Under consideration for publication in Math. Struct. in Comp. Science

Relative monads formalised

THORSTEN ALTENKIRCH[1†], JAMES CHAPMAN[2‡] and TARMO UUSTALU[2 §]

[1] *School of Computer Science, University of Nottingham,*
Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK
[2] *Institute of Cybernetics, Tallinn University of Technology,*
Akadeemia tee 21, 12618 Tallinn, Estonia

Received July 31, 2010

Relative monads are a generalisation of ordinary monads where the underlying functor need not be an endofunctor. In this paper, we describe a formalisation of the basic theory of relative monads in the interactive theorem prover and dependently typed programming language Agda. This comprises the requisite basic category theory, the central concepts of the theory of relative monads and adjunctions, compared to their ordinary counterparts, and two running examples from programming theory.

1 Introduction

Relative monads (Altenkirch, Chapman, and Uustalu 2010) are a recent generalisation of ordinary monads to cover similar structures where the underlying functor need not be an endofunctor. Our interest in this generalisation was triggered by some structures from programming theory that, in many ways, are strikingly similar to monads (even respecting the same laws), have similar programming applications, but nonetheless fail to be monads. Some examples of relative monads include untyped and typed λ-terms, finite-dimensional vector spaces, Hughes's arrows. Our research effort on relative monads is ongoing.

In this paper, we describe a formalisation of the basic theory of relative monads in the interactive theorem prover and dependently typed programming language Agda (Agda team 2010). This formalisation work is motivated by a number of basic observations.

First of all, by moving from a conventional functional programming language like Haskell (The Haskell Team 2010) to a dependently typed language like Agda (Agda team 2010), we gain the opportunity to formally certify our programs. Indeed, we are able to move from programs to "deliverables", which contain partial specifications of programs as types; and the correctness of the assertions can be automatically verified by a type checker.

† T. Altenkirch was supported by the Engineering and Physical Sciences Research Council (EPSRC) grant no. EP/G034109/1.
‡ J. Chapman and T. Uustalu were supported by the Estonian Centre of Excellence in Computer Science, EXCS, financed by the European Regional Development Fund.
§ T. Uustalu was also supported by the Estonian Science Foundation grant no. 6940.

Second, since monads are the most successful pattern in conventional functional programming, we should expect that they play a central role in dependently typed functional programming. But in fact, with dependent types, we should go beyond monads. Once we move to a more fine-grained type system, such as Agda's, it becomes clear that the conventional notion of a monad is unnecessarily restrictive, since it fails to account for the monad-like structures we encounter in programming that are not endofunctors. We would like to contend that, in a dependently typed setting, it is only natural to go on from monads to relative monads, as they are more general, but not more complex. (In Haskell, the opportunity does not arise: we can only speak of endofunctors, in fact, only endofunctors on the category of Haskell types.)

Third, we think that the present work is an interesting example of the mutual influence of theoretical work and formalisation. Not only does the formal development necessitate a review and refinement of structures inherited form mathematics, but the development generates demands on the language and system used—here Agda. In fact, in the case of our study of relative monads, we worked on the formalisation in parallel with the theoretical work and indeed both influenced each other.

The third point is reflected in our approach to extensionality. Agda's propositional equality is proof irrelevant but not extensional with respect to functions. We use both properties of propositional equality in our formalisation. We prove proof irrelevance and assume extensionality of functions. At first, it sounds somewhat drastic to just assume extensionality: Is it not dangerous? Why do we not just assume absurdity and have done with it? However, it has been shown by Hofmann (Hofmann 1995) that extensionality is a conservative extension of intensional type theory. Altenkirch et al. (Altenkirch, McBride and Swierstra 2007) have shown that it possible to introduce a type theory with an extensional propositional equality without giving up decidable type checking and other computational properties of the system. Another deficiency of Agda's propositional equality is that it does not support quotients, but we have no need for them in this formalisation.

Previous formalisations of category theory in type theory and higher-order logic include (Dyckhoff 1985; Altucher and Panangaden 1990; Dybjer and Gaspes 1994; Huet and Saïbi 1998; Carvalho 1998; Wilander 2005; Coquand and Spiwack 2007; Sozeau 2010; O'Keefe 2004; Dawson 2007). The formalisations in intensional type theory have used setoids (i.e., sets with an equivalence relation) to model the homsets of a category while being content with a set for the collection of objects. Our experience is that using setoids quickly becomes unwieldy and introduces an unacceptable overhead to the formal development. Moreover, it is not clear what should be a set and what should be a setoid. E.g., in the case of a category, one could argue that both objects and homsets should be setoids to give a direct account of constructions such as the arrow category. These choices would then lead to many related but different implementations of the same concept, which is clearly unacceptable and also unfeasible.

Our intention in this work is not to develop a category theory library for Agda and we do not argue that the approach taken here is the only way or the best way to do this. Instead, this development has been carried out as part of our research on relative monads not as an after-the-fact verification, but as a day-to-day tool for prototyping and sanity checking. However, this development is intended to be a substantial experiment in formalising category theory in Agda for the purposes of finding out whether our approach and the type theory and implementation of Agda are fit for purpose. We will address these points in the conclusion.

We have carried out our formalisation in the development version of Agda (= version 2.2.7; the latest release version is 2.2.6). This is because of certain vital performance improvements in the handling of records in the new version. We have striven to make the development as self-contained as possible, not relying on libraries, for stability in this situation. The formalisation comes in two versions. The version described in the paper is without a universe hierarchy (uses `--type-in-type`) to simplify the presentation. But we also offer a stratified version relying on `--universe-polymorphism`, which is more bureaucratic, but might give more confidence. As already mentioned above, we make extensive use of proof-irrelevance of propositional equations (provable in Agda) and extensionality of functions (assumed). Both developments are available online (Chapman 2010).

Apart from some necessary basics of categories, the formalisation covers essentially Sections 2.1, 2.2 and 2.3 of (Altenkirch, Chapman, and Uustalu 2010) on relative monads, relative adjunctions, Kleisli and Eilenberg-Moore adjunctions, with a special focus on two examples: well-scoped untyped λ-terms and typed λ-terms. We are in the process of formalizing the later sections on relative monads as monoids in suitable functor categories and the examples of finite-dimensional vector spaces and Hughes's arrows. Progress in some of these directions depends on further improvements to Agda.

The paper is organised as follows. In Section 2, we introduce our approach using the example of monoids. As we discuss this example, we introduce Agda's syntax and develop the required utilities for our formalisation. In Section 3, we introduce the background category theory that we need in order to discuss relative monads: categories, functors, natural transformations, and functor categories. Next, we introduce relative monads and several examples (well-scoped and well-typed λ-terms) in Section 4. When we introduce relative versions of structures, we always relate them to their ordinary counterparts. In Section 5, we introduce relative adjunctions, describe how they can be composed to form relative monads, and how relative monads can be split into relative adjunctions. We also develop the examples further.

The paper assumes no knowledge of category theory or monads and adjunctions, in the sense that we define everything as we go. But familiarity with these topics will help the reader appreciate what we do.

2 Monoids

Basic algebraic structures can be defined as follows: first we define some data (e.g., a set and some operations on it), which provides the basic structure; and second we define some laws, which govern how the data (e.g., an operation) behaves. An example of something which fits this pattern is a monoid. A monoid is given by the following data: a set; a distinguished element; and a binary operation on the set.

$$(S, \epsilon, \cdot)$$

These data are subject to the following laws:

Left unit law
$$\epsilon \cdot m = m \quad \text{where } m \in S$$
Right unit law
$$m \cdot \epsilon = m \quad \text{where } m \in S$$
Associativity
$$(m \cdot n) \cdot o = m \cdot (n \cdot o) \quad \text{where } m, n, n \in S$$

In Agda (Agda team 2010), we can use a dependent record to represent a monoid quite naturally. Both the data and the laws are contained in the record as a sequence of fields. The laws are represented as propositional equations. The data is explicitly represented as a sequence of fields in a record which is itself a `Set` (Agda's notation for a type). Each field has an explicit type which can refer to the fields that came before (Eg. `ε`'s type is the preceding field `S`). The three equational laws are represented as fields in the same record:

```
record Monoid : Set where
  field S   : Set
        ε   : S
        _·_ : S → S → S
        lid : {m : S} → ε · m ≅ m
        rid : {m : S} → m · ε ≅ m
        ass : {m n o : S} → (m · n) · o ≅ m · (n · o)
```

Let us look more closely at the Agda syntax introduced in this definition: first, we begin the definition with `record` then the name of the record (`Monoid` in this case); next, we could give some parameters, but none are necessary here; finally, we say that this has type `Set` and end the line with `where`. The fields of the record are preceded by the `field` keyword. Next, we name three fields and give their types. Agda support unicode characters, so `ε` is a perfectly good identifier. Infix operators (like `_·_`) are named with underscores, indicating where the arguments go, and `_→_` is the non-dependent function space, indicating that, in the case of `_·_` that it takes two elements of `S` and always returns an element of `S`. Next we give the laws. They are fields, so they must have names and their types are the equations which they enforce. Universal quantification (or Π-type, they are the same in Agda) is written as `(a : A) → B`. In this definition we use implicit universal quantification `{a : A} → B`, as these arguments can often be inferred. This is just a notational convenience: we are free to leave explicit arguments implicit (by giving them as an underscore `_`) or give implicit arguments explicitly (by enclosing them in curly braces `{n}`).

Compare this type of definition with Haskell, where we can define the operations of a monoid and define an instance of a monoid (e.g. natural numbers, zero and addition), but we cannot show inside the system that this instance would obey the laws of a monoid. In Agda, we can do both: we can write programs that make use of algebraic structure and we can reason about them, and in the process make extra guarantees that the we really have the structures that we say we do. In fact, we go further than just allowing such guarantees to be state; we *insist* that they are met. A monoid carries with it the guarantee that the the monoid laws are satisfied. If it does not satisfy the laws, we cannot give it the type of monoid.

Our intention is not to *encode* monoids in type theory, but to explain what a monoid *is* in type theory, and then from this point on, use the language of type theory, as opposed to informal mathematical notation, to discuss monoids and other structures. Having given the example of monoids in both informal notation and formally in type theory, we will now only give formal definitions. If type theory is to be a language of mathematics, then one should be able to use it as a language of mathematical explanations. It is our opinion that Haskell is already quite effective as a language for explaining programming ideas, on a computer, or on a whiteboard, or the back of an envelope. It is our hope that type theory (as it is in Agda and related languages) can be a language for explaining not just programming ideas but also mathematical ideas.

Before moving on to consider the categorical machinery we will need later, we will first show, by way of an example, that natural numbers with zero and addition form a monoid.

First, we give the inductive definition of natural numbers in Agda. We define a inductive datatype with two constructors: `z` for zero; and `s` for successor which takes a natural number and returns another natural number.

```
data Nat : Set where
  z : Nat
  s : Nat → Nat
```

Any natural number can be seen to have either `z` or `s` as its outermost constructor. For this reason when writing functions that consume natural numbers we need only consider the canonical cases of `z` or `s n`.

We define addition as an infix operator `_+_` by recursion on the first argument. In Agda, we do not need to indicate that we are doing recursion on the first argument in our definition, instead Agda's termination checker checks that our recursion is valid after the fact. It is valid in this case, as our recursive call takes a structurally smaller first argument. Indeed, we use only structural recursion in this paper.

```
_+_ : Nat → Nat → Nat
z   + n = n
s m + n = s (m + n)
```

Next, we can show that `Nat`, `z` and `_+_` form a monoid:

```
Nat+Mon : Monoid
Nat+Mon = record{S   = Nat;
                 ε   = z;
                 _·_ = _+_;
                 lid = refl;
                 rid = ?;
                 ass = ?}
```

We have defined a new record `Nat+Mon` with type `Monoid` and begun to fill in the fields. The data is as follows: the underlying set is `Nat`; the unit element is `z`; and the binary operation is given by `_+_`. Next we must give proofs of the laws, specialised to this monoid. The left unit law is trivial; its type is `z + m = m`. This holds up to definitional equality (=). It is just the first line of the *definition* of `_+_`. Hence, it automatically computes to `m = m` and can be proved by reflexivity. The other two laws, whose types are `m + z = m` and `(m + n) + o = m + (n + o)` respectively, do not compute any further and require more elaborate proofs. In Agda, unfinished parts of a definition are denoted by a question mark `?`. It should be noted that the formalisation contains no unfinished parts. In this paper we will give some incomplete definitions and fill them in later (as we do here) and also leave some definitions incomplete. In the latter case we will give a sketch of the missing term but omit its precise form due to reasons of space and/or readibility.

Before proving these as lemmas and filling in the two question marks, we take the opportunity to examine propositional equality and a useful general lemma. In this paper, we rely on the following definition of propositional equality:

```
data _≅_ {A : Set} : {A' : Set} → A → A' → Set where
  refl : {a : A} → a ≅ a
```

This is a heterogeneous version of Martin-Löf's identity type (known as John Major equality). It allows us to state an equality between elements of different types. However, its only canonical inhabitant is reflexivity, where both the types and the terms on the either side of the equation are identical.

The useful lemma, which we will use very soon, states that functions take equal arguments to equal results (they `resp`ect equality):

```
resp : ∀{A}{B : A → Set}(f : ∀ a → B a){a a' : A} →
       a ≅ a' → f a ≅ f a'
resp f p = ?
```

Using the notation ∀, we can introduce an universally quantified variable, but leave its type to be inferred. In this case, the type of `A` must be `Set`, as it is the domain of `B`, and the type of `a` must be `A`, as it is applied to `B`. The type of the question mark is `f a ≅ f a'`. To carry out the proof, we pattern-match on `p`, which is a proof of the equation `a ≅ a'`. The only inhabitant of an equation is `refl`, and hence `a` and `a'` must be identical. By performing this pattern match, `a` and `a'` get unified in the context and the goal (the type of the question mark) is simplified to `f a ≅ f a`, which is inhabited by `refl` as well. The finished proof is just:

```
resp f refl = refl
```

In a situation where the same function is on the outside on both sides of an equation we are trying to prove, we can use `resp` to 'peel off' the function and reduce our problem to proving what is underneath.

Having stated and proved this lemma, we can now prove the right unit law for the `Nat`Monoid+ monoid: `{n : Nat} → n + z ≅ n`. The proof proceeds by induction (induction and recursion are the same in Agda) on the implicit argument `n`, which we provide explicitly here, so we can pattern-match on it. In the zero case, it is given by `refl` of type `z ≅ z`. In the successor case, the goal computes to `s (n + z) ≅ s n`. We use `resp` to peel off the function `s` and then apply our inductive hypothesis `rid+ {n}`.

```
rid+ : {n : Nat} → n + z ≅ n
rid+ {z}   = refl
rid+ {s n} = resp s (rid+ {n})
```

We can prove the associativity condition in a similar way. Here we need only refer to the first implicit argument explicitly:

```
ass+ : {m n o : Nat} → (m + n) + o ≅ m + (n + o)
ass+ {z}   = refl
ass+ {s m} = resp s (ass+ {m})
```

Having proved these two lemmas, we can finish `Nat+Mon`:

```
Nat+Mon : Monoid
Nat+Mon = record{S   = Nat;
                 ε   = z;
                 _·_ = _+_;
                 lid = refl;
                 rid = rid+;
                 ass = ass+}
```

3 Basic category theory

In this section, we develop some basic category theory that we will make use of later. A category is an algebraic structure like any other. We define it as before: we introduce some data, a set of objects, a set of morphisms between two objects, for any object, an identity morphism and, for any two morphisms between appropriate objects, a composition morphism. Then we give some equations which govern how this structure behaves: identity is left and right unit of composition, and composition is associative. These conditions are often represented pictorially as categorical diagrams. Categorical diagrams are a very nice representation of equations. In Agda, we just represent the equations directly using propositional equality.

```
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)
```

A simple example of a category is the category of small sets. The objects are sets and the morphisms are simple functions between them. The identity is given by the identity function and the composition is given by function composition. The laws hold definitionally.

```
Sets : Cat
Sets = record{Obj  = Set;
              Hom  = λ X Y → X → Y;
              iden = id;
              comp = λ f g → f • g;
              idl  = refl;
              idr  = refl;
              ass  = refl}
```

Another simple example is given any category $\mathbf{C}$, we can form the category $\mathbf{C}^{op}$ by turning round all the arrows. In Agda, we define this as a postfix operator `_Op`. The category `C Op` has the same objects as `C`. The morphisms from `X` to `Y` in `C Op` are the morphisms from `Y` to `X` in `C`. The identity in `C Op` is given by the identity of `C` and the composition in `C Op` is given by reversing the order of the composition of `C`. The left unit law follows from right identity of `C`, and the right unit from the left identity of `C`. The associativity follows from the associativity of `C` after applying symmetry.

```
_Op : Cat → Cat
C Op = record{Obj  = Obj C;
              Hom  = λ X Y → Hom C Y X;
              iden = iden C;
              comp = λ f g → comp C g f;
              idl  = idr C;
              idr  = idl C;
              ass  = sym (ass C)}
```

For the first time in the definition of `_Op`, we are referring to the fields of one record in the definition of another and how this works warrants further explanation. The type of the `Obj` field is `Set`, so we must give a set on the right of the `=` sign. But `Obj` appears again on the right and this time takes an argument `C`. When we define a record (e.g., `Cat`) Agda defines projections functions with the same names as the field names (e.g., `Cat.Obj`). The projections are functions from the record type to the field type (e.g., `Cat.Obj : Cat → Set`). We can open the name space of the record (e.g., `open Cat`) to bring the projections into scope and avoid having to give the name of the record as part of the projection name. We do this after every record and hence we can refer to the projections directly (e.g., `Obj`). Hence in the line `Obj = Obj C;`, the `Obj` on the left is the field name of `C Op` and the `Obj` on the right is the projection function from the record `Cat`. The remainder of the definition proceeds as described above.

Next we define functors. This time the record `Fun` is parametrised by the categories (`C` and `D`) at the domain and codomain of the functor. The record has fields for the *object map* and the *morphism map*. It also has fields for the two functor laws which guarantee that the morphism map does the right thing with the identity and composition morphisms, i.e., that identity in `C` is mapped to identity in `D` and that composition in `C` is mapped to composition in `D`. By including these laws in the definition of a functor, we guarantee that anything that has the right to call itself a functor respects the functor laws.

```
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)
```

Having defined functors (morphisms between categories), we can define natural transformations (morphisms between functors):

```
record NatT {C D}(F G : Fun C D) : Set where
  field cmp : ∀ {X} → Hom D (OMap F X) (OMap G X)
        nat : ∀{X Y}{f : Hom C X Y} →
              comp D (HMap G f) cmp ≅ comp D cmp (HMap F f)
```

The natural transformation is made up of its *components*, given by a function which for any object `X` in `C` (taken implicitly) gives us a morphism from `OMap F X` to `OMap G X`, and a *naturality condition*, which states that, given any morphism `f` in `C`, we can go along the morphism and then the component or we can go along the component first and then the morphism.

Given definitions of categories, functors and natural transformations we might aspire to defining a functor category where the objects are functors and the morphisms are natural transformations. Let us start to realise this aspiration:

```
FunctorCat : Cat → Cat → Cat
FunctorCat C D = record{Obj  = Fun C D;
                        Hom  = NatT;
                        id   = ?;
                        comp = ?;
                        idl  = ?;
                        idr  = ?;
                        ass  = ?}
```

The morphisms in this category are natural transformations, so we need to define the identity natural transformation and composition of natural transformations. The identity natural transformation is from a functor `F` to itself. The component at `X` in `C` must be a morphism in `D` from `OMap F X` to itself. This is just the identity morphism in `D` on the object `OMap F X`. The naturality condition computes to `comp D (HMap F f) (iden D)` ≅ `comp D (iden D) (HMap F f)`. which follows from the left and right unit laws (`lid` and `rid`) in `D`.

```
idNat : ∀{C D}{F : Fun C D} → NatT F F
idNat {C}{D}{F} = record{cmp = iden D;
                         nat = λ{X}{Y}{f} → trans (idr D) (sym (idl D))}
```

Composition proceeds analogously. Just like the components of the identity natural transformation are given by identity in `D`, the components of composition are given by composition in `D`, specifically, composition of the components of the natural transformation α and β. We omit the proof term of the naturality condition and leave it as `?`. The omitted proof follows from the associativity law for `D` and the naturality of α and β.

```
compNat : ∀{C D}{F G H : Fun C D} → NatT G H → NatT F G → NatT F H
compNat {C}{D} α β = record {cmp = comp D (cmp α) (cmp β); nat = ?}
```

To have a category, we must prove the that `idNat` is left and right unit of `compNat` and `compNat` is associative. Let us look at left unit law first:

```
idlNat : ∀{C D}{F G : Fun C D}{α : NatT F G} → compNat idNat α ≅ α
idlNat  = ?
```

To prove `idlNat` we must prove that the natural transformations `compNat idNat α` and `α` are (propositionally) equal. Natural transformations are records, and to prove that two records are equal we must prove that their fields are equal. In this case this means we must prove that the components are equal and also the the proofs(!) of naturality are equal. Worry not, help is at hand in the form of proof irrelevance:

```
ir : {A A' : Set}{a : A}{a' : A'}{p q : a ≅ a'} → p ≅ q
ir {p = refl}{q = refl} = refl
```

In Agda, all proofs of a particular propositional equation are equal. The canonical inhabitant of a propositional equations is `refl`. When writing a program or proving a theorem by pattern matching, we need only consider canonical representatives of the arguments, and `refl` equals `refl` by (surprise, surprise) `refl`. So, we can just use proof irrelevance to avoid proving that the naturality proofs are equal right? Well, no. The problem is that the types of the two proofs (the equations of which they are proofs) are different. The naturality condition for `compNat (idNat G) α` is

```
comp D (HMap G f) (comp D (iden D) (cmp α)) ≅
comp D (comp D (iden D) (cmp α)) (HMap F f)
```

and for `α` it is

```
comp D (HMap G f) (cmp α) ≅ comp D (cmp α) (HMap F f)
```

These equations are different, but their respective left and right hand sides are provably equal. We just need to plug in the proofs that the components are equal in the right place. We define a lemma `fixtypes` to deal with this common situation where we have two equality proofs between provably equal equations. Once we have ensured that the equations (or rather their types) are equal, we can apply proof irrelevance to show that their proofs are equal.

```
fixtypes : ∀{A A'}{a a' : A}{a'' a''' : A'}
           {p : a ≅ a'}{q : a'' ≅ a'''} →
           a ≅ a'' → a' ≅ a''' → p ≅ q
fixtypes refl refl = ir
```

Given this principle, we are now in a position to prove that the naturality conditions are equal, or we would be, were it not for the fact that the naturality conditions are actually functions: they hold for any `X`, `Y` and `f`. Here Agda's notion of propositional equality lets us down. We really need extensionality: we would like functions to be equal, if they do the same thing on all possible arguments. We postulate this principle as follows:

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

```
                  (∀ a → f a ≅ g a) → f ≅ g
```

We could now prove the left identity law above by proving only that the components are equal and dispensing with the naturality conditions, using the facilities we just introduced. However, we can do better than this, we can prove once-and-for-all that two natural transformations are equal, if their components are equal. We can prove that the naturality conditions are equal, by plugging in the proof that components are equal using extensionality and by invoking proof irrelevance via `fixtypes`. We omit the proof term.

```
NatTEq : {C D : Cat}{F G : Fun C D}{α β : NatT F G} →
         cmp α ≅ cmp β → α ≅ β
NatTEq p = ?
```

This turns out to be a common pattern in our formalisation: whenever we need to prove that two records are equal and the records are made up of some fields giving some data and some laws that govern it, we need only prove equality of the data.

Having defined `NatTEq`, we can use it to reduce the problem of proving the three lemmas to proving that the components are equal.

```
idlNat : ∀{C D}{F G : Fun C D}{α : NatT F G} → compNat idNat α ≅ α
idrNat : ∀{C D}{F G : Fun C D}{α : NatT F G} → compNat α idNat ≅ α
assNat : ∀{C D}{E F G H : Fun C D}
         {α : NatT G H}{β : NatT F G}{η : NatT E F} →
         compNat (compNat α β) η ≅ compNat α (compNat β η)
```

Their components are equal by the left and right identity, and associativity laws of the category `D` respectively (and extensionality).

Having proved these three lemmas, we can now complete our definition of a functor category:

```
FunctorCat : Cat → Cat → Cat
FunctorCat C D = record{Obj  = Fun C D;
                        Hom  = NatT;
                        id   = idNat;
                        comp = compNat;
                        idl  = idlNat;
                        idr  = idrNat;
                        ass  = assNat}
```

4 Monads and relative monads

Relative monads are a generalisation of monads, from endofunctors, to functors that may go between different categories. We define ordinary monads first before introducing relative monads. In Manes' style, a monad on a category `C` consists of three pieces of data: a map `T` from objects of `C` to objects of `C`; an operation η that, for any object `X` of `C`, gives a morphism in `C` from `X` to `T X`; and an operation on morphisms of `C` called `bind` that (for any objects `X` and `Y`) lifts morphisms from `X` to `T Y` to a morphism from `T X` to `T Y`.

```
record Monad (C : Cat) : Set where
  field T    : Obj C → Obj C
```

```
        η    : ∀ {X} → Hom C X (T X)
        bind : ∀{X Y} → Hom C X (T Y) → Hom C (T X) (T Y)
        law1 : ∀{X} → bind (η {X}) ≅ iden C {T X}
        law2 : ∀{X Y}{f : Hom C X (T Y)} → comp C (bind f) η ≅ f
        law3 : ∀{X Y Z}{f : Hom C X (T Y)}{g : Hom C Y (T Z)} →
               bind (comp C (bind g) f)  ≅ comp C (bind g) (bind f)
```

A simple example of a monad is the so-called maybe monad:

```
data Maybe (A : Set) : Set where
  Just : A → Maybe A
  Nothing : Maybe A
```

Note first that this is not yet a monad; it is just a set. `Maybe` gives a canonical way to add a distinguished element to a set `A`. This element is usually used to denote failure of a function that would otherwise return an element of `A`. We define a function `mbind` which lifts an unreliable function on reliable input to an unreliable function on unreliable input:

```
mbind : {X Y : Set} → (X → Maybe Y) → Maybe X → Maybe Y
mbind f (Just x) = f x
mbind f Nothing  = Nothing
```

Next we prove two properties about `mbind`: that giving it `Just` as an argument (quite a reliable unreliable function) is the same as the identity function;

```
mlaw1 : ∀{A}(a : Maybe A) → mbind Just a ≅ id a
mlaw1 (Just a) = refl
mlaw1 Nothing  = refl
```

and that, given two appropriate functions, composing their lifted versions, or lifting one, composing, and then lifting the result gives the same outcome:

```
mlaw3 : ∀{A B C}{f : A → Maybe B}{g : B → Maybe C}(a : Maybe A) →
        mbind (mbind g • f) a ≅ (mbind g • mbind f) a
mlaw3 (Just a) = refl
mlaw3 Nothing  = refl
```

Having defined the `Maybe` type, `mbind`, and proved the two properties, we have all we need to define the `Maybe` monad on the category `Sets`. We use extensionality and `mlaw1` and `mlaw3` to prove the first and third monad laws, the second one holds definitionally.

```
MaybeMonad : Monad Sets
MaybeMonad = record{T    = Maybe;
                   η    = Just;
                   bind = mbind;
                   law1 = ext mlaw1;
                   law2 = refl;
                   law3 = ext mlaw3}
```

The definition of the relative monad is quite similar to that of the ordinary monad:

```
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)
```

The record `RMonad` takes the source and target categories `C` and `D` as implicit arguments and a functor `J` between them as an explicit argument. The idea is that functor `J` is some kind of embedding-like thing which *repairs* the mismatch in the categories in the remainder of the definition. `T` is a mapping from objects of `C` to objects of `D`, so η's type must be adjusted from the previous definition: for any object `X` in `C`, η gives a morphism in `D` from `OMap J X` to `T X`. `bind` must also be adjusted: this time, it takes as input maps in `D` from `OMap J X` to `T X`. The three laws remain the same, but their types are adjusted by replacing `C` with `D` and inserting `OMap J` where necessary.

A monad is a special case of a relative monad where the functor `J` is just the identity functor:

```
IdF : ∀ C → Fun C C
IdF C = record{OMap = id; HMap = id; fid = refl; fcomp = refl}
```

To define the special case there is nothing to prove we just plug in the contents of the fields of the monad into the fields of the relative monad:

```
specialM : {C : Cat} → Monad C → RMonad (IdF C)
specialM {C} M = record{T    = T M;
                        η    = η M;
                        bind = bind M;
                        law1 = law1 M;
                        law2 = law2 M;
                        law3 = law3 M}
```

Given a monad on some category `D`, we can restrict it to a relative monad by post-composing `T` with any functor `J` into `D`.

```
restrictM : {C D : Cat}(J : Fun C D) → Monad D → RMonad J
restrictM J M = record{T    = T M • OMap J;
                       η    = η M;
                       bind = bind M;
                       law1 = law1 M;
                       law2 = law2 M;
                       law3 = law3 M}
```

Under favourable conditions, restriction is the right adjoint of an interesting adjunction. Its left adjoint is investigated in Section 4 of (Altenkirch, Chapman, and Uustalu 2010).

4.1 Well-scoped λ-terms

So, our first example of a relative monad is just an ordinary non-relative one. A more informative example is the well-scoped λ-terms. They are well-scoped because their type `Tm` is indexed by the number of variables in scope and ensures that they cannot refer to any others. We will show that `Tm` is the object map `T` of a relative monad on where `J` is functor whose object map is given by `Fin`:

```
data Fin : Nat → Set where
  fz : ∀{n} → Fin (s n)
  fs : ∀{n} → Fin n → Fin (s n)
```

Variables are de Bruijn indices and represented as elements of finite sets. The finite set `Fin 0` is empty, the finite set `Fin 1` contains one element `fz`, the finite set `Fin 2` contains two elements `fz` and `fs fz`, etc.

```
data Tm : Nat → Set where
  var : ∀{n} → Fin n → Tm n
  lam : ∀{n} → Tm (s n) → Tm n
  app : ∀{n} → Tm n → Tm n → Tm n
```

Looking at the definition of the type `Tm`, we see that the `var` constructor embeds elements of `Fin n` (variables from a `n`-element scope) into the set `Tm n` (terms over this scope). The `lam` constructor is a scope-safe λ-abstraction. It takes a body over `s n` variables and gives back a term over `n` variables (the bound variable has been abstracted by the λ). The `app` constructor is for scope-safe application of a function over `n` variables to an argument over `n` variables. If the function and argument are over different numbers of variables we cannot even form the term corresponding to their application.

To show that the well-scoped terms form a relative monad, we must first fix `J` and the categories between which it operates. The object map of `J` should be `Fin`, hence the sets of objects of the source and target categories should be `Nat` and `Set`. We define a category `Nats` whose objects are natural numbers, to be understood as possible context sizes. The morphisms are functions from `Fin m → Fin n`. These morphisms can be thought of as renamings from `m` variables to `n` variables. Identity and composition are given by the identity function and composition of functions respectively. As is the case for the category of sets (where the morphisms are also functions), the laws hold definitionally.

```
Nats : Cat
Nats = record{Obj  = Nat;
              Hom  = λ m n → Fin m → Fin n;
              iden = id;
              comp = λ f g → f • g;
              idl  = refl;
              idr  = refl;
              ass  = refl}
```

The target of the functor `J` is the category of sets `Sets` defined earlier. We now give the definition for the functor `J`. On objects, it is `Fin` and, on maps, it is the identity function. The laws hold trivially due to the trivial operation on maps:

```
FinF : Fun Nats Sets
FinF = record{OMap  = Fin;
              HMap  = id;
              fid   = refl;
              fcomp = refl}
```

We can now begin to show that we have a relative monad on `FinF`:

```
TmRMonad : RMonad FinF
TmRMonad = record{T = Tm; η = var; bind = ?; law1 = ?; law2 = ?; law3 = ?}
```

We fix the object map `T` to be `Tm` which gives `η` the type `{n : Nat} → Fin n → Tm n`. It is just the constructor `var`. The type of bind is more interesting: `{m n : Nat} → (Fin m → Tm n) → Tm m → Tm n`. A function from `Fin m → Tm n` is a substitution; for every variable in `Fin m` it gives a term over `n` variables. `var` is the identity substitution: it replaces a variable with itself (seen as term). `bind` performs a substitution; it applies a substitution from `m` variables to terms over `n` variables to a term over `m` variables to give a term over `n` variables. Put another way it lifts a substitution which is an operation on variables to an operation on terms. There are several ways to proceed from here to define substitution. We chose to first define renaming (we already have a category of renamings) and use it to define substitution. This approach has a clear mathematicaly structure, a simple termination argument (all our definitions are structurally recursive), and makes the proofs of the monad laws relatively straightforward.

First, we define some type synonyms for renamings, the identity renaming, and composition of renaming. This makes the types that follow easier to read.

```
Ren : Nat → Nat → Set
Ren m n = Fin m → Fin n

renId : ∀{n} → Ren n n
renId = id

renComp : ∀{m n o} → Ren n o → Ren m n → Ren m o
renComp f g = f • g
```

Next, we define the action of renaming `ren` which applies a renaming to a term, or, put another way, takes a renaming from `m` to `n` and lifts it to a function from terms over `m` variables to terms over `n` variables. We define it by recursion on the term. In the variable case, we peel off the `var` constructor, apply the renaming, and replace the `var`. In the application case, we can just push the renaming down to the subterms. In the case of λ-abstraction, we weaken the renaming using `wk` (defined below) and then apply it to the body.

```
ren : ∀{m n} → Ren m n → Tm m → Tm n
ren f (var i)   = var (f i)
ren f (app t u) = app (ren f t) (ren f u)
ren f (lam t)   = lam (ren (wk f) t)
```

To push renaming under a binder, we need a helper function `wk` which weakens a renaming: the variable `fz` is passed straight through and the others are mapped to the successor (the weakening of variables) of their original values:

```
wk : ∀{m n} → Ren m n → Ren (s m) (s n)
wk f fz     = fz
wk f (fs i) = fs (f i)
```

Next, we prove that `wk` maps the identity renaming to itself and `ren` maps the identity renaming to the identity function:

```
wkid : ∀{n}(i : Fin (s n)) → wk renId i ≅ renId i
wkid fz     = refl
wkid (fs i) = refl

renid : ∀{n}(t : Tm n) → ren renId t ≅ id t
renid (var i)   = refl
renid (app t u) = resp2 app (renid t) (renid u)
renid (lam t)   = resp lam (trans (resp (λ f → ren f t) (ext wkid))
                                  (renid t))
```

Notice that, in the proof `renid`, we need the proof `wkid`, as, in the program `ren`, we needed the program `wk`.

Next, we prove the `wk` takes composition of renamings to composition of renamings and `ren` takes composition of renamings to composition of functions. Notice again that the `lam` cases require the corresponding properties for `wk`.

```
wkcomp : ∀{m n o}(f : Ren n o)(g : Ren m n)(i : Fin (s m)) →
            wk (renComp f g) i ≅ renComp (wk f) (wk g) i
wkcomp f g fz     = refl
wkcomp f g (fs i) = refl

rencomp : ∀{m n o}(f : Ren n o)(g : Ren m n)(t : Tm m) →
          ren (renComp f g) t ≅ (ren f • ren g) t
rencomp f g (var i)   = refl
rencomp f g (app t u) = resp2 app (rencomp f g t) (rencomp f g u)
rencomp f g (lam t)   =
  resp lam (trans (resp (λ f → ren f t) (ext (wkcomp f g)))
                  (rencomp (wk f) (wk g) t))
```

These properties indicate that `wk` and `ren` are morphism maps of two functors. `wk` is a morphism map for the object endomap `s` on the category of renamings and `ren` the morphism map for the object map `Tm` from the category of renamings to the category of sets. We stop short of defining these functors explicitly; we define only what we will need later.

Having dealt with renaming, we can now move on to substitution. We define a type synonym for substitution, as we did for renaming, but defer the definition of identity and substitution until later:

```
Sub : Nat → Nat → Set
Sub m n = Fin m → Tm n
```

The first operation we define is analogous to wk; we must be able to weaken a substitution to push it under a λ-abstraction. As before, the variable fz is passed straight through. In the other case we apply the substitution and weaken the resultant term. Notice that fs weakens a variable and ren fs weakens a term.

```
lift : ∀{m n} → Sub m n → Sub (s m) (s n)
lift f fz     = var fz
lift f (fs i) = ren fs (f i)
```

Having defined lift, we can define the action of substitutions on terms. In the variable case, we apply the substitution; in the application case, we pass it to the subterms; in the λ case we weaken it and pass it to the body.

```
sub : ∀{m n} → Sub m n → Tm m → Tm n
sub f (var i)   = f i
sub f (app t u) = app (sub f t) (sub f u)
sub f (lam t)   = lam (sub (lift f) t)
```

The identity substitution is just var. When applied, it removes the var constructor and then reapplies it again:

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

To compose substitutions, we must lift the first one f to a function from terms to terms sub f. Then we can compose:

```
subComp : ∀{m n o} → Sub n o → Sub m n → Sub m o
subComp f g = sub f • g
```

As we did for renamings we prove that lift takes the identity substitution to itself and that sub take the identity substitution to the identity function on terms:

```
liftid : ∀{n}(i : Fin (s n)) → lift subId i ≅ subId i
liftid fz     = refl
liftid (fs i) = refl

subid : ∀{n}(t : Tm n) → sub subId t ≅ id t
subid (var i)   = refl
subid (app t u) = resp2 app (subid t) (subid u)
subid (lam t)   = resp lam (trans (resp (λ f → sub f t) (ext liftid))
                                  (subid t))
```

To prove the required properties of lift and sub for composition, we need some extra lemmas. We have defined ren using wk, lift using ren and sub using lift. Hence, we need some lemmas regarding how they interact:

```
liftwk : ∀{m n o}(f : Sub n o)(g : Ren m n)(i : Fin (s m)) →
          (lift f • wk g) i ≅ lift (f • g) i
liftwk f g fz     = refl
liftwk f g (fs i) = refl

subren : ∀{m n o}(f : Sub n o)(g : Ren m n)(t : Tm m) →
         (sub f • ren g) t ≅ sub (f • g) t
subren f g (var i)   = refl
subren f g (app t u) = resp2 app (subren f g t) (subren f g u)
subren f g (lam t)   =
  resp lam (trans (subren (lift f) (wk g) t)
                  (resp (λ f → sub f t) (ext (liftwk f g))))

renwklift : ∀{m n o}(f : Ren n o)(g : Sub m n)(i : Fin (s m)) →
              (ren (wk f) • lift g) i ≅ lift (ren f • g) i
renwklift f g fz     = refl
renwklift f g (fs i) = trans (sym (rencomp (g i) (wk f) fs))
                             (rencomp (g i) fs f)

rensub : ∀{m n o}(f : Ren n o)(g : Sub m n)(t : Tm m) →
         (ren f • sub g) t ≅ sub (ren f • g) t
rensub f g (var i)   = refl
rensub f g (app t u) = resp2 app (rensub f g t) (rensub f g u)
rensub f g (lam t)   =
  resp lam (trans (rensub (wk f) (lift g) t)
                  (resp (λ f → sub f t) (ext (renwklift f g))))
```

Having proved these lemmas relating `wk`, `ren`, `lift` and `sub`, we can now prove the properties of composition we need. `lift` takes composition of substitutions to composition of substitutions and `sub` takes composition of substitutions to composition of functions:

```
liftcomp : ∀{m n o}(f : Sub n o)(g : Sub m n)(i : Fin (s m)) →
           lift (subComp f g) i ≅ subComp (lift f) (lift g) i
liftcomp f g fz     = refl
liftcomp f g (fs i) = trans (rensub fs f (g i))
                            (sym (subren (lift f) fs (g i)))

subcomp : ∀{m n o}(f : Sub n o)(g : Sub m n)(t : Tm m) →
          sub (subComp f g) t ≅ (sub f • sub g) t
subcomp f g (var i)   = refl
subcomp f g (app t u) = resp2 app (subcomp f g t) (subcomp f g u)
subcomp f g (lam t)   =
  resp lam (trans (resp (λ f → sub f t) (ext (liftcomp f g)))
                  (subcomp (lift f) (lift g) t))
```

Similarly to renaming, there are two functors lurking here. `lift` is the morphism map of an endofunctor on the category of substitutions and `sub` is the morphism map of a functor from the category of substitutions to the category of sets.

Having defined `sub` and proved `subid` and `subcomp`, we can fill in the rest of the definition of the relative monad. `bind` is `sub`, the first law follows from `subid`, the second law holds definitionally and the third law follows from `subcomp`.

```
TmRMonad : RMonad FinF
TmRMonad = record{T    = Tm;
                  η    = var;
                  bind = sub;
                  law1 = ext subid;
                  law2 = refl;
                  law3 = ext (subcomp _ _)}
```

4.2 Well-typed λ-terms

The well-typed λ-terms also form a relative monad. First we define some types: an inert base type; and a simple function space.

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

Contexts are just sequences of types. They will play the role that natural numbers played in the well-scoped terms: indicating how many variables are in scope, but also what their types are. In this sense, they can be thought of as like natural numbers where the successor is labelled with a type.

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

Variables are de Bruijn indices as before. The type of variables is very much like `Fin`, but we have contexts instead of natural numbers to indicate the scope and the type is given as an extra type index. For the zeroth variable `vz`, we can see that its type is guaranteed to be the same as the type at the end of the context. The successor `vs` is essentially weakening restricted to variables, it introduces a new variable at the end of the context and preserves the type.

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

The type `Tm` receives the same treatment. It guarantees the that terms are well-scoped as before, but now they are also well-typed. An ill-typed term is not a term at all.

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

To show that this datatype gives a relative monad, we must define substitution and prove the same properties of it as before (that it respects identity and composition). The programs we must write are very similar and so are the proofs. We give the programs in full, but omit the proofs of the substitution properties. Having proved these properties, we must also fix some categorical structures (What are the categories? What is J?) to complete the relative monad definition. We will describe these in full.

Given two contexts, renaming is defined to be a type of functions which takes a type as an implicit argument and returns a function from variables over one context to variables over the other. Notice that the types of the variables preserved. Identity and composition are given by identity and composition of functions.

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

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
```

Before defining the action of a renaming on a term, we explain how to weaken a renaming (to push it under a λ-abstraction) by introducing a new type (for the bound variable) at the end of the context. The type is taken implicitly. The action `ren` is defined by recursion on the term. In the variable case, the renaming is applied, in the application case, it is passed to the subterms and, in the λ-abstraction case, it is weakened before being passed to the body.

```
wk : ∀{Γ Δ σ} → Ren Γ Δ → Ren (Γ < σ) (Δ < σ)
wk f vz     = vz
wk f (vs i) = vs (f i)

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

We require the following properties of `wk` and `ren`:

```
wkid    : ∀{Γ σ τ}(x : Var (Γ < τ) σ) → wk renId x ≅ renId x
renid   : ∀{Γ σ}(t : Tm Γ σ) → ren renId t ≅ id t
wkcomp  : ∀ {B Γ Δ}(f : Ren Γ Δ)(g : Ren B Γ){σ τ}(x : Var (B < σ) τ) →
          wk (renComp f g) x ≅ renComp (wk f) (wk g) x
rencomp : ∀ {B Γ Δ}(f : Ren Γ Δ)(g : Ren B Γ){σ}(t : Tm B σ) →
          ren (renComp f g) t ≅ (ren f • ren g) t
```

We omit the proofs, as they have exactly the same structure as their counterparts for the well-scoped terms.

Given two contexts, substitution is defined to be a type of functions which takes a type as an implicit argument and returns a type-preserving function from variables over one context to terms over the other.

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

Before defining the action of a substitution on a term, we explain how to weaken a substitution. The zeroth variable is passed straight through, for other variables, we apply the substitution and weaken the result (`ren vs` weakens a term).

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

The action of substitution on a term is defined by recursion on the term. In the variable case, the substitution is applied, in the application case, it is passed to the subterms, in the λ-abstraction case, it is weakened and then passed to the body.

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

The identity is given by the `var` constructor. Composition is given by lifting the first substitution to an operation on terms by applying `sub` and then using function composition.

```
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
```

We require the following properties of substitution:

```
liftid    : ∀{Γ σ τ}(x : Var (Γ < σ) τ) → lift subId x ≅ subId x
subid     : ∀{Γ σ}(t : Tm Γ σ) → sub subId t ≅ id t
liftwk    : ∀{B Γ Δ}(f : Sub Γ Δ)(g : Ren B Γ){σ τ}(x : Var (B < σ) τ) →
            (lift f • wk g) x ≅ lift (f • g) x
subren    : ∀{B Γ Δ}(f : Sub Γ Δ)(g : Ren B Γ){σ}(t : Tm B σ) →
            (sub f • ren g) t ≅ sub (f • g) t
renwklift : ∀{B Γ Δ}(f : Ren Γ Δ)(g : Sub B Γ){σ τ}(x : Var (B < σ) τ) →
            (ren (wk f) • lift g) x ≅ lift (ren f • g) x
rensub    : ∀{B Γ Δ}(f : Ren Γ Δ)(g : Sub B Γ){σ}(t : Tm B σ) →
            (ren f •  sub g) t ≅ sub (ren f • g) t
liftcomp  : ∀{B Γ Δ}(f : Sub Γ Δ)(g : Sub B Γ){σ τ}(x : Var (B < σ) τ) →
            lift (subComp f g) x ≅ subComp (lift f) (lift g) x
subcomp   : ∀{B Γ Δ}(f : Sub Γ Δ)(g : Sub B Γ){σ}(t : Tm B σ) →
            sub (subComp f g) t ≅ (sub f • sub g) t
```

The proofs are omitted.

We now consider the categorical structure. We start with a category of renamings as before. This time, the renamings are typed and the objects of the category are contexts instead of natural numbers, the morphisms are renamings as before. Identity and composition morphisms are given by identity and composition renamings, these are in turn given by the identity function and composition of functions, hence the categorical laws hold definitionally (after applying extensionality for functions with implicit arguments `iext`).

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

Instead of just have a `J` be a functor from `ConCat` to `Sets` this time it is a functor from `ConCat` to a functor category. The functor category is from the discrete category of types to `Sets`. The discrete category of types has types as objects, and only identity morphisms. We represent this as a category where set of morphisms is only inhabited when the objects are equal. Identity is given by reflexivity of equality, composition by transitivity and proof irrelevance guarantees the required laws:

```
TyCat : Cat
TyCat = record{Obj  = Ty;
               Hom  = λ X Y → X ≅ Y;
               iden = refl;
               comp = λ p q → trans q p;
               idl  = ir;
               idr  = ir;
               ass  = ir}
```

Next we define the functor `VAR : Fun Concat (FunctorCat TyCat Sets)` that plays the role of `J`. We define its object and morphism maps separately. First we define the object map:

```
OVAR : Con → Fun TyCat Sets
OVAR Γ = record{OMap  = Var Γ;
                HMap  = subst (Var Γ);
                fid   = refl;
                fcomp = λ{_ _ _ p q} → ext (substtrans (Var Γ) q p)}
```

For any object in `ConCat`, a context Γ, it gives a functor whose object map is given by `Var Γ` and whose morphism map is given by substitutivity of equality. The type of `subst` says that the equality is substitutive. Its proof says it maps reflexive equations to the identity function:

```
subst : ∀{A}(P : A → Set){a a' : A} → a ≅ a' → P a → P a'
subst P refl p = p
```

Here `P = Var Γ`, `A = Ty`, so this amounts to saying we can substitute by equal types.

To prove the functor laws, we check that `subst` does the right thing with identity (reflexivity) and composition (transitivity) in `TyCat`. The identity property `fid` holds definitionally as `subst` computes when the equality proof is `refl`. For `fcomp`, we need a lemma about substituting by transitive equations which states that `subst` takes composition of equations to composition of functions:

```
substtrans : ∀{A}(P : A → Set){a a' a''}(p : a ≅ a')(q : a' ≅ a'') →
             ∀ x → subst P (trans p q) x ≅ (subst P q • subst P p) x
substtrans P refl refl x = refl
```

Next define the operation on maps for `VAR`:

```
HVAR : ∀{Γ Δ}(f : Ren Γ Δ) → NatT (OVAR Γ) (OVAR Δ)
HVAR f = record{cmp = f; nat = λ {_ _ p} → ext (substren p f)}
```

It takes a morphism in `ConCat`, which is a renaming `f : Ren Γ Δ`, and defines a natural transformation between the functors `OVAR Γ` and `OVAR Δ`. The component of the natural transformation `cmp` is just the renaming `f` except the component takes the type argument explicitly. To prove the naturality condition `nat`, we need a lemma about commuting renamings and the substitution of equality proofs:

```
substren : {Γ Δ : Con}{σ τ : Ty}(p : σ ≅ τ)(f : Ren Γ Δ)(i : Var Γ σ) →
           (subst (Var Δ) p • f) i ≅ (f • subst (Var Γ) p) i
substren refl f i = refl
```

Having defined the object and morphism maps `OVAR` and `HVAR`, we can put them together to form a the functor `VAR`:

```
VAR : Fun ConCat (FunctorCat TyCat Sets)
VAR = record{
  OMap = OVAR; HMap = HVAR; fid = NatTEq refl; fcomp = NatTEq refl}
```

We use the previously defined operation `NatTEq`, which allows us to assert that natural transformation are equal, if their components are equal, to dismiss the proof obligations for the functor laws. In both case the components are definitionally equal.

Next, we begin to fill in the definition of the relative monad for `Tm`:

```
TmRMonad : RMonad VAR
TmRMonad = record {T = ?; η = ?; bind = ?; law1 = ?; law2 = ?; law3 = ?}
```

In the case of the well-scoped, terms we plugged in `Tm`, `var`, and `sub` for `T`, `η`, and `bind` respectively. Here we have a functor category as our target category, so we must so we must lift `Tm` to be a functor, `var` to be a natural transformation, and `sub` to be an operation on natural transformations before we can use them. The laws will be lifted to equations between natural transformations, but using `NatTEq` we can reduce this to equations between their components.

We lift `Tm` to a functor `TM` which, for any context, Γ gives a functor whose object map is `Tm Γ` and whose morphism map is given by `subst`. This is just like the definition of `OVAR`, but with `Var` replaced with `Tm`.

```
TM : Con → Fun TyCat Sets
TM Γ = record{OMap  = Tm Γ;
              HMap  = subst (Tm Γ);
              fid   = refl;
              fcomp = λ {_ _ _ p q} → ext (substtrans (Tm Γ) q p)}
```

We lift `var` to a natural transformation `ηnat` whose components are given by `var`.

```
ηnat : {Γ : Con} → NatT (OVAR Γ) (TM Γ)
ηnat = record{cmp = var; nat = λ{_ _ p} → ext (substsub p var)}
```

The naturality condition requires a lemma like the one we required for commuting substitution of equations and renamings, except this time it is for substitutions instead:

```
substsub : {Γ Δ : Con}{σ τ : Ty}(p : σ ≅ τ)(f : Sub Γ Δ)(i : Var Γ σ) →
           (subst (Tm Δ) p • f) i ≅ (f •  subst (Var Γ) p) i
substsub refl f i = refl
```

We lift `sub` to an operation on natural transformation `bindnat` whose components apply `sub` to the components of the natural transformation it takes as an argument:

```
bindnat : ∀{Γ Δ} → NatT (OVAR Γ) (TM Δ) → NatT (TM Γ) (TM Δ)
bindnat α = record{cmp = sub (cmp α);
                   nat = λ{_ _ p} → ext (substcmp α p)}
```

We require another commutativity lemma here for the naturality condition of `bindnat`:

```
substcmp : {Γ Δ : Con}(α : NatT (OVAR Γ) (TM Δ)){σ τ : Ty}
           (p : σ ≅ τ)(t : Tm Γ σ) →
           (subst (Tm Δ) p  • sub (cmp α)) t
           ≅
           (sub (cmp α) •  subst (Tm Γ) p) t
substcmp α refl t = refl
```

We can now fill in the rest of the definition of `TmRMonad` giving `TM`, `ηnat`, `bind` for `T`, `η` and `bind` respectively. To prove the laws, we apply `NatTEq` and then they follow from `subid`, reflexivity and `subcomp` respectively.

```
TmRMonad : RMonad VAR
TmRMonad = record{T    = TM;
                  η    = ηnat;
                  bind = bindnat;
                  law1 = NatTEq (iext λ _ → ext subid);
                  law2 = NatTEq refl;
                  law3 = NatTEq (iext (λ _ → ext (subcomp _ _)))}
```

5 Adjunctions and relative adjunctions

An adjunction is structure between two categories `C` and `D`. It is carried by two functors `L` and `R`. There are many options regarding the further data and laws. We choose to accompany the two functors by a natural bijection between `Hom D (OMap L X) Y` and `Hom C X (OMap R Y)` natural in `X` and `Y`. The fields `left` and `right` give the functions in the two directions, `lawa` and `lawb` assert they are mutually inverse, and `natleft` and `natright` assert the naturality conditions. We could excise the definition of natural isomorphism, but as we only use it here, we stick with this flat definition to avoid unnecessary packing and unpacking.

```
record Adj (C D : Cat) : Set where
  field L : Fun C D
```

```
        R : Fun D C
        left : {X : Obj C}{Y : Obj D} →
               Hom D (OMap L X) Y → Hom C X (OMap R Y)
        right : {X : Obj C}{Y : Obj D} →
                Hom C X (OMap R Y) → Hom D (OMap L X) Y
        lawa : {X : Obj C}{Y : Obj D}(f : Hom D (OMap L X) Y) →
               right (left f) ≅ f
        lawb : {X : Obj C}{Y : Obj D}(f : Hom C X (OMap R Y)) →
               left (right f) ≅ f
        natleft : {X X' : Obj C}{Y Y' : Obj D}
                  (f : Hom C X' X)(g : Hom D Y Y')
                  (h : Hom D (OMap L X) Y) →
                  comp C (HMap R g) (comp C (left h) f)
                  ≅
                  left (comp D g (comp D h (HMap L f)))
        natright : {X X' : Obj C}{Y Y' : Obj D}
                   (f : Hom C X' X)(g : Hom D Y Y')
                   (h : Hom C X (OMap R Y)) →
                   right (comp C (HMap R g) (comp C h f))
                   ≅
                   comp D g (comp D (right h) (HMap L f))
```

Relative adjunctions are defined similarly. A relative adjunction goes between a functor `J : Fun C D` and a category `E`. It is a given by two functors `L : Fun C E` and `R : Fun E D` and further data. This time, the bijection is between `Hom E (OMap L X) Y` and `Hom D (OMap J X) (OMap R Y)`. As `X` is taken from `C` not `D`, we must apply `OMap J` to rectify this mismatch in the type of the bijection. The isomorphism and naturality conditions are adjusted accordingly.

```
record RAdj {C D : Cat}(J : Fun C D)(E : Cat) : Set where
  field L : Fun C E
        R : Fun E D
        left : {X : Obj C}{Y : Obj E} →
               Hom E (OMap L X) Y → Hom D (OMap J X) (OMap R Y)
        right : {X : Obj C}{Y : Obj E} →
                Hom D (OMap J X) (OMap R Y) → Hom E (OMap L X) Y
        lawa : {X : Obj C}{Y : Obj E}(f : Hom E (OMap L X) Y) →
               right (left f) ≅ f
        lawb : {X : Obj C}{Y : Obj E}(f : Hom D (OMap J X) (OMap R Y)) →
               left (right f) ≅ f
        natleft : {X X' : Obj C}{Y Y' : Obj E}
                  (f : Hom C X' X)(g : Hom E Y Y')
                  (h : Hom E (OMap L X) Y) →
                  comp D (HMap R g) (comp D (left h) (HMap J f))
                  ≅
                  left (comp E g (comp E h (HMap L f)))
        natright : {X X' : Obj C}{Y Y' : Obj E}
                   (f : Hom C X' X)(g : Hom E Y Y')
                   (h : Hom D (OMap J X) (OMap R Y)) →
                   right (comp D (HMap R g) (comp D h (HMap J f)))
                   ≅
                   comp E g (comp E (right h) (HMap L f))
```

An ordinary adjunction is a special case of a relative adjunctions where the functor `J` is the identity functor `IdF`.

Given an ordinary adjunction between two categories `D` and `E`, we can restrict it to a relative adjunction between a functor `J : Fun C D` and the category `E` by post-composing (using functor composition `_○_`) the left adjoint `L` with `J`. The bijection follows from the bijection of the adjunction where `X = OMap J X`. The naturality follows from the naturality of the adjunction where `f = HMap J f`:

```
restrictA : {C D E : Cat}(J : Fun C D) → Adj D E → RAdj J E
restrictA J A = record{L         = L A ○ J;
                       R         = R A;
                       left      = left A;
                       right     = right A;
                       lawa      = lawa A;
                       lawb      = lawb A;
                       natleft   = natleft A • HMap J;
                       natright  = natright A • HMap J}
```

The adjoint functors of any adjunction can be composed to form a monad. This construction can be generalised to relative monads. Given a relative adjunction on `J` between functors `F` and `G` the object map of the relative monad `T` arises by composing the object maps of the functors `OMap G • OMap F`. To define η, we must produce a map from `OMap J X` to `OMap G (OMap F X)` which we do by applying `left` at the identity in `D` at `OMap L X`. For `bind`, we need to define a function which takes morphisms of from `OMap J X` to `OMap G (OMap F Y)` to morphisms from `OMap G (OMap F X)` to `OMap G (OMap F Y)` for any `X` and `Y`. We first apply then `right` to give a morphism of type `OMap F X` to `OMap F Y` and then we apply `HMap G`. The laws follow from the laws of the monad and from functor laws. We omit the proof terms.

```
Adj2Mon : ∀{C D E}{J : Fun C D} → RAdj J E → RMonad J
Adj2Mon A = record{T    = OMap (R A) • OMap (L A);
                   η    = left A (iden E);
                   bind = HMap (R A) • right A;
                   law1 = ?;
                   law2 = ?;
                   law3 = ?}
```

Any monad can be split into an adjunction. There are two canonical ways to do this: one is due to Kleisli; and the other is due to Eilenberg and Moore. These constructions can be generalised to relative monads. We consider the relative Kleisli adjunction first.

5.1 Kleisli

The relative Kleisli category is defined for a relative monad `M` on `J : Fun C D`. The category `Kl M` is has objects of `C` as its objects. Its morphisms are given by morphisms in `D` from `OMap J X` to `T M Y`. Identity is given by the η of the relative monad `M` and composition of two functions `f : Hom D (OMap J Y) (T M Z)` and `g : Hom D (OMap J X) (T M Y)` is given by composition in `D` of `bind M f` with `g`. The left identity law follows from the first law of the relative monad and the left identity law of the category `D`, the right identity law follows immediately from the second relative monad law, and the associativity law follows from the third relative monad law and associativity in `D`.

```
Kl : ∀{C D}{J : Fun C D}  → RMonad J → Cat
Kl {C}{D}{J} M = record{
  Obj  = Obj C;
  Hom  = λ X Y → Hom D (OMap J X) (T M Y);
  iden = η M;
  comp = λ f g → comp D (bind M f) g;
  idl  = λ{_ _ f} → trans (resp (λ g → comp D g f) (law1 M)) (idl D);
  idr  = law2 M;
  ass  =  λ{W}{X}{Y}{Z}{f}{g}{h} →
    trans (resp (λ f → comp D f h) (law3 M)) (ass D)}
```

The Kleisli category of an ordinary monad can be recovered by setting `C = D` and `J = IdF`.

We define the left adjoint of the relative Kleisli adjunction as follows:

```
RKlL : ∀{C D}{J : Fun C D}(M : RMonad J) → Fun C (Kl M)
RKlL {C}{D}{J} M = record{
  OMap = id; HMap = λ f → comp D (η M) (HMap J f); fid = ?; fcomp = ?}
```

On objects, it does nothing (the objects are the underlying objects of `C`). On morphism is lifts a morphism `f : Hom C X Y` to a morphism `HMap J f : Hom D (OMap J X) (OMap J Y)` and composes it with `η M Y` to give a morphism in `Hom D (OMap J X) (T M Y)`. The `fid` law follows from the `fid` law for `J` and right identity of `D`. The `fcomp` law follows from `fcomp` for `J`, associativity in `D`, and the second relative monad law for `M`.

The right adjoint is even more straightforward:

```
RKlR : ∀{C D}{J : Fun C D}(M : RMonad J) → Fun (Kl M) D
RKlR M = record{OMap = T M; HMap = bind M; fid = law1 M; fcomp = law3 M}
```

We can show that this forms an adjunction by defining:

```
KlAdj : ∀{C D}{J : Fun C D}(M : RMonad J) → RAdj J (Kl M)
KlAdj M = record{L = RKlL M;
                 R = RKlR M;
                 left     = id;
                 right    = id;
                 lawa     = λ _ → refl;
                 lawb     = λ _ → refl;
                 natleft  = ?;
```

```
                natright = ?}
```

The bijection holds definitionally and both naturality conditions follow from the second relative monad law for `M` and associativity in `D`. We omit the proof terms.

Again, we can recover the the ordinary Kleisli adjunction by setting `C = D` and `J = IdF C`. The Kleisli categories for the examples of the well-typed and well-scoped terms are their respective categories of substitutions. We omit the the definitions but note that we have defined and proved all the pieces we need already to do so.

5.2 Eilenberg-Moore

The Eilenberg-Moore category of a monad is the category of algebras for the monad and algebra morphisms between them. The ordinary notion of an algebra for a monad $T : \mathbf{C} \to \mathbf{C}$ is given by a pair of an object A in $\mathbf{C}$ and a morphism a from $T\,A$ to A, subject to some laws. Instead of attempting to generalise this notion directly we take an alternative, but equivalent, version and generalise that. We take an algebra for a monad T to be pair of an object A in $\mathbf{C}$ as before and an operation a on morphisms that, for any object X in $\mathbf{C}$ and for any morphism from X to A, gives a morphism from $T\,X$ to A. Applying a to A and the identity morphisms on A recovers the original version.

An algebra for a relative monad `M` over a functor `J : Fun C D` is defined as follows:

```
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)
```

It has a carrier `acar` in `D` and an algebra structure `astr` that, for any `Z : Obj C` (taken implicitly), takes a morphism from `OMap J Z` to `acar` to a morphism from `T M Z` to `acar`, subject to two laws which state that the algebra structure `astr` interacts appropriately with the `η` and `bind` of the monad. These laws play the same role as the usual laws for an algebra of a monad and in the case of an ordinary monad (an relative monad on the identity functor) they are equivalent.

The definition of an algebra morphism for a relative monad contains a morphism on the underlying objects of the algebra, as usual, and a homomorphism condition stating we can apply the morphism first and then the algebra structure, or the other way around, to yield the same result. The formulation of this condition is slightly different from the usual formulation of an algebra morphism for a monad as we have a different notion of algebra structure.

```
record RAlgMorph {C D : Cat}{J : Fun C D}{M : RMonad J}
       (A B : RAlg M) : Set where
  field amor : Hom D (acar A) (acar B)
        ahom : ∀{Z}{f : Hom D (OMap J Z) (acar A)} →
               comp D amor (astr A f) ≅ astr B (comp D amor f)
```

We define a useful lemma stating that any two algebra morphisms are equal, if the underlying morphisms are equal. This is analogous to the NatTEq lemma for natural transformations and such properties are very useful, whenever we define a category where the morphisms are records. We give only its type here. The proof follows from extensionality and proof irrelevance after using the fixtypes lemma.

```
RAlgMorphEq : ∀{C D}{J : Fun C D}{M : RMonad J}{X Y : RAlg M}
              {f g : RAlgMorph X Y} → amor f ≅ amor g → f ≅ g
```

Next we define identity and composition of morphisms:

```
IdMorph : ∀{C D}{J : Fun C D}{M : RMonad J}{A : RAlg M} → RAlgMorph A A
IdMorph {C}{D} = record{amor = iden D; ahom = ?}

CompMorph : ∀{C D}{J : Fun C D}{M : RMonad J}{X Y Z : RAlg M} →
            RAlgMorph Y Z → RAlgMorph X Y → RAlgMorph X Z
CompMorph {C}{D} f g = record{amor = comp D (amor f) (amor g); ahom = ?}
```

The underlying morphisms are just formed by identity and composition (of the underlying morphisms of the algebra morphisms being composed) in D. For the identity morphism, the naturality condition follows from left unit law in D and, for the composition morphism, it follows from the naturality properties of the morphisms being composed and associativity in D. To be able to define an relative EM category we must also prove the left and right unit laws, and associativity laws for algebra morphisms. After applying RAlgMorphEq, they each follow from the corresponding property of the category D:

```
idlMorph : ∀{C D}{J : Fun C D}{M : RMonad J}
           {X Y : RAlg M}{f : RAlgMorph X Y} → CompMorph IdMorph f ≅ f
idlMorph {C}{D} = RAlgMorphEq (idl D)

idrMorph : ∀{C D}{J : Fun C D}{M : RMonad J}
           {X Y : RAlg M}{f : RAlgMorph X Y} → CompMorph f IdMorph ≅ f
idrMorph {C}{D} = RAlgMorphEq (idr D)

assMorph : ∀{C D}{J : Fun C D}{M : RMonad J}{W X Y Z : RAlg M}
           {f : RAlgMorph Y Z}{g : RAlgMorph X Y}{h : RAlgMorph W X} →
           CompMorph (CompMorph f g) h ≅ CompMorph f (CompMorph g h)
assMorph {C}{D} = RAlgMorphEq (ass D)
```

Having defined, algebras, algebra morphism, identity, composition, and proved the laws, we can put them together to define the EM category:

```
EM : ∀{C D}{J : Fun C D} → RMonad J → Cat
EM M = record{Obj  = RAlg M;
              Hom  = RAlgMorph;
              iden = IdMorph;
              comp = CompMorph;
              idl  = idlMorph;
              idr  = idrMorph;
              ass  = assMorph}
```

Having defined the category EM, we can now define the left and right adjoint functors that make up the relative EM adjunction:

```
REML : ∀{C D}{J : Fun C D}(M : RMonad J) → Fun C (EM M)
REML {C}{D}{J} M = record {
  OMap  = λ X → record{acar  = T M X;
                      astr  = bind M;
                      alaw1 = sym (law2 M);
                      alaw2 = law3 M};
  HMap  = λ f → record {amor = bind M (comp D (η M) (HMap J f));
                       ahom = sym (law3 M)};
  fid   = ?;
  fcomp = ?}
```

For objects of C, the left adjoint creates algebras by applying the monad map T M to them and defining the algebra structure to be bind M. The laws follow from the second and third monad laws. It is interesting to note that in the case of ordinary monads (where C = D and J = IdF), this version of algebras uses bind exactly where the standard version would use the multiplication of the monad μ. For morphisms, we lift morphisms from X to Y in C to morphisms in D from T M X to T M Y by applying HMap J, composing with η M Y and then applying bind M. The homomorphism condition ahom of the algebra morphism follows from the third monad law. It remains to prove the functor laws for the left adjoint. These follow from the corresponding functor laws for J, the monad laws (first and third respectively) and the laws of the category D.

The right adjoint is much simpler as it is throws away the algebra structure rather than constructs it:

```
REMR : ∀{C D}{J : Fun C D}(M : RMonad J) → Fun (EM M) D
REMR M = record{OMap  = acar; HMap  = amor; fid   = refl; fcomp = refl}
```

On objects, it projects the underlying object and on morphisms it projects the underlying morphism. The functor laws hold definitionally.

Next we combine these functors to form a relative adjunction. This time the bijection does not hold definitionally and we have to do some work:

```
REMAdj : ∀{C D}{J : Fun C D}(M : RMonad J) → RAdj J (EM M)
REMAdj {C}{D} M = record{L = REML M;
                         R = REMR M;
                         left     = λ f → comp D (amor f) (η M);
                         right    = λ{X}{B} f →
                           record{amor = astr B f; ahom = sym (alaw2 B)};
                         lawa     = ?;
                         lawb     = ?;
                         natleft  = ?;
                         natright = ?}
```

To go left, we must construct a morphism of type Hom D (OMap J X) (AObj B) from an algebra morphism `f` between an algebra whose object is `T M X` and an algebra `B`. `amor f` gives us a morphism of type `Hom D (T M X) (acar B)` and by composing with `η M X` we get a morphism of the right type. To go `right`, we must reverse the process: given a morphism `f : Hom D (OMap J X) (acar B)`, we must construct an algebra morphism. We apply the algebra structure of `B` at `X` to `f` to give the underlying morphism (whose type is `Hom D (T M X) (acar B)`) and the homomorphism condition follows from the second algebra law for `B`. The laws `lawa`, `lawb`, `natleft`, and `natright` follow from the first monad law for `M`, the first algebra law for `B`, the second monad law for `M`, and the first algebra law for `B` respectively.

5.3 Set-model of the well-typed λ-terms

The set-model of the well-typed λ-terms form an relative EM algebra for the relative monad `TM` in quite a natural way. The idea is to define interpretation of object level types as Agda types, and an evaluator (the interpretation of terms) which maps terms to values (the interpretation of types) given a suitable environment (environments are interpretations of contexts). One could also say that a term `t : Tm Γ σ` is interpreted as a function from the interpretation of the context `Env Γ` to the interpretation of the type `Val σ`.

The values form the carrier of the algebra and the evaluator forms the algebra structure.

Let us look at the construction in detail. First we define the interpretation of object types `Ty` as Agda types `Sets`.

```
Val : Ty → Set
Val ι       = One
Val (σ ⇒ τ) = Val σ → Val τ
```

The inert base type is interpreted as the unit type but another type such as `Nat` would do the same job as it only contains neutral terms for the the simply-typed λ-calculus. The object level function space is interpreted as the meta-level (Agda) function space. Together with the datatype `Ty`, the set-valued function `Val` is a very simple, and very useful, example of a universe construction.

Next we interpret contexts as environments, they are like substitutions, but emit values over a type instead of terms over a context and type. We also define an operation `_<<_` to extend an environment by a new value which we will need for the evaluator.

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

_<<_ : ∀{Γ σ} → Env Γ → Val σ → Env (Γ < σ)
(γ << v) vz     = v
(γ << v) (vs x) = γ x
```

Given `Val`, `Env` and `_<<_` we have all that we need to define a simple evaluator:

```
eval : ∀{Γ σ} → Env Γ → Tm Γ σ → Val σ
eval γ (var x)   = γ x
eval γ (app t u) = eval γ t (eval γ u)
eval γ (lam t)   = λ v → eval (γ << v) t
```

It takes a environment over Γ, a term in context Γ of type σ and emits a value of type σ. In the variable case, it looks up the variable in the environment. In the case of application, we evaluate the function in the environment to give a real Agda function which we can then run it on the evaluated argument. In the case of λ-abstraction, we must produce a function. We define this as a meta level (Agda) λ-abstraction which takes a value of appropriate type. When this argument value arrives the evaluator will evaluate the body `t` in the original environment extended with the new value `v`. We have essentially created a closure. That is the end of our program. We now need to prove some properties and perform some categorical constructions to show that the set model forms an algebra.

We need five lemmas to proceed. The first lemma shows that we can commute evaluation with substitution by a type equation:

```
substeval : ∀{σ τ}(p : σ ≅ τ){Γ : Con}{γ : Env Γ}(t : Tm Γ σ) →
      (subst Val p  • eval γ) t ≅ (eval γ • subst (Tm Γ) p) t
substeval refl t = refl
```

The second lemma show that `wk` commutes with `_<<_`:

```
wk<< : ∀{Γ Δ}(α  : Ren Γ Δ)(β : Env Δ){σ}(v : Val σ) →
       ∀{ρ}(y : Var (Γ < σ) ρ) → ((β • α) << v) y ≅ (β << v) (wk α y)
wk<< α β v vz     = refl
wk<< α β v (vs x) = refl
```

The third shows that we can commute renaming and evaluation:

```
reneval : ∀{Γ Δ σ}(α : Ren Γ Δ)(β : Env Δ)(t : Tm Γ σ) →
          eval (eval β • var • α) t ≅ (eval β • ren α) t
reneval α β (var x) = refl
reneval α β (app t u) =
  resp2 (λ f x → f x) (reneval α β t) (reneval α β u)
reneval α β (lam t) = ext λ v →
  trans (resp (λ γ → eval γ t) (iext λ _ → ext (wk<< α β v)))
        (reneval (wk α) (β << v) t)
```

The fourth shows that we can lift and context extension in evaluation:

```
lifteval : ∀{Γ Δ σ τ}(α : Sub Γ Δ)(β : Env Δ)
           (v : Val σ)(y : Var (Γ < σ) τ) →
           ((eval β • α) << v) y ≅ (eval (β << v) • lift α) y
lifteval α β v vz     = refl
lifteval α β v (vs x) = reneval vs (β << v) (α x)
```

The fifth lemma shows that we can commute evaluation and substitution:

```
subeval : ∀{Γ Δ σ}(α : Sub Γ Δ)(β : Env Δ)(t : Tm Γ σ) →
          eval (eval β • α) t ≅ (eval β • sub α) t
subeval α β (var x)   = refl
subeval α β (app t u) =
  resp2 (λ f x → f x) (subeval α β t) (subeval α β u)
subeval α β (lam t)   = ext λ v →
  trans (resp (λ γ → eval γ t) (iext λ _ → ext (lifteval α β v)))
```

```
        (subeval (lift α) (β << v) t)
```

Given these lemmas, we can show that the set model forms an relative EM algebra:

```
modelRAlg : RAlg TmRMonad
modelRAlg = record{
  acar  = record{OMap  = Val;
                 HMap  = subst Val;
                 fid   = ext λ _ → refl;
                 fcomp = λ {_ _ _ p q} → ext (substtrans Val q p)};
  astr  = λ α → record{cmp = eval (cmp α);
                       nat = λ{_ _ p} → ext (substeval p)};
  alaw1 = NatTEq refl;
  alaw2 = NatTEq (iext (λ _ → ext (subeval _ _)))}
```

Let us go through the definition of `modelRAlg` step by step. To define an algebra, we must give a carrier `acar` which is an object in `FuntorCat TyCat Sets` which is a functor from `TyCat` to `Sets`. Its object map is given by `Val` and its morphism map by `subst Val`. The functor laws follows from reflexivity (as `subst` computes when the equality proof is reflexive) and the property of `subst` for transitivity (which is composition of morphisms in `TyCat`) we proved earlier. To define the algebra structure, we must give a morphism in `FunctorCat TyCat Sets` which for any context `Γ` is an operation on natural transformations. This operation takes a natural transformation whose components given by `∀ {σ} → Var Γ σ → Val σ` to a natural transformation whose components have type `∀ {σ} → Tm Γ σ → Val σ`. This is exactly the type of the evaluator `eval`, so we plug the components of the given natural transformation into the evaluator. The naturality condition is taken care of by the lemma `substeval` we proved above. All that remains is to prove the two algebra laws. After applying `NatTEq` in both cases, the first law holds definitionally and the second follows from the last lemma `subeval` we proved above.

5.4 Extensional λ-models of the well-scoped λ-calculus

The is no set-model of the well-scoped (untyped) λ-terms. So, instead we do something more general: we give a specification for a model and show that any such model yields a relative EM-algebra.

We define an extensional λ-model in the same style as we have defined other mathematical structures. It has some data: a set `S`; an operation `eval` that, given an appropriate environment, evaluates a well-scoped term to give a value in `S`; and an operation `ap` which performs application on values in `S`. The behaviour of these data is governed by four laws: three laws governing how the syntax is evaluated (notice that the law for `lam` applies only in the presence of an argument); and an extensionality principle.

```
record LambdaModel : Set where
  field S      : Set
        eval   : ∀{n} → (Fin n → S) → Tm n → S
        ap     : S → S → S
        lawvar : ∀{n}{i : Fin n}{γ : Fin n → S} → eval γ (var i) ≅ γ i
        lawapp : ∀{n}{t u : Tm n}{γ : Fin n → S} →
```

```
                  eval γ (app t u) ≅ ap (eval γ t) (eval γ u)
        lawlam : ∀{n}{t : Tm (s n)}{γ : Fin n → S}{s : S} →
                  ap (eval γ (lam t)) s ≅ eval (γ << s) t
        lawext : ∀{f g : S} → ((a : S) → ap f a ≅ ap g a) → f ≅ g
```

In the definition of `lamlaw`, we need to be able to extend the environment, so we implement an operation to do this:

```
_<<_ : ∀{n X} → (Fin n → X) → X → Fin (s n) → X
(f << x) fz     = x
(f << x) (fs i) = f i
```

To define an EM-algebra, we require versions of exactly the same lemmas as we required for set-model in the typed-case. We omit their proofs and given only the types. There is one key difference in the proofs which is that we must explicitly use the equations `lawvar`, `lawapp`, `lawlam`. In the case of the set-model, where `eval` and `ap` are functions, these principles held definitionally and were invisible in the proofs.

```
wk<<    : ∀(l : LambdaModel){m n}(α  : Fin m → Fin n)(β : Fin n → S l)
          (v : S l) → (y : Fin (s m)) →
          ((β • α) << v) y ≅ (β << v) (wk α y)
reneval : ∀(l : LambdaModel){m n}(α : Fin m → Fin n)(β : Fin n → S l)
          (t : Tm m) →
          eval l (eval l β • (var • α)) t ≅ (eval l β • ren α) t
lift<<  : ∀(l : LambdaModel){m n}(γ  : Fin m → Tm n)(α : Fin n → S l)
          (a  : S l)(i : Fin (s m)) →
          ((eval l α • γ ) << a) i ≅ (eval l (α << a) • lift γ) i
subeval : ∀(l : LambdaModel){m n}(t : Tm m)
          (γ : Fin m → Tm n)(α : Fin n → S l) →
          eval l (eval l α • γ) t ≅ (eval l α • sub γ) t
```

Having proved `subeval` (using the three previous lemmas), we can define a relative EM algebra on the monad `TmRMonad`:

```
TmRAlg : LambdaModel → RAlg TmRMonad
TmRAlg l = record{acar  = S l;
                  astr  = eval l;
                  alaw1 = ext λ _ → sym (lawvar l);
                  alaw2 =  ext λ t → subeval l t _ _}
```

The object is given by the set `S` and the map by `eval`. The first law follows from `lawvar` (recall that this followed definitionally in the set-model) and the second from the lemma `subeval`.

6 Conclusion

We have presented a self-contained and detailed account of our formalisation of relative monads and relative adjunctions, with some examples.

Due to reasons of space, we have not described everything we have formalised. Every construction we have performed for relative monads has also been carried out for ordinary monads as well. Here in the paper we have included only the definitions of ordinary monads and adjunctions. We have done this for both styles of monads, i.e., for Manes style with an object mapping `T` and operations η `bind` and also as a functor `T` with natural transformations η and μ. We have also formalised the Yoneda lemma in Agda (with the intention of using it for the example of arrows as a relative monad on the Yoneda embedding), and a further example of finite-dimensional vector spaces.

To complete the formalisation of section 2 of (Altenkirch, Chapman, and Uustalu 2010), we would very much like to able to show that the relative Kleisli and Eilenberg-Moore constructions of a relative monad are respectively initial and final in the category of its splittings into relative adjunctions. This is currently not possible in Agda for performance reasons (not even for ordinary monads). There is a space leak due to a loss of sharing when using records. This is particularly acute for our formalisation which uses records extensively. Defining the category of splittings is currently one step beyond what we can achieve.

Formalisation of the further theory of relative monads (Sections 3, 4 of (Altenkirch, Chapman, and Uustalu 2010)) requires Kan extensions. We expect that our calculus of coends as presented in Section 3.1 of (Altenkirch, Chapman, and Uustalu 2010) lends itself to formalisation, but this is future work. We also plan to advance our running examples, as we add more of the relative monads machinery, and also to give a treatment of the further example of arrows as relative monads (Section 5 of (Altenkirch, Chapman, and Uustalu 2010)).

We have already discussed the question of extensionality. For the formalisation done so far it is sufficent to assume functional extensionality but going further would require quotient types in particular to develop the coend calculus. This can be achieved by adding further postulates which state that every setoid gives rise to a set. Formally this corresponds to using a type theory with exact coequalizers, this is often refered to as a *predicative topos* (van den Berg 2010). Clearly, adding postulates to Agda is unsatisfying for several reasons: first of all arbitrary postulates could be easily unsound, and second the resulting type theory is computationally not well behanved forcing us to prove equations which should just be definitional equalities. We hope that the situation will improve once implementations of Observational Type Theory (Altenkirch, McBride and Swierstra 2007) become available and ready for serious formalisations.

References

The Agda team. (2010). `http://wiki.portal.chalmers.se/agda/`

Altenkirch, T., Chapman, J. and Uustalu, T. (2010) Monads need not be endofunctors. In Ong, L. (editor) Proc. of 13th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2010. Lect. Notes in Comput. Sci. Volume 6014. 297–311. Springer.

Altenkirch, T., McBride, C. and Swiestra, W. (2007) Observational equality, now! Proc. of Workshop on Programming Languages Meet Program Verification, PLPV 2007. 57–68. ACM, New York.

Altucher, J. A. and Panangaden, P. (1990) A mechanically assisted constructive proof in category theory. In Stickel, M. E. (editor) Proc. of 10th Int. Conf. on Automated Deduction, CADE-10. Lect. Notes in Artif. Intell. Volume 449. 500–513. Springer.

Carvalho, A. (1998) Category Theory in Coq. Diploma Thesis. Institute Superior Técnico, Universidade Técnica de Lisboa.

Chapman, J. (2010) A formalisation of relative monads in Agda. `http://cs.ioc.ee/~james/relmon.html`

Coquand, T. and Spiwack, A. (2007) Towards constructive homological algebra in type theory. Kauers, M. et al. (editors) Proc. of 14th Symp. on he Integration of Symbolic Computation and Mechanized Reasoning, 6th Int. Conf. on Mathematical Knowledge Management, CALCULEMUS/MKM 2007. Lect. Notes in Comput. Sci. Volume 4573. 40–54. Springer.

Dawson, J. E. (2007) Compound monads and the Kleisli category. Unpublished note. Available online at `http://users.cecs.anu.edu.au/~jeremy/pubs/cmkc/`.

Dyckhoff, R. (1985) Category theory as an extension of Martin-Löf type theory. Techn. report, Dept. of Computer Science, Univ. lof St. Andrews.

Dybjer, P. and Gaspes, V. (1994) Implementing a category of sets in ALF. Technical Report. Dept. of Computer Science, Chalmers Univ. of Technology.

Dyckhoff, R. (1985) Category theory as an extension of Martin-Löf type theory. Techn. report, Dept. of Computer Science, Univ. lof St. Andrews.

The Haskell Team. (2010) Haskell Website. `http://www.haskell.org`.

Hofmann, M. (1995) Extensional concepts in intensional type theory. PhD Thesis. University of Edinburgh.

Huet, G. and Saïbi, A. (2000) Constructive category theory. In Plotkin, G., Stirling, C. and Tofte, M. Proof, Language, and Interaction: Essays in Honour of Robin Milner. 239–275. MIT Press.

O'Keefe, G. (2004) Towards a readable formalisation of category theory. Electron. Notes in Theor. Comput. Sci. 91, 212–228.

Sozeau, M. (2010) Cat: category theory with classes. A Coq formalization. Available online at `http://mattam.org/repos/coq/cat/`.

van den Berg, B. (2006) Predicative topos theory and models for constructive set theory. PhD thesis. Universiteit Utrecht.

Wilander, O. (2005) An E-bicategory of E-categories exemplifying a type-theoretic approach to bicategories. Technical Report. University of Uppsala.