Friedrich-Alexander-Universität
Erlangen-Nürnberg

Chair for Computer Science 8
Theoretical Computer Science

Implementing Categorical Notions of Partiality and Delay in Agda

Bachelor Thesis in Computer Science (Consolidated Version)

Leon Vatthauer

Advisor:

Sergey Goncharov

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Erlangen, July 17, 2024

Abstract

Moggi famously showed how to use category theory (specifically monads) to model the semantics of effectful computations. In this thesis we will examine how to model possibly non-terminating computations, which requires a monad supporting some form of partiality. For that we will consider categorical properties that a monad that models partiality should satisfy and then compare concrete monads in view of these properties.

Capretta's delay monad is a typical example for a partiality monad, but it comes with a too intensional notion of built-in equality. Since fixing this seems to be impossible without additional axioms, we will examine a novel approach of defining a partiality monad that works in a general setting by making use of previous research on iteration theories and drawing on the inherent connection between partiality and iteration.

Finally, we will show that in the category of setoids this partiality monad instantiates to a quotient of the delay monad, yielding a concrete description of the partiality monad in this category.

Contents

1 Introduction

Haskell is considered a purely functional programming language, though the notion of purity referenced is an informal one, not to be confused with the standard notion of pure function, which describes functions that do not have any side effects. Indeed, as a programming language that offers general recursion, Haskell does at least have to include partiality as a side effect. To illustrate this, consider the following standard list reversal function

```
reverse :: [a] -> [a]
reverse l = revAcc l []
  where
    revAcc []     a = a
    revAcc (x:xs) a = revAcc xs (x:a)
```

and regard the following definition of an infinite list

```
ones :: [Int]
ones = 1 : ones
```

Of course evaluation of the term `reverse ones` will never terminate, hence it is clear that `reverse` is a partial function. Thus, in order to reason about Haskell programs, or generally programs of any programming language offering general recursion, one needs to be able to model partiality as a side effect.

Generally for modelling programming languages there are three prevailing methods. First is the operational approach studied by Plotkin [3], where partial functions are used that map programs to their resulting values, secondly there is the denotational approach by Scott [6], where programming languages are interpreted mathematically, by functions that capture the "meaning" of programs. For this thesis we will consider the third, categorical approach that has been introduced by Moggi [5]. In the categorical approach programs are interpreted in categories, where objects represent types and monads are used to model side effects. The goal for this thesis is thus to study monads which are suitable for modeling partiality.

We use the dependently typed programming language Agda [23] as a safe and type-checked environment for reasoning in category theory, therefore in Chapter 3 we start out by quickly showcasing the Agda programming language as well as the category theory library that we will be working with. In Chapter 4 we will then consider various properties that partiality monads should satisfy and inspect Capretta's delay monad [12], which has been introduced in type theory as a coinductive data type and then studied as a monad in the category of setoids. We will examine the delay monad in a general categorical setting, where we prove strength and commutativity of this monad. However, it is not a minimal partiality monad, i.e. one that captures no other side effect besides some form of non-termination, since the monad comes with a too intensional notion of equality. In order to achieve minimality one can consider the quotient of the delay monad where a less intensional notion of equality is used. However, it is believed to be impossible to show that the monadic structure is preserved under such a quotient. In [16] the axiom of countable choice has been identified as a sufficient assumption under which the monad structure is preserved.

In order to define a partiality monad using no such assumptions, we will draw on the inherent connection between iteration and recursion in Chapter 5 to define a suitable partiality monad, by relating to previous research on iteration theories. This monad has first been introduced and studied in [19] under weaker assumptions than we use, concretely by weakening the notion of Elgot algebra to the notion of uniform iteration algebra, which uses fewer axioms. During mechanization of the results concerning this monad it turned out that under the weaker assumptions, desirable properties like commutativity seem not to be provable, resulting in our adaptation of this monad. Lastly, in Chapter 6 we will study this partiality monad in the category of setoids, where notably the axiom of countable choice is provable. In this category, the partiality monad turns out to be equivalent to a certain quotient of the delay monad.

2 Preliminaries

We assume familiarity with basic categorical notions, in particular: categories, functors, functor algebras and natural transformations, as well as special objects like (co)products, terminal and initial objects and special classes of morphisms like isomorphisms (isos), epimorphisms (epis) and monomorphisms (monos). In this chapter we will introduce notation that will be used throughout the thesis and also introduce some notions that are crucial to this thesis in more detail. We write $|\mathscr{C}|$ for the objects of a category $\mathscr{C}$, id_X for the identity morphism on X, $(-)\circ(-)$ for the composition of morphisms and $\mathscr{C}(X,Y)$ for the set of morphisms between X and Y. We will also sometimes omit indices of the identity and of natural transformations in favor of readability.

2.1 Distributive Categories

Let us first introduce notation for binary (co)products by giving their usual diagrams:

$$\begin{tikzcd}
A & A\times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \arrow[ul, "f"] \arrow[u, dashed, "\exists!\langle f,g\rangle"] \arrow[ur, "g"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "f"'] & A+B \arrow[d, "\exists!{[f,g]}"] & B \arrow[l, "i_2"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd}$$

We will furthermore overload this notation and write $f\times g := \langle f\circ\pi_1, g\circ\pi_2\rangle$ and $f+g := [i_1\circ f, i_2\circ g]$ on morphisms. To avoid parentheses we will use the convention that products bind stronger than coproducts.

We write 1 for the terminal object together with the unique morphism $! : A \to 1$ and 0 for the initial object with the unique morphism $¡ : A \to 0$.

Categories with finite products (i.e. binary products and a terminal object) are also called Cartesian and categories with finite coproducts (i.e. binary coproducts and an initial object) are called coCartesian.

Definition 2.1 (Distributive Category (🐓)). A Cartesian and coCartesian category $\mathscr{C}$ is called distributive if the canonical (left) distributivity morphism $dstl^{-1}$ is an isomorphism:

$$\begin{tikzcd}
X\times Y+X\times Z \arrow[r, bend left, "dstl^{-1}:=[id\times i_1, id\times i_2]"] & X\times(Y+Z) \arrow[l, bend left, "dstl"]
\end{tikzcd}$$

Remark 2.2. Definition 2.1 can equivalently be expressed by requiring that the canonical right distributivity morphism is an iso, giving these inverse morphisms:

$$\begin{tikzcd}
Y\times X+Z\times X \arrow[r, bend left, "dstr^{-1}:=[i_1\times id, i_2\times id]"] & (Y+Z)\times X \arrow[l, bend left, "dstr"]
\end{tikzcd}$$

These two can be derived from each other by taking either

$$dstr := (swap+swap)\circ dstl\circ swap$$

or

$$dstl := (swap+swap)\circ dstr\circ swap$$

where $swap := \langle\pi_2,\pi_1\rangle : A\times B \to B\times A$.

Proposition 2.3 (🐓). *The distribution morphisms can be viewed as natural transformations i.e. they satisfy the following diagrams:*

$$\begin{tikzcd}
X\times(Y+Z) \arrow[r, "f\times(g+h)"] \arrow[d, "dstl"] & A\times(B+C) \arrow[d, "dstl"] \\
X\times Y+X\times Z \arrow[r, "f\times g+f\times h"] & A\times B+A\times C
\end{tikzcd}$$

$$\begin{tikzcd}
(Y+Z)\times X \arrow[r, "(g+h)\times f"] \arrow[d, "dstr"'] & (B+C)\times A \arrow[d, "dstr"] \\
Y\times X+Z\times X \arrow[r, "g\times f+h\times f"] & B\times A+C\times A
\end{tikzcd}$$

Proposition 2.4 (🐓). *The distribution morphisms satisfy the following properties:*

1. $dstl\circ(id\times i_1) = i_1$
2. $dstl\circ(id\times i_2) = i_2$
3. $[\pi_1,\pi_1]\circ dstl = \pi_1$
4. $(\pi_2+\pi_2)\circ dstl = \pi_2$
5. $dstl\circ swap = (swap+swap)\circ dstr$
6. $dstr\circ(i_1\times id) = i_1$
7. $dstr\circ(i_2\times id) = i_2$
8. $(\pi_1+\pi_1)\circ dstr = \pi_1$
9. $[\pi_2,\pi_2]\circ dstr = \pi_2$
10. $dstr\circ swap = (swap+swap)\circ dstl$

Definition 2.5 (Exponential Object (🐓)). Let $\mathscr{C}$ be a Cartesian category and $X,Y \in |\mathscr{C}|$. An object X^Y is called an exponential object (of X and Y) if there exists an evaluation morphism $eval : X^Y\times Y \to X$ and for any $f : X\times Y \to Z$ there exists a morphism $curry\ f : X \to Z^Y$ that is unique with respect to the following diagram:

$$\begin{tikzcd}
Z\times Y \arrow[r, "curry\, f\times id"] \arrow[dr, "f"'] & X^Y\times Y \arrow[d, "eval"] \\
& X
\end{tikzcd}$$

Proposition 2.6 (🐓)**.** *Every exponential object X^Y satisfies the following properties:*

1. *The mapping $curry : \mathscr{C}(X \times Y, Z) \to \mathscr{C}(X \to Z^Y)$ is injective,*
2. *$curry(eval \circ (f \times id)) = f$ for any $f : X \times Y \to Z$,*
3. *$curry\ f \circ g = curry(f \circ (g \times id))$ for any $f : X \times Y \to Z, g : A \to X$.*

A Cartesian closed category is a Cartesian category $\mathscr{C}$ that also has an exponential object X^Y for any $X,Y \in |\mathscr{C}|$. The internal logic of Cartesian closed categories is the simply typed λ-calculus, which makes them a suitable environment for interpreting programming languages. For the rest of this thesis we will work in an ambient distributive category $\mathscr{C}$, that however need not be Cartesian closed as to be more general.

2.2 F-Coalgebras

Let $F : \mathscr{C} \to \mathscr{C}$ be an endofunctor. Recall that F-algebras are tuples $(X, \alpha : FX \to X)$ consisting of an object of $\mathscr{C}$ and a morphism out of the functor. Initial F-algebras have been studied extensively as a means of modeling inductive data types together with induction and recursion principles [7]. For this thesis we will be more interested in the dual concept namely terminal coalgebras; let us formally introduce them now.

Definition 2.7 (F-Coalgebra (🐓))**.** A tuple $(X \in |\mathscr{C}|, \alpha : X \to FX)$ is called an *F-coalgebra* (hereafter referred to as just *coalgebra*).

Definition 2.8 (Coalgebra Morphisms (🐓))**.** Let $(X, \alpha : X \to FX)$ and $(Y, \beta : Y \to FY)$ be two coalgebras. A morphism between these coalgebras is a morphism $f : X \to Y$ such that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "\alpha"] \arrow[d, "f"'] & FX \arrow[d, "Ff"] \\
Y \arrow[r, "\beta"'] & FY
\end{tikzcd}$$

Coalgebras on a given functor together with their morphisms form a category that we call *Coalgs*(F).

Proposition 2.9 (🐓)**.** *Coalgs(F) is a category.*

The terminal object of *Coalgs*(F) is sometimes called *final coalgebra*, we will however call it the *terminal coalgebra* for consistency with initial F-algebras. Similarly to initial F-algebras, the final coalgebra can be used for modeling the semantics of coinductive data types where terminality of the coalgebra yields corecursion as a definitional principle and coinduction as a proof principle. Let us make the universal property of terminal coalgebras concrete.

Definition 2.10 (Terminal Coalgebra (🐓))**.** A coalgebra $(T, t : T \to FT)$ is called a terminal coalgebra if for any other coalgebra $(X, \alpha : X \to FX)$ there exists a unique morphism $[\![\alpha]\!] : X \to T$ satisfying:

$$\begin{tikzcd}
X \arrow[r, "\alpha"] \arrow[d, dashed, "{[\![\alpha]\!]}"'] & FX \arrow[d, "{F[\![\alpha]\!]}"] \\
T \arrow[r, "t"] & FT
\end{tikzcd}$$

We use the common notation νF to denote the terminal coalgebra for F (if it exists).

We will discuss the concrete form that induction and coinduction take in a type theory in Chapter 3. Let us now reiterate a famous Lemma concerning terminal F-coalgebras.

Lemma 2.11 (Lambek's Lemma [1] (🐓))**.** *Let $(T, t : T \to FT)$ be a terminal coalgebra. Then t is an isomorphism.*

2.3 Monads

Monads are widely known in functional programming as a means for modeling effects in "pure" languages and are also central to this thesis. Let us recall the basic definitions[2][5].

Definition 2.12 (Monad (🐓))**.** A monad $\mathbf{T}$ on a category $\mathscr{C}$ is a triple (T, η, μ), where $T : \mathscr{C} \to \mathscr{C}$ is an endofunctor and $\eta : Id \to T, \mu : TT \to T$ are natural transformations, satisfying the following laws:

$$\mu_X \circ \mu_{TX} = \mu_X \circ T\mu_X \tag{M1}$$

$$\mu_X \circ \eta_{TX} = id_{TX} \tag{M2}$$

$$\mu_X \circ T\eta_X = id_{TX} \tag{M3}$$

These laws are expressed by the following diagrams:

$$\begin{tikzcd}
TTTX \arrow[r, "\mu_{TX}"] \arrow[d, "T\mu_X"'] & TTX \arrow[d, "\mu_X"] \\
TTX \arrow[r, "\mu_X"'] & TX
\end{tikzcd}$$

$$\begin{tikzcd}
TX \arrow[r, "\eta"] \arrow[dr, "id"'] & TTX \arrow[d, "\mu"] & TX \arrow[l, "T"'] \arrow[dl, "id"] \\
& TX &
\end{tikzcd}$$

Definition 2.13 (Monad Morphism (🐓))**.** A morphism between monads $(S : \mathscr{C} \to \mathscr{C}, \eta^S, \mu^S)$ and $(T : \mathscr{C} \to \mathscr{C}, \eta^T, \mu^T)$ is a natural transformation $\alpha : S \to T$ between the underlying functors such that the following diagrams commute.

$$\begin{tikzcd}
X \arrow[r, "\eta^S"] \arrow[dr, "\eta^T"'] & SX \arrow[d, "\alpha"] \\
& TX
\end{tikzcd}$$

$$\begin{tikzcd}
SSX \arrow[r, "S\alpha"] \arrow[d, "\mu^S"'] & STX \arrow[r, "\alpha"] & TTX \arrow[d, "\mu^T"] \\
SX \arrow[rr, "\alpha"'] & & TX
\end{tikzcd}$$

This yields a category of monads on a given category $\mathscr{C}$ that we call *Monads*($\mathscr{C}$).

Proposition 2.14. *Monads*($\mathscr{C}$) *is a category.*

Monads can also be specified in a second equivalent way that is better suited to describe computation.

Definition 2.15 (Kleisli Triple (🐓))**.** A Kleisli triple on a category $\mathscr{C}$ is a triple $(F, \eta, (-)^*)$, where $F : |C| \to |C|$ is a mapping on objects, $(\eta_X : X \to FX)_{X \in |C|}$ is a family of morphisms and for every morphism $f : X \to FY$ there exists a morphism $f^* : FX \to FY$ called the Kleisli lifting, where the following laws hold:

$$\eta^*_X \quad = id_{FX} \tag{K1}$$

$$f^* \circ \eta_X = f \qquad \text{for any } f : X \to FY \tag{K2}$$

$$f^* \circ g* = (f^* \circ g)^* \text{ for any } f : Y \to FZ, g : X \to FY \tag{K3}$$

Let $f : X \to TY, g : Y \to TZ$ be two programs, where T is a Kleisli triple. These programs can be composed by taking: $f^* \circ g : X \to TZ$, which is called Kleisli composition. Haskell's do-notation is a useful tool for writing Kleisli composition in a legible way. We will sometimes express $(f^* \circ g)x$ equivalently as

```
do y <- g x
   f y
```

This yields the category of programs for a Kleisli triple that is called the Kleisli category.

Definition 2.16 (Kleisli Category (🐓))**.** Given a monad T on a category $\mathscr{C}$, the Kleisli category $\mathscr{C}^T$ is defined as:

- $|\mathscr{C}^T| = |C|$
- $\mathscr{C}^T(X,Y) = \mathscr{C}(X,TY)$
- Composition of programs is Kleisli composition.
- The identity morphisms are the unit morphisms of T, $id_X = \eta_X : X \to TX$

The laws of categories then follow from the Kleisli triple laws.

Proposition 2.17 ([4] (🐓))**.** *The notions of Kleisli triple and monad are equivalent.*

For the rest of this thesis we will use both equivalent notions interchangeably to make definitions easier.

2.4 Strong and Commutative Monads

Consider the following program in do-notation

```
do y <- g x
   f (x , y)
```

where $g : X \to TY$ and $f : X \times Y \to TZ$ are programs and $\mathbf{T}$ is a monad. Kleisli composition does not suffice for interpreting this program, we will get stuck at

$$X \xrightarrow{\langle id, g \rangle} X \times TY \xrightarrow{?} T(X \times Y) \xrightarrow{f^*} TZ.$$

Instead, one needs the following stronger notion of monad.

Definition 2.18 (Strong Monad (🐓))**.** A monad (T, η, μ) on a Cartesian category $\mathscr{C}$ is called strong if there exists a natural transformation $\tau_{X,Y} : X \times TY \to T(X \times Y)$ that satisfies the following conditions:

$$T\pi_2 \circ \tau_{1,X} \quad = \pi_2 \tag{S1}$$

$$\tau_{X,Y} \circ (id_X \times \eta_Y) \quad = \eta_{X \times Y} \tag{S2}$$

$$\tau_{X,Y} \circ (id_X \times \mu_Y) \quad = \mu_{X \times Y} \circ T\tau_{X,Y} \circ \tau_{X,TY} \tag{S3}$$

$$M\alpha_{X,Y,Z} \circ \tau_{X \times Y,Z} = \tau_{X,Y \times Z} \circ (id_X \times \tau_{Y,Z}) \circ \alpha_{X,Y,TZ} \tag{S4}$$

where $\alpha_{X,Y,Z} = \langle\langle \pi_1, \pi_1 \circ \pi_2 \rangle, \pi_2 \circ \pi_2 \rangle : X \times (Y \times Z) \to (X \times Y) \times Z$ is the associativity morphism on products.

Definition 2.19 (Strong Monad Morphism)**.** A morphism between two strong monads $(S : \mathscr{C} \to \mathscr{C}, \eta^S, \mu^S, \tau^S)$ and $(T : \mathscr{C} \to \mathscr{C}, \eta^T, \mu^T, \tau^T)$ is a morphism between monads as in Definition 2.13 where additionally the following diagram commutes.

$$\begin{tikzcd}
X \times SY \arrow[r, "id \times \alpha"] \arrow[d, "\tau^S"'] & X \times TY \arrow[d, "\tau^T"'] \\
S(X \times Y) \arrow[r, "\alpha"] & T(X \times Y)
\end{tikzcd}$$

As with monads this yields a category of strong monads on $\mathscr{C}$ that we call *StrongMonads*($\mathscr{C}$).

Let us now consider the following two programs

```
do x <- p
   y <- q
   return (x, y)
```

```
do y <- q
   x <- p
   return (x, y)
```

Where $p : TX$ and $q : TY$ are computations of some monad T. A monad where these programs are equal, is called commutative.

Definition 2.20 (Commutative Monad (🐓))**.** A strong monad $\mathbf{T}$ is called commutative if the (right) strength τ commutes with the induced left strength

$$\sigma_{X,Y} = Tswap \circ \tau_{Y,X} \circ swap : TX \times Y \to T(X \times Y)$$

that satisfies symmetrical conditions to the ones τ satisfies. Concretely, $\mathbf{T}$ is called commutative if the following diagram commutes:

$$\begin{tikzcd}
TX \times TY \arrow[r, "\tau"] \arrow[d, "\sigma"'] & T(TX \times Y) \arrow[d, "\sigma^*"] \\
T(X \times TY) \arrow[r, "\tau^*"'] & T(X \times Y)
\end{tikzcd}$$

2.5 Free Objects

Free objects, roughly speaking, are constructions for instantiating structure declarations in a minimal way. We will rely on free structures in Chapter 5 to define a monad in a general setting. We recall the definition to establish some notation and then describe how to obtain a monad via existence of free objects.

Definition 2.21 (Free Object (🧲))**.** Let $\mathscr{C}$,$\mathscr{D}$ be categories and $U : \mathscr{C} \to \mathscr{D}$ be a forgetful functor (whose construction usually is obvious). A free object on some object $X \in |\mathscr{D}|$ is an object $FX \in |\mathscr{C}|$ together with a morphism $\eta : X \to UFX$ such that for any $Y \in |\mathscr{C}|$ and $f : X \to UY$ there exists a unique morphism $f^\star : FX \to Y$ satisfying:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\eta"'] & UY \\
UFX \arrow[ur, dashed, "Uf^\star"'] &
\end{tikzcd}$$

Proposition 2.22 (🧲,🧲)**.** *Let $U : \mathscr{C} \to \mathscr{D}$ be a forgetful functor. If for every $X \in |\mathscr{D}|$ a free object $FX \in |C|$ exists then $(X \mapsto UFX, \eta : X \to UFX, (f : X \to UFY)^\star : UFX \to UFY)$ is a Kleisli triple on $\mathscr{D}$.*

3 Implementing Category Theory in Agda

There are many formalizations of category theory in proof assistants like Coq or Agda. The benefits of such a formalization are clear: having a usable formalization allows one to reason about categorical notions in a type checked environment that makes errors less likely. Ideally such a development will bring researchers together and enable them to work in a unified setting that enables efficient communication of ideas and concepts. In this thesis we will work with the dependently typed programming language Agda [23] and the agda-categories [20] library that serves as an extensive foundation of categorical definitions. This chapter shall serve as a quick introduction to the relevant parts of Agda's type theory, the agda-categories library, and the formalization of this thesis.

3.1 The Underlying Type Theory

Agda implements a Martin-Löf style dependent type theory with *inductive* and *coinductive types* as well as an infinite hierarchy of universes $\texttt{Set}_0$, $\texttt{Set}_1$, …, where usually $\texttt{Set}_0$ is abbreviated as `Set`. Recall that inductive types usually come with a principle for defining functions from inductive types, called *recursion* and a principle for proving facts about the inhabitants of inductive types, called *induction*. These are standard notions and need no further introduction. Coinductive types come with dual principles that are however lesser known. Dually to inductive types that are defined by their *constructors*, coinductive types are defined by their *destructors* or their observational behavior. Take the type of streams over a type `A`, for example. In Agda one would define this type as a coinductive record like so:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

i.e. the type of streams over `A` is defined by the two destructors `head : Stream A → A` and `tail : Stream A → Stream A` that return the head and the tail of the stream respectively. Now, *corecursion* is a principle for defining functions into coinductive types by specifying how results of the function may be observed. Take for example the following function which defines an infinite stream repeating the same argument and is defined by use of Agda's *copatterns*.

```
repeat : {A : Set} (a : A) → Stream A
head (repeat a) = a
tail (repeat a) = repeat a
```

Let us introduce the usual notion of stream bisimilarity. Given two streams, they are bisimilar if their heads are equal and their tails are bisimilar.

```
record _≈_ {A} (s : Stream A) (t : Stream A) : Set where
  coinductive
  field
    head : head s ≡ head t
    tail : tail s ≈ tail t
```

In this definition `_≡_` is the built-in propositional equality in Agda with the single constructor `refl`. We can now use coinduction as a proof principle to proof a fact about streams.

```
repeat-eq : ∀ {A} (a : A) → repeat a ≈ tail (repeat a)
head (repeat-eq {A} a) = refl
tail (repeat-eq {A} a) = repeat-eq a
```

Where in the coinductive step we were able to assume that `repeat a ≈ tail(repeat a)` already holds and showed that thus `tail(repeat a) ≈ tail(tail(repeat a))` holds.

Streams are always infinite and thus this representation of coinductive types as coinductive records is well suited for them. However, consider the type of *possibly* infinite lists, that we will call `coList`. In pseudo notation this type can be defined as

```
codata coList (A : Set) : Set where
  nil : coList A
  _::_ : A → coList A → coList A
```

That is, the coinductive type `coList` is defined by the constructors `nil` and `_::_`. Agda does implement a second way of defining coinductive types that allows exactly such definitions, however the use of these sometimes called *positive coinductive types* is discouraged, since it is known to break subject reduction [21][22]. Instead, sticking to coinductive records, we can define `coList` as two mutual types, one inductive and the other coinductive:

```
mutual
  data coList (A : Set) : Set where
    nil : coList A
    _::_ : A → coList′ A → coList A
  record coList′ (A : Set) : Set where
    coinductive
    field force : coList A
```

Unfortunately, this does add the overhead of having to define functions on `coList` as mutual recursive functions, e.g. the `repeat` function from before can be defined as

```
mutual
  repeat  : {A : Set} (a : A) → coList A
  repeat′ : {A : Set} (a : A) → coList′ A
  repeat a = a ∷ repeat′ a
  force (repeat′ a) = repeat a
```

or more succinctly using a λ-function

```
repeat : {A : Set} (a : A) → coList A
repeat a = a ∷ λ { .force → repeat a }
```

In Chapter 6 we will work with such a coinductive type that is defined by constructors, hence to avoid the overhead of defining every data type twice and using mutual function definitions in the thesis, we will work in a type theory that does offer coinductive types with constructors and their respective corecursion and coinduction principles. However, in the formalization we stick to using coinductive records as to implement best practices. The translation between the two styles is straightforward, as illustrated by the previous example.

3.2 Setoid Enriched Categories

Let us now consider how to implement category theory in Agda. The usual textbook definition of a category glosses over some design decisions that have to be made when implementing it in type theory. One would usually see something like this:

Definition 3.1 (Category)**.** A category consists of

- A collection of objects
- A collection of morphisms between objects
- For every two morphisms $f : X \to Y, g : Y \to Z$ another morphism $g \circ f : X \to Z$ called the composition
- For every object X a morphism $id_X : X \to X$ called the identity

where the composition is associative, and the identity morphisms are identities with respect to the composition.

Here *collection* refers to something that behaves set-like, which is not a set and is needed to prevent size issues (there is no set of all sets, otherwise we would obtain Russel's paradox, but there is a collection of all sets), it is not immediately clear how to translate this to type theory. Furthermore, in mathematical textbooks equality between morphisms is usually taken for granted, i.e. there is some global notion of equality that is clear to everyone. In type theory we need to be more thorough as there is no global notion of equality, eligible for all purposes, e.g. the standard notion of propositional equality has issues when dealing with functions in that it requires extra axioms like functional extensionality.

The definition of category that we will work with can be seen in Listing 1 (unnecessary information has been stripped). The key differences to the definition above are firstly that instead of talking about collections, Agda's infinite `Set` hierarchy is utilized to prevent size issues. This notion of category is thus parametrized by 3 universe levels, one for objects, one for morphisms and one for equalities. A consequence is that the category does not contain a type of all morphisms, instead it contains a type of morphisms for any pair of objects. Furthermore, the types of morphisms are equipped with an equivalence relation _≈_, making them setoids. This addresses the aforementioned issue of how to implement equality between morphisms: the notion of equality is just added to the definition of a category. This version of the notion of category is also called a *setoid-enriched category*.

As a consequence of using a custom equality relation, proofs like `∘-resp-≈` are needed throughout the library to make sure that operations on morphisms respect the equivalence relation. In the thesis we will omit such proofs, but they are contained in our formalization. Lastly, the designers of agda-categories also include symmetric proofs like `sym-assoc` to definitions, in this case to guarantee that the opposite category of the opposite category is equal to the original category, and a similar reason for requiring `identity²`, we won't address the need for these proofs and just accept the requirement as given for the rest of the thesis.

```
record Category (o ℓ e : Level) : Set (suc (o ⊔ ℓ ⊔ e)) where
  field
    Obj : Set o
    _⇒_ : Obj → Obj → Set ℓ
    _≈_ : ∀ {A B : Obj } → (A ⇒ B) → (A ⇒ B) → Set e

    id  : ∀ {A : Obj} → (A ⇒ A)
    _∘_ : ∀ {A B C : Obj} → (B ⇒ C) → (A ⇒ B) → (A ⇒ C)

   assoc     : ∀ {A B C D} {f : A ⇒ B} {g : B ⇒ C} {h : C ⇒ D}
     → (h ∘ g) ∘ f ≈ h ∘ (g ∘ f)
   sym-assoc : ∀ {A B C D} {f : A ⇒ B} {g : B ⇒ C} {h : C ⇒ D}
     → h ∘ (g ∘ f) ≈ (h ∘ g) ∘ f
    identityˡ : ∀ {A B} {f : A ⇒ B} → id ∘ f ≈ f
    identityʳ : ∀ {A B} {f : A ⇒ B} → f ∘ id ≈ f
    identity² : ∀ {A} → id ∘ id {A} ≈ id {A}
    equiv     : ∀ {A B} → IsEquivalence (_≈_ {A} {B})
    ∘-resp-≈  : ∀ {A B C} {f h : B ⇒ C} {g i : A ⇒ B}
      → f ≈ h
      → g ≈ i
      → f ∘ g ≈ h ∘ i
```

Listing 1: Definition of Category [20]

From this it should be clear how other basic notions like functors or natural transformations look in the library.

3.3 The formalization

Every result and used fact (except for Proposition 4.4) in this thesis has been proven either in our own formalization[1] or in the agda-categories library [20]. The formalization is meant to be used as a reference alongside this thesis, where concrete details of proofs can be looked up and verified. The preferred format for viewing the formalization is as automatically generated clickable HTML code[2], where multiple annotations explaining the structure have been added in

[1] https://git8.cs.fau.de/theses/bsc-leon-vatthauer

[2] https://wwwcip.cs.fau.de/ hy84coky/bsc-thesis/

Markdown, however concrete explanations of the proofs and their main ideas are mostly just contained in this thesis.

In the future this formalization may be adapted into a separate library that uses the agda-categories library as a basis but is more focussed on the study of partiality monads and iteration theories. As such the formalization has been structured similar to the agda-categories library, where key concepts such as monads correspond to separate top-level folders, which contain the core definitions as well as folders for sub-concepts and their properties.

4 Partiality Monads

Moggi's categorical semantics [5] describe a way to interpret an effectful programming language in a category. For this one needs a (strong) monad T capturing the desired effects, then we can take the elements of TA as denotations for programs of type A. The Kleisli category of T can be viewed as the category of programs, which gives us a way of composing programs (Kleisli composition).

For this thesis we will restrict ourselves to monads for modeling partiality, the goal of this chapter is to capture what it means to be a partiality monad and look at two common examples.

4.1 Properties of Partiality Monads

We will now look at how to express the following non-controversial properties of a minimal partiality monad categorically:

1. Irrelevance of execution order
2. Partiality of programs
3. No other effect besides some form of non-termination

The first property of course holds for any commutative monad, the other two are more interesting.

To ensure that programs are partial, we recall the following notion by Cockett and Lack [9], that axiomatizes the notion of partiality in a category:

Definition 4.1 (Restriction Structure ()). A restriction structure on a category $\mathscr{C}$ is a mapping $dom : \mathscr{C}(X,Y) \to \mathscr{C}(X,X)$ with the following properties:

$$f \circ (\mathrm{dom}\, f) = f$$
$$(\mathrm{dom}\, f) \circ (\mathrm{dom}\, g) = (\mathrm{dom}\, g) \circ (\mathrm{dom}\, f)$$
$$\mathrm{dom}\,(g \circ (\mathrm{dom}\, f)) = (\mathrm{dom}\, g) \circ (\mathrm{dom}\, f)$$
$$(\mathrm{dom}\, h) \circ f = f \circ \mathrm{dom}\,(h \circ f)$$

for any $X,Y,Z \in |\mathscr{C}|, f : X \to Y, g : X \to Z, h : Y \to Z$.

The morphism $\mathrm{dom}\, f : X \to X$ represents the domain of definiteness of $f : X \to Y$. In the category of partial functions this takes the following form:

$$(\mathrm{dom}\, f)(x) = \begin{cases} x & \text{if } f(x) \text{ is defined} \\ \text{undefined} & \text{else} \end{cases}$$

That is, $\mathrm{dom}\, f$ is only defined on values where f is defined and for those values it behaves like the identity function.

Definition 4.2 (Restriction Category ()). Every category has a trivial restriction structure by taking $dom(f : X \to Y) = id_X$. We call categories with a non-trivial restriction structure *restriction categories*.

For a suitable defined partiality monad T the Kleisli category $\mathscr{C}^T$ should be a restriction category.

Lastly, we also recall the following notion by Bucalo et al. [11] which captures what it means for a monad to have no other side effect besides some sort of non-termination:

Definition 4.3 (Equational Lifting Monad ()). A commutative monad T is called an *equational lifting monad* if the following diagram commutes:

$$\begin{tikzcd}
TX \arrow[r, "\Delta"] \arrow[dr, "T\langle \eta{,}id \rangle"'] & TX \times TX \arrow[d, "\tau"] \\
 & T(TX \times X)
\end{tikzcd}$$

where $\Delta_X : X \to X \times X$ is the diagonal morphism.

To make the equational lifting property more comprehensible we can alternatively state it using do-notation. The equational lifting property states that the following programs must be equal:

```
do x <- p                   do x <- p
   return (x , p)              return (x , return x)
```

That is, if some computation $p : TX$ terminates with the result $x : X$, then $p = return\, x$ must hold afterwards. This of course implies that running p multiple times yields the same result as running p once.

Proposition 4.4 ([11]). *If T is an equational lifting monad the Kleisli category $\mathscr{C}^T$ is a restriction category.*

Definition 4.3 combines all three properties stated at the beginning of the section, so when studying partiality monads in this thesis, we ideally expect them to be equational lifting monads. For the rest of this chapter we will use these definitions to compare two monads that are commonly used to model partiality.

4.2 The Maybe Monad

The endofunctor $MX = X + 1$ extends to a monad by taking $\eta_X = i_1 : X \to X + 1$ and $\mu_X = [id, i_2] : (X+1)+1 \to X+1$. The monad laws follow easily (). This is generally known as the maybe monad and can be viewed as the canonical example of an equational lifting monad.

Theorem 4.5 (). *M is an equational lifting monad.*

In the setting of classical mathematics this monad is therefore sufficient for modeling partiality, but in general it will not be useful for modeling non-termination as a side effect, since one would need to know beforehand whether a program terminates or not. For the purpose of modeling possibly non-terminating computations another monad has been introduced by Capretta [12].

4.3 The Delay Monad

Capretta's delay monad [12] is a coinductive datatype whose inhabitants can be viewed as suspended computations. This datatype is usually defined by the two coinductive constructors *now* $: X \to DX$ and *later* $: DX \to DX$, where *now* lifts a value inside a computation and *later* intuitively delays a computation by one time unit. See Chapter 6 for a type theoretical study of this monad. Categorically we obtain the delay monad by the terminal coalgebras $DX = \nu A.X + A$, which we assume to exist. In this section we will show that these terminal coalgebras indeed yield a monad that is strong and commutative.

Since DX is defined as a terminal coalgebra, we can define morphisms via corecursion and prove theorems by coinduction. By Lemma 2.11 the coalgebra structure $out : DX \to X + DX$ is an isomorphism, whose inverse can be decomposed into the two constructors mentioned before: $out^{-1} = [now, later] : X + DX \to DX$.

Lemma 4.6 (🐓)**.** *The following conditions hold:*

- *$now : X \to DX$ and $later : DX \to DX$ satisfy:*
$$out \circ now = i_1 \qquad out \circ later = i_2 \qquad \text{(D1)}$$
- *For any $f : X \to DY$ there exists a unique morphism $f^* : DX \to DY$ such that the following commutes.*
$$\begin{tikzcd} DX \arrow[r, "out"] \arrow[d, dashed, "f^*"] & X + DX \arrow[d, "{[out \circ f, i_2 \circ f^*]}"] \\ DY \arrow[r, "out"] & Y + DY \end{tikzcd} \qquad \text{(D2)}$$
- *There exists a unique morphism $\tau : X \times DY \to D(X \times Y)$ such that:*
$$\begin{tikzcd} X \times DY \arrow[r, "id \times out"] \arrow[d, dashed, "\tau"] & X \times (Y + DY) \arrow[r, "dstl"] & X \times Y + X \times DY \arrow[d, "id + \tau"'] \\ D(X \times Y) \arrow[rr, "out"] & & X \times Y + D(X \times Y) \end{tikzcd}$$
(D3)

Lemma 4.7 (🐓)**.** *The following properties of* **D** *hold:*

1. $out \circ Df = (f + Df) \circ out$
2. $f^* = [f, (later \circ f)^*] \circ out$
3. $later \circ f^* = (later \circ f)^* = f^* \circ later$

Lemma 4.8 (🐓)**.** $\mathbf{D} := (D, now, (-)^*)$ *is a Kleisli triple.*

Terminality of the coalgebras $(DX, out : DX \to X + DX)_{X \in |\mathscr{C}|}$ yields the following proof principle.

Remark 4.9 (Proof by coinduction)**.** Given two morphisms $f, g : X \to DY$. To show that $f = g$ it suffices to show that there exists a coalgebra structure $\alpha : X \to Y + X$ such that the following diagrams commute:

$$\begin{tikzcd} X \arrow[r, "\alpha"] \arrow[d, "f"] & Y + X \arrow[d, "id + f"] \\ DY \arrow[r, "out"] & Y + DY \end{tikzcd}$$

$$\begin{tikzcd} X \arrow[r, "\alpha"] \arrow[d, "g"] & Y + X \arrow[d, "id + g"] \\ DY \arrow[r, "out"] & Y + DY \end{tikzcd}$$

Uniqueness of the coalgebra morphism $[\![\alpha]\!] : (X, \alpha) \to (DY, out)$ then results in $f = g$.

Lemma 4.10 (🐓)**.** **D** *is a strong monad.*

To prove that **D** is commutative we will use another proof principle previously called the *Solution Theorem* [10] or *Parametric Corecursion* [8]. In our setting this takes the following form.

Definition 4.11 (🐓)**.** We call a morphism $g : X \to D(Y + X)$ *guarded* if there exists a morphism $h : X \to Y + D(Y + X)$ such that the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, "g"] \arrow[d, "h"'] & D(Y + X) \arrow[d, "out"] \\ Y + D(Y + X) \arrow[r, "i_1 + id"'] & (Y + X) + D(Y + X) \end{tikzcd}$$

Corollary 4.12 (Solution Theorem (🐓))**.** *Let $g : X \to D(Y + X)$ be guarded. Solutions of g are unique, i.e. given two morphisms $f, i : X \to DY$ then $f = [now, f]^* \circ g$ and $i = [now, i]^* \circ g$ already implies $f = i$.*

Let us record some facts that we will use to prove commutativity of **D**:

Corollary 4.13 (🐓, 🐓)**.** *These properties of τ and σ hold:*

$$\begin{aligned} out \circ \tau &= (id + \tau) \circ dstl \circ (id \times out) && (\tau_1) \\ out \circ \sigma &= (id + \sigma) \circ dstr \circ (out \times id) && (\sigma_1) \\ \tau \circ (id \times out^{-1}) &= out^{-1} \circ (id + \tau) \circ dstl && (\tau_2) \\ \sigma \circ (out^{-1} \times id) &= out^{-1} \circ (id + \sigma) \circ dstr && (\sigma_2) \end{aligned}$$

Theorem 4.14 (🐓)**.** **D** *is commutative.*

We have now seen that **D** is strong and commutative, however it is not an equational lifting monad, since besides modeling non-termination, the delay monad also counts the execution time of a computation. This is a result of the too intensional notion of equality that this monad comes with.

In Chapter 6 we will see a way to remedy this: taking the quotient of the delay monad where execution time is ignored. This will then yield an equational lifting monad on the category of setoids. However, in a general setting it is generally believed to be impossible to define a monad structure on this quotient. Chapman et al. [16] have identified the axiom of countable choice (which crucially holds in the category of setoids) as a sufficient requirement for defining a monad structure on the quotient of **D**.

5 Iteration Algebras and Monads

In this chapter we will draw on the inherent connection between partiality and iteration to establish a partiality monad in a general setting without axioms by utilizing previous research on iteration theories.

5.1 Elgot Algebras

Recall the following notion from [13], previously called *complete Elgot algebra.*

Definition 5.1 (Guarded Elgot Algebra (🐔))**.** Given a functor $H : \mathscr{C} \to \mathscr{C}$, a *(H-)guarded Elgot algebra* consists of:

- An object $A \in |\mathscr{C}|$,
- a H-algebra structure $a : HA \to A$,
- and for every $f : X \to A + HX$ an *iteration* $f^\sharp : X \to A$, satisfying the following axioms:
  - **Fixpoint**: $f^\sharp = [id, a \circ H(f^\sharp)] \circ f$
    for any $f : X \to A + HX$,
  - **Uniformity**: $(id + Hh) \circ f = g \circ h$ implies $f^\sharp = g^\sharp \circ h$
    for any $f : X \to A + HX, g : Y \to A + HY, h : X \to Y$,
  - **Compositionality**: $((f^\sharp + id) \circ h)^\sharp = ([(id + Hi_1) \circ f, i_2 \circ Hi_2] \circ [i_1, h])^\sharp \circ i_2$
    for any $f : X \to A + HX, h : Y \to X + HY$.

Consider an Elgot algebra over the identity functor $Id : \mathscr{C} \to \mathscr{C}$ together with the trivial Id-algebra structure $id : A \to A$. Morphisms of the form $f : X \to A + X$ can then be viewed as modeling one iteration of a loop, where $X \in |\mathscr{C}|$ is the state space and $A \in |\mathscr{C}|$ the object of values. Intuitively, in such a setting the iteration operator $(-)^\sharp$ runs such a morphism in a loop until it terminates (or diverges), thus assigning it a solution. This is what the **Fixpoint** axiom guarantees. On the other hand the **Uniformity** axiom states how to handle loop invariants and finally, the **Compositionality** axiom is the most sophisticated one, stating that compatible iterations can be combined into a single iteration with a merged state space.

The previous intuition gives rise to the following simpler definition that has been introduced in [19].

Definition 5.2 (Elgot Algebra (🐔))**.** A *(unguarded) Elgot Algebra* [19] consists of:

- An object $A \in |\mathscr{C}|$,
- and for every $f : X \to A + X$ an *iteration* $f^\sharp : X \to A$, satisfying the following axioms:
  - **Fixpoint**: $f^\sharp = [id, f^\sharp] \circ f$
    for any $f : X \to A + X$,
  - **Uniformity**: $(id + h) \circ f = g \circ h$ implies $f^\sharp = g^\sharp \circ h$
    for any $f : X \to A + X, g : Y \to A + Y, h : X \to Y$,
  - **Folding**: $((f^\sharp + id) \circ h)^\sharp = [(id + i_1) \circ f, i_2 \circ h]^\sharp$
    for any $f : X \to A + X, h : Y \to X + Y$.

Note that the **Uniformity** axiom requires an identity to be proven, before it can be applied. However, we will omit these proofs in most parts of the thesis, since they mostly follow by simple rewriting on (co)products, the full proofs can be looked up in the accompanying formalization.

Now, in this setting the simpler **Folding** axiom replaces the sophisticated **Compositionality** axiom. Indeed, for Id-guarded Elgot algebras with a trivial algebra structure, the **Folding** and **Compositionality** axioms are equivalent [19], which is partly illustrated by the following Lemma.

Lemma 5.3 (🐔)**.** *Every Elgot algebra* $(A, (-)^\sharp)$ *satisfies the following additional axioms*

- ***Compositionality:*** $((f^\sharp + id) \circ h)^\sharp = ([(id + i_1) \circ f, i_2 \circ i_2] \circ [i_1, h])^\sharp \circ i_2$
  for any $f : X \to A + X, h : Y \to X + Y$,
- ***Stutter:*** $(([h, h] + id) \circ f)^\sharp = (i_1 \circ h, [h + i_1, i_2 \circ i_2])^\sharp \circ inr$
  for any $f : X \to (Y + Y) + X, h : Y \to A$,
- ***Diamond:*** $((id + \Delta) \circ f)^\sharp = ([i_1, ((id + \Delta) \circ f)^\sharp + id] \circ f)^\sharp$
  for any $f : X \to A + (X + X)$.

Note that in [19] it has been shown that the **Diamond** axiom implies **Compositionality**, yielding another definition of Elgot algebras only requiring the **Fixpoint**, **Uniformity** and **Diamond** axioms.

Let us now consider morphisms that are coherent with respect to the iteration operator. A special case being morphisms between Elgot algebras.

Definition 5.4 (Iteration Preserving Morphisms (🐔))**.** Let $(A, (-)^{\sharp_a}), (B, (-)^{\sharp_b})$ be two Elgot algebras.

A morphism $f : X \times A \to B$ is called *right iteration preserving* if

$$f \circ (id \times h^{\sharp_a}) = ((f + id) \circ dstl \circ (id \times h))^{\sharp_b}$$

for any $h : Y \to A + Y$.

Symmetrically a morphism $g : A \times X \to B$ is called *left iteration preserving* if

$$f \circ (h^{\sharp_a} \times id) = ((f + id) \circ dstr \circ (h \times id))^{\sharp_b}$$

for any $h : Y \to A + Y$.

Let us also consider the special case where $X = 1$. A morphism $f : A \to B$ is called *iteration preserving* if

$$f \circ h^{\sharp_a} = ((f + id) \circ h)^{\sharp_b}$$

for any $h : Y \to A + Y$.

We will now study the category of Elgot algebras and iteration preserving morphisms that we call *ElgotAlgs*($\mathscr{C}$). Let us introduce notation for morphisms between Elgot algebras: we denote an Elgot algebra morphism $f:(A,(-)^{\sharp_a}) \to (B,(-)^{\sharp_b})$ as $f : A \hookrightarrow B$, where we omit stating the iteration operator.

Lemma 5.5 (🐓)**.** *ElgotAlgs*($\mathscr{C}$) *is a category.*

Products and exponentials of Elgot algebras can be formed in a canonical way, which is illustrated by the following two Lemmas.

Lemma 5.6 (🐓)**.** *If $\mathscr{C}$ is a Cartesian category, so is ElgotAlgs($\mathscr{C}$).*

Lemma 5.7 (🐓)**.** *Given $X \in |\mathscr{C}|$ and $A \in |ElgotAlgs(\mathscr{C})|$. The exponential X^A (if it exists) can be equipped with an Elgot algebra structure.*

5.2 The Initial Pre-Elgot Monad

In this section we will study the monad that arises from existence of all free Elgot algebras. We will show that this is an equational lifting monad and also the initial strong pre-Elgot monad. Starting in this section we will now omit indices of the iteration operator of Elgot algebras for the sake of readability.

Let us first recall the following notion that was introduced in [14] and reformulated in [19].

Definition 5.8 (Elgot Monad)**.** An Elgot monad consists of

- A monad **T**,
- for every $f : X \to T(Y + X)$ an iteration $f^\dagger : X \to TY$ satisfying:
  - **Fixpoint**: $f^\dagger = [\eta, f^\dagger]^* \circ f$
    for any $f : X \to T(Y + X)$,
  - **Uniformity**: $f \circ h = T(id + h)$ implies $f^\dagger \circ g = g^\dagger$
    for any $f : X \to T(Y + X), g : Z \to T(Y + Z), h : Z \to X$,
  - **Naturality**: $g^* \circ f^\dagger = ([Ti_1 \circ g, \eta \circ i_2]^* \circ f)^\dagger$
    for any $f : X \to T(Y + X), g : Y \to TZ$,
  - **Codiagonal**: $f^{\dagger\dagger} = (T[id, i_2] \circ f)^\dagger$
    for any $f : X \to T((Y + X) + X)$.

If the monad **T** is strong with strength τ and $\tau \circ (id \times f^\dagger) = (T dstl \circ \tau \circ (id \times f))^\dagger$ for any $f : X \to T(Y + X)$, then **T** is a strong Elgot monad.

We regard Elgot monads as minimal semantic structures for interpreting side-effecting while loops, as has been argued in [17], [18]. The following notion has been introduced in [19] as a weaker approximation of the notion of Elgot monad, using less sophisticated axioms.

Definition 5.9 (Pre-Elgot Monad (🐓))**.** A monad **T** is called pre-Elgot if every TX extends to an Elgot algebra such that for every $f : X \to TY$ the Kleisli lifting $f^* : TX \to TY$ is iteration preserving.

If the monad **T** is additionally strong and the strength τ is right iteration preserving we call **T** strong pre-Elgot.

(Strong) pre-Elgot monads form a subcategory of *Monads*($\mathscr{C}$) where objects are (strong) pre-Elgot monads and morphisms between pre-Elgot monads are natural transformations α as in Definition 2.13 such that additionally each α_X is iteration preserving. Similarly, morphisms between strong pre-Elgot monads are natural transformations α as in Definition 2.19 such that each α_X is iteration preserving. We call these categories *PreElgot*($\mathscr{C}$) and *StrongPreElgot*($\mathscr{C}$) respectively.

Lemma 5.10 (🐓, 🐓)**.** *PreElgot($\mathscr{C}$) and StrongPreElgot($\mathscr{C}$) are categories.*

Assuming a form of the axiom of countable choice it has been proven in [19] that the initial pre-Elgot monad and the initial Elgot monad coincide, thus closing the expressivity gap in such a setting. However, it is believed to be impossible to close this gap in a general setting.

Proposition 5.11 (🐓)**.** *Existence of all free Elgot algebras yields a monad that we call* **K**.

We will need a notion of stability for **K** to make progress, since we do not assume $\mathscr{C}$ to be Cartesian closed.

Definition 5.12 (Right-Stable Free Elgot Algebra (🐓, 🐓))**.** Let KY be a free Elgot algebra on $Y \in |\mathscr{C}|$. We call KY *right-stable* if for every $A \in ElgotAlgs(\mathscr{C}), X \in |\mathscr{C}|$, and $f : X \times Y \to A$ there exists a unique right iteration preserving $f^{\blacktriangleright} : X \times KY \to A$ such that

$$\begin{tikzcd}
X \times Y \arrow[r, "f"] \arrow[d, "id \times \eta"'] & A \\
X \times KY \arrow[ur, "f^{\blacktriangleright}"'] &
\end{tikzcd}$$

commutes.

A symmetrical variant of the previous definition is sometimes useful.

Definition 5.13 (Left-Stable Free Elgot Algebra (🐓, 🐓))**.** Let KY be a free Elgot algebra on $Y \in |\mathscr{C}|$. We call KY *left-stable* if for every $A \in ElgotAlgs(\mathscr{C}), X \in |\mathscr{C}|$, and $f : Y \times X \to A$ there exists a unique left iteration preserving $f^{\blacktriangleleft} : KY \times X \to A$ such that

$$\begin{tikzcd}
X \times Y \arrow[r, "f"] \arrow[d, "\eta \times id"'] & A \\
KX \times Y \arrow[ur, "f^{\blacktriangleleft}"'] &
\end{tikzcd}$$

commutes.

Lemma 5.14 (🐓)**.** *Definitions 5.12 and 5.13 are equivalent in the sense that they imply each other.*

Lemma 5.15 (🐓)**.** *In a Cartesian closed category every free Elgot algebra is stable.*

For the rest of this chapter we will assume every KX to exist and be stable. Under these assumptions we show that $\mathbf{K}$ is an equational lifting monad and in fact the initial strong pre-Elgot monad. Let us first introduce a proof principle similar to the one introduced in Remark 4.9.

Remark 5.16 (Proof by right-stability (🐓))**.** Given two morphisms $g, h : X \times KY \to A$ where $X, Y \in |\mathscr{C}|, A \in |ElgotAlgs(\mathscr{C})|$. To show that $g = h$, it suffices to show that g and h are right iteration preserving and there exists a morphism $f : X \times Y \to A$ such that

$$\begin{tikzcd}
X \times KY \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & A \\
X \times Y \arrow[u, "id \times \eta"] \arrow[ru, "f"'] &
\end{tikzcd}$$

commutes.

Of course there is also a symmetric version of this.

Remark 5.17 (Proof by left-stability (🐓))**.** Given two morphisms $g, h : KX \times Y \to A$ where $X, Y \in |\mathscr{C}|, A \in |ElgotAlgs(\mathscr{C})|$. To show that $g = h$, it suffices to show that g and h are left iteration preserving and there exists a morphism $f : X \times Y \to A$ such that

$$\begin{tikzcd}
KX \times Y \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & A \\
X \times Y \arrow[u, "\eta \times id"] \arrow[ru, "f"'] &
\end{tikzcd}$$

commutes.

Lemma 5.18 (🐓)**.** $\mathbf{K}$ *is a strong monad.*

As we did when proving commutativity of $\mathbf{D}$, let us record some facts about τ and the induced σ, before proving commutativity of $\mathbf{K}$.

Corollary 5.19 (🐓)**.** *σ is left iteration preserving and satisfies $\sigma \circ (\eta \times id) = \eta$ and the following properties of τ and σ hold.*

$$\tau \circ (f^* \times g^*) = (\tau \circ (id \times g))^* \circ \tau \circ (f^* \times id) \tag{τ_1}$$

$$\sigma \circ (f^* \times g^*) = (\sigma \circ (f \times id))^* \circ \sigma \circ (id \times g^*) \tag{σ_1}$$

The following Lemma is central to the proof of commutativity.

Lemma 5.20 (🐓)**.** *Given $f : X \to KY + X, g : Z \to KA + Z$,*
$\tau^* \circ \sigma \circ (((\eta + id) \circ f)^\sharp \times ((\eta + id) \circ g)^\sharp) = \sigma^* \circ \tau \circ (((\eta + id) \circ f)^\sharp \times ((\eta + id) \circ g)^\sharp)$.

Lemma 5.21 (🐓)**.** $\mathbf{K}$ *is a commutative monad.*

Theorem 5.22 (🐓)**.** $\mathbf{K}$ *is an equational lifting monad.*

Theorem 5.23 (🐓,🐓)**.** $\mathbf{K}$ *is the initial (strong) pre-Elgot monad.*

6 A Case Study on Setoids

In Chapter 4 we have argued that the delay monad is not an equational lifting monad, because it does not only model partiality, but it also considers computation time in its built-in notion of equality. One way to remedy this is to take the quotient of the delay monad where computations with the same result are identified. In this chapter we will use the quotients-as-setoid approach, i.e. we will work in the category of setoids and show that the quotiented delay monad is an instance of the previously defined monad **K** in this category.

6.1 Setoids in Type Theory

We will now introduce the category that the rest of the chapter will take place in. Let us start with some basic definitions.

Definition 6.1 (Setoid (🐓))**.** A setoid is a tuple $(A, \overset{A}{=})$ where A (usually called the *carrier*) is a type and $\overset{A}{=}$ is an equivalence relation on the inhabitants of A.

For brevity, we will not use the tuple notation most of the time, instead we will just say 'Let A be a setoid' and implicitly call the equivalence relation $\overset{A}{=}$.

Definition 6.2 (Setoid Morphism (🐓))**.** A morphism between setoids A and B constitutes a function $f : A \to B$ between the carriers, such that f respects the equivalences, i.e. for any $x,y : A$, $x \overset{A}{=} y$ implies $f\,x \overset{B}{=} f\,y$. We will denote setoid morphisms as $A \leadsto B$.

Let us now consider the function space setoid, which is of special interest, since it carries a notion of equality between functions.

Definition 6.3 (Function Space Setoid)**.** Given two setoids A and B, the function space setoid on these setoids is defined as $(A \leadsto B, \doteq)$ or just $A \leadsto B$, where $\doteq$ is the point wise equality on setoid morphisms.

Setoids together with setoid morphisms form a category that we will call *Setoids*. Properties of *Setoids* have already been examined in [15], however we will reiterate some of these properties now to introduce notation that will be used for the rest of the chapter.

Proposition 6.4 (🐓)**.** *Setoids is a distributive category.*

Proposition 6.5 (🐓)**.** *Setoids is Cartesian closed.*

6.2 Quotienting the Delay Monad

In this section we will introduce data types only using inference rules. For that we adopt the convention that coinductive types are introduced by doubled lines while inductive types are introduced with a single line.

Now, recall from previous chapters that Capretta's delay monad [12] is a coinductive type defined by the two constructors:

$$\frac{x : A}{\mathit{now}\, x : D\,A} \qquad \frac{x : D\,A}{\mathit{later}\, x : D\,A}$$

Furthermore, let us recall two different notions of bisimilarity between inhabitants of the delay type that have been studied previously in [16]. Afterwards, we will reiterate some facts that have been proven in [16] to then finally prove that the quotiented delay type extends to an instance of the monad **K** that has been introduced in Chapter 5.

Let A be a setoid. Lifting the equivalence $\overset{A}{=}$ to $D\ A$ yields another equivalence called *strong bisimilarity.* This equivalence is defined by the rules

$$\frac{x \overset{A}{=} y}{x \sim y} \qquad \frac{x \sim y}{\mathit{later}\, x \sim \mathit{later}\, y}$$

Proposition 6.6 ([16] (🐓))**.** $(D\,A, \sim)$ *is a setoid and admits a monad structure.*

Computations in $(D\,A, \sim)$ are only identified if they evaluate to the same result in the same number of steps. In many contexts this behavior is too intensional. Instead, we will now consider the quotient of this setoid, where all computations that evaluate to the same result are identified. Let us first define a relation that states that two computations evaluate to the same result

$$\frac{x \overset{A}{=} y}{\mathit{now}\, x \downarrow y} \qquad \frac{x \downarrow c}{\mathit{later}\, x \downarrow c}\ .$$

Now, we call two computations p and q *weakly bisimilar* or $p \approx q$ if they evaluate to the same result, or don't evaluate at all, which is specified by the rules

$$\frac{a \overset{A}{=} b \qquad x \downarrow a \qquad y \downarrow b}{x \approx y} \qquad \frac{x \approx y}{\mathit{later}\, x \approx \mathit{later}\, y}$$

Proposition 6.7 ([12] (🐓))**.** $(D\,A, \approx)$ *is a setoid and admits a monad structure.*

For the rest of this chapter we will abbreviate $\tilde{D}\,A = (D_A, \sim)$ and $\tilde{\tilde{D}}\,A = (D_A, \approx)$.

Lemma 6.8 (🐓)**.** *Every $\tilde{\tilde{D}}\,A$ can be equipped with an Elgot algebra structure.*

In the next proof a notion of *discretized* setoid is needed, i.e. given a setoid Z, we can discretize Z by replacing the equivalence relation with propositional equality, yielding $|Z| := (Z, \equiv)$. Now, the following corollary describes how to transform an iteration on $\tilde{\tilde{D}}\,A$ into an iteration on $\tilde{D}\,A$.

Corollary 6.9 (🐓)**.** *Given a setoid morphism $g : X \rightsquigarrow \tilde{\tilde{D}}\ A + X$, there exists a setoid morphism $\bar{g} : |X| \rightsquigarrow \tilde{D}\,A + |X|$ such that $g^{\sharp}\,x \sim \bar{g}^{\tilde{\sharp}}\,x$ for any $x : X$.*

Theorem 6.10 (🐓)**.** *Every $\tilde{\tilde{D}}\,A$ can be equipped with a free Elgot algebra structure.*

We have shown in Theorem 6.10 that every $\tilde{\tilde{D}}\ A$ extends to a free Elgot algebra. Together with Proposition 6.5 and Lemma 5.15 this yields a description for the monad **K** which has been defined in Chapter 5, in the category *Setoids*.

7 Conclusion

We have considered a novel approach to defining a monad suitable for modelling partiality from first principles, which has first been introduced in [19]. Using the dependently typed programming language Agda, we were able to formally verify important properties of this monad: it is an equational lifting monad, i.e. a monad that offers no other side effect besides some form of non-termination and furthermore it turns out to be the initial pre-Elgot monad. Moreover, we have considered a concrete description of this monad in the category of setoids, where it turns out to be a quotient of the delay monad.

With this thesis we have thus created a small Agda library that contains categorical concepts concerning partiality and iteration theories. Future work might improve on this library by formalizing important results concerning partiality monads, such as the fact that every equational lifting monad has a restriction category as its Kleisli category. Furthermore, one can continue studying the delay monad in a categorical setting, by modeling the quotient by weak bisimilarity of the delay monad through a certain coequalizer, as has been done in [19], and then identifying assumptions under which this constitutes a suitable monad for modeling partiality.

Bibliography

[1] J. Lambek, ‘A fixpoint theorem for complete categories,’ *Mathematische Zeitschrift*, vol. 103, pp. 151–161, 1968.

[2] S. M. Lane, ‘Categories for the working mathematician,’ 1971. [Online]. Available: `https://api.semanticscholar.org/CorpusID:122892655`.

[3] G. D. Plotkin, ‘Call-by-name, call-by-value and the λ-calculus,’ *Theoretical computer science*, vol. 1, no. 2, pp. 125–159, 1975.

[4] E. G. Manes, ‘Algebraic theories in a category,’ *Algebraic Theories*, pp. 161–279, 1976.

[5] E. Moggi, ‘Notions of computation and monads,’ *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, Jul. 1991, ISSN: 0890-5401. DOI: `10.1016/0890-5401(91)90052-4`. [Online]. Available: `https://doi.org/10.1016/0890-5401(91)90052-4`.

[6] D. S. Scott, ‘A type-theoretical alternative to iswim, cuch, owhy,’ *Theoretical Computer Science*, vol. 121, no. 1-2, pp. 411–440, 1993.

[7] V. Vene, *Categorical programming with inductive and coinductive types*. Citeseer, 2000.

[8] L. S. Moss, ‘Parametric corecursion,’ *Theoretical Computer Science*, vol. 260, no. 1, pp. 139–163, 2001, Coalgebraic Methods in Computer Science 1998, ISSN: 0304-3975. DOI: `https://doi.org/10.1016/S0304-3975(00)00126-2`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0304397500001262`.

[9] J. R. B. Cockett and S. Lack, ‘Restriction categories i: Categories of partial maps,’ *Theor. Comput. Sci.*, vol. 270, no. 1–2, pp. 223–259, Jan. 2002, ISSN: 0304-3975. DOI: `10.1016/S0304-3975(00)00382-0`. [Online]. Available: `https://doi.org/10.1016/S0304-3975(00)00382-0`.

[10] P. Aczel, J. Adámek, S. Milius, and J. Velebil, ‘Infinite trees and completely iterative theories: A coalgebraic view,’ *Theor. Comput. Sci.*, vol. 300, no. 1–3, pp. 1–45, May 2003, ISSN: 0304-3975. DOI: `10.1016/S0304-3975(02)00728-4`. [Online]. Available: `https://doi.org/10.1016/S0304-3975(02)00728-4`.

[11] A. Bucalo, C. Führmann, and A. Simpson, ‘An equational notion of lifting monad,’ *Theor. Comput. Sci.*, vol. 294, no. 1–2, pp. 31–60, Feb. 2003, ISSN: 0304-3975. DOI: `10.1016/S0304-3975(01)00243-2`. [Online]. Available: `https://doi.org/10.1016/S0304-3975(01)00243-2`.

[12] V. Capretta, ‘General recursion via coinductive types,’ *CoRR*, vol. abs/cs/0505037, 2005. arXiv: `cs/0505037`. [Online]. Available: `http://arxiv.org/abs/cs/0505037`.

[13] J. Adámek, S. Milius, and J. Velebil, ‘Elgot algebras,’ *CoRR*, vol. abs/cs/0609040, 2006. arXiv: `cs/0609040`. [Online]. Available: `http://arxiv.org/abs/cs/0609040`.

[14] J. Adámek, S. Milius, and J. Velebil, ‘Elgot theories: A new perspective on the equational properties of iteration,’ *Mathematical Structures in Computer Science*, vol. 21, no. 2, pp. 417–480, 2011. DOI: `10.1017/S0960129510000496`.

[15] Y. Kinoshita and J. Power, ‘Category theoretic structure of setoids,’ *Theoretical Computer Science*, vol. 546, pp. 145–163, 2014, Models of Interaction: Essays in Honour of Glynn Winskel, ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2014.03.006`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0304397514001819`.

[16] J. Chapman, T. Uustalu, and N. Veltri, ‘Quotienting the delay monad by weak bisimilarity,’ in *Proceedings of the 12th International Colloquium on Theoretical Aspects of Computing - ICTAC 2015 - Volume 9399*, Berlin, Heidelberg: Springer-Verlag, 2015, pp. 110–125, ISBN: 9783319251493. DOI: `10.1007/978-3-319-25150-9_8`. [Online]. Available: `https://doi.org/10.1007/978-3-319-25150-9_8`.

[17] S. Goncharov, L. Schröder, C. Rauch, and M. Piróg, ‘Unifying guarded and unguarded iteration,’ in *Foundations of Software Science and Computation Structures: 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 20*, Springer, 2017, pp. 517–533.

[18] S. Goncharov, L. Schröder, C. Rauch, and J. Jakob, ‘Unguarded recursion on coinductive resumptions,’ *Logical Methods in Computer Science*, vol. 14, 2018.

[19] S. Goncharov, ‘Uniform elgot iteration in foundations,’ *CoRR*, vol. abs/2102.11828, 2021. arXiv: `2102.11828`. [Online]. Available: `https://arxiv.org/abs/2102.11828`.

[20] J. Z. S. Hu and J. Carette, ‘Formalizing category theory in agda,’ in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 327–342, ISBN: 9781450382991. DOI: `10.1145/3437992.3439922`. [Online]. Available: `https://doi.org/10.1145/3437992.3439922`.

[21] T. A. Team, *Agda user manual*, version 2.6.4.3, Mar. 2024. [Online]. Available: `https://agda.readthedocs.io/en/v2.6.4.3/`.

[22] T. C. D. Team, *The coq reference manual*, version 8.19.1, Mar. 2024. [Online]. Available: `https://coq.inria.fr/doc/V8.19.0/refman/`.

[23] Agda Developers, *Agda*, version 2.6.5. [Online]. Available: `https://agda.readthedocs.io/`.