since each of the names are distinct from each other and the free variables.

## 2.6 de Bruijn NbE

Another approach to resolve the fresh variable problem is to use a different representation of lambda calculus terms where variables are simpler. In general, a variable can be thought of as a pointer to the lambda that binds it. Named terms do this by labelling lambdas and referencing them in variables using their names. de Bruijn terms take a different approach which we describe below, but fundamentally both approaches serve the purpose, so represent the same set of terms.

de Bruijn terms are defined as follows. [4]

```
data DbExpr = DbVar Int
            | DbLam DbExpr
            | DbApp DbExpr DbExpr
```

In de Bruijn terms, lambda terms are not named, as seen in the definition of `DbLam`. Instead, each variable refers to the lambda that binds it using a de Bruijn index: the number of lambdas between the occurrence of the variable and the lambda which binds it.

| Named Notation | de Bruijn Notation |
|---|---|
| $\lambda x.x$ | $\lambda.0$ |
| $(\lambda x.x)(\lambda y.y)$ | $(\lambda.0)(\lambda.0)$ |
| $\lambda x.x(\lambda y.xy)$ | $\lambda.0(\lambda.10)$ |

Figure 2.2: Examples of de Bruijn terms and their equaivalent named terms

Variables with indices greater than the number of bound lambdas are treated as free. For example, we could represent the term $y(\lambda x.xy)$ as $0(\lambda.01)$ using de Bruijn terms. In the above term we have implicitly bound the free variable $y$ to 0 at the top level of the term. Note that we refer to $y$ using the index 1 inside the lambda since the lambda introduces a bound variable in the context of its body. 2 $(\lambda.03)$ is an equivalent term where $y$ is implicitly bound to 2 at the top level.

The changes to the syntax are reflected in the definition of the de Bruijn normal and neutral terms.

```
data NormalForm = NfNeutralForm NeutralForm
                | NfLam NormalForm

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

We also redefine the semantic set for de Bruijn terms following an implementation by Andreas Abel [2]

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

data NeutralV = NeVLevel Int
              | NeVApp NeutralV V
```

The `eval` function for de Bruijn terms operates similarly to the gensym evaluation function.

```
type Env = Map Int V

eval :: Env -> DbExpr -> V
```

In the de Bruijn implementation, the environment maps from de Bruijn indices to the semantic set, instead of from names.

```
eval env (DbVar x) = case lookup x env of
    Just y -> y
    Nothing -> undefined
```

In the variable case we again lookup the semantic interpretation of the given variable from the environment. In contrast to the gensym approach, we bind free variables in addition to bound variables in the context before the execution of `eval`. As seen in a previous example, there are many equivalent terms in the de Bruijn syntax depending on the choice of free variable indices. We make the choice explicit by binding free variables indices in the environment before the execution of `eval`.

This introduces the possibility that eval does not terminate successfully, however in practice this case will never be evaluated

```
eval env (DbLam m) = Function f where
f :: V -> V
f v = eval env' m where
    env' = insert 0 v (mapKeys (+1) env)
```

As in the named approach, the semantic representation chosen for functions is a Haskell function that takes a semantic value `v`, and returns the semantic interpretation of the lambda body with the bound variable interpreted as `v`. They key difference in the deBruijn case when working with de Bruijn terms is the construction of the new environment. We first increment all other indicies by 1 since we have introduced a new lambda between each variable and its associated binding. Then, to introduce the new variable bound by the lambda into the environment, we bind the 0th index to `v` in the shifted environment.

The application case remains exactly the same as in the gensym implementation, so the approach for contracting redexes is identical.

```
reify :: Int -> V -> NormalForm
```

Reification for de Bruijn terms also follows a similar structure to the gensym approach. However, instead of taking a list

```
reify n (Function f) = NfLam (reify (n + 1) (f (Neutral (NeVLevel n))))


reify n (Neutral m)  = NfNeutralForm (reifyNeutral n m)
    where
        reifyNeutral :: Int -> NeutralV -> NeutralForm
        reifyNeutral n (NeVLevel k) = NeVar (n - 1 - k)
        reifyNeutral n (NeVApp p q) = NeApp (reifyNeutral n p) (reify n q)


normaliseDbExpr :: Int -> DbExpr -> DbExpr
normaliseDbExpr n = normalToExpr . reify n . eval initialEnv
    where
        initialEnv = Map.fromList [(k, Neutral  (NeVLevel (n - 1 - k)))
                                  | k <-[0..(n - 1)]   ]
```

### 2.6.1 Fresh variables solution 2 - Locally nameless terms

Uses de Bruijn Indicies for syntax and deBruijn levels for semantics

# Chapter 3

# Representation of the Simply Typed Lambda Calculus

Before implementing NbE for the STLC, we define datatypes representing the STLC in Haskell.

## 3.1 Types

We first define a type syntax `Ty` to represent the monotypes of the STLC.

```
data Ty = BaseTy | Ty :-> Ty
infixr 9 :->
```

In this type syntax there is a single base type `BaseTy` and an infix type constructor `:->`, where the type `A :-> B` represents the function type from type `A` to type `B`. For the implementation of NbE in Chapter 4, a single base type is sufficient to capture the structure of simply typed terms. Multiple base types would have introduced unnecessary complexity. Normalising terms with polymorphic types is beyond the scope of this project.

`infixr :-> 9` specifies that `:->` is right associative, so as per convention, the type `A :-> (B :-> C)` can instead be written `A :-> B :-> C`.

## 3.2 First Attempt at Typed Terms

For typed NbE, we only normalise well-typed terms of the lambda calculus. To ensure that our terms are well-typed, expressions should track the type of the term and its typing context. We consider the following implementation of typed expressions.

```
data Expr = Var Ty [Ty] Int
          | Lam Ty [Ty] Expr
          | App Ty [Ty] Expr Expr
```

This implementation uses the de Bruijn style explored in Chapter 2, with an additional `Ty` parameter to store the type of the term, and a `[Ty]` parameter to store the typing context.

However, with this implementation it is possible to construct terms which are not well-typed such as `Var BaseTy [] 0`. This is because the set of representable terms is exactly the set of untyped terms, many of which are not well-typed. We could have a separate type-checking function to verify whether terms are well-typed before normalisation, but there would be no guarantee that the normalisation function produces well-typed terms.

Instead, we opt for an approach where the only inhabitants of `Expr` are well-typed terms. This guarantees that terms are well-typed before and after normalisation, and removes the additional complexity of a type-checker. To implement a datatype of terms that are well-typed by construction, we need to do more than simply add the type and typing context of each term. We also need to restrict the construction of terms based on the typing judgement rules of the STLC. In Section 3.5, we present a method for specifying these type restrictions developed by Richard Eisenberg [3], using GADTs.

## 3.3 Introduction to Generalised Algebraic Datatypes (GADTs)

GADTs are a generalisation of algebraic datatypes, where the type signature of each constructor is explicitly specified. The canonical example of a GADT is the length-indexed vector, where the length of each vector is tracked in its type.

To track the length of the vector in its type, we first need a way of representing numbers at the type-level. A standard way of representing the natural numbers at value-level is as follows:

```
data Nat = Zero | Succ Nat
```

We use the `DataKinds` compiler extension to automatically create a kind `Nat`, with the same structure as the original `Nat` datatype. The promoted kind `Nat` has the inhabitants `'Zero` and `'Succ`, which are denoted as types with apostrophes to prevent ambiguity with their value-level counterparts. These inhabitants are type constructors, where `'Zero` has kind `Nat`, and `'Succ` has kind `Nat` → `Nat`.

| Level | Original ADT | Promoted Kind |
|---|---|---|
| Kinds | | Nat |
| Types | Nat | 'Zero, 'Succ |
| Values | Zero, Succ | |

Figure 3.1: Illustration of the promoted kinds and types automatically created by `DataKinds`

We use the `'Zero` and `'Succ` type constructors to represent numbers at the type level. Using the `GADTs` extension we now define the datatype for length-indexed vectors using GADT syntax.

```
data Vec :: * -> Nat -> * where
    ZeroVec :: Vec a 'Zero
    SuccVec :: a -> Vec a n -> Vec a ('Succ n)
```

[3]

A value of type `Vec a n` is a vector of `n` elements of type `a`. For this definition we also require the `KindSignatures` compiler extension to specify the kind signature of `Vec` in the first line of the definition. This specifies that the first type parameter of `Vec`, denoting the type of the elements of the vector, should be of kind `*`. This is because the type of the elements of the vector could be any concrete type, all of which are inhabitants of `*`. The first line also specifies that the second type parameter of `Vec`, denoting the length of the vector, should be a type inhabiting the promoted kind `Nat`, which we use to represent a type-level number. The returned kind `*` in the kind signature specifies that each vector has a concrete type of kind `*`.

Since `Vec` is a GADT, the types of each constructor are given explicitly. The `ZeroVec` constructor creates a vector with elements of any type (since `a` is universally quantified over) of length `'Zero`. The `SuccVec` constructor takes a value of type `a` and a vector of `n` elements of type `a`, and returns a new vector of length `'Succ n`. For example, the vector `SuccVec "a" (SuccVec "b" ZeroVec)` has type `Vec String ('Succ ('Succ 'Zero))`.

An immediate advantage of using GADT length-indexed vectors over standard lists is that we can define a new function `head'` which only operates on vectors containing at least one element.

```
head' :: Vec a (Succ n) -> a
head' (SuccVec x xs) = x
```

The additional type precision awarded by GADTs moves errors from run-time to compile-time.

## 3.4 Elem GADT

Before using GADTs to construct the set of well-typed expressions, we first define the `Elem` GADT. A value of type `Elem xs x` is a proof that the type `x` is an element of the list of types `xs`.

```
data Elem :: [a] -> a -> * where
    Head :: Elem (x ': xs) x
    Tail :: Elem xs x -> Elem (y ': xs) x
```

[3]

The `Head` constructor produces a proof that `x` is an element of any list beginning with `x`. Given a value of type `Elem xs x`, the `Tail` constructor produces a proof that `x` is an element of the extended list `y:xs` for any element `y`. For example, the value `Tail Head` could have type `Elem '["a","b","c","d"] "b"` or type `Elem '[4, 5] 5`.

Note that all elements of said lists are at the type level, so we need a promoted type-level version of the `(:)` list constructor. Promotion to the type-level constructor `'(:)` is handled by `DataKinds`, however we need to enable the `TypeOperators` extension to allow the use of the infix operators at type-level. Without the `TypeOperators` extension we could use the syntax `Elem ('(:) x xs)` in the definition of `Head`, however this is harder to read.

Note that `[a]` is the kind consisting of lists with type-level elements of the same kind `a`, rather than the standard list of value-level elements of the same type. To write this polymorphic kind signature we need the `PolyKinds` extension, which extends the `KindSignatures` extension by enabling polymorphic kind signatures. Since `KindSignatures` is a dependency of `PolyKinds` it is implicitly enabled when using `PolyKinds`, so we can replace the `KindSignatures` extension declaration with `PolyKinds`.

## 3.5 Typed Expression Syntax

We are now ready to define the `Expr` GADT which represents the set of well-typed terms of the STLC.

```
data Expr :: [Ty] -> Ty -> * where
    Var :: Elem ctx ty -> Expr ctx ty
    Lam :: Expr (arg ': ctx) result -> Expr ctx (arg ':-> result)
    App :: Expr ctx (arg ':-> result) -> Expr ctx arg -> Expr ctx result
```

A value of type `Expr ctx ty` is a well-typed expression of the STLC of type `ty` in the typing context `ctx`. Hence, `Expr` encodes the set of valid typing judgements, where each constructor encodes a typing judgement rule in its type.

The `Var` constructor corresponds to the variable typing judgement rule. A variable can only be a well-typed expression of type `ty` if `ty` is present in the typing context. `Var` takes a value of type `Elem ctx ty` which proves that `ty` is in the typing context `ctx` by specifying which element of the context the variable is referring to. Note that instead of indexing variables by names or de Bruijn indices as we saw in Chapter 2, typed variables are indexed by an `Elem` value. However, the values of type `Elem xs x` act very similarly to de Bruijn indices, as `Elem` values refer to bound variables by how many bindings there are between the instance of the variable and where it was bound. The $i$th element of the typing context corresponds to the type of the element at de Bruijn index $i$. For example `Var (Tail Head)` could have type `Expr '[BaseTy, BaseTy :-> BaseTy, BaseTy] ('BaseTy :-> 'BaseTy)`, where `Tail Head` has the type `Elem '[BaseTy, BaseTy :-> BaseTy, BaseTy] ('BaseTy :-> 'BaseTy)`. The `Elem` value tells us where to find the type in the typing context, which in turn specifies how many new bound variables have been introduced since the variable we are interested in was bound, exactly like a de Bruijn index.

The `Lam` constructor corresponds to the abstraction typing judgement rule. Given a well-typed expression of type `result` in the context `arg:ctx`, we can abstract out the first variable of context into a bound variable with a lambda expression, producing a new term with the function type `arg :-> result` in the weakened context. We refer to the argument of the `Lam` constructor as the body of the lambda.

The `App` constructor corresponds directly to the application typing judgement rule, where we can apply one term to another if they share the same context `ctx` and the type of the second term matches the argument type `arg` of the first term.

Using this implementation for `Expr`, it is guaranteed that an expression of type `Expr ctx ty` is well-typed. In Chapter 4, we will see how `Expr`'s constructors are used to form new expressions guaranteed to be well-typed by their type.

Note that the apostrophes on type-level constructors are not required for successful compilation, as GHC can infer whether the constructor is type-level or value-level automatically.

# Chapter 4

# Normalising the Typed Lambda Calculus

In this chapter, we translate an Agda implementation of NbE developed by Andras Kovacs [5] for the STLC into Haskell.

## 4.1 Target Syntax

We construct set of the simply typed normal terms `NormalForm` by combining the approach seen for the untyped terms in Chapter 2 with the GADT syntax from Chapter 3 to ensure the inhabitants are well-typed.

```
data NormalForm :: [Ty] -> Ty -> * where
    NormalNeutral :: NeutralForm ctx ty -> NormalForm ctx ty
    NormalLam :: NormalForm (arg ': ctx) result -> NormalForm ctx (arg :-> result)

data NeutralForm :: [Ty] -> Ty -> * where
    NeutralVar :: Elem ctx ty -> NeutralForm ctx ty
    NeutralApp :: NeutralForm ctx (arg :-> result) -> NormalForm ctx arg
               -> NeutralForm ctx result
```

## 4.2 Order Preserving Embeddings

In the lambda case of `eval`, NbE evaluates the body of the lambda in a stronger context where a new bound variable is introduced (as seen in Chapter 2). To continue the evaluation of the body ...

```
data OPE :: [Ty] -> [Ty] -> * where
    Empty :: OPE '[] '[]
    Drop  :: OPE ctx1 ctx2 -> OPE (x : ctx1) ctx2
    Keep  :: OPE ctx1 ctx2 -> OPE (x : ctx1) (x : ctx2)
```

A value of type `OPE a b` is a proof that the list of types `b` is a subsequence of `a`. The three constructors correspond to the different ways of constructing such a proof. The `Empty` constructor corresponds to the trivial case that the empty list contains itself. Given a proof that `ctx2` is embedded in `ctx1`, the `Drop` constructor extends the proof to show that `ctx2` is embedded in `x:ctx1`, and the `Keep` constructor extends the proof to show that `x:ctx2` is embedded in `x:ctx1`. For example, `Drop (Keep Empty))` could have the (Haskell) type `OPE '[BaseTy :-> BaseTy, BaseTy] '[BaseTy]`.

## 4.3 Semantic Set

The normalisation function should return a `NormalForm` with the same type and typing context as the original term, so should have the type signature `Expr ctx ty` → `NormalForm ctx ty`. Because `reify` is a pure function of the semantic set `V`, this set must also be indexed by the typing context and type of the expression to preserve the typing information (as seen in the kind signature below).

```
data V :: [Ty] -> Ty -> * where
    Base :: NormalExpr ctx BaseTy -> V ctx BaseTy
    Function :: (forall ctx' .  OPE ctx' ctx -> V ctx' arg -> V ctx' result)
             -> V ctx (arg :-> result)
```

The semantic set for typed NbE is the key difference between the typed and untyped implementations of NbE this dissertation presents.

The `Base` constructor specifies that values of `V`with type `BaseTy` are normal forms. The `Function` constructor creates a value of function type `arg :-> result` given a function with type signature

```
forall ctx' . OPE ctx' ctx -> V ctx' arg -> V ctx' result
```

By default, in GHC, there is only have one "level" of polymorphism, where type variables are implicitly universally quantified. However, within the function type signature above, we want to treat `ctx` as fixed, and quantify over all larger contexts. To achieve this we use higher-ranked polymorphism with the explicit `forall` syntax to delay the quantification of `ctx'` until after `ctx` has been bound. Then, by treating `OPE` as a relation on typing contexts, we specify that `ctx'` must be larger than `ctx`. We can think of the `OPE` argument as a predicate on the quantification, in that the function first takes a proof that `ctx'` contains `ctx`. To enable higher ranked polymorphism, we use the `Rank2Types` extension. The result is that the `Function` constructor takes a function with partially applied type `V ctx' arg` → `V ctx' result` where `ctx'` is any contexr larger than the original context, and returns a semantic value in the original context with type `arg :-> result`

## 4.4 Evaluation

### 4.4.1 Environment

```
data Env :: [Ty] -> [Ty] -> * where
    EmptyEnv :: Env '[] ctxV
    ConsEnv  :: Env ctx ctxV -> V ctxV ty -> Env (ty : ctx) ctxV


envLookup :: Elem ctx ty -> Env ctx ctxV -> V ctxV ty
envLookup Head     (ConsEnv _    v) = v
envLookup (Tail n) (ConsEnv prev _) = envLookup n prev
```

### 4.4.2 Variable Case

```
eval :: Env ctx ctxV -> Expr ctx ty -> V ctxV ty
eval env (Var n) = envLookup n env
```

### 4.4.3 Lambda Case

```
eval (env :: Env ctx ctxV) (Lam (body :: Expr (arg:ctx) result)) = Function f
    where
        f :: (SingContext ctxV') => OPE ctxV' ctxV -> V ctxV' arg -> V ctxV' result
        f ope v = eval (ConsEnv (strengthenEnv ope env) v) body


strengthenElem :: OPE strong weak -> Elem weak ty -> Elem strong ty
strengthenElem Empty      v        = v
strengthenElem (Drop ope) v        = Tail (strengthenElem ope v)
strengthenElem (Keep ope) Head     = Head
strengthenElem (Keep ope) (Tail v) = Tail (strengthenElem ope v)

strengthenNormal :: OPE strong weak -> NormalExpr weak ty -> NormalExpr strong ty
strengthenNormal ope (NormalNeutral n) = NormalNeutral (strengthenNeutral ope n)
```

```
strengthenNormal ope (NormalLam n)     = NormalLam (strengthenNormal (Keep ope) n)

strengthenNeutral :: OPE strong weak -> NeutralExpr weak ty -> NeutralExpr strong ty
strengthenNeutral ope (NeutralVar n)   = NeutralVar (strengthenElem ope n)
strengthenNeutral ope (NeutralApp f a) = NeutralApp (strengthenNeutral ope f) (strengthenNormal ope a)

strengthenV :: (SingContext strong) => OPE strong weak -> V weak ty -> V strong ty
strengthenV ope                       (Base nf) = Base (strengthenNormal ope nf)
strengthenV (ope :: OPE strong weak) (Function (f :: forall strong . (SingContext strong) => OPE strong weal
    where
        f' :: (SingContext stronger) => OPE stronger strong -> V stronger arg -> V stronger result
        f' ope' = f (composeOPEs ope ope')

strengthenEnv :: (SingContext c) => OPE c b -> Env a b -> Env a c
strengthenEnv _   EmptyEnv          = EmptyEnv
strengthenEnv ope (ConsEnv tail v) = ConsEnv (strengthenEnv ope tail) (strengthenV ope v)
```

### 4.4.4 Application Case

```
eval env (App f a) = appV (eval env f) (eval env a)
    where
        appV (Function f') a' = f' (idOPEFromEnv env) a'

        idOPEFromEnv :: (SingContext ctxV) => Env ctx ctxV -> OPE ctxV ctxV
        idOPEFromEnv _ = idOpe


class SingContext ctx where
    idOpe :: OPE ctx ctx

instance SingContext '[] where
    idOpe = Empty

instance (SingContext xs, SingTy x) => SingContext (x:xs) where
    idOpe = Keep idOpe
```

## 4.5 Reification

```
reify :: V ctx ty -> NormalExpr ctx ty
reify (Base nf)     = nf
reify (Function f) = NormalLam (reify (f ope (evalNeutral (NeutralVar Head))))
    where
        ope = weakenContext (Function f)

        weakenContext :: (SingContext ctx) => V ctx ty -> OPE (x:ctx) ctx
        weakenContext _ = wk

evalNeutral :: (SingTy ty, SingContext ctx) => NeutralExpr ctx ty -> V ctx ty
evalNeutral = evalNeutral' singTy

evalNeutral' :: (SingContext ctx) => STy ty -> NeutralExpr ctx ty -> V ctx ty
evalNeutral' SBaseTy       n                                   = Base (NormalNeutral n)
evalNeutral' (SArrow _ _)  (n :: NeutralExpr ctx (arg :-> result)) = Function f
    where
        f :: (SingContext strongerCtx) => OPE strongerCtx ctx -> V strongerCtx arg -> V strongerCtx result
        f ope v = evalNeutral (NeutralApp (strengthenNeutral ope n) (reify v))


data STy :: Ty -> * where
    SBaseTy :: STy BaseTy
    SArrow  :: (SingTy a, SingTy b) => STy a -> STy b -> STy (a :-> b)
```

```
class SingTy a where
    singTy :: STy a

instance SingTy 'BaseTy where
    singTy = SBaseTy

instance (SingTy a, SingTy b) => SingTy (a :-> b) where
    singTy = SArrow singTy singTy
```

```
class SingContext ctx where
    idOpe :: OPE ctx ctx
    wk :: OPE (x:ctx) ctx
    wk = Drop idOpe
```

## 4.6 Normalisation

```
normalise :: (SingContext ctx) => Expr ctx ty -> NormalExpr ctx ty
normalise = reify . eval initialEnv
```

```
class SingContext ctx where
    idOpe :: OPE ctx ctx
    wk :: OPE (x:ctx) ctx
    wk = Drop idOpe
    initialEnv :: Env ctx ctx

instance SingContext '[] where
    idOpe = Empty
    initialEnv = EmptyEnv

instance (SingContext xs, SingTy x) => SingContext (x:xs) where
    idOpe = Keep idOpe
    initialEnv = ConsEnv (strengthenEnv wk initialEnv) (evalNeutral (NeutralVar Head))
```

## 4.7 Notes

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

Advantage over ADTs: type refinement by constructor

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) [7]

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

### 4.7.1 Solution: Order Preserving Embeddings (OPEs)

Following implementation in Agda [5], 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

### 4.7.2 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

### 4.7.3 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 [6]

### 4.7.4 Solution: Singleton pattern

Method of Type to value known as reflection [6]

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] Nate Foster. *Cornell University CS 4110 - Lecture 15*. URL: https://www.cs.cornell.edu/courses/cs4110/2018fa/lectures/lecture15.pdf.

[5] 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.

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

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

[8] 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.