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.

Tuesday 11th May, 2021

Declaration

Lucas O'Dowd-Jones, Tuesday 11th May, 2021

Contents

Executive Summary

My research hypothesis is that Haskell extensions are powerful enough to implement Normalisation by Evaluation for the Simply Typed Lambda Calculus.

The aims of this project are

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

In this project we

- Implement two variations of Normalisation by Evaluation for the untyped lambda calculus in Haskell, see Chapter 2, following mathematical specifications.
- Translate an Agda implementation of Normalisation by Evaluation into Haskell using compiler extensions, see Chapter 4, which demonstrates that the research hypothesis was correct.
- Evaluate the practical use of the features enabled by compiler extensions, and recommend their use in instances where type-safety is the primary priority, see Chapter 5.

Acknowledgements

Thanks to Alex Kavvos for his guidance, support and concrete examples. Thanks also to the programming languages research group for listening to my presentation, and offering useful insights.

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 terms of the lambda calculus (referred to as "syntax") as elements of a mathematical "semantic" set. NbE then "reifies" semantic values 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. NbE is useful for the following reasons.

1. Since the foundations of NbE are mathematical, we can prove that our implementation is correct and study its behaviour formally [5]. 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.

2. Some research suggests that NbE can improve the speed of compilation of functional languages [7].

3. Research is ongoing into whether dependent type theories such as Coq can use NbE to check for equality between dependent types by normalising type-level programs [2]

To become familiar with the general operation of NbE, in Chapter 2 we present two implementations for NbE of the untyped lambda calculus. The first implementation normalises the named lambda calculus. This implementation works, but its correctness depends on fresh name generation which makes it difficult to reason about. This issue motivates a second implementation of NbE, where we normalise a nameless representation of the lambda terms instead. Since variable names are not part of this syntax, fresh variables are much easier to generate.

Then we move to the central challenge of this project: translating an Agda implementation of NbE for the Simply Typed Lambda Calculus (STLC) into Haskell. The Agda implementation we follow makes use of advanced type-level features such as dependent types and dependent pattern matching. However, Agda is currently unsuitable for general-purpose programming for the following reasons

1. It is a type theory and proof assistant primarily, rather than a general-purpose programming language

2. It has a steep learning curve due to the theoretical understanding required to develop programs

3. The community supporting it is small and mainly academic

Haskell is a mature language suitable for industry use [CITE HASKELL INDUSTRY USAGE] as it strikes a balance between theory and practice, allowing developers to take advantage of type-safety without as much theoretical overhead. However, recent compiler options have enabled the use of advanced type-features more akin to dependently typed languages such as Agda.

Through implementing NbE for the STLC, this project will assess whether Haskell can be used to emulate features of languages with full dependent types. Members of the Haskell community are actively working on bringing full dependent types to Haskell [1], but this project serves as an evaluation of how well the existing tooling for complex types works in practice.

In Chapter 3, we use introduce and use Generalised Algebraic Datatypes (GADTs) in conjunction with the `DataKinds` and `PolyKinds` extensions to define terms that are well-typed by construction. In Chapter 4 we implement the well-typed normalisation function. For this 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 a method inspired by the singleton pattern [8].

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

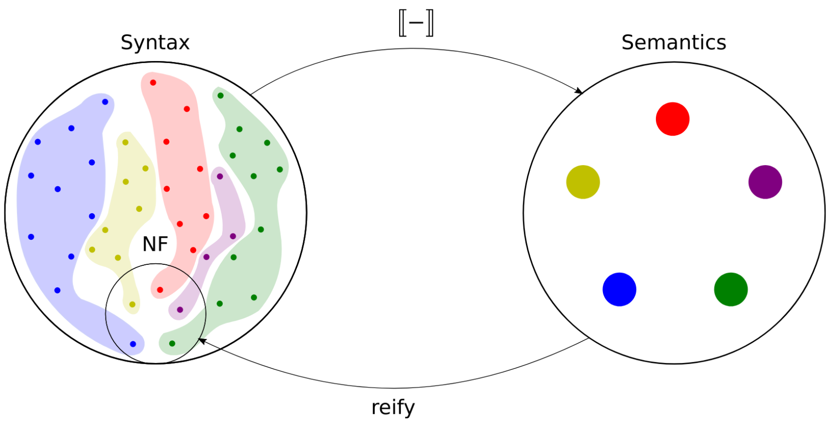

Figure 2.1: A visual overview of the NbE algorithm from [6]

NbE proceeds in two steps. The first is to evaluate terms of the lambda calculus into a semantic set. In Figure 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 composition of `eval` and `reify` yields the `normalise` function which maps terms to their normal forms.

Figure 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.

2.2 Gensym NbE

In this section we implement NbE for the untyped lambda calculus in Haskell, following a mathematical specification by Lindley [6]. We begin by representing the sets that NbE will operate on as data types in Haskell, namely the term syntax, set of normal forms and semantic set. Once these sets are defined, we implement the `eval` and `reify` functions which compose to give the `normalise` function.

2.2.1 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`.

2.2.2 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` → `V`. Lindley's specification [6] describes the semantic set as a mathematical function space, however since our implementation uses Haskell as the mathematics, semantic functions correspond to Haskell functions. The `NeutralV` data type represents base set of the recursive function space, whose elements are semantic variables and applications.

2.2.3 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 entry 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` $\to$ `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 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.

2.2.4 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 α-equivalence, we can choose any variable name that satisfies these requirements.

One approach to resolve this issue suggested by [6] 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 direct 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 fresh 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.

2.2.5 Normalisation

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.

Some terms such as $(\lambda x.xx)(\lambda x.xx)$ do not have a normal form. However, the `normalise` type signature guarantees that all expressions can be normalised to a normal form. The resolution to this paradox is that in cases where terms do not have a normal form, `normalise` will not terminate so will never produce a normal form. In particular, such expressions do not have a representation in the semantic state, so `eval` never terminates. For example, when evaluating $(\lambda x.xx)(\lambda x.xx)$, `eval` reaches a stage where it must evaluate the Haskell application `(\v -> app v v) (\v -> app v v)`. Following the definition of `app`, we find that this evaluates to the same expression, and thus will never finish evaluating. This is similar to the corresponding β-reduction, where $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$, except that in NbE this infinte contraction happens in the semantics.

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.

2.3 de Bruijn NbE

2.3.1 Motivation

While the Gensym approach works, generating fresh variables is a fragile procedure. The correctness of `normalise` depends on it, however mistakes in the implementation would be easy to make and difficult to notice. Another drawback of the Gensym approach is that `reify` is indexed by an infinite stream of variable names, which is difficult to reason about.

One approach to resolve the fresh variable problem is to remove variable names from the syntax altogether, by using an alternative representation of lambda calculus terms. 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 same purpose, so represent the same set of terms.

2.3.2 Introduction to de Bruijn 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. If we think of the set of bound variables at each point of the term as a stack, with the most recently bound variable at the top, then the de Bruijn index of a variable is the number of bindings you have to pop from the scope stack to reach the variable.

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 equivalent 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.

We now define NbE for the de Bruijn lambda calculus terms. Our implementation is inspired by an implementation by Andreas Abel [2], but diverges from it to improve type-safety.

2.3.3 Syntax

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
```

2.3.4 Semantics

We also redefine the semantic set for de Bruijn terms.

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

data NeutralV = NeVFreeVar Int
              | NeVBoundVar Int
              | NeVApp NeutralV V
```

Our implementation uses distinct constructors to distinguish between free and bound variables in the semantic set, which makes it easier to understand the operation of the code and improves type safety compared to Abel's implementation [2].

2.3.5 Evaluation

```
type Env = Map Int V
```

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

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

```
eval :: Env -> DbExpr -> V
eval env (DbVar x) = case lookup x env of
    Just y -> y
    Nothing -> Neutral (NeVFreeVar (x - size env))
```

In the variable case we lookup the semantic interpretation of bound variables from the environment.

In Abel's implementation, all the free variables are bound to semantic values in the initial environment before evaluation begins [2]. Since we assume that all free variables are present in the environment, in the `Nothing` case of the variable lookup, we would have to return `undefined`. Thus, the algorithm is only correct if the initial environment binds all free variables. Avoiding fragile pre-processing and implicit assumptions was the motivation for moving to de Bruijn notation, so we have modified Abel's implementation to improve type-safety.

Instead of assuming that all free variables are already bound in the environment, we generate their semantic interpretations as we evaluate the term. We start with an empty initial environment, so the `Nothing` case corresponds to evaluation of a free variable, as it did in the named implementation.

de Bruijn indices are not suitable for indexing variables in the semantic set, as they are dependent on the number of bindings in the context in which they occur. This context will change as redexes are contracted and lambda bindings removed by `eval`. We choose a semantic representation for free variables where all occurrences of the same free variable are identified with the same semantic value. `eval` returns a semantic variable with the index of same variable at the outermost context of the term. It calculates this by subtracting the number of bindings in the current context (`size env`) from the syntactic index `x`. Thus, every occurrence of the free variable will be identified with the same semantic value, regardless of where it occurs in the term.

```
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 to the named implementation is the construction of the new environment. We first increment all other indices 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 since the approach for contracting redexes is identical.

2.3.6 Reification

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

Reification for de Bruijn terms also follows a similar structure to the Gensym approach. However, instead of taking a stream of fresh variables, the de Bruijn `reify` function only needs the number of bound variables in the semantic value of type `V`. This is significantly easier to reason about than the Gensym `reify` function.

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

As in the untyped implementation, to reify a function `f` we return a lambda where the body of the term is the reification of `f` at a fresh variable.

We use de Bruijn levels to represent bound variables in the semantic set, as Abel does in his implementation. de Bruijn levels work in the opposite way to indices: the de Bruijn level 0 corresponds to the oldest bound variable in scope, rather than the newest.

Since `n` variables are bound in `f`, the de Bruijn levels which have already been bound are $0, 1, \ldots, n-1$. Thus, `NeVBoundVar n` is a fresh variable. Since we have introduced a new bound variable, we increment `n` by one when reifying the body of `f`.

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

Since the first `n` de Bruijn indices refer to bound variables, to reify a free variable, we increment the index of the variable at the outermost context of the term `k` by `n`. To reify a bound variable, we translate the de Bruijn level `k` back to its corresponding de Bruijn index `n - 1 - k`.

2.3.7 Normalisation

```
normalise :: DbExpr -> NormalForm
normalise = reify 0 . eval initialEnv where
    initialEnv = Map.empty
```

To `normalise` de Bruijn expressions, we first evaluate the term in the empty environment and then reify the expression which initially has no bound variables. As in the untyped implementation, we have the same guarantees and caveats for normalisation. However, we have achieved our goal of removing the need for a fresh name stream.

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.

```haskell
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.

```haskell
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` (no variable is well-typed if the typing context is empty). 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 Generalised Algebraic Datatypes (GADTs).

3.3 Introduction to Generalised Algebraic Datatypes

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 [3].

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

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.

[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. 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 nth element of the context corresponds to the type of the bound variable with de Bruijn index n.

The Var constructor corresponds to the variable typing judgement rule. Instead of indexing variables by names or de Bruijn indices as we saw in Chapter 2, typed variables are indexed by an Elem value.

A variable can only be a well-typed expression of type ty if ty is present in the typing context. The argument of type Elem ctx ty proves that ty is in the typing context ctx. Thus, it is impossible to construct a variable which is not well-typed.

From inspecting the constructors of Elem, we notice that all Elem values take the form

```
Tail ( ... (Tail Head) ... )
```

The Elem value with n tail constructors refers to nth most recently bound variable in the typing context, and hence to the variable with de Bruijn index n. 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). Here Tail Head refers to the bound variable with de Bruijn index 1 (since it has one Tail constructor), which we see from the context has type BaseTy :-> BaseTy.

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 ctx. 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. From this point we omit the apostrophe unless it ambiguous whether a constructor operates at the value-level or type-level.

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 the 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 NormalForm's 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

Following Kovacs' implementation, before we can define the semantic set we need a datatype to represent Order Preserving Embeddings (OPEs) on contexts.

```
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 of `OPE` correspond to the different ways of constructing such a proof. The `Empty` constructor corresponds to the trivial statement 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 type `OPE '[BaseTy :-> BaseTy, BaseTy] '[BaseTy]`.

From a programs as proof perspective, `OPE` is a binary relation on the set of contexts.

4.3 Semantic Set

In Andras Kovacs' implementation of NbE [5], the type of the semantic set is determined by the function Ty^{N}. This function takes the type and context of an expression as values, and returns the type of the

```
Tyᴺ : Ty → Con → Set
Tyᴺ ι       Γ = Nf Γ ι
Tyᴺ (A ⇒ B) Γ = ∀ {Δ} → OPE Δ Γ → Tyᴺ A Δ → Tyᴺ B Δ
```

Figure 4.1: Agda implementation of the typed semantic set from [5]

corresponding semantic set, where `Set` can be thought of as kind `*`. In Haskell we do not have a way to directly express a function which takes values and returns types. Instead we define the GADT `V` which represents the typed semantic set.

```
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)
```

`V` is indexed by a context of kind `[Ty]` and a type of kind `Ty`, which were value-level function arguments in Figure 4.1. `V` is necessarily indexed by these parameters at the type level since `normalise` should return a normal form with the same context and type as the given syntactic expression. `reify` is a pure function of the semantic set, so `V` must preserve this type-level information.

Ty^{N} pattern matches on the type argument and is defined in two cases: a base type case and a function case, where the Agda implementation uses ι for `BaseTy`. In our implementation the expression type is not given as a value-level argument that can be pattern matched on, so instead we use constructors to emulate a similar behaviour. From the return type of the constructors we see that values of `V` with type `BaseTy` can only be created by the `Base` constructor. Thus, if we are given a value of type `V ctx 'BaseTy`, GHC can deduce the value takes the form `Base n` where `n :: NormalExpr ctx Expr`. `n` is an inhabitant of the type returned by the Agda implementation in the base type case. Similarly, given a value of type `V ctx (A :-> B)`, GHC can deduce that the value must have the form `Function f`, since only the `Function` constructor can produce semantic values with function type. Again, we can extract a semantic value of the correct type since the type of `f` is exactly the type specified by the function type case of Ty^{N}. Thus, we have successfully mimicked the behaviour of Ty^{N} in Haskell.

The argument to the `Function` constructor has following the 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, as seen in Figure 4.1, we want to treat `ctx` as fixed, before quantifying over `ctx'`, where we use `ctx` instead of Γ and `ctx'` instead of Δ. 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. To enable higher ranked polymorphism, we use the `Rank2Types` extension.

The `OPE` parameter acts as a precondition requiring a proof that `ctx'` contains `ctx`, so intuitively this type quantifies over all contexts containing `ctx`.

4.4 Evaluation

In this section we implement the typed `eval` function.

4.4.1 Environment

As in the untyped implementation of NbE, during evaluation we need an environment to track bound variables and their associated semantic values. Since our variables have the same structure as de Bruijn indices, it suffices to use a list as the environment, where the nth element of the list corresponds to the variable with de Bruijn index n.

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

From the constructors, we see that the `Env` GADT has the same value-level structure as the standard Haskell list. However, we have used GADT syntax to enforce additional restrictions on the types of the elements in the environment. From the kind signature, we see that `Env` is indexed by two typing contexts. From the constructor types, we see that the first of these tracks the types of the semantic elements stored in the environment. The second typing context is the typing context shared by all semantic values stored in the environment. The motivation for indexing `Env` by these typing contexts is due to `envLookup` function defined below in 4.4.2.

4.4.2 Variable Case

Now that we have defined the environment, we can begin implementing the evaluation function `eval`.

```
eval :: Env ctx ctxV -> Expr ctx ty -> V ctxV ty
```

As before, we pattern match on each of the three expression constructors. We start with the variable case.

```
eval env (Var n) = envLookup n env
    where
        envLookup :: Elem ctx ty -> Env ctx ctxV -> V ctxV ty
        envLookup Head     (ConsEnv _    v) = v
        envLookup (Tail n) (ConsEnv tail _) = envLookup n tail
```

As in the untyped case, for variables we lookup the semantic value associated with the bound variable from the environment. Since variables are indexed by an `Elem` value, `envLookup` takes an `Elem` value and returns the associated semantic value.

The additional type information encoded in `Env` allows us to make guarantees about the lookup. Since `n` is a value of type `Elem ctx ty`, we know that `ty` exists somewhere in the list `ctx`. Since `Env` is also indexed by the same context `ctx`, which tracks the types of the semantic values in the environment, it is guaranteed that the semantic value at the corresponding de Bruijn index exists in the environment with type `ty`. Thus, `envLookup` doesn't have to handle the case where the variable is not in the environment like in the untyped implementation; the types guarantee that all variables possible in the context `ctx` are bound in the environment.

Since all semantic values in the environment have the same typing context `ctxV`, we can also be sure that the returned semantic value has the context `ctxV`.

Note that `envLookup` does not pattern match on the `EmptyEnv` case. The types ensure that this case is never required since `EmptyEnv` has type `Env '[] ctxV`, and there are no inhabitants of the type `Elem '[] x` for any `x`. Here `Elem ctx ty` acts like a pre-condition on `envLookup`.

4.4.3 Lambda Case

Before implementing the lambda case of `eval` we define the function `strengthenEnv` with type signature `OPE ctxV' ctxV -> Env ctx ctxV -> Env ctx ctxV'`. `strengthenEnv` takes an `OPE ctxV' ctxV` and an environment, and returns the same environment where each semantic value has the expanded context `ctxV'`. To implement `strengthenEnv`, we first build a series of functions for strengthening each constituent part of a semantic value.

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

First we strengthen `Elem` values which index variables. We use the naming convention `OPE strong weak` since `strong` contains at least as many bindings as `weak`. We can perform this context strengthening since if `ty` is an element of the context `weak`, then it is also an element of any context `strong` containing `weak`.

```
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 n) = NeutralApp (strengthenNeutral ope f)
                                                    (strengthenNormal ope n)
```

`strengthenNeutral` and `strengthenNormal` strengthen entire expressions. Intuitively any well-typed expression in one context should also be well-typed in any larger context containing original, since adding variable bindings will not invalidate existing type judgements. `strengthenNormal` essentially encodes the weakening property for the normal form subset of the STLC.

In the `NormalLam` case, type refinement guarantees that the context for the body of the lambda binds an additional variable. Hence, we have to expand the given OPE with an additional `Keep`. In the `strengthenNeutral` case we use `strengthenElem` to point variables to their updated de Bruijn indices in the expanded context.

```
strengthenV :: OPE strong weak -> V weak ty -> V strong ty
strengthenV ope (Base nf)     = Base (strengthenNormal ope nf)
strengthenV ope (Function f) = Function f'
    where
        f' ope' v = f (composeOPEs ope ope') v

composeOPEs :: OPE b c -> OPE a b -> OPE a c
composeOPEs v        Empty    = v
composeOPEs v        (Drop u) = Drop (composeOPEs v u)
composeOPEs (Drop v) (Keep u) = Drop (composeOPEs v u)
composeOPEs (Keep v) (Keep u) = Keep (composeOPEs v u)
```

`strengthenV` takes an `OPE` and a value of the semantic set, and strengthens the semantic value according to the given `OPE`.

If `ope' :: strongest strong` and `v :: V strongest arg`, then to satisfy the type restrictions of the semantic set in the `Function` case, we need to apply `v` to `f` with an `OPE` of type `OPE strongest weak`. We produce such an `OPE` using the `composeOPEs` function, which implements transitivity for two `OPE`s.

However, GHC automatically infers that `f' :: OPE Any strong -> V Any arg -> V Any result`, which leads to a failure to compile at type-checking. This error occurs since all the contexts which GHC has inferred as `Any` must be the same to satisfy type-checking, and since values of type `Any` can be different, GHC cannot deduce necessary type equalities between these contexts.

```
strengthenV :: 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 . OPE strong weak -> V strong arg -> V strong result))
    = Function f'
      where
          f' :: OPE strongest strong -> V strongest arg -> V strongest result
          f' ope' = f (composeOPEs ope ope')
```

To resolve this issue we use type annotations, which are enabled by the `ScopedTypeVariables` extension. `ope :: OPE strong weak` binds the type variables `strong` and `weak` in the body of the case. We also bind the type variables for the constructor argument `f`. We can then use these bound variables to specify the type of `f'`, where we explicitly declare that the contexts inferred as `Any` must be the same context `strongest`. With this additional type information, type-checking succeeds and the program compiles.

```
strengthenEnv :: OPE ctxV' ctxV -> Env ctx ctxV -> Env ctx ctxV'
strengthenEnv _   EmptyEnv         = EmptyEnv
strengthenEnv ope (ConsEnv tail v) = ConsEnv (strengthenEnv ope tail) (strengthenV ope v)
```

`strengthenEnv` maps over the elements of the environment, strengthening each one in turn with the `strengthenV` function.

Now we are ready to implement the lambda case of the `eval` function.

```
eval env (Lam body) = Function f
    where
        f ope v = eval (ConsEnv (strengthenEnv ope env) v) body
```

As in the untyped implementation, in the lambda case `eval` returns a semantic function that maps from a semantic argument `v` to the evaluated body of the lambda. We evaluate the body in an updated environment where we have bound the variable `Var Head` to `v`. However, if `ope :: ctxV' ctxV` then `v` has the context `ctxV'` whereas all the elements of `env` have context `ctxV`, so we cannot immediately add `v` to `env`. Instead, we have to strengthen the contexts of all the elements of the `env` using the given `ope` argument and `strengthenEnv`, which yields the strengthened environment of type `Env ctx ctxV'`. Now the contexts of the elements of the modified environment match the context `ctxV'` of `v`, so we can add `v` to the modified environment.

As in the `Function` case of `strengthenV`, the type of `f` that GHC infers is

```
OPE Any ctxV -> V Any arg -> V Any result
```

which is too general and causes compilation to fail at type-checking.

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

Once we have annotated the argument types, type-checking and compilation succeed.

4.4.4 Application Case

```
idₑ : ∀ {Γ} → OPE Γ Γ
idₑ {•}     = •
idₑ {Γ , A} = keep (idₑ {Γ})
```

Figure 4.2: Agda implementation of the identity OPE from [5]

Before implementing the lambda case of `eval`, we need to define the identity `OPE` function from Figure 4.2 in Haskell. This function pattern matches on the type-level context Γ and produces the identity `OPE` proving that Γ is contained in Γ. This is known as a dependent pattern match, as the result of the function is dependent of the type of its arguments. In Haskell, standard functions cannot pattern match on types, so we use an alternative construction, inspired by the singleton pattern [8], to pattern match at the type level.

```
class SingContext ctx where
    idOpe :: OPE ctx ctx

instance SingContext '[] where
    idOpe = Empty

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

We create a class `SingContext` with a single method `idOpe`, which will return the identity `OPE` for the context `ctx`. We then use class instances to pattern match on the type of `ctx`. In the empty context case, `idOpe` returns the `Empty` `OPE`. In the `x:xs` case, we require that the type `xs` has a `SingContext` instance which grants us the ability to use the `idOpe :: OPE xs xs` function in the instance definition. Applying the `OPE` for the tail of the list to the `Keep` constructor produces the identity `OPE` for the whole list.

The class instances essentially form a map from types to implementations of `idOpe`. Since `ctx` is a type variable that might not be known until runtime, the process of resolving the correct implementation to use could take place at runtime. This is surprising given that without extensions, GHC erases all type information during compilation.

We are now ready to define the application case of the `eval` function

```
eval env (App f n) = appV (eval env f) (eval env n)
    where
        appV (Function f') n' = f' (idOPEFromEnv env) n'
        idOPEFromEnv :: (SingContext ctxV) => Env ctx ctxV -> OPE ctxV ctxV
        idOPEFromEnv _ = idOpe
```

We first evaluate both terms of the application into the semantics, and then perform application at the semantic level using the `appV` function.

From the definition of `Function`, we can evaluate our function with a semantic argument in any context stronger than `ctxV`, where the `OPE` argument is a witness to the stronger context. Since `n'` has the same context `ctxV` as `Function f'`, we evaluate the semantic function `f'` at `ctxV` by choosing the `OPE` of type `OPE ctxV ctxV`. We produce such a value using the `idOPEFromEnv` function. This function takes an environment but throws away the value since the only information needed to produce the `OPE` is the type-level semantic context `ctxV`. The `SingContext ctxV` constraint awards us the `idOPE :: OPE ctxV ctxV` function, which handles the type level pattern match on `ctxV` and produces the identity `OPE` of `ctxV`.

To ensure that `ctxV` is an element of the `SingContext` class, we add the same constraint to `eval`'s type signature to give `eval :: (SingContext ctxV) => Env ctx ctxV -> Expr ctx ty -> V ctxV ty`. Without this additional constraint, GHC cannot be sure that `ctxV` is a member of the `SingContext` class, so the constraint on `idOPEFromEnv` is not satisfied, and compilation fails. However, any function that uses `eval` now also has to prove that `ctxV` has a `SingContext` instance to satisfy `eval`'s constraint. By adding class constraints, we implicitly propagate this guarantee back through the call-stack. Eventually we reach a point where the `ctxV` variable is instantiated with a concrete type that GHC can infer has a `SingContext` instance, which it can't do for arbitrary type variables like `ctxV`. This adds boilerplate code to type signatures, but is necessary to ensure type-checking succeeds and does not impact the implementation of the functions themselves. For successful compilation we would need to retrofit previously defined code with the necessary class constraints, but since these constraints do not affect the operation of functions, we omit them for brevity.

Since it is guaranteed by the `Expr` GADT that `f` has a function type, it is guaranteed by `eval` that its semantic interpretation `eval env f` also has a function type. Thus, we only need to define the `Function` case of `appV`, since `Function` is the only constructor of `V` that can produce a semantic value of function type. In fact, when the `appV (Base v) a = ...` case is included, GHC detects that the pattern match is redundant and issues an `Inaccessible code` warning at compile time.

4.5 Reification

In this section we implement the `reify` function, which takes elements of the semantic set to their canonical well-typed normal forms.

```
    reify :: V ctx ty -> NormalForm ctx ty
    reify (Base nf) = nf
```

Since semantic values with type `BaseTy` are normal forms, `reify` returns the corresponding normal form `nf`.

```
uᴺ : ∀ {A Γ} → Ne Γ A → Tyᴺ A Γ
uᴺ {ι}       n = ne n
uᴺ {A ⇒ B} n = λ σ aᴺ → uᴺ (app (Neₑ σ n) (qᴺ aᴺ))
```

Figure 4.3: Agda implementation of `etaExpand` from [5]

It would be desirable to normalise expressions into η-expanded normal form, where all the arguments of expressions with function type are abstracted. `etaExpand` is the function responsible for evaluating syntactic expressions into η-expanded semantic expressions. To implement `etaExpand` we follow the Agda definition in Figure 4.3, where we use `etaExpand` for u^{N} and `reify` for q^{N}. u^{N} pattern matches on the type `A` at the type-level. Since this is not possible directly in Haskell, we use the same class instance trick from the application case of `eval`.

```
    class SingTy ty where
        etaExpand :: NeutralForm ctx ty -> V ctx ty
```

```
instance SingTy 'BaseTy where
    etaExpand n = Base (NormalNeutral n)

instance (SingTy result) => SingTy (arg :-> result) where
    etaExpand (n :: NeutralForm ctx (arg :-> result)) = Function f
        where
            f :: OPE ctx' ctx -> V ctx' arg -> V ctx' result
            f ope v = etaExpand (NeutralApp (strengthenNeutral ope n) (reify v))
```

We create a class `SingTy` with a single function `etaExpand`, where the type of the neutral form is bound to the type `ty` parameterising the class. The `'BaseTy` instance is used when the argument `n` is an expression of type `BaseTy`. In the body of the instance, GHC infers that `n :: NeutralForm ctx 'BaseTy`, which allows us to apply the `Base` constructor. The `:->` instance is used when `n` has a function type. In this case we use type annotations again since the automatically inferred type of `f` is too general for successful compilation. This annotation also shows us that in the function case, GHC is able to infer that `n :: NeutralForm ctx (arg :-> result)`. In this instance, `etaExpand` returns a function, which will eventually be reified into a lambda. However, the body of this function may itself may require η-expansion, depending on the type `arg` of the argument `v`. Since the `NeutralApp` constructor creates a neutral value of type `result`, `etaExpand` requires that `result` also has a `SingTy` instance in the instance declaration.

In the untyped implementation of NbE, evaluation and reification take place in two distinct phases. However, from the definition of `etaExpand` we see that in the typed implementation, expressions are bounced back and forth between syntactic and semantic representations. This makes the implementation more difficult to reason about, but is a necessary trade-off to achieve the additional type security and features such as η-expansion that the typed implementation provides.

```
class SingContext ctx where
    idOpe :: OPE ctx ctx
    bindOpe :: OPE (arg:ctx) ctx
    bindOpe = Drop idOpe
```

We extend the existing class `SingContext` to include `bindOpe`, which produces an `OPE` from a given context `ctx` to one where we have bound a new variable `arg:ctx`. Since `bindOpe` is defined the same way in both the `Empty` and `(:)` cases, we set a default implementation in the class definition and leave the individual instances unaltered.

```
reify (Function f) = NormalLam (reify (f extendedOpe boundVar))
    where
        boundVar = etaExpand (NeutralVar Head)

        extendedOpe = extendOpe (Function f)

        extendOpe :: (SingContext ctx) => V ctx ty -> OPE (arg:ctx) ctx
        extendOpe _ = bindOpe
```

To reify a function `f :: OPE ctx' ctx -> V ctx' arg -> V ctx' result`, we evaluate the `f` at a semantic bound variable, and reify the result (which represents the body of the function `f`). `boundVar` has the type `V (arg:ctx) arg`, so we use the `extendedOpe` function to produce an `OPE` for a context where we have bound an additional variable of type `arg` in the context. All instances of the argument in the body of `f` are replaced with `etaExpand (NeutralVar Head)`. From the definition of `etaExpand`, we see that if `NeutralVar Head` has type `BaseTy`, `etaExpand` can immediately evaluate the variable into the semantic set. On the other hand, if `NeutralVar Head` has a function type, `etaExpand` evaluates to a function which waits for an argument to be supplied before continuing evaluation of the variable.

The other section of the program where we evaluate a semantic function `f` is the application case of `eval` (see Section 4.4.4). These cases motivate the use of `OPE`s, since in the application case we need to evaluate `f` in the same context as the semantic value, whereas in this case we need to evaluate `f` in an extended context where an additional bound variable is introduced. Whilst `OPE`s are responsible for a significant amount of the additional work in implementing well-typed normaliation, they are an elegant solution to evaluating semantic functions in different contexts.

4.6 Normalisation

```
uᶜᴺ : ∀ {Γ} → Conᴺ Γ Γ
uᶜᴺ {•}     = •
uᶜᴺ {Γ , A} = Conᴺₑ wk uᶜᴺ , uᴺ (var vz)
```

Figure 4.4: Agda implementation of **initialEnv** from [5]

Before we define `normalise`, we need to specify the initial environment to evaluate our expression in. We want `normalise` to return a normal form with the same context as the original term. Since `eval :: Env ctx ctxV -> Expr ctx ty -> V ctxV ty` and `reify :: V ctxV ty -> NormalForm ctxV ty`, we can achieve this by evaluating with an environment of type `Env ctx ctx`. The initial environment is a function of the context of the given expression, as seen in Figure 4.4, where $u^{c\ N}$ is the Agda function that produces the initial environment.

```
class SingContext ctx where
    idOpe :: OPE ctx ctx
    bindOpe :: OPE (x:ctx) ctx
    bindOpe = 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 bindOpe initialEnv) (etaExpand (NeutralVar Head))
```

We emulate the type-level pattern match by further extending the `SingContext` class with the `initialEnv` function. For each element in the context, we add a new semantic variable to the environment, strengthening the contexts of all the semantic elements in the environment accordingly using `strengthenEnv bindOpe`. This allows us to normalise expressions with free variables where the type of each free variable is given in the context. `etaExpand` requires that `x` has a `SingTy` instance, so we implicitly pass this guarantee by adding a `SingTy` constraint to the instance.

Grouping functions under a single type class like this could be problematic for more complicated applications, as it may lead to an accumulation of class constraints. For example, consider the following `hypothetical` function.

```
hypothetical :: (SingContext ctx) => Expr ctx ty -> OPE (ty:ctx) (ty:ctx)
hypothetical _ = idOpe
```

Even through `idOpe` only depends on the length of a context rather than its contents, to use the `idOpe` implementation for `ty:ctx`, the `(:)` instance for `SingContext` requires that `ty` is a member of `SingTy`. Thus, `hypothetical` does not type-check and compilation fails. In cases like these it would be necessary to split the type-level pattern-matching functions across distinct classes, to avoid the accumulation of constraints on a single class instance. This could lead to significant amounts of boilerplate in applications using classes for type-level pattern matching more frequently. In our implementation, adding this constraint does not prevent other functions from type-checking.

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

The type signature of `normalise` is the motivation for all the additional type-level restrictions in this implementation. Given a well-typed expression, `normalise` is guaranteed by the type system to return a well-typed normal form with the same context and type as the original expression. The type information in our implementation acts as a proof of this property, which is verified at compile time by the type-checker.

To normalise an expression, the user must specify a concrete context and type for the expression using type annotations. This instantiates type variables which enables polymorphic functions to resolve to specific instances. For example,

```
normalise (Lam (Var Head) :: Expr '[] (BaseTy :-> BaseTy))
    = Lam (Var 0)
```

whereas

```
normalise (Lam (Var Head) :: Expr '[] ((BaseTy :-> BaseTy) :-> BaseTy :-> BaseTy))
    = Lam (Lam (App (Var 1) (Var 0)))
```

where the returned terms are expressed in de Bruijn syntax for brevity.

This implementation returns expressions in η-expaned form as well as β-normal form, so different type annotations can lead to different terms even if the syntax is the same.

Below we attempt to normalise a term with a polymorphic type signature.

```
normalise (Lam (Var Head) :: Expr '[] (a :-> a))
```

However, this expression cannot be normalised and GHC produces an error since `a` is a type variable. Without a concrete type, GHC is unable to resolve instances of polymorphic functions such as `etaExpand` at runtime.

Whilst it is guaranteed that the returned expression of `normalise` is normal form, it is not guaranteed that it is the correct normal form for the given expression. Andras Kovacs goes further in his paper and formulates a machine-checked proof of correctness for his implementation of NbE in Agda [5]. It would be interesting to see if this too can be translated into Haskell.

Chapter 5

Critical Evaluation

5.1 Testing

In all our implementations, the type system ensures that `normalise` returns a normal form, however it is not guaranteed that it is the correct normal form for the given term. Although producing such a guarantee is out of the scope of this project, in this section we validate our implementations with unit tests.

We discuss the testing of the STLC NbE implementation, but testing for the de Bruijn and Gensym NbE implementations can be easily derived by removing the types and changing the term syntax of our tests. Testing for all three implementations is included in the full source code.

To test `normalise`, we want to generate large, well-typed expressions with many redexes to contract. We choose the Church numerals and operations on them as it is easy to construct large terms with predictable normal forms. The Church numerals are a representation of the natural numbers in the lambda calculus of the form $\lambda fx.f(f \ldots (fx) \ldots)$, where the numeral n is encoded by n applications of f [10].

```
type ChurchNumeralTy = ((BaseTy :-> BaseTy) :-> BaseTy :-> BaseTy)

type ChurchNumeral = Expr '[] ChurchNumeralTy
```

We define the type `ChurchNumeralTy` as the concrete type of Church numerals in the `Expr` syntax, where `BaseTy :-> BaseTy` is the type of the function f, and `BaseTy` is the type of its argument `x`. Instead of `BaseTy`, we could have used any type as the encoding only depends on the structure of the term. However, in our implementation to satisfy `SingTy` constraints we must choose a concrete type rather than a type variable. It makes sense to use the simplest type possible to avoid additional complexity. We also have to choose a concrete context to satisfy the `SingContext` constraints. Since Church numerals contain no free variables, we specify that our encoding has an empty initial context in the definition of the type synonym `ChurchNumeral`.

```
app2 :: Expr ctx (a :-> b :-> c) -> Expr ctx a -> Expr ctx b -> Expr ctx c
app2 f x y = App (App f x) y

toChurchNumeral :: Int -> ChurchNumeral
toChurchNumeral 0 = Lam (Lam (Var Head))
toChurchNumeral i = App churchSucc (toChurchNumeral (i - 1))
    where
        churchSucc :: Expr '[] (ChurchNumeralTy :-> ChurchNumeralTy)
        churchSucc = Lam (Lam (Lam (App f (app2 n f x))))

        n = Var (Tail (Tail Head))
        f = Var (Tail Head)
        x = Var Head
```

We use the `toChurchNumeral` function to create an expression that takes an `Int` `i` and produces an expression which will normalise to `i`th Church numeral. It does this by repeatedly applying the successor term `churchSucc` in the syntax. `normalise` will contract these redexes to produce a Church numeral, since

Church numerals are already in normal form.

For example, `toChurchNumeral 2` produces the following complicated expression.

```
App (Lam (Lam (Lam (App (Var 1) (App (App (Var 2) (Var 1)) (Var 0))))))
(App (Lam (Lam (Lam (App (Var 1) (App (App (Var 2) (Var 1)) (Var 0))))))
(Lam (Lam (Var 0))))
```

where we have replaced `Elem` variables with de Bruijn indices for brevity. However, when we normalise the above term we get

```
Lam (Lam (App (Var 1) (App (Var 1) (Var 0))))
```

which we recognise as the Church numeral for 2, $\lambda fx.f(fx)$. This is encouraging evidence that `normalise` is correct. Since we can generate Church numerals, we define operations on them to build more complex terms.

```
addChurchNumeral :: ChurchNumeral -> ChurchNumeral -> ChurchNumeral
addChurchNumeral = app2 churchAdd
    where
        churchAdd :: ClosedExpr (ChurchNumeralTy :-> ChurchNumeralTy :-> ChurchNumeralTy)
        churchAdd = Lam (Lam (Lam (Lam (app2 m f (app2 n f x)))))

        m = Var (Tail (Tail (Tail Head)))
        n = Var (Tail (Tail Head))
        f = Var (Tail Head)
        x = Var Head
```

The `addChurchNumeral` function applies the syntactic Church addition combinator `churchAdd` to two Church numerals. To verify that `normalise` is working correctly, we can verify that addition in the Church encoding using `normalise` matches standard addition.

```
prop_add :: Int -> Int -> Bool
prop_add m n = normaliseDB (addChurchNumeral (toChurchNumeral m) (toChurchNumeral n))
            == normaliseDB (toChurchNumeral (m + n))
```

`prop_add` represents the desired property that for any two natural numbers `n` and `m`, adding their Church numerals and normalising the result yields the same expression as adding the integers and normalising the Church numeral. This function returns `True` for all non-negative values of `n` and `m` we've checked, which is strong evidence that `normalise` is producing the correct normal form. To automate the checking of this property we could use the `QuickCheck` package. Notice that we use `normaliseDB` in `prop_add`, which first normalises the expression as usual using `normalise`, but then converts it into a de Bruijn expression. We do this because `Expr` is not part of the `Eq` class, for reasons we explore in Section 5.2.1

We developed a similar property for Church multiplication. Additionally, we verified that `normalise` can evaluate expressions constructed using the Church boolean encoding.

5.2 Evaluation of Language Features

In this section we evaluate how well Haskell's advanced language features worked during the implementation of NbE for the STLC. We also generalise our experiences to consider whether these features would be useful for general-purpose programming in Haskell, and what issues could prevent their use.

5.2.1 GADTs

We found that `GADTs` is an excellent extension for emulating dependent types when combined with `PolyKinds`. GADT syntax allowed us to define types such as `Expr` that would not be possible in standard Haskell. It also enabled us to translate Agda `data` definitions such as `OPE` and type generating functions such as Ty^{N}, that otherwise could not be expressed in Haskell.

The NbE algorithm highlighted the mathematical rigour that GADTs allow us to encode in Haskell, such as proving the weakening property of the STLC using `strengthenNormal`, the transitivity of `OPE`s using `composeOPEs`, and the subject reduction property of `normalise`. We can be confident in the correctness of our implementation thanks to the programs as proofs perspective GADTs has awarded us. In cases where definitions were not correct, the types were not satisfied, providing a stronger layer of security against errors than standard Haskell. For example, modifying any of the cases in `strengthenElem` creates a type error.

However, during the analysis and testing of our implementation, we found that GADTs indexed by types can create difficulties for deriving useful class instances. For example, it would be desirable for `NormalForm` to be a member of the `Eq` class, so that during testing we could directly check if two normal forms are equal (see Section 5.1).

```
instance Eq (NeutralForm ctx ty) where
    (NeutralVar n) == (NeutralVar m) = n == m
    (==) (NeutralApp (n :: NeutralForm ctx (arg1 :-> ty)) m)
         (NeutralApp (x :: NeutralForm ctx (arg2 :-> ty)) y)
          = n == x && m == y
    _ == _ = False
```

Here we have attempted to implement an `Eq` instance for `NeutralForm`, which is needed to determine equality between normal forms. The variable case doesn't pose any problems, as GHC can automatically derive an `Eq` instance for `Elem`. However, in the application case we cannot be sure that `arg1` and `arg2` are equal, since they are universally quantified type variables. Because of this, `n` and `x` may have different types, and so we cannot determine equality between them. Thus, the instance fails to compile.

We attempted to benchmark the typed NbE implementation, to evaluate how much performance overhead the additional type information incurred compared to the untyped implementations. However, to use Haskell's benchmarking features, we need a `NormalForm` instance for the `NFData` class. As `NormalForm` is a GADT, GHC is unable to automatically derive the instance. This further exemplifies how the class limitation makes GADTs more difficult to work with, and it would be interesting to explore whether this limitation could be overcome.

In general, we believe that GADTs are a useful tool for Haskell developers to improve the type-safety of their programs by taking advantage of the programs as proof paradigm. However, the instance deriving limitation shows that the extension is not completely mature yet, and in cases where type-safety is not paramount, the additional effort GADTs require may outweigh the reward.

5.2.2 Dependent Pattern Matching

For our implementation, the class instance trick for performing dependent pattern matching on types worked well. Passing singleton class constraints through the program did add syntactic noise and required some thought, but passing constraints through function type signatures had no effect on the function implementations.

However, in some situations we were forced to add type constraints to data types themselves. For example, we had to add a `SingTy arg` constraint to the `Lam` constructor. This is more concerning, as the implementation guided the structure of the data. For our project this was not an issue, but such a constraint could have implications on other functions, which reduces the code's modularity. For example, any function which produces a `Lam` value would have to ensure the argument type is a member of `SingTy`, even though this constraint is not relevant for the expression itself.

Our boilerplate overhead was small, since we only needed dependent pattern matches for a few functions. However, as noted in Section 4.6 for more complex applications with many dependent pattern matches, this boilerplate could become more difficult to manage. The boilerplate was also compact since our implementation only required pattern matches on the structure of types. For example, `idOpe` only depended on the length of the type-level context, so there was no need to identify its contents. This meant we only used a portion of the singleton pattern, which can be used to extract all the type-level information into the value level, at the cost of additional boilerplate.

Richard Eisenberg's proposals for built-in dependent pattern matches using a syntax to similar Agda would be a much cleaner approach than `GADTs`, but the current solution for managing dependent pattern matches is the `singleton` library [9]. This library uses template Haskell to automatically generate the instances and data types needed for "faking" dependent pattern matches, and would be useful for dependent pattern matching at scale.

5.2.3 Arbitrary-Rank Polymorphism

When defining the `Function` constructor of the STLC semantic set, we used explicit `forall` syntax to implement nested polymorphism. Whilst this satisfied most type requirements, when constructing concrete `Function` elements, GHC could not infer necessary type variable equalities to complete type-checking. We resolved this issue by using type annotations to bind type variables in function arguments.

An interesting area for further study could be to identify why type inference was not able to derive the type variable equalities in these cases. We suspect that it could be due to the higher ranked polymorphism, as `Function` was the only definition in our implementation that made use of the `RankNTypes` extension. If this is the case, then the `RankNTypes` extension can reduce type-safety by introducing `Any` types, which can be satisfied with incorrect implementations.

Chapter 6

Conclusion

6.1 Summary

In this dissertation we produced three variations of NbE. The first normalised the untyped lambda calculus using the Gensym approach, following an implementation by Lindley [6]. Issues with fresh variable names inspired a second implementation for untyped de Bruijn terms which we adapted from a specification by Abel [2]. GADTs were used to define the STLC in Haskell, before successfully translating an implementation of NbE for the SLTC by Kovacs [5] from Agda to Haskell. In the evaluation, we verified that that our implementations are correct, and found that GHC compiler extensions are powerful enough to emulate some features of dependently typed languages, albeit with practical drawbacks to their use.

6.2 Evaluation of Aims

We achieved both of our original aims, although there are areas for further study in both.

Our first aim was to produce various implementations of NbE in Haskell, which we achieved in Chapters 2 and 4 and verified in Section 5.1. This aim was intentionally left very open, since this was an exploratory project where it was difficult to predict the problems that would arise, and the amount of time each implementation would take. Each implementation served a valuable purpose. The Gensym implementation helped us understand the operation of NbE. The de Bruijn implementation was useful in deepening this understanding, which we evidenced by adding our own variations on the guiding implementation. The typed implementation allowed us to test advanced typing features, and presented challenges when Haskell fell short of Agda's features.

Our second aims was to explore how successful modern features of Haskell are in implementing an algorithm with complex types, which we discussed in Chapter 5. We saw that all the advanced features worked well at achieveing their specific goals, and made it possible to construct programs that would not be possible in standard Haskell. However, the recurring conclusion was that these features incur a significant amount of additional work in practice, and can lead to difficult problems. Thus, their use is hard to justify unless the type-security of the program is the priority. The release of Dependent Haskell [1] should address some of these issues by making dependent features a more central part of the language.

6.3 Areas for Further Study

As part of our second aim, we would have liked to benchmark NbE for the STLC against the untyped implementations to evaluate the performance trade-off of the additional type information. As mentioned in Section 5.2.1, we were unable to construct the necessary class instances to benchmark our normalisation, however this is an interesting problem that others are already researching [11]. If solved, it would also be interesting to benchmark the Haskell implementation of typed NbE against Kovac's Agda implementation, as a significant difference in performance could be compelling reason to choose one over the other in practice.

In Section 5.2.3, we suggested an investigation into whether the `RankNTypes` extension leads to overly general type inference.

In Section 4.6, we identified the additional challenge of translating Kovacs' full proof of correctness for the typed NbE implementation from Agda to Haskell. If possible, we would have extremely strong evidence that the Haskell `normalise` implementation produces the correct normal form.

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] 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] Stephanie Weirich Richard A. Eisenberg. "Dependently Typed Programming with Singletons". In: (). URL: https://richarde.dev/papers/2012/singletons/paper.pdf.

[9] Jan Stolarek Richard Eisenberg. *Singletons Package on Hackage*. URL: https://hackage.haskell.org/package/singletons.

[10] Mathias Ricken. *Comp 311 - Review 2*. URL: https://www.cs.rice.edu/~javaplt/311/Readings/supplemental.pdf.

[11] Koen Pauwels Georgios Karachalias Michiel Derhaeg Tom Schrijvers. "Bidirectional Type Class Instances". In: (). URL: https://arxiv.org/pdf/1906.12242.pdf.