TALLINN UNIVERSITY OF TECHNOLOGY
Institute of Cybernetics
Software Science Laboratory

Normalisation by evaluation for data types

Master's thesis

Student:	Hendrik Maarand
Student code:	122184IAPMM
Supervisor:	James Chapman

Tallinn
2014

Autorideklaratsioon

Deklareerin, et käesolev lõputöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud.

....	....
(kuupäev)	(lõputöö kaitsja allkiri)

Abstract

Normalisation is the process of finding the normal form of a term. This gives us a mechanism for deciding if two terms have the same meaning—we just need to check if their normal forms are equal. In this thesis we look at normalisation by evaluation, a semantic approach to normalisation, for simply typed lambda calculus augmented with natural numbers, lists, pairs, and streams. We give an implementation of the normaliser together with a proof of its correctness in the dependently typed programming language Agda.

Annotatsioon

Normaliseerimine on protsess termi normaalkuju leidmiseks. See annab meile meetodi otsustamaks, kas kaks termi on samatähenduslikud—me peame lihtsalt kontrollima, kas nende normaalkujud on võrdsed. Selles töös on uurimise all normaliseerimine läbi väärtustamise, mis on semantiline lähenemine normaliseerimisele. Meie objektkeeleks on lihtsalt tüübitud lambda-arvutus laiendatuna naturaalarvude, listide, paaride ning striimidega. Töö tulemusena me anname normaliseerija implementatsiooni koos tõestusega selle korrektsusest sõltuvate tüüpidega programmeerimiskeeles Agda.

Contents

1 Introduction

The main purpose of this thesis is to give an implementation of normalisation by evaluation for simply typed lambda calculus extended with natural numbers, lists, pairs, and streams, together with a proof of its correctness.

Normalisation is the process of simplifying terms (finding their normal forms) in a formal system. We want two terms to be considered equal (they have the same meaning) if and only if they have the same normal form. The process of obtaining a normal form may change the structure of the term, but it must keep the meaning of it. When the system is a programming language, the normaliser should therefore transform one program into another program with the same meaning. As two terms which have the same normal form have the same meaning, normal forms are a useful mechanism for deciding the equality of two terms. If a system is normalising, then every term is reducible to its normal form. If we are able to show that in a programming language every program has a normal form, then it means that our programs will not go wrong or loop infinitely.

Normalisation is a central component in proof assistants like Agda and Coq, which are based on intuitionistic type theory. There it is necessary to perform normalisation on the types, since in those systems types can contain values and therefore we need to perform computation to normalise the types. All this happens during type checking, before the programs are executed.

As an example of normalisation, we can think about arithmetic expressions. If we have an expression like $4 + 5$, for example, then we know from arithmetic, that it is the same as 9. There are also many other expressions which are essentially the same as 9. $1 + 8$ is one possible example. Somehow it seems that 9 is the best representative for this class of expressions, we cannot simplify it any further. We choose 9 to be the normal form of these expressions.

In this thesis we are going to implement normalisation using evaluation as the central component of the normaliser. Evaluation is the process of computing the value of an expression. For programs, it finds the meaning of a program—the value it evaluates to. To normalise a term, we first compute the value of the expression in some model. We then use an inverse of the evaluation function to extract the normal form of the original expression from the value we got in the previous step.

1.1 Lambda calculus

Lambda calculus was invented by Church in the 1930s as part of a general theory of functions and logic, intended to be a foundation for mathematics. Although this full system was shown to be inconsistent, the subsystem dealing with only functions became a successful model for the computable functions [5]. Lambda calculus is also the foundation of functional programming languages.

The simplest representation of lambda calculus is type free. This means that every expression (considered as a function) may be applied to every other expression (considered as an argument). A simple example is the identity function $I = \lambda x.x$, which may be applied to any argument x to give the same x as a result. Interestingly, I can also be applied to itself.

Lambda calculus has two basic operations, called application and abstraction. Application, which is usually written as $f\ a$, denotes f considered as an algorithm applied to a considered as an input. In a type free representation, it is allowed to have expressions like $f\ f$, which mean f is applied to itself. This is a useful method for simulating recursion inside lambda calculus.

Abstraction is a definition of an anonymous function. If m is an expression containing x, then $\lambda x.m[x]$ denotes the function which maps x to $m[x]$. The variable x does not need actually to occur in m, in which case $\lambda x.m[x]$ is a constant function with the value m.

The following is a simple example about application and abstraction:

$$(\lambda x.x^2 + 1)\ 3 = 3^2 + 1 = 10$$

$(\lambda x.x^2 + 1)\ 3$ is essentially the function $x \mapsto x^2 + 1$ applied to the argument 3 resulting in $3^2 + 1$. In general,

$$(\lambda x.m)\ n = m[n/x]$$

where $m[n/x]$ is the substitution of n for x in m. This is called β-conversion.

An abstraction binds the free variable x in m. For example, $\lambda x.y\ x$ has x as a bound variable and y as a free variable. It is assumed that bound variables in an expression are different from the free ones. This can be achieved by renaming the bound variables. For example, $\lambda x.x$ becomes $\lambda y.y$. These two expressions denote the same program. Two expressions which differ only by the names of the bound variables are considered the same. This is called α-conversion.

The notion of η-conversion is concerned with the idea of extensionality. Under η-conversion, every function f is equal to its expansion $\lambda x.f\ x$ whenever x does not occur free in f.

Simply typed lambda calculus requires each expression to be associated with a type. This is to avoid inconsistencies in the system. In the most basic version, there is a single base type ι and a single constructor $\sigma \to \tau$ which is used to denote the type of functions taking something of type σ to something of type τ.

Using a typed representation limits the number of programs we can write. In the case of application, $f\ a$, there is a strict rule for the term f, which must be of a function type and for the term a which must be of an appropriate type. If f is of type $\sigma \to \tau$ then it must be that a is of type σ. The result of the application is of type τ. In the abstraction case, if $m : \tau$ is an expression containing $x : \sigma$, then the lambda abstraction $\lambda x : \sigma.m[x]$ is of type $\sigma \to \tau$.

1.2 Normalisation by evaluation

One possible way of carrying out normalisation is by a stepwise reduction of terms. This requires a predefined set of reduction rules of the form $t \to t'$ to mean that t can be transformed to t' in a single step. The resulting term, after repeatedly applying these rules until no more rules apply, is called a normal form. This is known as reduction based normalisation. In this thesis we take another approach to normalisation—normalisation by evaluation.

Normalisation by evaluation is based on the idea, that a normal form of a term can be obtained by first interpreting (evaluating) the term in a suitable model and then writing a function “reify” (or “quote”) which maps an object in this model to a normal form representing it. The normalisation function is obtained by composing reify with the interpretation function [8]. This is a semantic approach to normalisation and it is sometimes called “reduction-free normalisation”.

In an informal notation, the normalisation function *norm* is given as the composition of *reify* and *eval*.

$$\mathit{norm}\ t = \mathit{reify} \circ \mathit{eval}\ t$$

Therefore, we require that *reify* is the left inverse of *eval*. The aim of the normalisation function is to pick a unique representative from each equivalence class. We are interested in the correctness of our normaliser, that it actually picks the right things for normal forms. The correctness is expressed in the following way.

$$t \sim t' \iff \mathit{norm}\ t \equiv \mathit{norm}\ t'$$

This means that two terms t and t' are provably equal if they are equal in the model. To prove this, we need to prove that the condition holds in both directions. For the *if* direction (soundness), we must first prove a lemma about evaluating convertible terms:

$$t \sim t' \implies \mathit{eval}\ t \equiv \mathit{eval}\ t'$$

For proving the converse direction (completeness), it is necessary to prove an underlying lemma, that a term is provably equal to its normal form:

$$t \sim \mathit{norm}\ t$$

Another property we are also interested in is the stability of our normaliser, that it preserves normal forms (for terms that are already normal forms).

$$n \equiv \mathit{norm}\ n$$

In this thesis we have developed a formally verified implementation of normalisation by evaluation for simply typed lambda calculus in the dependently typed programming language Agda [17].

1.3 Agda

Our goal is to write a normaliser for lambda calculus and to prove its correctness. One option would be to build the normaliser in any programming language and then use some external tools (annotating the source code with invariants, using a theorem prover) or perhaps pen and paper to prove properties about the program. Another option is to use a language with a more powerful type system, which allows us to encode invariants about our programs directly in the same environment. For the formalisation in this thesis, we have used the dependently typed programming language Agda.

In ordinary programming languages types and values are two entities which are quite clearly separated. In a dependently typed language they become more related: types can contain (depend on) arbitrary values. This makes it possible to encode properties of values as types, whose elements are proofs that the property is true. To prove a property, we must write a program of the required type. For it to be a consistent system, it is required for all programs to be total [14].

One common example for dependently typed programming is to define the data type of length-indexed lists, called vectors, and some operations on it. The type of vectors, `Vec A n` is a dependent type. `A` is the type of the elements contained in this vector. The type also contains a value `n` which represents the length of the vector. In Agda, vectors can be defined in the following way.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _,_ : ∀{m} → A → Vec A m → Vec A (suc m)
```

The type of this definition says that vectors are parametrized by a type `A`. The type of vectors of `A` is then `ℕ → Set`. All together, this means that vectors contain elements of type `A` and are indexed by natural numbers. The next two lines describe how one can create a vector: either you can create an empty vector `[]` (which has length zero) or you can take an element of type `A` and a vector containing `m` (which may be zero) elements of type `A` and as a result get a vector of length `suc m`.

In Agda, a parameter in curly braces is an implicit parameter. This means that we do not have to supply it when using the constructor, Agda tries to figure out the value itself. The underscore has several meanings. In the definition of vectors, it is used as a placeholder for the values required by the constructor. It can also be used in places where we do not care about the value of an expression at that point or we do not want to give a name to a variable.

Now that we can express the length of a list like data structure in its type, it would be very unfortunate if we would not be able to operate on the elements of this type without preserving this information. As an example operation, we will consider concatenating two vectors.

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

First, we have defined addition for natural numbers, since we will need to add together the length of two vectors when concatenating them.

```
_++_ : ∀{A m n} → Vec A m → Vec A n → Vec A (m + n)
```

```
[]      ++ b = b
(x , a) ++ b = x , (a ++ b)
```

The function `++` takes two vectors, one with length `m` and the other with length `n`, and produces a vector of length `m + n`. The actual definition of the function is identical to what we would have for lists, we do not have to do anything for this extra property to hold. We will now look at why this worked out so well.

In the first case, we are concatenating an empty vector to some other vector. We want the result to be the other vector, since adding an empty vector to something should not change it. The length of the resulting vector is therefore `n`, which is the length of the second argument. Due to the definition of `+`, `zero + n` and `n` are definitionally equal, meaning that Agda cannot distinguish between the two, we defined them to be equal, so we are done. For the second case, the length of the first vector is `suc m` and the length of the second is `n`. By the definition of `+`, `suc m + n = suc (m + n)`. Vector `a` has length `m` and we know by induction that `a ++ b` has length `m + n`. By the definition of `,` prepending an element to a vector of length `m` gives us a vector of length `suc m`, therefore the length of the result is `suc (m + n)`.

One important thing about functions in Agda is that they have to be structurally recursive. Every recursive call in the definition of a function must be on a structurally smaller argument. In the definition of `++` we can see that for the first case, this requirement is fulfilled, because there is no recursive call. In the second case, the first argument has one element fewer than the original call.

As said before, we would like to prove properties about our programs (which are written in Agda) in Agda. To conduct proofs, we are often interested in equality. For example, we would like to be able to prove properties like `m + 0 ≅ m`, which do not hold definitionally. In addition to definitional equality, we can also have propositional equality, which we use when we want to prove properties about our programs. To formulate a property, we need to express it in the type of an Agda program. Propositional equality can be defined in the following way.

```
data _≅_ {A : Set}(x : A) : A → Set where
  refl : x ≅ x
```

This may seem a bit strange. Surely, two things are equal if they are the same, but is this enough for proving non-trivial properties about programs? Actually, this is quite good, because we can define symmetry, transitivity and congruence using this definition. If we would declare these properties also as members of the data type, then working with the proofs would become much more tedious. In this thesis we will use the propositional equality defined in `Relation.Binary.HeterogeneousEquality` in Agda standard library. It is a more liberal version of the definition given above.

As an example, we will now prove that the empty vector is the right unit of vector concatenation. The empty vector is also the left unit of concatenation, but that holds definitionally.

```
++unitR : ∀{A m} (xs : Vec A m) → xs ++ [] ≅ xs
++unitR [] = refl
```

```
++unitR {m = suc m} (x , xs) = proof
  (x , xs) ++ []
  ≅⟨⟩
  x , (xs ++ [])
  ≅⟨ cong₂ (λ m' xs' → _,_ {m = m'} x xs') (+unitR m) (++unitR xs) ⟩
  x , xs
  ■
```

If the vector `xs` is actually empty, then the property holds definitionally. In the second case, the vector is not empty. The proof for it uses equational reasoning, which allows to write proofs with multiple steps in a nice way. The lines that begin with ≅ are justifications for those steps. The first step holds definitionally, and that is why we do not need to give any justifications. We do not actually need to write out that step, but it is here to make it clear what is happening.

For the next step, we need to show that two vectors are equal if the head elements are the same and the tails are almost the same, only on one side there is an empty vector appended to it. Notice that this is actually the same property we are currently proving, just on a smaller vector. This means we already have a proof of it using the inductive hypothesis.

To show that the full vectors are equivalent, we use `cong` which is a construct to show that if `x ≅ x'` then `f x ≅ f x'` for any function `f`. We would like to write

```
cong (λ xs' → x , xs') (++unitR xs)
```

to mean that `λ xs' → x , xs'` produces equal results when applied to equal arguments. In this case, the arguments would be equal by the inductive hypothesis. However, this will not work, as on the left hand side the length of the tail is `m + zero` and on the other side it is just `m`. Therefore, we need to abstract over two things: the tail of the vector and the length of the tail. To show that `m + zero ≅ m` we use `+unitR` which exactly states that `zero` is the right unit of addition.

```
+unitR : (k : ℕ) → k + zero ≅ k
+unitR zero    = refl
+unitR (suc k) = cong suc (+unitR k)
```

1.4 Relative monads

The design of the formalisation (the program and the proof) in this thesis is guided by the idea of relative monads [3]. The structure of the components is chosen to highlight the relative monadic properties of the evaluator. In this section we will look at relative monads, some background category theory, and how to formalise them in Agda. The Agda definitions are based on the formalisation of relative monads [4].

A monad is a structure known from category theory. Before defining monads we need to define the underlying structures. A category is an algebraic structure that consists of objects and morphisms. It has two basic properties: morphisms can be composed and there exists an identity morphism for every object. The composition of morphisms is associative and the identity morphism is the left and right unit of composition. In Agda, this can be defined as a record.

```
record Cat : Set where
  field Obj  : Set
        Hom  : Obj → Obj → Set
        iden : ∀{X} → Hom X X
        comp : ∀{X Y Z} → Hom Y Z → Hom X Y → Hom X Z
        idl  : ∀{X Y}{f : Hom X Y} → comp iden f ≅ f
        idr  : ∀{X Y}{f : Hom X Y} → comp f iden ≅ f
        ass  : ∀{W X Y Z}{f : Hom Y Z}{g : Hom X Y}{h : Hom W X} →
               comp (comp f g) h ≅ comp f (comp g h)
```

Records are a way to group together values of different type. They are similar to structs in other languages, but the type of a field in a record can depend on the values of other fields of the same record, which appear before it in the declaration. As an example, the type of `Hom` depends on the value of `Obj`.

We also need to define the category `Fam` where the objects are families of sets and the morphisms are functions between them.

```
Fam : Set → Cat
Fam I = record {
  Obj  = I → Set;
  Hom  = λ A B → ∀{i} → A i → B i;
  iden = id;
  comp = λ f g → f ∘ g;
  idl  = refl;
  idr  = refl;
  ass  = refl}
```

A functor is a mapping between two categories. Our definition of functors is a record, which is parametrized by two categories, `C` and `D`, as the domain and codomain of the functor. The record defines the object map `OMap` and the morphism map `HMap`. The object map transforms objects in the first category to objects in the second category. The morphism map transforms morphisms in the first category to morphisms in the second. The record also includes the functor laws which state that identity in `C` is mapped to identity in `D` and that composition in `C` is mapped to composition in `D`.

```
record Fun (C D : Cat) : Set where
  field OMap : Obj C → Obj D
        HMap : ∀{X Y} → Hom C X Y → Hom D (OMap X) (OMap Y)
        fid  : ∀{X} → HMap (iden C {X}) ≅ iden D {OMap X}
        fcomp : ∀{X Y Z}{f : Hom C Y Z}{g : Hom C X Y} →
                HMap (comp C f g) ≅ comp D (HMap f) (HMap g)
```

A relative monad is not defined on a single category, as ordinary monads, but on a functor `J` from a source category `C` to a target category `D`. The functor `J` should be thought of as a tool to fix the mismatch between the two categories.

```
record RMonad {C D : Cat}(J : Fun C D) : Set where
  field T : Obj C → Obj D
        η : ∀{X} → Hom D (OMap J X) (T X)
        bind : ∀{X Y} → Hom D (OMap J X) (T Y) → Hom D (T X) (T Y)
        law1 : ∀{X} → bind (η {X}) ≅ iden D {T X}
        law2 : ∀{X Y}{f : Hom D (OMap J X) (T Y)} →
               comp D (bind f) η ≅ f
        law3 : ∀{X Y Z}{f : Hom D (OMap J X) (T Y)} →
               {g : Hom D (OMap J Y) (T Z)} →
               bind (comp D (bind g) f) ≅ comp D (bind g) (bind f)
```

An algebra for a relative monad `M` over a functor `J` is defined as follows. It has a carrier `acar` in `D` and an algebra structure `astr` which for any `Z : Obj C` takes a morphism from `OMap J Z` to `acar` in `D` to a morphism from `T M Z` to `acar` in `D`. The algebra must also satisfy two laws, which state that the algebra structure `astr` interacts appropriately with the η and `bind` of the monad.

```
record RAlg {C D : Cat}{J : Fun C D}(M : RMonad J) : Set where
  field acar : Obj D
        astr : ∀{Z} → Hom D (OMap J Z) acar → Hom D (T M Z) acar
        alaw1 : ∀{Z}{f : Hom D (OMap J Z) acar} →
                f ≅ comp D (astr f) (η M)
        alaw2 : ∀{Z}{W}{k : Hom D (OMap J Z) (T M W)} →
                {f : Hom D (OMap J W) acar } →
                astr (comp D (astr f) k) ≅ comp D (astr f) (bind  M k)
```

1.5 Related work

The first known proof of normalisation for typed lambda calculus is from 1942 by Turing [9]. Normalisation by evaluation itself was invented by Martin-Löf [13]. There it appears as a special case of presenting a normalisation proof. Instead of proving that each term has a normal form, one creates a function which computes the normal form together with the proof that the result is actually a normal form.

Normalisation by evaluation for typed lambda calculus with β and η conversion was discovered by Berger and Schwichtenberg [6]. They needed a normalisation algorithm for their proof system MINLOG and normalisation by evaluation provided a simple solution.

Recently, Allais, McBride and Boutillier [2] have considered normalisation by evaluation to solve the decision problem for an equational theory enriched with monoid, functor and fusion laws. For example, they achieve that `map swap ∘ map swap ≅ id`, where `swap` swaps the elements of a pair. Their implementation of normalisation by evaluation is more syntactic in nature than ours and differs from the traditional approach.

Normalising infinite structures like streams has been investigated by de Vries and Severi [16]. Their approach to normalisation is based on reduction, where they produce possibly infinite normal forms.

1.6 Overview

In the following two chapters, we look at simply typed lambda calculus: how to represent a language like simply typed lambda calculus in Agda and how to implement normalisation for it. As a result, we have a normaliser for simply typed lambda calculus with additional data types: natural numbers, lists, pairs, and streams.

Chapter two is devoted to the syntactic aspects of lambda calculus. There we look at representing the syntax, renaming, and substitution, and we show some properties about how these things fit together. Substitution is used in the conversion relation so it appears in the correctness proof. At the end of the chapter, we look at how to extend the development to additional data types.

Chapter three is mostly about the semantic parts of the normalisation process. We start by defining the interpretation function, which maps terms to values. After that, we shortly discuss normal forms, and then define the function `reify` to read values back to normal forms. Similarly to the previous chapter, this chapter also includes various properties related to the development, most of which lead up to the proof of correctness of the normaliser. The end of the chapter is about extending the normaliser for additional data types.

2 Syntactic aspects of lambda calculus

In this chapter we will look at how to represent our object language, simply typed lambda calculus, in Agda. This is introduced using the basic system with only base and function types. This representation is taken from the formalisation of relative monads [4]. At the end of the chapter, we will see how to extend our representation for additional data types (not present in [4]).

2.1 Syntax

As we are working with a typed language, the first thing we should do is to define the types in our language. We have a base type ι and function types $\sigma \Rightarrow \tau$. This can be represented in Agda as a data type with two constructors. The constructor for base types is a constant and the one for functions takes two types and gives back a new type.

```
data Ty : Set where
  ι     : Ty
  _⇒_ : Ty → Ty → Ty
```

Next, we will define contexts, which are necessary for assigning a type to a term. We are going to represent contexts as *cons* lists (sequences of types) growing in the "wrong" direction. A context can be empty or it can be made up of a context and a type.

```
data Con : Set where
  ε   : Con
  _<_ : Con → Ty → Con
```

We are going to represent variables as de Bruijn indices [7]. This is a nameless approach to variables: a variable is identified by how far it is from the binding lambda. This means that variables are basically natural numbers and the value of the number is its location in the context.

```
data Var : Con → Ty → Set where
  vze  : ∀{Γ σ} → Var (Γ < σ) σ
  vsu  : ∀{Γ σ τ} → Var Γ σ → Var (Γ < τ) σ
```

Finally, we can define terms. There are only three syntactic constructs in our object language: variables, lambda abstractions, and applications. If we have a variable, we can always turn it into a term. If we have a term of type τ and a variable of type σ in the last position in the context, then we can turn it into a function $\sigma \Rightarrow \tau$. If we have a function, we can apply it to an argument of a suitable type and get the result.

```
data Tm (Γ : Con) : Ty → Set where
  var : ∀{σ} → Var Γ σ → Tm Γ σ
  lam : ∀{σ τ} → Tm (Γ < σ) τ → Tm Γ (σ ⇒ τ)
  app : ∀{σ τ} → Tm Γ (σ ⇒ τ) → Tm Γ σ → Tm Γ τ
```

An important observation is that in a dependently typed language we can define the syntax and typing rules together. This gives us a "no junk" representation of our language, we have to consider only those expressions which are well-typed.

2.2 Renaming

Renaming a term is an operation which replaces a variable in a term with another variable. We will define the type of renamings as a function mapping variables in one context to variables in another.

```
Ren : Con → Con → Set
Ren Γ Δ = ∀{σ} → Var Γ σ → Var Δ σ
```

We are going to need an operation for weakening a renaming before we can actually rename a term. Weakening is necessary for pushing the renaming inside the lambda term by extending the context with the bound variable.

```
wk : ∀{Γ Δ σ} → Ren Γ Δ → Ren (Γ < σ) (Δ < σ)
wk ρ vze     = vze
wk ρ (vsu y) = vsu (ρ y)
```

Now that we have weakening, we can define how to actually rename a term. Renaming a term is defined by induction on the term.

```
ren : ∀{Γ Δ} → Ren Γ Δ → ∀{σ} → Tm Γ σ → Tm Δ σ
ren α (var x)   = var (α x)
ren α (lam t)   = lam (ren (wk α) t)
ren α (app t u) = app (ren α t) (ren α u)
```

If the term is a variable, we just need to apply the renaming. In the case of a lambda, we first need to weaken the renaming and then apply it to the body of the lambda. For application, we apply the renaming to the subterms.

As an example of renaming a term, we now look at applying the identity function to a variable of type $\iota \Rightarrow \iota$. In a more convenient notation it would be $(\lambda x.x)\, y$.

```
idapp : ∀{Γ} → Tm (Γ < (ι ⇒ ι)) (ι ⇒ ι)
idapp = app (lam (var vze)) (var vze)
```

A key thing to notice here is that the `var vze` inside the lambda is different from the second argument of `app`. We now apply the `vsu` renaming to the term, which means that variables get incremented by one, with the exception that when going under the lambda, the renaming is weakened. This means that the variable bound by the current lambda will not be renamed. The following is the result of applying the renaming.

```
app (lam (var vze)) (var (vsu vze))
```

We can also define the identity renaming and the composition of two renamings. The identity is just the ordinary identity function and composition is just the ordinary function composition.

```
renId : ∀{Γ} → Ren Γ Γ
renId = id
```

```
renComp : ∀{B Γ Δ} → Ren Γ Δ → Ren B Γ → Ren B Δ
renComp f g = f ∘ g
```

Now that we have defined what a renaming is, we are going to show some of its properties. To do that, we need to be able to say when two renamings are equal. Since renamings are functions, we need to be able to say when two functions are equal. For enabling this, we postulate extensionality for functions.

```
postulate ext : {A : Set}{B B' : A → Set} →
                {f : ∀ a → B a}{g : ∀ a → B' a} →
                (∀ a → f a ≅ g a) → f ≅ g
```

`ext` states that two functions `f` and `g` are equal if their results are equal for every possible input. Postulating something is essentially adding an axiom to Agda. This is something which should be done with care as we might introduce inconsistencies into the system. However, it has been shown that extensionality is a conservative extension of intensional type theory [11]. We also postulate `iext`, which is basically the same, except that it applies to functions taking implicit arguments.

Weakening the identity renaming should be the same as the identity renaming and renaming terms using the identity renaming should keep the terms unchanged. We are not going to show many proofs of the properties in this thesis. We will give the first ones as an example. Also, we will often omit the implicit arguments from the definitions. This is purely to avoid excessive clutter. For the full details, we refer the reader to the full formalisation available online [12].

```
wkid : ∀{Γ σ τ}(x : Var (Γ < τ) σ) → wk renId x ≅ renId x
wkid vze     = refl
wkid (vsu y) = refl

renid : ∀{Γ σ}(t : Tm Γ σ) → ren renId t ≅ t
renid (var x) = refl
renid (lam y) = proof
  lam (ren (wk renId) y)
  ≅⟨ cong (λ (f : Ren _ _) → lam (ren f y)) (iext (λ _ → ext wkid)) ⟩
  lam (ren renId y)
  ≅⟨ cong lam (renid y) ⟩
  lam y
  ■
renid (app t u) = cong₂ app (renid t) (renid u)
```

As `wk` is defined by induction on the variable, the proof of `wkid` is by induction on the variable as well. If the proof is `refl`, then it means that the property is so simple that even Agda can see that it holds.

As `ren` is defined by induction on the term, proof of `renid` is by induction on the term. In the lambda case we need to apply extensionality to show `wk renId ≅ renId`. This is because a renaming is a function and we want to show the equality of the two renamings.

```
wkcomp : (f : Ren Γ Δ)(g : Ren B Γ)(x : Var (B < σ) τ) →
            wk (renComp f g) x ≅ renComp (wk f) (wk g) x

rencomp : (f : Ren Γ Δ)(g : Ren B Γ)(t : Tm B σ) →
            ren (renComp f g) t ≅ (ren f ∘ ren g) t
```

Proofs of `wkcomp` and `rencomp` are very similar to proofs of `wkid` and `renid`, respectively.

As a final note about renamings, we can observe that our definition of renamings forms a category on the contexts.

```
RenCat : Cat
RenCat = record {
  Obj  = Con;
  Hom  = Ren;
  iden = renId;
  comp = renComp;
  idl  = iext (λ _ → refl);
  idr  = iext (λ _ → refl);
  ass  = iext (λ _ → refl)}
```

The objects of the category are the contexts, the morphisms are the renamings, identity morphism is the identity renaming and the composition of morphisms is the composition of renamings. The three properties almost hold definitionally, we need to apply extensionality of implicit functions since our definition of renamings expects the type as the implicit parameter.

Now we can define the functor `VarF` which will play the role of `J` in our instance of a relative monad. Objects are transformed using `Var` and morphisms are transformed using `id`. Since `HMap` is `id`, the properties `fid` and `fcomp` hold definitionally.

```
VarF : Fun RenCat (Fam Ty)
VarF = record {
  OMap = Var;
  HMap = id;
  fid = refl;
  fcomp = refl}
```

2.3 Substitution

Substitution is the operation of substituting one term for a variable in another. For example, applying a function to an argument requires that we substitute this argument term for the parameter in the function body. We will define a substitution as a function mapping variables in one context to terms in another.

```
Sub : Con → Con → Set
Sub Γ Δ = ∀{σ} → Var Γ σ → Tm Δ σ
```

Similarly to renamings, we now want to have an operation for weakening a substitution. If have the zeroth variable, we just return itself. For other variables, we first apply the substitution to the variable and then weaken the resulting term using `ren vsu`.

```
lift : ∀{Γ Δ σ} → Sub Γ Δ → Sub (Γ < σ) (Δ < σ)
lift f vze     = var vze
lift f (vsu x) = ren vsu (f x)
```

The actual operation of substituting terms for variables is defined by induction on the term. For variables, we just apply the substitution. In the lambda case, we lift it and then apply it to the body, and in the application case, we apply it to both subterms.

```
sub : ∀{Γ Δ} → Sub Γ Δ → ∀{σ} → Tm Γ σ → Tm Δ σ
sub f (var x)   = f x
sub f (lam t)   = lam (sub (lift f) t)
sub f (app t u) = app (sub f t) (sub f u)
```

To extend an existing substitution, we must bind a new term to the zeroth variable and shift everything else by one.

```
sub<< : ∀{Γ Δ} → Sub Γ Δ → ∀{σ} → Tm Δ σ → Sub (Γ < σ) Δ
sub<< f t vze     = t
sub<< f t (vsu x) = f x
```

The identity of substitutions cannot be the ordinary identity function as it was for renamings, since the domain of a substitution is a variable and the range is a term. The identity substitution is actually the `var` constructor of terms. Two substitutions `f` and `g` are composed by precomposing `g` with `sub f`, the substitution `f` operating on terms.

```
subId : ∀{Γ} → Sub Γ Γ
subId = var

subComp : ∀{B Γ Δ} → Sub Γ Δ → Sub B Γ → Sub B Δ
subComp f g = sub f ∘ g
```

As an example of how substitution works, we now look at applying a variable of type $\iota \Rightarrow \iota$ to a variable of type ι. In a more convenient notation it would be written $x\, y$, the variable x applied to the variable y.

```
appvars : ∀{Γ} → Tm ((Γ < ι) < (ι ⇒ ι)) ι
appvars = app (var vze) (var (vsu vze))
```

We now want to substitute the identity function `lam (var vze)` for the zeroth variable, the function in the application. To do that, we extend the identity substitution with the identity function and then apply it to the term.

```
sub (sub<< var (lam (var vze))) appvars
```

As the result we get the following term. The first argument of the application is now an actual function, not a variable. Notice, that the second argument has become `var vze`.

```
app (lam (var vze)) (var vze)
```

Lifting an identity substitution should be the same as the identity function. Substitution using the identity substitution should not change the terms. The proof of `liftid` is by induction on the variable and the proof of `subid` is by induction on the term. These are again very similar to the proofs of `wkid` and `renid`.

```
liftid : (x : Var (Γ < σ) τ) → lift subId x ≅ subId x
subid : (t : Tm Γ σ) → sub subId t ≅ id t
```

Next, we have proved some properties relating together operations on renamings and substitutions.

```
liftwk : (f : Sub Γ Δ)(g : Ren B Γ)(x : Var (B < σ) τ) →
            (lift f ∘ wk g) x ≅ lift (f ∘ g) x

subren : (f : Sub Γ Δ)(g : Ren B Γ)(t : Tm B σ) →
            (sub f ∘ ren g) t ≅ sub (f ∘ g) t
```

The proof of `liftwk` is by induction on the variable. The proof of `subren` is by induction on the term. In the lambda case we need to apply `liftwk`.

```
renwklift : (f : Ren Γ Δ)(g : Sub B Γ)(x : Var (B < σ) τ) →
              (ren (wk f) ∘ lift g) x ≅ lift (ren f ∘ g) x

rensub :  (f : Ren Γ Δ)(g : Sub B Γ)(t : Tm B σ) →
              (ren f ∘  sub g) t ≅ sub (ren f ∘ g) t
```

The proof of `renwklift` is by induction on the variable. In the successor case, we need to use `rencomp`. The proof of `rensub` is by induction on the term. In the lambda case, we need to apply `renwklift`.

```
liftcomp : (f : Sub Γ Δ)(g : Sub B Γ)(x : Var (B < σ) τ) →
              lift (subComp f g) x ≅ subComp (lift f) (lift g) x

subcomp : (f : Sub Γ Δ)(g : Sub B Γ){σ}(t : Tm B σ) →
              sub (subComp f g) t ≅ (sub f ∘ sub g) t
```

The proof of `liftcomp` is by induction on the variable. In the successor case we need to apply both `rensub` and `subren`. The proof of `subcomp` is by induction on the term. In the lambda case we need to apply `liftcomp`.

After defining substitution, we can now give the proof that `Tm` forms a relative monad on the `VarF` functor defined earlier. `T` is the `Tm` type constructor, η is the `var` constructor and `bind` is the substitution operation. The first law is proved using `subid` and handling extensionality. For the third law, we need to use `subcomp` under extensionality. The second law holds definitionally.

```
TmRMonad : RMonad VarF
TmRMonad = record {
  T = Tm;
  η = var;
  bind = sub;
  law1 = iext (λ σ → ext subid);
  law2 = refl;
  law3 = λ {_ _ _ f g} → iext (λ σ → ext (subcomp g f))}
```

2.4 Equational theory

This section is about the convertibility relation ∼ which is defined as a data type. It can be thought of as a specification of normalisation. Two terms are convertible (provably equal) if they can be related together using the elements of this data type. Since we have transitivity as one of the elements, two terms can be related through multiple steps. This is similar to the definition of propositional equality given in the Introduction, but here we have more than one constructor.

```
data _∼_ {Γ : Con} : ∀{σ : Ty} → Tm Γ σ → Tm Γ σ → Set where
  refl∼  : t ∼ t
  sym∼   : t ∼ u → u ∼ t
  trans∼ : t ∼ u → u ∼ v → t ∼ v
  beta∼  : app (lam t) u ∼ sub (sub<< var u) t
  eta∼   : t ∼ lam (app (ren vsu t) (var vze))
  congapp∼ : t ∼ t' → u ∼ u' → app t u ∼ app t' u'
  conglam∼ : t ∼ t' → lam t ∼ lam t'
```

`refl∼`, `sym∼` and `trans∼` are the reflexivity, symmetry and transitivity properties for our convertibility relation. `beta∼` and `eta∼` state that our relation respects β-conversion and η-conversion. `congapp∼` states that two applications are convertible if their corresponding subterms are convertible. `conglam∼` states that two lambda terms are convertible if the bodies of the lambda terms are convertible.

Notice that propositional equality implies convertibility. If we have two equal terms, then it must be that they are convertible.

```
≅to∼ : {t t' : Tm Γ σ} → t ≅ t' → t ∼ t'
≅to∼ refl = refl∼
```

2.5 Extensions

In this section, we are going to extend our treatment of simply typed lambda calculus with additional data types. It will mostly be about highlighting the necessary changes to accommodate these new types. Code for the modifications will be shown as excerpts, not in full detail.

2.5.1 Natural numbers and lists

Natural numbers and lists are both inductive data types and share similarities in their structure. Therefore, it is natural to consider them together. When viewed separately, the combination of function types with the type of natural numbers is also known as Gödel's T [10].

We begin by extending the definition of types in our language. `nat` is the type of natural numbers and `[ σ ]` is the type of lists with elements of type σ. We use the three dots to indicate that the definition is incomplete.

```
data Ty : Set where
  ...
  nat  : Ty
  [_]  : Ty → Ty
```

We do not need to make any modifications to the definitions of contexts and variables. For natural numbers, we extend the definition of terms with `ze` and `su` for constructing natural numbers and primitive recursion `rec` for eliminating them. The situation is similar for lists. We have `nil` for the empty list, `cons` for appending an element to the head of the list and `fold` for eliminating the list.

```
data Tm (Γ : Con) : Ty → Set where
  ...
  ze   : Tm Γ nat
  su   : Tm Γ nat → Tm Γ nat
  rec  : ∀{σ} → Tm Γ σ → Tm Γ (σ ⇒ σ) → Tm Γ nat → Tm Γ σ
  nil  : ∀{σ} → Tm Γ [ σ ]
  cons : ∀{σ} → Tm Γ σ → Tm Γ [ σ ] → Tm Γ [ σ ]
  fold : ∀{σ τ} → Tm Γ τ → Tm Γ (σ ⇒ τ ⇒ τ) →
                  Tm Γ [ σ ] → Tm Γ τ
```

Now we look at how to rename these newly added terms. For `ze` and `nil` we do not need to do anything since both are constants. For other cases we apply the renaming to the subterms.

```
ren : ∀{Γ Δ} → Ren Δ Γ → ∀{σ} → Tm Δ σ → Tm Γ σ
...
ren ρ ze            = ze
ren ρ (su t)        = su (ren ρ t)
ren ρ (rec z f n)   = rec (ren ρ z) (ren ρ f) (ren ρ n)
ren ρ nil           = nil
ren ρ (cons h t)    = cons (ren ρ h) (ren ρ t)
ren ρ (fold a f l)  = fold (ren ρ a) (ren ρ f) (ren ρ l)
```

As before, we are not going to show the proofs in full detail and most of the time only show the types. The proof for `renid` is here as an example of how the proofs work on natural numbers and lists.

```
renid : ∀{Γ σ}(t : Tm Γ σ) → ren renId t ≅ t
...
renid ze            = refl
renid (su t)        = cong su (renid t)
renid (rec z f n)   = cong₃ rec (renid z) (renid f) (renid n)
renid nil          = refl
renid (cons h t)   = cong₂ cons (renid h) (renid t)
renid (fold a f l) = cong₃ fold (renid a) (renid f) (renid l)
```

For `ze` and `nil` the proof is just `refl`, which makes sense, since the operation of renaming those terms does not involve anything. For other cases we need to apply the inductive hypothesis to the constituents. The process is very similar for `rencomp`.

Applying a substitution to a term is very similar to renaming a term. For `ze` and `nil` we do nothing and for others we apply the substitution to the constituents.

```
sub : ∀{Γ Δ} → Sub Γ Δ → ∀{σ} → Tm Γ σ → Tm Δ σ
...
sub f ze            = ze
sub f (su n)        = su (sub f n)
sub f (rec z g n)   = rec (sub f z) (sub f g) (sub f n)
sub f nil           = nil
sub f (cons t u)    = cons (sub f t) (sub f u)
sub f (fold a fn l) = fold (sub f a) (sub f fn) (sub f l)
```

Additions to the proof of `subid` are very similar to the additions to the proof of `renid`. We do not need to change anything in the proofs of `liftwk` and `renwklift` as these are defined by induction on the variable. For `subren`, `rensub` and `subcomp` we need to add similar cases as we showed for `renid`.

Finally, we will show how to extend the convertibility relation to consider the new data types.

```
data _∼_ {Γ : Con} : ∀{σ : Ty} → Tm Γ σ → Tm Γ σ → Set where
  ...
  congsu∼  : t ∼ t’ → su t ∼ su t’
  congrec∼ : z ∼ z’ → f ∼ f’ → n ∼ n’ →
             rec z f n ∼ rec z’ f’ n’
  congrecze∼ : rec z f ze ∼ z
  congrecsu∼ : rec z f (su n) ∼ app f (rec z f n)
  congcons∼  : x ∼ x’ → xs ∼ xs’ → cons x xs ∼ cons x’ xs’
  congfold∼  : z ∼ z’ → f ∼ f’ → n ∼ n’ →
               fold z f n ∼ fold z’ f’ n’
  congfoldnil∼  : fold z f nil ∼ z
  congfoldcons∼ : fold z f (cons x xs)
                  ∼
                  app (app f x) (fold z f xs)
```

Notice that we have no cases for `ze` and `nil`, these are already covered by `refl∼`. We define that two natural numbers are convertible if their predecessors are convertible. We have three cases for the recursor. `congrec∼` states that two `rec` terms are convertible if their corresponding subterms are convertible. `congrecze∼` expresses the behaviour of `rec` when the argument is zero, `congrecsu∼` expresses the behaviour when the argument is a successor. The situation is similar for fold.

2.5.2 Pairs and streams

A pair is a data type which contains two pieces of data: the first and the second projection. In other words, it is a tuple with two elements. A stream can be thought of as an infinite sequence of values. It is similar to a pair in that we also interact with it using observations—we can observe what the element at position n is by taking the n-th projection of the stream. By this analogy, a stream can be seen as an infinite tuple, with the exception that all elements of a stream must be of the same type, while a tuple can have elements of different types.

Now we will look at how to represent pairs and streams in our language. We begin by extending the set of types in our language to support pairs and streams. $\sigma \wedge \tau$ is the type of pairs, where the first component is of type σ and the second is of type τ. < σ > is the type of streams which contain elements of type σ.

```
data Ty : Set where
  ...
  _∧_  : Ty → Ty → Ty
  <_>  : Ty → Ty
```

We continue by extending the definition of terms with constructors for creating and eliminating pairs and streams. `,,` is for creating a pair from two existing terms and `fst` and `snd` are for projecting out the desired element. `tup` is for creating a stream from a function from natural numbers to terms. To project out a value from a stream, we use `proj`. Which element is projected is decided by the first argument, which is the position of the element in the stream.

```
data Tm (Γ : Con) : Ty → Set where
  ...
  _,,_ : ∀{σ τ} → Tm Γ σ → Tm Γ τ → Tm Γ (σ ∧ τ)
  fst  : ∀{σ τ} → Tm Γ (σ ∧ τ) → Tm Γ σ
  snd  : ∀{σ τ} → Tm Γ (σ ∧ τ) → Tm Γ τ
  tup  : ∀{σ} → (ℕ → Tm Γ σ) → Tm Γ < σ >
  proj : ∀{σ} → ℕ → Tm Γ < σ > → Tm Γ σ
```

Renaming pairs is rather similar to what we have done before. For streams, we need to handle the natural number appearing in the terms.

```
ren : ∀{Γ Δ} → Ren Γ Δ→ ∀{σ} → Tm Γ σ → Tm Δ σ
...
ren α (t ,, u)    = ren α t ,, ren α u
ren α (fst t)     = fst (ren α t)
ren α (snd t)     = snd (ren α t)
ren α (tup f)     = tup (λ n → ren α (f n))
ren α (proj n s) = proj n (ren α s)
```

We again use `renid` as a simple example of how to prove properties about the new data types.

```
renid : ∀{Γ σ}(t : Tm Γ σ) → ren renId t ≅  t
```

```
...
renid (a ,, b)   = cong₂ _,,_ (renid a) (renid b)
renid (fst t)    = cong fst (renid t)
renid (snd t)    = cong snd (renid t)
renid (tup f)    = cong tup (ext (λ n → renid (f n)))
renid (proj n f) = cong (proj n) (renid f)
```

The idea is to use the inductive hypothesis for the subterms. `tup` also requires the use of extensionality since we are working with a function. The proof of `rencomp` is very similar.

Again, applying a substitution is very similar to applying a renaming. The idea is to push the substitution inside the term. For `tup` we apply the substitution to the result of `g n`, that is the element at position `n`. Notice that this is done only when that element is actually needed.

```
sub : ∀{Γ Δ} → Sub Γ Δ → ∀{σ} → Tm Γ σ → Tm Δ σ
...
sub f (a ,, b)   = sub f a ,, sub f b
sub f (fst t)    = fst (sub f t)
sub f (snd t)    = snd (sub f t)
sub f (tup g)    = tup (λ n → sub f (g n))
sub f (proj n s) = proj n (sub f s)
```

Additions to the proof of `subid` are very similar to the additions to the proof of `renid`. We do not need to change anything in the proofs of `liftwk` and `renwklift` as these are defined by induction on the variable. For `subren`, `rensub` and `subcomp` we need to add similar cases as we showed for `renid`.

Finally, we will show how to extend the convertibility relation for pairs and streams.

```
data _∼_ {Γ : Con} : ∀{σ : Ty} → Tm Γ σ → Tm Γ σ → Set where
  ...
  congpair∼   : a ∼ a' → b ∼ b' → (a ,, b) ∼ (a' ,, b')
  paireta∼    : t ∼ (fst t ,, snd t)
  pairfst∼    : a ∼ fst (a ,, b)
  pairsnd∼    : b ∼ snd (a ,, b)
  congfst∼    : a ∼ a' → fst a ∼ fst a'
  congsnd∼    : a ∼ a' → snd a ∼ snd a'
  congtup∼    : (∀ n → f n ∼ g n) → tup f ∼ tup g
  congproj∼   : {n : ℕ} → f ∼ g → proj n f ∼ proj n g
  streambeta∼ : {n : ℕ} → proj n (tup f) ∼ f n
  streameta∼  : s ∼ tup (λ n → proj n s)
```

`congpair∼` gives us that two pairs are convertible if their both components are convertible. `paireta∼` states that a pair is equal to its expansion. `pairfst∼` and `pairsnd∼` give us that a component of a pair is equal to the corresponding projection. `congfst∼` and `congsnd∼` state that the projections of two pairs are convertible if the two pairs are convertible.

`congtup∼` states that two streams are convertible if the underlying functions are convertible. `congproj∼` states that the projections of convertible streams are convertible. `streambeta∼` and `streameta∼` illustrate that `tup` and `proj` cancel each other out.

3 Normalisation by evaluation

This chapter is about normalising the terms in our object language. We will look at evaluating the terms and extracting the normal forms from the values. At the same time, we will prove essential properties about the process, leading up to the correctness proof of the normaliser. We will finish this chapter with the extensions of our language.

3.1 Evaluation

Evaluator is a function from terms (syntax) to values (semantics). First, we will define the values (model) into which we want to evaluate our language.

```
Val : Con → Ty → Set
Val Γ ι         = Nf Γ ι
Val Γ (σ ⇒ τ) = ∀{Δ} → Ren Γ Δ → Val Δ σ → Val Δ τ
```

It is important to note that values are not defined as a data type (which was the case for terms, for example). The type of values is defined as a function returning a type. Given a context Γ and a type σ, it gives an Agda type which should be used for representing a value of type σ.

For base types, a value is just a normal form of base type. Normal forms are defined in section 3.2. They can be thought of as a subset of terms. For functions, the value is an actual Agda function. In addition to just taking a value of type σ to a value of type τ, this representation also takes implicitly a new context and a renaming from the old context to this new (future) context, so that the function can be used in a different context than where it was defined.

This definition of functions is slightly incomplete. The definition above is good enough for implementing the normalisation algorithm, but for the proofs we actually extended the definition by an additional property. A function value then becomes an existentially quantified type stating that there exists a function from values of type σ to values of type τ such that renaming the result of the function is the same as renaming the argument and the function. The evaluator must additionally create values that preserve that property. We have decided to exclude this property from the remainder of this document to keep the code simpler.

```
Val Γ (σ ⇒ τ) = Σ (∀{Δ} → Ren Γ Δ → Val Δ σ → Val Δ τ)
  λ f → ∀{Δ Δ'}(ρ : Ren Γ Δ)(ρ' : Ren Δ Δ')(v : Val Δ σ) →
        renval ρ' (f ρ v) ≅ f (ρ' ∘ ρ) (renval ρ' v)
```

We are also going to need a method for renaming values. For base types, renaming values is just renaming normal forms, which is similar to renaming terms. For functions, we need to change the renaming used when applying this function to a parameter.

```
renval : ∀{Γ Δ σ} → Ren Γ Δ → Val Γ σ → Val Δ σ
renval {σ = ι} α x      = renNf α x
renval {σ = σ ⇒ τ} α v = λ {E} β v' → v (renComp β α) v'
```

Evaluating a term requires an environment for mapping variables to values. We are going to represent it similarly to renamings and substitutions. Given two contexts, an environment is a function which implicitly takes a type and gives back a function from variables in the first context to values in the second context.

```
Env : Con → Con → Set
Env Γ Δ = ∀{σ} → Var Γ σ → Val Δ σ

_<<_ : ∀{Γ Δ} → Env Γ Δ → ∀{σ} → Val Δ σ → Env (Γ < σ) Δ
(γ << v) vze     = v
(γ << v) (vsu x) = γ x
```

Extending an environment is defined by what the result of applying the extended environment to a variable should be. For the zeroth variable, the value which was used for extending the environment is returned. For other variables, the predecessor of the variable in the original environment is returned.

Next, we are going to define evaluation together with some of its properties.

```
mutual
  eval : ∀{Γ Δ σ} → Env Γ Δ → Tm Γ σ → Val Δ σ
  eval γ (var x) = γ x
  eval γ (lam t) = λ α v → eval ((renval α ∘ γ) << v) t

  evallem : (γ : Env Γ Δ)(ρ : Ren Δ Δ₁)(t : Tm Γ σ) →
                renval ρ (eval γ t) ≅ eval (renval ρ ∘ γ) t
```

Our evaluator `eval` is defined by induction on the structure of the term which is evaluated. For variables, we just look the value up from the environment. For the lambda case, we extend the environment so that `v` is now the variable bound by the lambda and then evaluate the body. The evaluator is defined mutually with a property about pushing a renaming into the environment. The proof of `evallem` is by induction on the term `t`.

We now look at an example how the evaluator works. First, we define the identity function on base types, which is just $\lambda x.x$

```
id : ∀{Γ} → Tm Γ (ι ⇒ ι)
id = lam (var vze)
```

This defines the term `id` to be a function on base types that given an argument returns the zeroth variable—the variable bound by the lambda constructor `lam` in the definition. We make this example a bit more interesting by applying this identity function to a variable of type $\iota \Rightarrow \iota$. The term is now essentially $(\lambda x.x)\, y$

```
idapp : ∀{Γ} → Tm (Γ < (ι ⇒ ι)) (ι ⇒ ι)
idapp = app (lam (var vze)) (var vze)
```

Notice that in the last position of the context there is a function on base types, meaning that the zeroth variable is of that type. We will evaluate this term in the identity environment, similarly to what the normaliser will do. This results in the following meta level function.

```
λ {Δ} α v → ne (napp (nvar (α vze)) v)
```

This is a function which implicitly takes the new context, then takes a renaming from the old context to the new context, and then takes a value in the new context. As a result, it gives back the zeroth variable from the definition of `idapp` applied to the value `v`. Why a variable of type $\iota \Rightarrow \iota$ was expanded to this function definition, is because we evaluated the term in the identity environment and in the identity environment we get the value of a variable by reflecting it (turning it into a value). Reflection is defined in section 3.3.

We have proved some properties about how the evaluator fits together with renamings, substitutions and environments.

```
wk<< : (α : Ren Γ Δ)(β : Env Δ E)(v : Val E σ) →
              ∀{ρ}(y : Var(Γ < σ) ρ) →
              ((β ∘ α) << v) y ≅ ((β << v) ∘ wk α) y

reneval : (α : Ren Γ Δ)(β : Env Δ E)(t : Tm Γ σ) →
              eval (β ∘ α) t ≅ (eval β ∘ ren α) t
```

The proof of `wk<<` is by induction on the variable `y`. `reneval` states that first renaming a term and then evaluating it is the same as evaluating the term in the renamed environment (an environment composed with a renaming is a renamed environment, `Env Δ E` becomes `Env Γ E`). The proof of it is by induction on the term `t`. In the lambda case, we need to apply `wk<<`.

```
lifteval : (α : Sub Γ Δ)(β : Env Δ E)(v : Val E σ) →
              (y : Var (Γ < σ) τ) →
              ((eval β ∘ α) << v) y ≅ (eval (β << v) ∘ lift α) y

subeval : (α : Sub Γ Δ)(β : Env Δ E)(t : Tm Γ σ) →
              eval (eval β ∘ α) t ≅ (eval β ∘ sub α) t
```

The proof of `lifteval` is by induction on the variable `y`. In the successor case, we need to apply `reneval`. `subeval` states that first applying the substitution and then evaluating the term is the same as pushing the substitution inside the environment. Indeed, composing `eval` applied to an environment with a substitution gives a function from variables to values, which is an environment. The proof of `subeval` is by induction on the term `t`. In the lambda case, we need to apply `lifteval`.

We can now show that our representation of lambda calculus forms a relative Eilenberg-Moore algebra. The carrier of the algebra is `Val Γ` so it has the type `Ty → Set`. The structure is given by the evaluator. The first law holds definitionally and the second law follows from the property `subeval`.

```
EMRAlg : Con → RAlg TmRMonad
EMRAlg Γ = record {
  acar  = Val Γ;
  astr  = λ {Γ} → λ γ → eval γ;
  alaw1 = refl;
  alaw2 = λ {Γ Δ α γ} → iext (λ σ → ext (subeval α γ))}
```

3.2 Normal forms

Terms and normal forms are rather similar in structure, but normal forms have additional constraints on them. We are going to represent normal forms as two mutually recursive data types: normal forms and neutral terms.

```
mutual
  data Nf (Γ : Con) : Ty → Set where
    nlam  : ∀{σ τ} → Nf (Γ < σ) τ → Nf Γ (σ ⇒ τ)
    ne    : Ne Γ ι → Nf Γ ι

  data Ne (Γ : Con) : Ty → Set where
    nvar  : ∀{σ} → Var Γ σ → Ne Γ σ
    napp  : ∀{σ τ} → Ne Γ (σ ⇒ τ) → Nf Γ σ → Ne Γ τ
```

To be a normal form, the term has to be in constructor form. A function has to be represented as a lambda abstraction. Neutral terms are terms which contain variables in key positions and due to that, their computation is stuck. Variables are not only a placeholder, but they also represent the unknowns of a program. Variables are neutral terms. Function applications, where the term in the function position is a neutral term and the argument is a normal form are also neutral terms. Since there are no more β-reductions and η-expansions to perform, this representation of a normal form is called β-normal η-long form.

In the proof of the normalisation algorithm, we are going to need to embed normal forms back into terms. This is achieved using a mutually defined embedding function for normal forms and neutral terms.

```
mutual
  embNf : ∀{Γ σ} → Nf Γ σ → Tm Γ σ
  embNf (nlam n) = lam (embNf n)
  embNf (ne x)   = embNe x

  embNe : ∀{Γ σ} → Ne Γ σ → Tm Γ σ
  embNe (nvar x)   = var x
  embNe (napp t u) = app (embNe t) (embNf u)
```

We are also going to need to rename normal forms. This is done using the same renaming for mapping variables to variables, as was used for terms. Since the renaming operation for normal forms and neutral terms is very similar to the renaming operation for terms, we will not give its full definition.

```
mutual
  renNf : ∀{Γ Δ} → Ren Δ Γ → ∀{σ} → Nf Δ σ → Nf Γ σ
  renNe : ∀{Γ Δ} → Ren Δ Γ → ∀{σ} → Ne Δ σ → Ne Γ σ
```

We also prove analogues of `renid` and `rencomp` for normal forms. The property is defined mutually between normal forms and neutral terms, but the proofs for the corresponding parts are very similar to the proofs of the same properties on terms.

```
mutual
  rennfid : (n : Nf Γ σ) → renNf renId n ≅ n
  renneid : (n : Ne Γ σ) → renNe renId n ≅ n

mutual
  rennfcomp : (ρ' : Ren Δ E)(ρ : Ren Γ Δ)(v : Nf Γ σ) →
                  renNf ρ' (renNf ρ v) ≅ renNf (ρ' ∘ ρ) v

  rennecomp : (ρ' : Ren Δ E)(ρ : Ren Γ Δ)(v : Ne Γ σ) →
                  renNe ρ' (renNe ρ v) ≅ renNe (ρ' ∘ ρ) v
```

3.3 Normalisation

In this section, we will define the last components needed for our normaliser, the normaliser itself and the soundness and completeness proofs of it. We start by defining the functions `reify` and its companion `reflect`, to read values back to normal forms and to get values out of neutral terms, respectively.

```
mutual
  reify : ∀{Γ} σ → Val Γ σ → Nf Γ σ
  reify ι x        = x
  reify (σ ⇒ τ) x = nlam (reify τ (x vsu (reflect σ (var vze))))

  reflect : ∀{Γ} σ → Ne Γ σ → Val Γ σ
  reflect ι v        = ne v
  reflect (σ ⇒ τ) n = λ α v →
                    reflect τ (napp (renNe α n) (reify σ v))
```

Both of these functions are type directed, they are defined by induction on the type. Reifying at base type is simple, since a value of base type is already a normal form. For function types, we construct a lambda abstraction where the body is the given function applied to `vsu` renaming (which just does a shift) and a fresh zeroth variable. To reflect a neutral term at base type is again simple, since we need to convert a neutral term to a normal form, the `ne` constructor does exactly that. For function types, we create an actual function which takes a renaming and a value, and gives back the given function applied to the parameter of the lambda. Reflecting is the method for creating fresh variables, as can be seen in the lambda case of `reify`.

```
idE : ∀{Γ} → Env Γ Γ
idE x = reflect _ (nvar x)

norm : ∀{Γ σ} → Tm Γ σ → Nf Γ σ
norm t = reify _ (eval idE t)
```

To normalise a term, we just have to put together evaluation with reification. The term will be evaluated in the identity environment.

We will now look at the same example as we did for evaluation $(\lambda x.x)y$. There we applied the identity function to a variable of type $\iota \Rightarrow \iota$. Applying the normalisation function `norm` to the original term gives the following result.

```
nlam (ne (napp (nvar (vsu vze)) (ne (nvar vze))))
```

If we were to embed this normal form into terms, we would get the following term.

```
lam (app (var (vsu vze)) (var vze))
```

The zeroth variable from the definition of `idapp` has been η-expanded into a lambda. This means, that our normalisation function transformed the original term $(\lambda x.x)\, y$ to $\lambda z.y\, z$.

Our goal now is to prove the correctness of this normaliser. Recall from the Introduction that correctness was expressed using the following property: $t \sim t' \iff norm\ t \equiv norm\ t'$. To prove this, we need to show that the property holds in both directions. For this, we need to prove some additional properties, the first one is `evalsound`.

```
evalsound : t ∼ t' → γ ≅ γ' → eval γ t ≅ eval γ' t'
```

It states that if we evaluate two convertible terms in equal environments, then we get equal results. It is actually the main thing needed to prove for the soundness of our normaliser. Its proof is by induction on the proof `t ∼ t'`.

```
soundness : {t t' : Tm Γ σ} → t ∼ t' → norm t ≅ norm t'
soundness p = cong (reify _) (evalsound p refl)
```

`soundness` is the *if* direction of the correctness property. Its proof is quite simple since we already have `evalsound` and `norm` is just `reify` composed with `eval`.

To prove the *only if* direction, we first need to prove the completeness lemma: $t \sim norm\ t$. In the following proofs leading up to that, we are going to need a way to relate together a term and a value. We will do that using logical relations [15].

```
_∋_R_ : ∀{Γ} σ → (t : Tm Γ σ) → (v : Val Γ σ) → Set
ι ∋ t R v        = t ∼ embNf (reify ι v)
(σ ⇒ τ) ∋ t R f = ∀{Δ} → (ρ : Ren _ Δ)(u : Tm Δ σ) →
                   (v : Val Δ σ) →
                   σ ∋ u R v → τ ∋ app (ren ρ t) u R f ρ v
```

`σ ∋ t R v` means that the term `t` is related to the value `v`. They are related in the sense that they represent the same thing, only one is syntactic and the other is semantic. For base types, we require that embedding the reified value is related to the original term. For function types, we require that given a renaming and another term and a value, which are related, then the results of applying the functions to the new arguments are also related.

```
_E_ : ∀{Γ Δ} → (ρ : Sub Γ Δ) → (η : Env Γ Δ) → Set
ρ E η = ∀{σ} → (x : Var _ σ) → σ ∋ ρ x R η x
```

`E` is basically just the relation `R` for substitutions and environments. Whenever we have a substitution and an environment which are related, then the results of looking up a variable are also related.

The first major component of the completeness proof is the `fund-thm` which is the so-called fundamental theorem of logical relations.

```
fund-thm : (t : Tm Γ σ)(ρ : Sub Γ Δ)(η : Env Γ Δ) →
                 ρ E η → σ ∋ sub ρ t R (eval η t)
```

It states that if we have a substitution and an environment which are related, then the result of applying the substitution to the term is related to the result of evaluating the term in the environment. The proof of it is by induction on the structure of the term. The other major component consists of two mutually defined lemmas `reifyR` and `reflectR`.

```
mutual
  reifyR : σ ∋ t R v → t ∼ embNf (reify σ v)
  reflectR : t ∼ embNe n → σ ∋ t R (reflect σ n)
```

`reifyR` means that if we have a relation between a term and a value, then we can turn it into a relation between a term and a normal form. `reflectR` means that if we have a relation between a term and a neutral term, then we can turn it into a relation between a term and a value. The proof is by induction on types for both properties at the same time.

```
completeness-lem : (t : Tm Γ σ) → t ∼ embNf (norm t)
completeness-lem t = proof
  t
  ∼⟨ ≅to∼ (sym (subid t)) ⟩
  sub subId t
  ∼⟨ reifyR _ (fund-thm t var idE idEE) ⟩
  embNf (norm t)
  ■
```

`completeness-lem` states that every term is related to its normal form. `idEE` is a proof that the identity substitution and the identity environment are related. The fundamental theorem gives us that the result of evaluating the term in the identity environment is related to the result of applying the identity substitution to the term. We then use `reifyR` to lift that to the convertibility relation.

```
completeness : (t t' : Tm Γ σ) → norm t ≅ norm t' → t ∼ t'
completeness t t' p = proof
  t
  ∼⟨ completeness-lem t ⟩
  embNf (norm t)
  ∼⟨ ≅to∼ (cong embNf p) ⟩
  embNf (norm t')
  ∼⟨ sym∼ (completeness-lem t') ⟩
  t'
  ■
```

This concludes our proof of correctness. `completeness` is proved using `completeness-lem` on both sides of the equation to get the convertibility of the terms. One can observe a similarity in the structure of the normalisation algorithm and its completeness proof.

Additionally, we will look at the stability of our normaliser. We want that every normal form is the normal form of itself. This is expressed as a mutually defined property in Agda. The property we are interested in is the one about normal forms, the one about neutral forms is needed for the proof of normal forms. `stabilityNf` is proved by induction on the normal form `n`, `stabilityNe` is proved by induction on the neutral term `n`.

```
mutual
  stabilityNf : (n : Nf Γ σ) → n ≅ norm (embNf n)
  stabilityNe : (n : Ne Γ σ) → eval idE (embNe n) ≅ (reflect _ n)
```

3.4 Extensions

In this section, we will look at the various changes necessary to extend our language to support additional data types. We follow the route taken in the previous chapter where we considered the extensions in two groups: natural numbers and lists, and pairs and streams.

3.4.1 Natural numbers and lists

We are now going to extend our definition of values to consider both natural numbers and lists.

```
Val : Con → Ty → Set
...
Val Γ nat   = Nf Γ nat
Val Γ [ σ ] = ListVal (Val Γ σ) (Ne Γ [ σ ])
```

Values of natural numbers are represented as normal forms of natural numbers. The case for lists hints that this is a bit of a cheat, since we cannot adopt the same strategy we followed for natural numbers.

For lists, first thing one might try is to use normal forms just like for natural numbers, since this idea works there. This creates a problem: what should the fold operation be for normal forms of type list? The problem appears in the cons case of a fold. Since our list is a normal form, then the head is also a normal form, but the function in fold expects a value. Another option one might try, is to use an ordinary lists of values. This works nicely with folds, but it has the problem of how to reflect a neutral term into a value, into an actual list. To avoid those problems, we use a special data type `ListVal`.

```
data ListVal (A B : Set) : Set where
  neLV   : B → ListVal A B
  nilLV  : ListVal A B
  consLV : A → ListVal A B → ListVal A B
```

This definition allows our lists to contain two kinds of things. From the definition of values, it is visible that we instantiate `A` to be `Val Γ σ` and `B` to be `Ne Γ [ σ ]`. This means that our lists can either be entirely made out of ordinary values of type σ or at some point the tail of the list or the whole list is a neutral term of type `[ σ ]`.

```
renval : ∀{Γ Δ σ} → Ren Γ Δ → Val Γ σ → Val Δ σ
...
renval {σ = nat} α v   = renNf α v
renval {σ = [ σ ]} α v = mapLV (renval {σ = σ} α) (renNe α) v
```

To rename a value of natural numbers we apply the renaming of normal forms, since we actually have a normal form. For lists, we map over the list with two functions. The first one is applied if the element at hand is a value and the second one is applied if the element is a neutral term.

```
eval : ∀{Γ Δ σ} → Env Γ Δ → Tm Γ σ → Val Δ σ
...
eval γ ze           = nze
eval γ (su t)       = nsu (eval γ t)
eval γ (rec z f n)  = natfold (eval γ z) (eval γ f) (eval γ n)
eval γ nil          = nilLV
eval γ (cons h t)   = consLV (eval γ h) (eval γ t)
eval γ (fold z f n) = listfold (eval γ z) (eval γ f) (eval γ n)
```

For evaluating `ze` and `su` we use the constructors from normal forms and for `nil` and `cons` we use the constructors from `ListVal`. `rec` and `fold` are turned into their semantic counterparts which are defined as follows.

```
natfold : Val Γ σ → Val Γ (σ ⇒ σ) → Val Γ nat → Val Γ σ
natfold z f (nenat x) = reflect σ (nrec (reify _ z) (reify _ f) x)
natfold z f nze       = z
natfold z f (nsu n)   = f renId (natfold {σ = σ} z f n)

listfold : Val Γ τ → Val Γ (σ ⇒ τ ⇒ τ) → Val Γ [ σ ] → Val Γ τ
listfold z f (neLV x)      = reflect τ (nfold (reify _ z) (reify _ f) x)
listfold z f nilLV         = z
listfold z f (consLV x xs) = (f renId x) renId (listfold z f xs)
```

The proofs in `evallem` for the constructors are straightforward. For both `rec` and `fold` case, we use a property that a renaming can be pushed inside a semantic fold.

```
renvalnatfold : (ρ : Ren Γ Δ) →
    renval ρ (natfold z f n)
    ≅
    natfold (renval ρ z) (renval ρ f) (renval ρ n)

renvallistfold : (ρ : Ren Γ Δ) →
    renval ρ (listfold z f n)
    ≅
    listfold (renval ρ z) (renval ρ f) (renval ρ n)
```

For `reneval` and `subeval` the additional proofs are quite simple. Mostly they are about using the inductive hypothesis in the right place.

For normal forms, we are only going to show the additions to the definition of normal forms and neutral terms. Renaming normal forms and neutral terms is very similar to renaming terms and so are the proofs, we will skip them here.

```
mutual
  data Nf (Γ : Con) : Ty → Set where
    ...
    nenat : Ne Γ nat → Nf Γ nat
    ne[]  : ∀{σ} → Ne Γ [ σ ] → Nf Γ [ σ ]
    nze   : Nf Γ nat
    nsu   : Nf Γ nat → Nf Γ nat
    nnil  : ∀{σ} → Nf Γ [ σ ]
    ncons : ∀{σ} → Nf Γ σ → Nf Γ [ σ ] → Nf Γ [ σ ]

  data Ne (Γ : Con) : Ty → Set where
    ...
    nrec  : ∀{σ} → Nf Γ σ → Nf Γ (σ ⇒ σ) → Ne Γ nat → Ne Γ σ
    nfold : ∀{σ τ} →
        Nf Γ τ → Nf Γ (σ ⇒ τ ⇒ τ) → Ne Γ [ σ ] → Ne Γ τ
```

The additions to `reify` and `reflect` are the following. For natural numbers, the algorithm is the same as it was for base types. To reify a value of type list, we need to look which one of the three constructors of `ListVal` we have. To reflect a neutral list, we just wrap it in the `ListVal` constructor for neutral lists. This is what makes this special data type useful for us.

```
mutual
  reify : ∀{Γ} σ → Val Γ σ → Nf Γ σ
  ...
  reify nat v               = v
  reify [ σ ] (neLV x)      = ne[] x
  reify [ σ ] nilLV         = nnil
  reify [ σ ] (consLV x xs) = ncons (reify _ x) (reify _ xs)

  reflect : ∀{Γ} σ → Ne Γ σ → Val Γ σ
  ...
  reflect nat n   = nenat n
  reflect [ σ ] n = neLV n
```

We do not need to change anything about our normalisation function, which is just composition of `reify` and `eval`. To scale up the proofs of soundness and completeness, we need to extend the logical relation `R`.

```
_∋_R_ : ∀{Γ} σ → (t : Tm Γ σ) → (v : Val Γ σ) → Set
...
nat ∋ t R nenat x = t ∼ embNe x
nat ∋ t R nze      = t ∼ ze
nat ∋ t R nsu v    = Σ (Tm _ nat) (λ t' → t ∼ su t' × nat ∋ t' R v)
[ σ ] ∋ t R neLV x       = t ∼ embNe x
[ σ ] ∋ t R nilLV        = t ∼ nil
[ σ ] ∋ t R consLV v vs = Σ (Tm _ σ) (λ h → Σ (Tm _ [ σ ])
     (λ hs → t ∼ cons h hs × σ ∋ h R v × [ σ ] ∋ hs R vs))
```

In the successor case, `v` is the predecessor of `nsu v`. To relate `v` with a term, we need to also have the predecessor of `t`. Because of this, the definition is an existentially quantified type, which says that there is a term `t'`, such that `su t'` is convertible to `t` and `t'` is related to `v`. The situation is similar for lists in the cons case, only there we need to have two existentially quantified types, one for the head and one for the tail of the list.

```
natfoldR : σ ∋ z R zv → (σ ⇒ σ) ∋ f R fv → nat ∋ n R nv →
           σ ∋ (rec z f n) R natfold zv fv nv

listfoldR : τ ∋ z R zv → (σ ⇒ τ ⇒ τ) ∋ f R fv →
            [ σ ] ∋ xs R xsv →
            τ ∋ (fold z f xs) R listfold zv fv xsv
```

To prove the fundamental theorem, we use two new lemmas, `natfoldR` and `listfoldR` in the cases for `rec` and `fold`. The first property says that we can relate the syntactic construct `rec` with its semantic counterpart `natfold` if their corresponding components are related by the logical relation `R`. `listfoldR` is similar, just for lists. For `reifyR` we need to induct also on the value we are reifying in the case of natural numbers and lists. The modifications to `fund-thm` and `reflectR` are straightforward. Once we have proved the fundamental theorem and modified `reifyR` and `reflectR` we are done, the remainder of the correctness proofs do not need any changes.

We will now look at examples of evaluation and normalisation on natural numbers and lists. First, we define addition on natural numbers.

```
add : ∀{Γ} → Tm Γ (nat ⇒ nat ⇒ nat)
add = lam (lam (rec (var vze) (lam (su (var vze))) (var (vsu vze))))
```

This may seem a bit confusing at first, but what it basically says is this. If the first argument is zero, then the result is the second argument. If the first argument is something else, then we apply that many successor constructors to the second argument. A more convenient notation would be $add = \lambda x.\lambda y.rec\ y\ (\lambda z.su\ z)\ x$.

Evaluating the addition function gives us a really similar result on values. `rec` has been replaced with `natfold` and we have the extra arguments for the value level functions.

```
λ {Δ} α v → λ {Δ₁} α₁ v₁ →
     natfold v₁ (λ {Δ₂} α₂ v₂ → nsu v₂) (renNf α₁ v)
```

Reifying this (normalising the original term) gives us back the original term. We now try to add together two natural numbers, two and two. Thus, we are evaluating the following term.

```
app (app add (su (su ze))) (su (su ze))
```

Since the function add is applied to two concrete arguments, not variables, the computation can be fully performed. Evaluating this term gives the number four as the result.

```
nsu (nsu (nsu (nsu nze)))
```

Since we represent values of natural numbers as normal forms, the normal form of the term is exactly the same as the result of evaluation.

For lists, we look at folding a list of natural numbers. As the initial list, we will have a list with three elements: zero, one and two.

```
0,1,2 : ∀{Γ} → Tm Γ [ nat ]
0,1,2 = cons ze (cons (su ze) (cons (su (su ze)) nil))
```

By folding over this list with the addition operation and zero as the default value, we are actually summing together the elements of the list.

```
fold ze add 0,1,2
```

Evaluating this term gives us the number three as the result.

```
nsu (nsu (nsu nze))
```

Again, due to our representation of values of natural numbers, the result of evaluation is the same as the result of normalisation.

3.4.2 Pairs and streams

We will represent pairs as regular Agda pairs defined in Data.Product. A pair is a record with two fields: the first projection and the second projection. The type of the second projection depends on the value of the first projection. We will not use this dependently typed aspect of Agda pairs for our representation of pair values. This definition of pairs is the same which we have used for existential quantification.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

The non-dependent version can be defined as

```
A × B = Σ A λ _ → B
```

We are going to define an infinite data structure to represent values of type stream. Our streams are parametrized by A, meaning that they contain values of type A. Every stream has a head (which is a single element) and a tail (which is another stream). This seems similar to lists, but the use of coinductive records allows us to have infinite streams, whereas lists are finite.

```
record Stream (A : Set) : Set where
  coinductive
  field shead : A
        stail : Stream A
```

Since streams are infinite, we cannot use the same equality which we did for finite data types. For streams, we are going to use bisimilarity.

```
record _S∼_ {A : Set}(s s' : Stream A) : Set where
  coinductive
  field hd∼ : shead s ≅ shead s'
        tl∼ : stail s S∼ stail s'

postulate SEq : ∀{A} → {s s' : Stream A} → s S∼ s' → s ≅ s'
```

`S∼` is strong bisimilarity for streams. The definition of `S∼` tells us, that if two streams are bisimilar, then inspecting the two streams in lockstep, we will always have that the heads are equal and the tails are bisimilar. We postulate that `S∼` implies equality. We cannot prove this, but this allows us to continue using equality in our proofs. The fact that the equality of streams is not just strong bisimilarity can be seen as a flaw in Agda.

Next, we define `lookup` and `tabulate` which are the semantic counterparts of `proj` and `tup` from terms.

```
lookup : ∀{A} → (s : Stream A) → (ℕ → A)
lookup s zero    = shead s
lookup s (suc n) = lookup (stail s) n

tabulate : ∀{A} → (ℕ → A) → Stream A
shead (tabulate f) = f zero
stail (tabulate f) = tabulate (λ n → f (suc n))
```

To look up the element at position zero in the stream, we return its head element. For the successor case, we look up the element at the preceding position in the tail. Accessing the field of a record can be done using the field name as a function and giving it the record as a parameter. This is possible only if the namespace of the record has been opened.

`tabulate` constructs a stream from a function. As the result is an infinite structure, we do not want to compute all of it at once. To avoid that, we use copatterns [1], a feature of Agda, which can be enabled by the `copatterns` compiler flag. In the definition of `tabulate`, instead of defining how to transform a function to a stream, we define it by what observations we can make on the result. Observations consist of an experiment (copattern) and its outcome (the right hand side of =). We can observe, that taking the head of the tabulated function `f` is the result of applying the function `f` to `zero`. We can also observe, that taking the tail of the tabulated function is the same as tabulating the original function `f` with the indices shifted by one.

Next, we show that `lookup` and `tabulate` cancel each other out.

```
lookuptab : ∀{A} → (f : ℕ → A) → (n : ℕ) →
    lookup (tabulate f) n ≅ f n
```

```
tablookup : ∀{A} → (s : Stream A) → tabulate (lookup s) S∼ s
```

The proof of `lookuptab` is by induction on the argument `n`. As `S∼` is an infinite structure, the proof of `tablookup` is defined using copatterns.

We will represent values of pair types as actual pairs and values of stream types as the coinductive record defined earlier.

```
Val : Con → Ty → Set
...
Val Γ (σ ∧ τ) = Val Γ σ × Val Γ τ
Val Γ < σ >   = Stream (Val Γ σ)
```

To rename a pair, we just rename the components. Renaming a stream is defined using copatterns. Taking the head of a renamed stream is the result of renaming the head element of the stream and taking the tail of the renamed stream is the result of renaming the tail.

```
renval : ∀{Γ Δ σ} → Ren Γ Δ → Val Γ σ → Val Δ σ
...
renval {σ = σ ∧ τ} α v =
    (renval {σ = σ} α (proj₁ v)) , (renval {σ = τ} α (proj₂ v))
shead (renval {σ = < σ >} α v) = renval {σ = σ} α (shead v)
stail (renval {σ = < σ >} α v) = renval {σ = < σ >} α (stail v)
```

A renaming can be pushed inside `tabulate` and `lookup`, which is described by the following two properties. The proof of `renvallookup` is by induction on the natural number `n`. The proof of `renvaltab` uses copatterns, since `S∼` is an infinite structure.

```
renvallookup : (α : Ren Γ Δ)(s : Stream (Val Γ σ))(n : ℕ) →
    renval α (lookup s n) ≅ lookup (renval α s) n

renvaltab :(f : ℕ → Val Γ σ) → (α : Ren Γ Δ) →
    renval α (tabulate f) S∼ tabulate (λ n → renval α (f n))
```

Now we will look how to evaluate pairs and streams.

```
eval : ∀{Γ Δ σ} → Env Γ Δ → Tm Γ σ → Val Δ σ
...
eval γ (a ,, b)   = (eval γ a) , (eval γ b)
eval γ (fst t)    = proj₁ (eval γ t)
eval γ (snd t)    = proj₂ (eval γ t)
eval γ (proj n s) = lookup (eval γ s) n
eval γ (tup f)    = tabulate (λ n → eval γ (f n))
```

To evaluate a pair, we evaluate both of its components and then combine the values to a pair. To evaluate `fst` and `snd`, we first evaluate the pair and then project out the result. For `proj` and `tup`, we need to transform them to `lookup` and `tabulate`.

In the proof of `evallem`, the cases for pairs are similar to other proofs. For the `proj` case we need to apply `renvallookup` and for the `tup` case we need to apply `renvaltab` to make the proof succeed.

For `reneval` and `subeval` the additional proofs are mostly about applying the inductive hypothesis in the right place.

To the definition of normal forms we add cases for constructing a pair and constructing a stream. To the definition of neutral terms we add cases for projections out of pairs and streams.

```
mutual
  data Nf (Γ : Con) : Ty → Set where
    ...
    _,-,_ : ∀{σ τ} → Nf Γ σ → Nf Γ τ → Nf Γ (σ ∧ τ)
    ntup  : ∀{σ} → (ℕ → Nf Γ σ) → Nf Γ < σ >

  data Ne (Γ : Con) : Ty → Set where
    ...
    nfst  : ∀{σ τ} → Ne Γ (σ ∧ τ) → Ne Γ σ
    nsnd  : ∀{σ τ} → Ne Γ (σ ∧ τ) → Ne Γ τ
    nproj : ∀{σ} → ℕ → Ne Γ < σ > → Ne Γ σ
```

To reify a pair, we just reify its components and from the results construct a pair of normal forms. To reify a stream, we reify the results of the lookup function on the stream. To reflect a pair, we reflect its components. To reflect a stream, we tabulate the function which reflects the projections.

```
mutual
  reify : ∀{Γ} σ → Val Γ σ → Nf Γ σ
  ...
  reify (σ ∧ τ) v = reify σ (proj₁ v) ,-, reify τ (proj₂ v)
  reify < σ > v   = ntup (λ n → reify σ (lookup v n))

  reflect : ∀{Γ} σ → Ne Γ σ → Val Γ σ
  ...
  reflect (σ ∧ τ) n = (reflect σ (nfst n)) , (reflect τ (nsnd n))
  reflect < σ > n   = tabulate (λ a → reflect σ (nproj a n))
```

The additions to the logical relation `R` are rather small compared to the case for lists and natural numbers.

```
_∋_R_ : ∀{Γ} σ → (t : Tm Γ σ) → (v : Val Γ σ) → Set
...
(σ ∧ τ) ∋ t R v = σ ∋ fst t R proj₁ v × τ ∋ snd t R proj₂ v
< σ > ∋ t R v   = ∀ n → σ ∋ proj n t R lookup v n
```

A syntactic pair is related to a semantic pair if their first projections are related and their second projections are related. A syntactic stream is related to a semantic stream if the results of the lookup functions `proj` and `lookup` are related for every index `n`.

Proving the fundamental theorem for the projections `fst`, `snd` and `proj` is easy, we just need to project out the right element from the proof of the whole. For `,,` we need to show that the theorem holds for both components. For `tup` we need to show that the theorem holds for every projection of the stream. For `reifyR` and `reflectR` we need to show that every projection of the pair or stream satisfies the same property, which follows by corresponding inductive hypothesis. This concludes the correctness proof of these extensions.

We will now look at examples of normalisation for pairs and streams. First, we look at taking the first projection of a pair consisting of a variable of type pair as the first and second projection. Since $\iota \wedge \iota$ is in the last position in the context, we know that `var vze` is of type $\iota \wedge \iota$.

```
fst-proj : ∀{Γ} → Tm (Γ < (ι ∧ ι)) (ι ∧ ι)
fst-proj = fst (var vze ,, var vze)
```

Evaluating this gives us the following value, a meta level pair.

```
ne (nfst (nvar vze)) , ne (nsnd (nvar vze))
```

Reifying and embedding it gives us the normal form of the original term, where the variable `var vze` has been η-expanded into a pair.

```
fst (var vze) ,, snd (var vze)
```

For streams, we first need to define the function which will be used for constructing the stream. We define it to give back a syntactic natural number one greater than the argument it was given. This means that the resulting stream has elements starting from one and increasing by one.

```
plus-one : ∀{Γ} → ℕ → Tm Γ nat
plus-one zero    = su ze
plus-one (suc n) = su (plus-one n)
```

To create a stream from it, we must apply the `tup` constructor to it.

```
stream : ∀{Γ} → Tm Γ < nat >
stream = tup plus-one
```

Evaluating the resulting stream gives us the following value. The result of looking up element at position `n` in the stream is the value of the term given by the function `plus-one` applied to `n`.

```
tabulate (λ n → eval idE (plus-one n))
```

Reifying this value gives us

```
ntup (λ n → lookup (tabulate (λ n₁ → eval idE (plus-one n₁))) n)
```

This is basically the result from evaluation wrapped inside `lookup` which is then wrapped inside the tabulation constructor for normal forms.

Normalising the first projection of the stream

```
proj (suc zero) stream
```

gives us the normal form of the element at that index.

```
nsu (nsu nze)
```

4 Conclusion

In this thesis, we have studied the problem of normalising programs in simply typed lambda calculus, which is the simplest form of a functional programming language. We worked our way up to a formally verified implementation of normalisation by evaluation for a basic system with only base and function types. We later extended our object language with some additional data types: natural numbers, lists, pairs, and streams. Both the implementation of the normaliser and the correctness proofs were done in the dependently typed programming language Agda.

The parts about streams are probably the most interesting. Handling infinite structures is intrinsically different from handling finite structures. Indeed, it is an active area of research [1]. When representing this in Agda, one also needs to represent this infinity in such a way that all the programs (and proofs) are still accepted by the termination checker. To accomplish this, we used copatterns, a new feature in Agda.

One possible direction for further development would be to use a more common definition of streams, where streams are constructed as an infinite unfolding of a function and a start state. The following is an example of an unfold on value level streams.

```
unfold : ∀{A S : Set} → (S → A × S) → S → Stream A
shead (unfold f s) = proj₁ (f s)
stail (unfold f s) = unfold f (proj₂ (f s))
```

In the current work, we have looked at specific examples of inductive and coinductive data types. Another possible direction for further work would be to look at an entire class of data types together. Also, one area which could be further investigated, is the part about the relative monadic properties of the whole normaliser, as we currently only focused on the evaluator.

This work was supported by the Estonian Science Foundation grant no. 9219.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming Infinite Structures by Observations. *SIGPLAN Not.*, 48(1):27–38, January 2013.

[2] Guillaume Allais, Conor McBride, and Pierre Boutillier. New Equations for Neutral Terms: A Sound and Complete Decision Procedure, Formalized. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, DTP '13, pages 13–24, New York, NY, USA, 2013. ACM.

[3] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Foundations of Software Science and Computational Structures*, pages 297–311, 2010.

[4] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Relative monads formalised. *Journal of Formalized Reasoning. Final version pending*, 2010.

[5] Henk Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[6] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 203–211. IEEE, 1991.

[7] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *INDAG. MATH*, 34:381–392, 1972.

[8] Peter Dybjer and Andrzej Filinski. Normalization and Partial Evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 137–192. Springer Berlin Heidelberg, 2002.

[9] R. O. Gandy. An early proof of normalisation by A. M. Turing. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 453–455. Academic Press, 1980.

[10] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. In *Gödel: Collected Works*, pages 241–251. Oxford University Press, 1958.

[11] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 1995.

[12] Hendrik Maarand. Formalisation of normalisation by evaluation for data types. `https://github.com/hendrikmaarand/lambda-calculus`, 2014.

[13] Per Martin-Löf. An Intuitionistic Theory of Types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, 1975.

[14] Ulf Norell. Dependently Typed Programming in Agda. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer Berlin Heidelberg, 2009.

[15] Gordon D. Plotkin. Lambda definability and logical relations. Technical report, University of Edinburgh, 1973.

[16] Paula G. Severi and Fer-Jan de Vries. Pure type systems with corecursion on streams: from finite to infinitary normalisation. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 141–152. ACM, 2012.

[17] Agda team. Agda. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2014.