Friedrich-Alexander-Universität
Technische Fakultät

Implementing Categorical Notions of Partiality and Delay in Agda

Leon Vatthauer

April 6, 2024

1. Partiality in Type Theory
2. Categorical Notions of Partiality
3. Implementation in Agda

Partiality in Haskell

In Haskell we are able to define arbitrary partial functions

- Some can be spotted easily by their definition:

```haskell
head :: [a] -> a
head (x:xs) = x
```

Partiality in Haskell

In Haskell we are able to define arbitrary partial functions

- Some can be spotted easily by their definition:

```haskell
head :: [a] -> a
head (x:xs) = x
```

ghci> head []
Exception: code-examples/example.hs:2:1-14:
Non-exhaustive patterns in function head

Partiality in Haskell

In Haskell we are able to define arbitrary partial functions

- Some can be spotted easily by their definition:

```haskell
head :: [a] -> a
head (x:xs) = x
```

ghci> head []
Exception: code-examples/example.hs:2:1-14:
Non-exhaustive patterns in function head

- others might be more subtle:

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

Partiality in Haskell

In Haskell we are able to define arbitrary partial functions

- Some can be spotted easily by their definition:

```haskell
head :: [a] -> a
head (x:xs) = x
```

```
ghci> head []
Exception: code-examples/example.hs:2:1-14:
Non-exhaustive patterns in function head
```

- others might be more subtle:

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

```
ghci> ones = 1 : ones
ghci> reverse ones
...
```

Partiality in Agda
The Maybe Monad

- In Agda every function has to be total and terminating, so how do we model partial functions?

Partiality in Agda

The Maybe Monad

- In Agda every function has to be total and terminating, so how do we model partial functions?
- Simple errors can be modelled with the maybe monad

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

```
head : ∀ A → List A → Maybe A
head nil          = nothing
head (cons x xs)  = just x
```

Partiality in Agda

The Maybe Monad

- In Agda every function has to be total and terminating, so how do we model partial functions?
- Simple errors can be modelled with the maybe monad

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

```
head : ∀ A → List A → Maybe A
head nil           = nothing
head (cons x xs) = just x
```

- What about `reverse` for (possibly) infinite lists:

```
data Colist (A : Set) : Set where
  [] : Colist A
  _∷_ : A → ∞ (Colist A) → Colist A
```

Partiality in Agda

Capretta's Delay Monad

- Capretta's Delay Monad is a **coinductive** data type whose inhabitants can be viewed as suspended computations.

```
data Delay (A : Set) : Set where
  now : A → Delay A
  later : ∞ (Delay A) → Delay A
```

Partiality in Agda

Capretta's Delay Monad

- Capretta's Delay Monad is a **coinductive** data type whose inhabitants can be viewed as suspended computations.

```
data Delay (A : Set) : Set where
  now : A → Delay A
  later : ∞ (Delay A) → Delay A
```

- The delay datatype contains a constant for non-termination:

```
never : Delay A
never = later (♯ never)
```

Partiality in Agda

Capretta's Delay Monad

- Capretta's Delay Monad is a **coinductive** data type whose inhabitants can be viewed as suspended computations.

```
data Delay (A : Set) : Set where
  now : A → Delay A
  later : ∞ (Delay A) → Delay A
```

- The delay datatype contains a constant for non-termination:

```
never : Delay A
never = later (♯ never)
```

- and we can define a function for *running* a computation (for some amount of steps):

```
run_for_steps : Delay A → ℕ → Delay A
run now   x for n      steps = now x
run later x for zero   steps = later x
run later x for suc n steps = run ♭ x for n steps
```

Partiality in Agda

Capretta's Delay Monad

- Now we can define a reverse function for possibly infinite lists:

```
reverse : ∀ {A : Set} → Colist A → Delay (Colist A)
reverse {A} l = revAcc l []
  where
    revAcc : Colist A → Colist A → Delay (Colist A)
    revAcc []        a = now a
    revAcc (x ∷ xs) a = later (♯ revAcc (♭ xs) (x ∷ (♯ a)))
```

Preliminaries

We work in category $\mathcal{C}$ that

- has finite products

Preliminaries

We work in category $\mathcal{C}$ that

- has finite products
- has finite coproducts

Preliminaries

We work in category $\mathcal{C}$ that

- has finite products
- has finite coproducts
- is distributive, i.e. the following is an iso:

$$(X \times Y) + (X \times Z) \xrightarrow{dstl^{-1} := [\langle id, inl \rangle, \langle id, inr \rangle]} X \times (Y + Z)$$

$$X \times (Y + Z) \xrightarrow{dstl} (X \times Y) + (X \times Z)$$

Preliminaries

We work in category $\mathcal{C}$ that

- has finite products
- has finite coproducts
- is distributive, i.e. the following is an iso:

$$(X \times Y) + (X \times Z) \xrightarrow{dstl^{-1} := [\langle id, inl \rangle, \langle id, inr \rangle]} X \times (Y + Z)$$

$$X \times (Y + Z) \xrightarrow{dstl} (X \times Y) + (X \times Z)$$

- has a natural numbers object $\mathbb{N}$ (which is stable)

Capturing Partiality

Moggi's categorical semantics [Mog91]

Goal: interpret an effectul programming language in a category $\mathcal{C}$

Capturing Partiality

Moggi's categorical semantics [Mog91]

Goal: interpret an effectful programming language in a category $\mathcal{C}$

- Take a Monad T on $\mathcal{C}$, we view objects A as types of values and objects TA as types of computations.

Capturing Partiality

Moggi's categorical semantics [Mog91]

Goal: interpret an effectful programming language in a category $\mathcal{C}$

- Take a Monad T on $\mathcal{C}$, we view objects A as types of values and objects TA as types of computations.
- Programs form a category $\mathcal{C}_T$ with $\mathcal{C}_T(X, Y) \coloneqq \mathcal{C}(X, TY)$

Capturing Partiality

Moggi's categorical semantics [Mog91]

Goal: interpret an effectful programming language in a category $\mathcal{C}$

- Take a Monad T on $\mathcal{C}$, we view objects A as types of values and objects TA as types of computations.
- Programs form a category $\mathcal{C}_T$ with $\mathcal{C}_T(X,Y) \coloneqq \mathcal{C}(X,TY)$

What properties should a monad T for modelling partiality have?

Capturing Partiality

Moggi's categorical semantics [Mog91]

Goal: interpret an effectful programming language in a category $\mathcal{C}$

- Take a Monad T on $\mathcal{C}$, we view objects A as types of values and objects TA as types of computations.
- Programs form a category $\mathcal{C}_T$ with $\mathcal{C}_T(X, Y) \coloneqq \mathcal{C}(X, TY)$

What properties should a monad T for modelling partiality have?

1. Commutativity (also entails strength)

Capturing Partiality

Moggi's categorical semantics [Mog91]

Goal: interpret an effectful programming language in a category $\mathcal{C}$

- Take a Monad T on $\mathcal{C}$, we view objects A as types of values and objects TA as types of computations.
- Programs form a category $\mathcal{C}_T$ with $\mathcal{C}_T(X, Y) := \mathcal{C}(X, TY)$

What properties should a monad T for modelling partiality have?

1. Commutativity (also entails strength)
2. Morphisms in $\mathcal{C}_T$ should be partial maps

Capturing Partiality
Moggi's categorical semantics [Mog91]

Goal: interpret an effectul programming language in a category $\mathcal{C}$

- Take a Monad T on $\mathcal{C}$, we view objects A as types of values and objects TA as types of computations.
- Programs form a category $\mathcal{C}_T$ with $\mathcal{C}_T(X, Y) := \mathcal{C}(X, TY)$

What properties should a monad T for modelling partiality have?

1. Commutativity (also entails strength)
2. Morphisms in $\mathcal{C}_T$ should be partial maps
3. There should be no other effect besides partiality

Capturing Partiality

Restriction Categories [CL02]

Definition

A restriction structure on $\mathcal{C}$ is a mapping $\text{dom} : \mathcal{C}(X, Y) \to \mathcal{C}(X, X)$ with the following properties:

$$f \circ (\text{dom } f) = f \tag{1}$$
$$(\text{dom } f) \circ (\text{dom } g) = (\text{dom } g) \circ (\text{dom } f) \tag{2}$$
$$\text{dom } (g \circ (\text{dom } f)) = (\text{dom } g) \circ (\text{dom } f) \tag{3}$$
$$(\text{dom } h) \circ f = f \circ \text{dom } (h \circ f) \tag{4}$$

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

Capturing Partiality

Restriction Categories [CL02]

Definition

A restriction structure on $\mathcal{C}$ is a mapping $\text{dom} : \mathcal{C}(X, Y) \to \mathcal{C}(X, X)$ with the following properties:

$$f \circ (\text{dom } f) = f \tag{1}$$
$$(\text{dom } f) \circ (\text{dom } g) = (\text{dom } g) \circ (\text{dom } f) \tag{2}$$
$$\text{dom } (g \circ (\text{dom } f)) = (\text{dom } g) \circ (\text{dom } f) \tag{3}$$
$$(\text{dom } h) \circ f = f \circ \text{dom } (h \circ f) \tag{4}$$

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

Intuitively dom f captures the domain of definedness of f.

Capturing Partiality

Restriction Categories [CL02]

Definition

A restriction structure on $\mathcal{C}$ is a mapping $\text{dom} : \mathcal{C}(X, Y) \to \mathcal{C}(X, X)$ with the following properties:

$$f \circ (\text{dom } f) = f \tag{1}$$

$$(\text{dom } f) \circ (\text{dom } g) = (\text{dom } g) \circ (\text{dom } f) \tag{2}$$

$$\text{dom } (g \circ (\text{dom } f)) = (\text{dom } g) \circ (\text{dom } f) \tag{3}$$

$$(\text{dom } h) \circ f = f \circ \text{dom } (h \circ f) \tag{4}$$

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

Intuitively dom f captures the domain of definedness of f.

Remark

Every category has a trivial restriction structure dom $f = id$, we call categories with a non-trivial restriction structure *restriction categories*.

Capturing Partiality

Equational Lifting Monads [BFS03]

The following criterion guarantees that some form of partiality is the only possible side-effect:

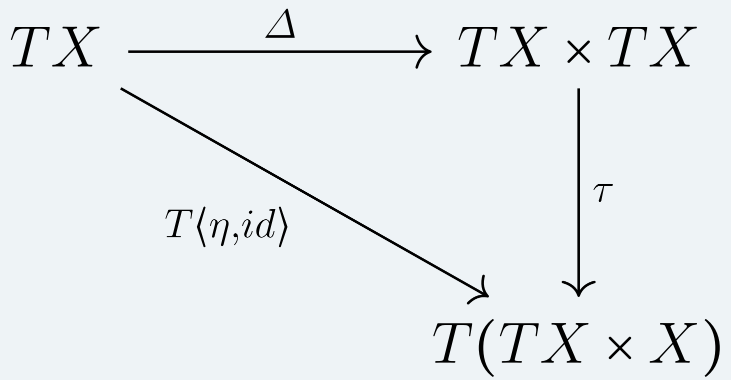
Capturing Partiality

Equational Lifting Monads [BFS03]

The following criterion guarantees that some form of partiality is the only possible side-effect:

Definition

A commutative monad T is called an *equational lifting monad* if the following diagram commutes:



Capturing Partiality

Equational Lifting Monads [BFS03]

The following criterion guarantees that some form of partiality is the only possible side-effect:

Definition

A commutative monad T is called an *equational lifting monad* if the following diagram commutes:

$$\begin{array}{ccc} TX & \xrightarrow{\Delta} & TX \times TX \\ & \searrow^{T\langle \eta, id \rangle} & \downarrow{\tau} \\ & & T(TX \times X) \end{array}$$

Theorem

The Kleisli category of an equational lifting monad is a restriction category.

Capturing Partiality
The Maybe Monad

- $MX = X + 1$

Capturing Partiality

The Maybe Monad

- $MX = X + 1$
- The maybe monad is strong and commutative:

$$\tau_{X,Y} := X \times (Y + 1) \xrightarrow{dstl} (X \times Y) + (X \times 1) \xrightarrow{id+!} (X \times Y) + 1$$

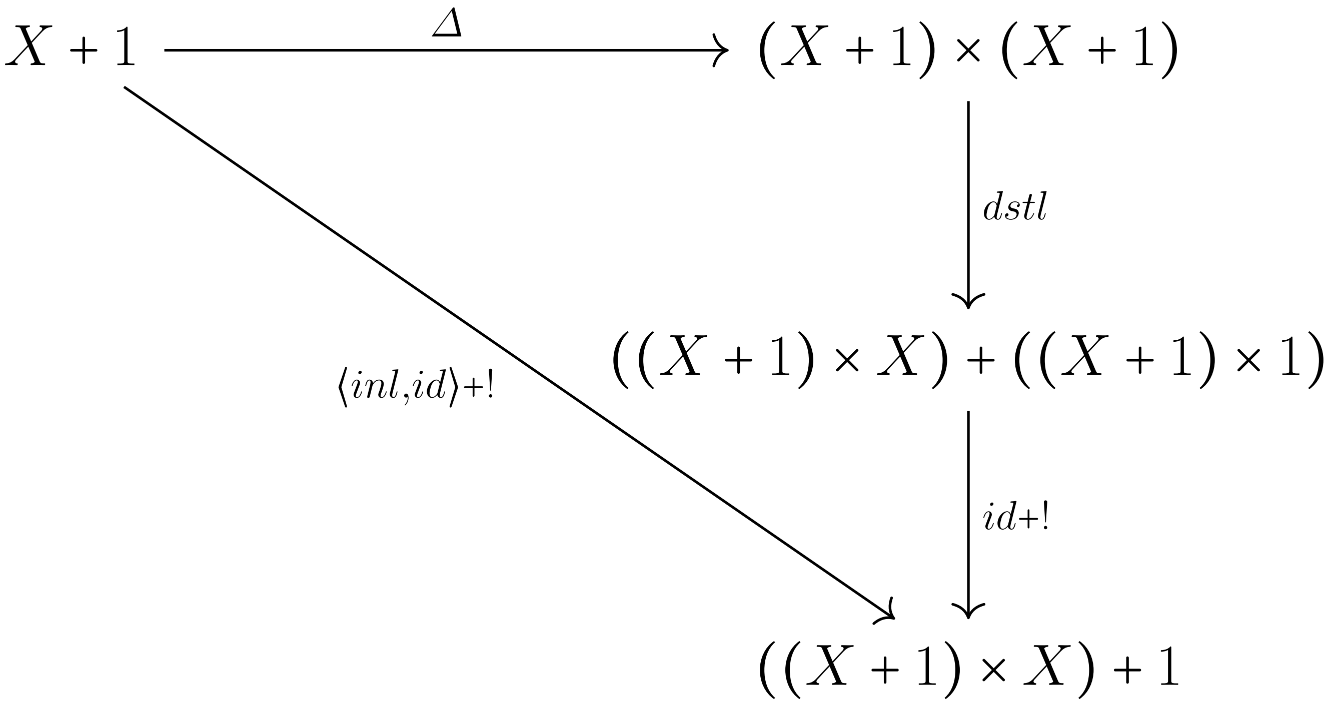
Capturing Partiality

The Maybe Monad

- $MX = X + 1$
- The maybe monad is strong and commutative:

$$\tau_{X,Y} := X \times (Y + 1) \xrightarrow{dstl} (X \times Y) + (X \times 1) \xrightarrow{id+!} (X \times Y) + 1$$

- and the following diagram commutes (i.e. it is an equational lifting monad):



Capturing Partiality

Capretta's Delay Monad [Cap05]

- Recall the delay codatatype:

$$\frac{x : X}{\overline{\overline{\textit{now}\ x : DX}}} \qquad \frac{c : DX}{\overline{\overline{\textit{later}\ c : DX}}}$$

Capturing Partiality

Capretta's Delay Monad [Cap05]

- Recall the delay codatatype:

$$\frac{x : X}{\overline{\overline{\mathit{now}\ x : DX}}} \qquad \frac{c : DX}{\overline{\overline{\mathit{later}\ c : DX}}}$$

- $DX = \nu A.X + A$

Capturing Partiality
Capretta's Delay Monad [Cap05]

- Recall the delay codatatype:

$$\frac{x : X}{\overline{\overline{\mathit{now}\ x : DX}}} \qquad \frac{c : DX}{\overline{\overline{\mathit{later}\ c : DX}}}$$

- $DX = \nu A.X + A$
- By Lambek we get $DX \cong X + DX$ which yields:

$$\begin{aligned} \mathit{out} &: DX \to X + DX \\ \mathit{out}^{-1} &: X + DX \to DX = [\mathit{now}, \mathit{later}] \end{aligned}$$

Capturing Partiality
Capretta's Delay Monad [Cap05]

- Recall the delay codatatype:

$$\frac{x : X}{\overline{\overline{\;now\ x : DX\;}}} \qquad \frac{c : DX}{\overline{\overline{\;later\ c : DX\;}}}$$

- $DX = \nu A.X + A$
- By Lambek we get $DX \cong X + DX$ which yields:

$$\begin{aligned} out &: DX \to X + DX \\ out^{-1} &: X + DX \to DX = [now, later] \end{aligned}$$

- D (if it exists) is a strong and commutative monad

Capturing Partiality

Capretta's Delay Monad [Cap05]

- Recall the delay codatatype:

$$\frac{x : X}{\overline{\overline{now\ x : DX}}} \qquad \frac{c : DX}{\overline{\overline{later\ c : DX}}}$$

- $DX = \nu A.X + A$
- By Lambek we get $DX \cong X + DX$ which yields:

$$\begin{aligned} out &: DX \to X + DX \\ out^{-1} &: X + DX \to DX = [now, later] \end{aligned}$$

- D (if it exists) is a strong and commutative monad
- D is not an equational lifting monad, because besides modelling partiality, it also counts steps (e.g. $now\ c \neq later\ (now\ c)$)

Capturing Partiality

Quotienting the Delay Monad [CUV15]

Following the work by Chapman, Uustalu and Veltri we can quotient D by the 'correct' kind of equality:

$$\frac{p \downarrow c \qquad q \downarrow c}{p \approx q} \qquad\qquad \frac{p \approx q}{later\, p \approx later\, q}$$

Capturing Partiality

Quotienting the Delay Monad [CUV15]

Following the work by Chapman, Uustalu and Veltri we can quotient D by the 'correct' kind of equality:

$$\frac{p \downarrow c \qquad q \downarrow c}{p \approx q} \qquad \frac{p \approx q}{later\ p \approx later\ q}$$

where

$$\frac{}{now\ c \downarrow c} \qquad \frac{x \downarrow c}{later\ x \downarrow c}$$

Capturing Partiality

Quotienting the Delay Monad [CUV15]

Following the work by Chapman, Uustalu and Veltri we can quotient D by the 'correct' kind of equality:

$$\frac{p \downarrow c \qquad q \downarrow c}{p \approx q} \qquad\qquad \frac{p \approx q}{later\ p \approx later\ q}$$

where

$$\frac{}{now\ c \downarrow c} \qquad\qquad \frac{x \downarrow c}{later\ x \downarrow c}$$

we can model this as the coequalizer:

$$D(X \times \mathbb{N}) \underset{D fst}{\overset{\iota^*}{\rightrightarrows}} DX \xrightarrow{\rho_X} D_{\approx}X$$

Capturing Partiality

Quotienting the Delay Monad [CUV15]

Following the work by Chapman, Uustalu and Veltri we can quotient D by the 'correct' kind of equality:

$$\frac{p \downarrow c \qquad q \downarrow c}{p \approx q} \qquad \frac{p \approx q}{\mathit{later}\ p \approx \mathit{later}\ q}$$

where

$$\frac{}{\mathit{now}\ c \downarrow c} \qquad \frac{x \downarrow c}{\mathit{later}\ x \downarrow c}$$

we can model this as the coequalizer:

$$D(X \times \mathbb{N}) \underset{Dfst}{\overset{\iota^*}{\rightrightarrows}} DX \xrightarrow{\rho_X} D_{\approx}X$$

Problem: Defining $\mu_X : D^2_{\approx}X \to D_{\approx}X$ requires countable choice.

Partiality from Iteration
Elgot Algebras

The following is an adaptation of Adámek, Milius and Velebil's *complete Elgot Algebras* [AMV06]:

Definition

A (unguarded) Elgot Algebra [Gon21] consists of:

- An object X
- for every $f : S \to X + S$ the iteration $f^{\#} : S \to X$, satisfying:
  - **Fixpoint**: $f^{\#} = [id, f^{\#}] \circ f$
  - **Uniformity**: $(id + h) \circ f = g \circ h \Rightarrow f^{\#} = g^{\#} \circ h$
    for $f : S \to X + S,\ g : R \to A + R,\ h : S \to R$
  - **Folding**: $((f^{\#} + id) \circ h)^{\#} = [(id + inl) \circ f, inr \circ h]^{\#}$
    for $f : S \to A + S,\ h : R \to S + R$

Partiality from Iteration
Elgot Algebras

The following is an adaptation of Adámek, Milius and Velebil's *complete Elgot Algebras* [AMV06]:

Definition

A (unguarded) Elgot Algebra [Gon21] consists of:

- An object X
- for every $f : S \to X + S$ the iteration $f^\# : S \to X$, satisfying:
  - **Fixpoint**: $f^\# = [id, f^\#] \circ f$
  - **Uniformity**: $(id + h) \circ f = g \circ h \Rightarrow f^\# = g^\# \circ h$
    for $f : S \to X + S,\ g : R \to A + R,\ h : S \to R$
  - **Folding**: $((f^\# + id) \circ h)^\# = [(id + inl) \circ f, inr \circ h]^\#$
    for $f : S \to A + S,\ h : R \to S + R$

Remark

Every Elgot algebra $(A, (-)^\#)$ comes with a divergence constant $\bot = (inr : 1 \to A + 1)^\# : 1 \to A$

Partiality from Iteration

Elgot Monads [AMV11] [GSR14]

Definition

A monad $\mathbf{T}$ is an Elgot monad if it has an iteration operator $(f : X \to T(Y + X))^\dagger : X \to TY$ satisfying:

- **Fixpoint**: $f^\dagger = [\eta, f^\dagger]^* \circ f$
  for $f : X \to T(Y + X)$
- **Uniformity**: $f \circ h = T(id + h) \circ g \Rightarrow f^\dagger \circ h = g^\dagger$
  for $f : X \to T(Y + X), g : Z \to T(Y + Z), h : Z \to X$
- **Naturality**: $g^* \circ f^\dagger = ([(T inl) \circ g, \eta \circ inr]^* \circ f)^\dagger$
  for $f : X \to T(Y + X), g : Y \to TZ$
- **Codiagonal**: $f^{\dagger\dagger} = (T[id, inr] \circ f)^\dagger$
  for $f : X \to T((Y + X) + X)$

Partiality from Iteration

Elgot Monads [AMV11] [GSR14]

Definition

A monad $\mathbf{T}$ is an Elgot monad if it has an iteration operator $(f : X \to T(Y + X))^\dagger : X \to TY$ satisfying:

- **Fixpoint**: $f^\dagger = [\eta, f^\dagger]^* \circ f$
  for $f : X \to T(Y + X)$
- **Uniformity**: $f \circ h = T(id + h) \circ g \Rightarrow f^\dagger \circ h = g^\dagger$
  for $f : X \to T(Y + X), g : Z \to T(Y + Z), h : Z \to X$
- **Naturality**: $g^* \circ f^\dagger = ([(T inl) \circ g, \eta \circ inr]^* \circ f)^\dagger$
  for $f : X \to T(Y + X), g : Y \to TZ$
- **Codiagonal**: $f^{\dagger\dagger} = (T[id, inr] \circ f)^\dagger$
  for $f : X \to T((Y + X) + X)$

Remark

Strong Elgot Monads are regarded as minimal semantic structures for interpreting effectful while-languages.

Partiality from Iteration

pre-Elgot Monads [Gon21]

Relaxing the requirements for Elgot monads we get the following weaker concept:

Definition

A monad $\mathbf{T}$ is called pre-Elgot if every TX extends to an Elgot algebra such that Kleisli lifting is iteration preversing, i.e.

$$h^* \circ f^\# = ((h^* + id) \circ f)^\# \qquad \text{for } f : Z \to TX + Z, h : X \to TY$$

Partiality from Iteration

pre-Elgot Monads [Gon21]

Relaxing the requirements for Elgot monads we get the following weaker concept:

Definition

A monad $\mathbf{T}$ is called pre-Elgot if every TX extends to an Elgot algebra such that Kleisli lifting is iteration preversing, i.e.

$$h^* \circ f^\# = ((h^* + id) \circ f)^\# \qquad \text{for } f : Z \to TX + Z, h : X \to TY$$

Theorem

Every Elgot monad is pre-Elgot

Partiality from iteration

The initial pre-Elgot Monad [Gon21]

- By defining KX as the free Elgot algebra over X we get a monad K (that we assume is stable)

Partiality from iteration

The initial pre-Elgot Monad [Gon21]

- By defining KX as the free Elgot algebra over X we get a monad K (that we assume is stable)
- K is strong and commutative

Partiality from iteration

The initial pre-Elgot Monad [Gon21]

- By defining KX as the free Elgot algebra over X we get a monad K (that we assume is stable)
- K is strong and commutative
- K is an equational lifting monad

Partiality from iteration

The initial pre-Elgot Monad [Gon21]

- By defining KX as the free Elgot algebra over X we get a monad K (that we assume is stable)
- K is strong and commutative
- K is an equational lifting monad
- K is the initial pre-Elgot monad

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $DX \cong X \times \mathbb{N} + 1$

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $DX \cong X \times \mathbb{N} + 1$
  - $D_{\approx}X \cong X + 1$

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $DX \cong X \times \mathbb{N} + 1$
  - $D_{\approx}X \cong X + 1$
  - $X + 1$ is the initial (pre-)Elgot monad ($\Rightarrow K \cong (-) + 1 \cong D_{\approx}$)

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $DX \cong X \times \mathbb{N} + 1$
  - $D_{\approx}X \cong X + 1$
  - $X + 1$ is the initial (pre-)Elgot monad ($\Rightarrow K \cong (-) + 1 \cong D_{\approx}$)
- **Assuming countable choice:**

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $DX \cong X \times \mathbb{N} + 1$
  - $D_{\approx}X \cong X + 1$
  - $X + 1$ is the initial (pre-)Elgot monad ($\Rightarrow K \cong (-) + 1 \cong D_{\approx}$)
- **Assuming countable choice:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide

Partiality from iteration

Closing the gap [Gon21]

Let's look at $\mathbf{K}$ under various assumptions:

- **Assuming excluded middle:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $DX \cong X \times \mathbb{N} + 1$
  - $D_{\approx}X \cong X + 1$
  - $X + 1$ is the initial (pre-)Elgot monad ($\Rightarrow K \cong (-) + 1 \cong D_{\approx}$)
- **Assuming countable choice:**
  - The initial pre-Elgot monad and the initial Elgot monad coincide
  - $D_{\approx}$ is the initial (pre-)Elgot Monad ($\Rightarrow K \cong D_{\approx}$).

1. Partiality in Type Theory
2. Categorical Notions of Partiality
3. **Implementation in Agda**

Goals

- Formalize the delay monad categorically and show that it is..

Goals

- Formalize the delay monad categorically and show that it is..
  - strong

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative
- Formalize K and show that it is..

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative
- Formalize K and show that it is..
  - strong

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative
- Formalize K and show that it is..
  - strong
  - commutative

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative
- Formalize K and show that it is..
  - strong
  - commutative
  - an equational lifting monad

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative
- Formalize K and show that it is..
  - strong
  - commutative
  - an equational lifting monad
  - the initial pre-Elgot monad

Goals

- Formalize the delay monad categorically and show that it is..
  - strong
  - commutative
- Formalize K and show that it is..
  - strong
  - commutative
  - an equational lifting monad
  - the initial pre-Elgot monad
- Take the category of setoids and show that K instantiates to $D_{\approx}$

Bibliography I

[AMV06] J. Adámek, S. Milius, and J. Velebil. "Elgot Algebras". In: *CoRR* abs/cs/0609040 (2006). URL: http://arxiv.org/abs/cs/0609040.

[AMV11] J. Adámek, S. Milius, and J. Velebil. "Elgot theories: a new perspective on the equational properties of iteration". In: *Mathematical Structures in Computer Science* 21.2 (2011), pp. 417–480. DOI: 10.1017/S0960129510000496.

[BFS03] A. Bucalo, C. Führmann, and A. Simpson. "An Equational Notion of Lifting Monad". In: *Theor. Comput. Sci.* 294.1–2 (Feb. 2003), pp. 31–60. DOI: 10.1016/S0304-3975(01)00243-2. URL: https://doi.org/10.1016/S0304-3975(01)00243-2.

[Cap05] V. Capretta. "General Recursion via Coinductive Types". In: *CoRR* abs/cs/0505037 (2005). URL: http://arxiv.org/abs/cs/0505037.

Bibliography II

[CL02] J. R. B. Cockett and S. Lack. "Restriction Categories I: Categories of Partial Maps". In: *Theor. Comput. Sci.* 270.1–2 (Jan. 2002), pp. 223–259. DOI: 10.1016/S0304-3975(00)00382-0. URL: https://doi.org/10.1016/S0304-3975(00)00382-0.

[CUV15] 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. DOI: 10.1007/978-3-319-25150-9_8. URL: https://doi.org/10.1007/978-3-319-25150-9_8.

[Gon21] S. Goncharov. "Uniform Elgot Iteration in Foundations". In: *CoRR* abs/2102.11828 (2021). URL: https://arxiv.org/abs/2102.11828.

[GSR14] S. Goncharov, L. Schröder, and C. Rauch. "(Co-)Algebraic Foundations for Effect Handling and Iteration". In: *CoRR* abs/1405.0854 (2014). URL: http://arxiv.org/abs/1405.0854.

Bibliography III

[Mog91] E. Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (July 1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4.

1. Partiality in Type Theory
2. Categorical Notions of Partiality
3. Implementation in Agda

App: Category Theory in Agda

Setoid-enriched Categories

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

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

  field
    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
    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
```