

# Totality versus Turing-Completeness?

Conor McBride

University of Strathclyde  
conor@strictlypositive.org

**Abstract.** In this literate Agda paper, I show that general recursive definitions can be represented in the free monad which supports the ‘effect’ of making a recursive call, without saying how these programs should be executed. Diverse semantics can be given by suitable monad morphisms. The Bove-Capretta construction of the domain of a general recursive function can be presented datatype-generically as an instance of this technique.

## 1 Introduction

Advocates of Total Functional Programming [17], such as myself, can prove prone to a false confession, namely that the price of functions which function is the loss of Turing-completeness. In a total language, to construct  $f : S \rightarrow T$  is to promise a canonical  $T$  eventually, given a canonical  $S$ . The alleged benefit of general recursion is just to inhibit such strong promises. To make a weaker promise, simply construct a total function of type  $S \rightarrow G T$  where  $G$  is a suitable monad.

The literature and lore of our discipline are littered with candidates for  $G$ , and this article will contribute another—the *free* monad with one operation  $f : S \rightarrow T$ . To work in such a monad is to *write* a general recursive function without prejudice as to how it might be *executed*. We are then free, in the technical sense, to choose any semantics for general recursion we like by giving a suitable *monad morphism* to another notion of partial computation. For example, Venanzio Capretta’s partiality monad [10], also known as the *completely iterative* monad on the operation  $yield : 1 \rightarrow 1$ , which might never deliver a value, but periodically offers its environment the choice of whether to interrupt computation or to continue.

Meanwhile, Ana Bove gave, with Capretta, a method for defining the *domain predicate* of a general recursive function simultaneously with the delivery of a value for every input satisfying that domain predicate [8]. Their technique gives a paradigmatic example of defining a datatype and its interpretation by *induction-recursion* in the sense of Peter Dybjer and Anton Setzer [11, 12]. Dybjer and Setzer further gave a coding scheme which renders first class the characterising data for inductive-recursive definitions. In this article, I show how to compute from the free monadic presentation of a general recursive function the code for its domain predicate. By doing so, I implement the Bove-Capretta method once for all, systematically delivering (but not, of course, discharging) the proof obligation required to strengthen the promise from partial  $f : S \rightarrow G T$  to the total  $f : S \rightarrow T$ .

Total functional languages remain *logically* incomplete in the sense of Gödel. There are termination proof obligations which we can formulate but not discharge within any given total language, even though the relevant programs—notably the language’s own evaluator—are total. Translated across the Curry-Howard correspondence, the argument for general recursion asserts that logical inconsistency is a price worth paying for logical completeness, notwithstanding the loss of the language’s value as *evidence*. Programmers are free to maintain that such dishonesty is essential to their capacity to earn a living, but a new generation of programming technology enables some of us to offer and deliver a higher standard of guarantee. *Faites vos jeux!*

## 2 The General Free Monad

Working (<http://github.com/pigworker/Totality>), in Agda, we may define a free monad which is general, both in the sense of being generated by any strictly positive functor, and in the sense of being suited to the modelling of general recursion.

```

data General (S : Set) (T : S → Set) (X : Set) : Set where
  !!      : X → General S T X
  _??_   : (s : S) → (T s → General S T X) → General S T X
infixr 5 _??_

```

At each step, we either output an  $X$ , or we make the request  $s ?? k$ , for some  $s : S$ , where  $k$  explains how to continue once a response in  $T s$  has been received. That is, values in  $\text{General } S T X$  are request-response trees with  $X$ -values at the leaves; each internal node is labelled by a request and branches over the possible meaningful responses. The key idea in this paper is to represent recursive calls as just such request-response interactions, and recursive definitions by just such trees.

$\text{General}$  datatypes come with a catamorphism, or ‘fold’ operator.<sup>1</sup>

```

fold : ∀ {l S T X} {Y : Set l} →
  (X → Y) → ((s : S) → (T s → Y) → Y) →
  General S T X → Y
fold r c (!! x)    = r x
fold r c (s ?? k) = c s λ t → fold r c (k t)

```

The ‘bind’ operation for the monad  $\text{General } S T$  substitutes computations for values to build larger computations. It is, of course, a **fold**.

```

_>>=>_c_ : ∀ {S T X Y} →
  General S T X → (X → General S T Y) → General S T Y
g >>=>_c_ k = fold k _??_ g
infixl 4 _>>=>_c_

```

We then acquire what Gordon Plotkin and John Power refer to as a *generic effect* [16]—the presentation of an individual request-response interaction:

```

call : ∀ {S T} (s : S) → General S T (T s)
call s = s ?? !!

```

Now we may say how to give a recursive definition for a function. For each argument  $s : S$ , we must build a request-response tree from individual **calls**, ultimately delivering a value in  $T s$ . We may thus define the ‘general recursive  $\Pi$ -type’,

```

PiG : (S : Set) (T : S → Set) → Set
PiG S T = (s : S) → General S T (T s)

```

to be a type of functions delivering the recursive *strategy* for computing a  $T s$  from some  $s : S$ .

For example, given the natural numbers,

```

data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

```

the following obfuscated identity function will not pass Agda’s syntactic check for guardedness of recursion.

<sup>1</sup> Whenever I intend a monoidal accumulation, I say ‘crush’, not ‘fold’.

```

fusc : Nat → Nat
fusc zero = zero
fusc (suc n) = suc (fusc (fusc n))

```

However, we can represent its definition without such controversy.

```

fusc : PiG Nat λ _ → Nat
fusc zero = !! zero
fusc (suc n) = call n >>=G λ fn → call fn >>=G λ ffn → !! (suc ffn)

```

Each `call` is only a *placeholder* for a recursive call to `fusc`. The latter tells us just how to expand the recursion *once*. Note that `fusc`'s *nested* recursive calls make use of the way `>>=G` allows values from earlier effects to influence the choice of later effects. Using only a free applicative functor would exactly exclude nested recursion.

Even so, it is fair to object that the ‘monadified’ definition is ugly compared to its direct but not obviously terminating counterpart, with more intermediate naming. Monadic programming is ugly in general, not just in `General!` Languages like Bauer and Pretnar’s *Eff* [6] show us that we can solve this problem, working in direct style for whatever effectful interface is locally available, but meaning the computation delivered by the appropriate Moggi-style translation into an explicitly monadic kernel [15]. There is no need to consider monadic style a just punishment, whatever your impurity.

By choosing the `General` monad, we have not committed to any notion of ‘infinite computation’. Rather, we are free to work with a variety of monads  $M$  which might represent the execution of a general recursive function, by giving a *monad morphism* from `General S T` to  $M$ , mapping each request to something which tries to deliver its response. Correspondingly, we shall need to define these concepts more formally.

### 3 Monads and Monad Morphisms, More or Less

This section is a formalisation of material which is largely standard. The reader familiar with monad morphisms should feel free to skim for notation without fear of missing significant developments.

Let us introduce the notion of a `Kleisli` structure on sets, as Altenkirch and Reus called it, known to Altenkirch, Chapman and Uustalu as a ‘relative’ monad [5, 4].

```

record Kleisli { i j } (M : Set i → Set j) : Set (lsuc (i ⊔ j)) where
  field
    return : ∀ { X } → X → M X
    _>>=_ : ∀ { A B } → M A → (A → M B) → M B
    _◇_ : ∀ { A B C : Set i } →
      (B → M C) → (A → M B) → (A → M C)
    (f ◇ g) a = g a >>= f
  infixl 4 _>>=_ _◇_

```

Although the ‘notion of computation’ is given by a mapping on value sets, that mapping need not be an *endofunctor*. We shall later find use for this flexibility when we interpret small computations as large descriptions of datatypes. The upshot is that we are obliged to work polymorphically in our set-theoretic magnitude. Given the fields `return` and `>>=`, we may equip ourselves with Kleisli composition in the usual way, replacing each value emerging from  $g$  with the computation indicated by  $f$ . Of course, we have

```

GeneralK : ∀ { S T } → Kleisli (General S T)
GeneralK = record { return = !!; _>>=_ = _>>=G_ }

```

The ‘Monad laws’ amount to requiring that `return` and `◇` give us a category.

```

record KleisliLaws { i j } { M : Set i → Set j } (KM : Kleisli M)
  : Set (Isuc (i ⊔ j)) where
open Kleisli KM
field
  .idLeft  : ∀ { A B } (g : A → M B) → return ◇ g ≡ g
  .idRight : ∀ { A B } (f : A → M B) → f ◇ return ≡ f
  .assoc   : ∀ { A B C D }
              (f : C → M D) (g : B → M C) (h : A → M B) →
              (f ◇ g) ◇ h ≡ f ◇ (g ◇ h)

```

The dots before the field names make those fields unavailable for computational purposes. Correspondingly, I have little compunction about postulating an extensional equality and reasoning by transforming functions.

```

postulate
  .ext : ∀ { i j } { A : Set i } { B : A → Set j } { f g : (a : A) → B a } →
        ((a : A) → f a ≡ g a) → f ≡ g

```

In order to improve the readability of proofs, I expose the reflexivity, symmetry and transitivity of equality in a way that lets us show our steps.

```

_=[_]=_ : ∀ { l } { X : Set l } (x : X) { y z } → x ≡ y → y ≡ z → x ≡ z
x =[ refl ] = q = q
_=<[_]=_ : ∀ { l } { X : Set l } (x : X) { y z } → y ≡ x → y ≡ z → x ≡ z
x =< refl ] = q = q
_□_ : ∀ { l } { X : Set l } (x : X) → x ≡ x
x □ = refl
infixr 2 _□ _=[_]=_ _=<[_]=_

```

I also make use of the way applicative forms respect equality.

```

[ ] : ∀ { l } { X : Set l } (x : X) → x ≡ x
[ x ] = refl
_-$-_ : ∀ { i j } { S : Set i } { T : Set j } { f g : S → T } { x y : S } →
        f ≡ g → x ≡ y → f x ≡ g y
refl -$ refl = refl
infixl 9 _-$-_

```

E.g., we may show that the usual law for iterating `>>=` is basically associativity.

```

.binds : ∀ { A B C } (a : M A) (f : B → M C) (g : A → M B) →
        a >>= (f ◇ g) ≡ a >>= g >>= f
binds a f g = assoc f g (const a) -$ [ < ]

```

Let us warm up to the proofs of the `KleisliLaws` with some basic properties of `fold`. Firstly, anything satisfying the defining equations of a `fold` is a `fold`.

```

.foldUnique : ∀ { l S T X } { Y : Set l } (f : General S T X → Y) r c →
  (∀ x → f (!! x) ≡ r x) → (∀ s k → f (s ?? k) ≡ c s (f · k)) →
  f ≡ fold r c
foldUnique f r c rq cq = ext help where
  help : (g : _) → _
  help (!! x) = f (!! x) =[ rq x ] = r x □

```

$$\begin{aligned}
\text{help } (s \text{ ?? } k) &= f (s \text{ ?? } k) && = [c q s k] = \\
& c s (f \cdot k) && = [c s] \text{ ext } (\lambda t \rightarrow \text{help } (k t)) = \\
& c s (\text{fold } r c \cdot k) \quad \square
\end{aligned}$$

An immediate consequence is that **fold**-ing the constructors gives the identity.

$$\begin{aligned}
.&\text{foldId} : \forall \{S T X\} \rightarrow \text{fold} !! \text{ ?? } _ \equiv \text{id} \{ \_ \} \{ \text{General } S T X \} \\
\text{foldId} &= \text{fold} !! \text{ ?? } _ = \langle \text{foldUnique id} !! \text{ ?? } _ (\lambda \_ \rightarrow \text{refl}) (\lambda \_ \_ \rightarrow \text{refl}) \rangle = \\
&\text{id} \quad \square
\end{aligned}$$

With a further induction, we can establish a fusion law for **fold** after  $\gg\equiv$ .

$$\begin{aligned}
.&\text{foldFusion} : \forall \{l S T X Y\} \{Z : \text{Set } l\} \\
&(r : Y \rightarrow Z) (c : (s : S) \rightarrow (T s \rightarrow Z) \rightarrow Z) (f : X \rightarrow \text{General } S T Y) \rightarrow \\
&(\text{fold } r c \cdot \text{fold } f \text{ ?? } _ ) \equiv \text{fold} (\text{fold } r c \cdot f) c \\
\text{foldFusion } r c f &= \text{ext help where} \\
\text{help} : (g : \_ ) \rightarrow \_ & \\
\text{help} (!! x) &= \text{refl} \\
\text{help} (s \text{ ?? } k) &= \\
&c s (\text{fold } r c \cdot \text{fold } f \text{ ?? } _ \cdot k) = [c s] \text{ ext } (\lambda t \rightarrow \text{help } (k t)) = \\
&c s (\text{fold} (\text{fold } r c \cdot f) c \cdot k) \quad \square
\end{aligned}$$

That is enough to establish the **KleisliLaws** for **GeneralK**.

$$\begin{aligned}
.&\text{GeneralKLaws} : \forall \{S T\} \rightarrow \text{KleisliLaws} (\text{GeneralK} \{S\} \{T\}) \\
\text{GeneralKLaws} &= \text{record} \\
&\{ \text{idLeft} = \lambda g \rightarrow [(\lambda k \rightarrow k \cdot g)] \text{ ext foldId}; \text{idRight} = \lambda \_ \rightarrow \text{refl} \\
&; \text{assoc} = \lambda f g h \rightarrow \\
&\quad (f \diamond g) \diamond h = [(\lambda k \rightarrow k \cdot h)] \text{ ext foldFusion } f \text{ ?? } _ g = \\
&\quad f \diamond (g \diamond h) \quad \square \\
&\} \text{ where open Kleisli GeneralK}
\end{aligned}$$

Now, let us consider when a polymorphic function  $m : \forall \{X\} \rightarrow M X \rightarrow N X$  is a *monad morphism* in this setting. Given **Kleisli**  $M$  and **Kleisli**  $N$ ,  $m \cdot -$  should map **return** and  $\diamond$  from  $M$  to  $N$ .

$$\begin{aligned}
\text{record Morphism } \{i j k\} \{M : \text{Set } i \rightarrow \text{Set } j\} \{N : \text{Set } i \rightarrow \text{Set } k\} & \\
& (KM : \text{Kleisli } M) (KN : \text{Kleisli } N) \\
(m : \forall \{X\} \rightarrow M X \rightarrow N X) : \text{Set } (\text{Isuc } (i \sqcup j \sqcup k)) \text{ where} & \\
\text{module } \_M = \text{Kleisli } KM; \text{module } \_N = \text{Kleisli } KN & \\
\text{field} & \\
.&\text{respl} : \{X : \text{Set } i\} \rightarrow \\
&\quad m \cdot \text{return}_M \{X\} \equiv \text{return}_N \{X\} \\
.&\text{respC} : \{A B C : \text{Set } i\} (f : B \rightarrow M C) (g : A \rightarrow M B) \rightarrow \\
&\quad m \cdot (f \diamond_M g) \equiv (m \cdot f) \diamond_N (m \cdot g)
\end{aligned}$$

The proofs, **idMorph** and **compMorph**, that monad morphisms are closed under identity and composition, are left as straightforward exercises for the reader.

Now, **General**  $S T$  is the free monad on the functor  $\Sigma S \lambda s \rightarrow T s \rightarrow -$  which captures a single request-response interaction. It is a free construction, turning functors into monads, in the sense that it is left adjoint to the forgetful map which turns monads back into functors. In other words, the monad morphisms from a free monad to  $M$  are exactly given by the polymorphic functions from the underlying functor to  $M$ . In our case, the monad morphisms

$$m : \forall \{X\} \rightarrow \text{General } S T X \rightarrow M X$$

are given exactly by the functions of type

$$\begin{aligned} & \forall \{X\} \rightarrow (\sum S \lambda s \rightarrow T s \rightarrow X) \rightarrow M X \cong \\ & (s : S) \rightarrow \forall \{X\} \rightarrow (T s \rightarrow X) \rightarrow M X \cong (s : S) \rightarrow M (T s) \end{aligned}$$

That is, the monad morphisms from `General S T` to `M` are exactly given by the ‘`M`-acting versions’ of our function.

```
morph : ∀ {l S T} {M : Set → Set l} (KM : Kleisli M)
      (h : (s : S) → M (T s))
      {X} → General S T X → M X
morph KM h = fold return (>>=ₘ · h) where open Kleisli KM
```

Let us show that `morph` makes `Morphisms`.

```
morphMorphism : ∀ {l S T} {M : Set → Set l}
  (KM : Kleisli M) (KLM : KleisliLaws KM) →
  (h : (s : S) → M (T s)) →
  Morphism (GeneralK {S} {T}) KM (morph KM h)
morphMorphism {-} {-} {-} KM KLM h =
  let module _G = Kleisli (GeneralK {S} {T})
      module _M = Kleisli KM; open KleisliLaws KLM
  in record
    { respl = refl
    ; respC = λ f g → morph KM h · (f ◊_G g) = [ refl ] =
              fold return_M (>>=ₘ · h) · fold f _??_ · g
              = [ (λ k → k · g) ] = (
```

Expanding `◊_G` and focusing our attention before the `· g`, we find a fusion opportunity.

```
fold return_M (>>=ₘ · h) · fold f _??_
= [ foldFusion return_M (>>=ₘ · h) f ] =
fold (morph KM h · f) (>>=ₘ · h)
= ( morphFusion KM KLM (morph KM h · f) h ) =
(morph KM h · f) ◊_M morph KM h □
```

We find that `foldFusion` leaves us with an operation which is almost the definition `morph KM h`, except that where we want `fold return_M`, we have `fold` of something else which we ought to be able to move after the `fold` by another fusion law, to be established forthwith. Meanwhile, plugging the `· g` back on the right, we are done.

```
) = (morph KM h · f) ◊_M (morph KM h · g) □}
```

The lemma we need allows us to fuse any `f ◊_M morph KM h` into a single `fold`.

```
.morphFusion : ∀ {l S T X Y}
  {M : Set → Set l} (KM : Kleisli M) (KLM : KleisliLaws KM)
  (f : X → M Y) (h : (s : S) → M (T s)) →
  let open Kleisli KM in
    f ◊ morph KM h ≡ fold {-} {-} {-} f (>>=ₘ · h)
morphFusion KM KLM f h = ext help where
  open Kleisli KM; open KleisliLaws KLM
  help : (g : _) → _
  help (!! x) = (f ◊ return) x = [ idRight f = [ x ] ] = f x □
  help (s ?? k) = (f ◊ morph KM h) (s ?? k)
                = [ refl ] = h s >>= (morph KM h · k) >>= f
```

```

=< binds (h s) f (morph KM h · k) >=
  h s >>= (f ◊ (morph KM h · k))
=< [ _>>= _ (h s) ] ≐ ext (λ t → help (k t)) >=
  fold f (_>>= _ · h) (s ?? k) ◻

```

Let us check that `morph` give us the *only* monad morphisms from `General S T`, using the uniqueness of `fold`.

```

.morphOnly : ∀ {l S T}
  {M : Set → Set l} (KM : Kleisli M) (KLM : KleisliLaws KM) →
  (m : {X : Set} → General S T X → M X) → Morphism GeneralK KM m →
  {X : Set} → m {X} ≐ morph KM (m · call) {X}
morphOnly KM KLM m mm = foldUnique m return_M (_>>= _ · m · call)
(λ x → m (!! x) ≐ [ resp!_m ≐ [ x ] ] = return_M x ◻)
(λ s k → m (s ?? k) ≐ [ refl ] =
  (m · (k ◊_G const (call s))) ◊ ≐ [ respC_m k (const (call s)) ≐ [ ◊ ] ] =
  m (call s) >>= _ (m · k) ◻)

```

where

```

module _G = Kleisli GeneralK
module _M = Kleisli KM; open KleisliLaws KLM
module _m = Morphism mm

```

## 4 General Recursion with the General Monad

`General` strategies are finite: they tell us how to expand one request in terms of a bounded number recursive calls. The operation which expands each such request is a monad endomorphism—exactly the one generated by our `f : PiG S T` itself, replacing each `call s` node in the tree by the whole tree given by `f s`.

```

expand : ∀ {S T X} → PiG S T → General S T X → General S T X
expand f = morph GeneralK f

```

You will have noticed that `call : PiG S T`, and that `expand call` just replaces one request with another, acting as the identity. As a recursive strategy, taking `f = λ s → call s` amounts to the often valid but seldom helpful ‘definition’:

$$f s = f s$$

By way of example, let us consider the evolution of state machines. We shall need Boolean values:

```

data Bool : Set where tt ff : Bool
if_then_else_ : {X : Set} → Bool → X → X → X
if tt then t else f = t
if ff then t else f = f

```

Now let us construct the method for computing the halting state of a machine, given its initial state and its one-step transition function.

```

halting : ∀ {S} → (S → Bool) → (S → S) → PiG S λ _ → S
halting stop step start with stop start
... | tt = !! start
... | ff = call (step start)

```

For Turing machines,  $S$  should pair a machine state with a tape,  $stop$  should check if the machine state is halting, and  $step$  should look up the current state and tape-symbol in the machine description then return the next state and tape. We can clearly explain how any old Turing machine computes without stepping beyond the confines of total programming, and without making any rash promises about what values such a computation might deliver.

## 5 The Petrol-Driven Semantics

It is one thing to describe a general-recursive computation but quite another to perform it. A simple way to give an arbitrary total approximation to partial computation is to provide an engine which consumes one unit of petrol for each recursive call it performs, then specify the initial fuel supply. The resulting program is primitive recursive, but makes no promise to deliver a value. Let us construct it as a monad morphism. We shall need the usual model of *finite* failure, allowing us to give up when we are out of fuel.

```
data Maybe (X : Set) : Set where
  yes : X → Maybe X
  no  : Maybe X
```

`Maybe` is monadic in the usual failure-propagating way.

```
MaybeK : Kleisli Maybe
MaybeK = record { return = yes
                  ; _>>_ = λ { (yes a) k → k a; no k → no } }
```

The proof `MaybeKL : KleisliLaws MaybeK` is a matter of elementary case analysis, so let us not dwell on it.

We may directly construct the monad morphism which executes a general recursion impatiently.

```
already : ∀ {S T X} → General S T X → Maybe X
already = morph MaybeK λ s → no
```

That is, `!!` becomes `yes` and `??` becomes `no`, so the recursion delivers a value only if it has terminated already. Now, if we have some petrol, we can run an `engine` which `expands` the recursion for a while, beforehand.

```
engine : ∀ {S T} (f : PiG S T) (n : Nat) {X} → General S T X → General S T X
engine f zero    = id
engine f (suc n) = engine f n · expand f
```

We obtain the petrol-driven (or step-indexed, if you prefer) semantics by composition.

```
petrol : ∀ {S T} → PiG S T → Nat → (s : S) → Maybe (T s)
petrol f n = already · engine f n · f
```

If we consider `Nat` with the usual order and `Maybe X` ordered by `no < yes x`, we can readily check that `petrol f n s` is monotone in  $n$ : supplying more fuel can only (but sadly not strictly) increase the risk of successfully delivering output.

An amusing possibility in a system such as Agda, supporting the partial evaluation of incomplete expressions, is to invoke `petrol` with `?` as the quantity of fuel. We are free to refine the `?` with `suc ?` and resume evaluation repeatedly for as long as

we are willing to wait in expectation of a **yes**. Whilst this may be a clunky way to signal continuing consent for execution, compared to the simple maintenance of the electricity supply, it certainly simulates the conventional experience of executing a general recursive program.

What, then, is the substance of the often repeated claim that a total language capable of this construction is not Turing-complete? Just this: there is more to delivering the run time execution semantics of programs than the pure evaluation of expressions. The *language* might thus be described as Turing-incomplete, even though the *system* by which you use it allows you to execute arbitrary recursive computations for as long as you are willing to tolerate. Such a pedantic quibble deserves to be taken seriously inasmuch as it speaks against casually classifying a *language* as Turing-complete or otherwise, without clarifying the variety of its semanticises and the relationships between them.

Whilst we are discussing the semanticises of total languages, it is worth remembering that we expect dependently typed languages to come with at least *two*: a run time execution semantics which computes only with closed terms, and an evaluation semantics which the typechecker applies to open terms. It is quite normal for general recursive languages to have a total typechecking algorithm.

## 6 Capretta’s Coinductive Semantics, via Abel and Chapman

Coinduction in dependent type theory remains a vexed issue: we are gradually making progress towards a presentation of productive programming for infinite data structures, but we can certainly not claim that we have a presentation which combines honesty, convenience and compositionality. The state of the art is the current Agda account due to Andreas Abel and colleagues, based on the notion of *copatterns* [3] which allow us to define lazy data by specifying observations of them, and on *sized types* [1] which give a more flexible semantic account of productivity at the cost of additional indexing.

Abel and Chapman [2] give a development of normalization for simply typed  $\lambda$ -calculus, using Capretta’s **Delay** monad [10] as a showcase for copatterns and sized types. I will follow their setup, then construct a monad morphism from **General**. The essence of their method is to define **Delay** as the data type of *observations* of lazy computations, mutually with the record type, **Delay**<sup>∞</sup>, of those lazy computations themselves.

```

mutual
  data Delay (i : Size) (X : Set) : Set where
    now : X → Delay i X
    later : Delay∞ i X → Delay i X
  record Delay∞ (i : Size) (X : Set) : Set where
    coinductive; constructor (↪)
    field force : {j : Size < i} → Delay j X
open Delay∞

```

Abel explains that **Size**, here, characterizes the *observation depth* to which one may iteratively **force** the lazy computation. Corecursive calls must reduce this depth, so cannot be used for the topmost observation. Pleasingly, they need not be rigidly guarded by constructors, because their sized types document their legitimate use. For example, we may define the *anamorphism*, or **unfold**, constructing a **Delay** *X* from a coalgebra for the underlying functor  $X + -$ .

```

data _+_ (S T : Set) : Set where
  inl : S → S + T
  inr : T → S + T

```

```

[-, -] : {S T X : Set} → (S → X) → (T → X) → S + T → X
[f, g] (inl s) = f s
[f, g] (inr t) = g t

```

**mutual**

```

unfold : ∀ {i X Y} → (Y → X + Y) → Y → Delay i X
unfold f y = [now, later · unfold∞ f] (f y)
unfold∞ : ∀ {i X Y} → (Y → X + Y) → Y → Delay∞ i X
force (unfold∞ f y) = unfold f y

```

Based on projection, copatterns favours products over sum, which is why most of the motivating examples are based on streams. As soon as we have a choice, mutual recursion becomes hard to avoid. Thus equipped, we can build a `Delay X` value by stepping a computation which can choose either to deliver an `X` or to continue.

Capretta explored the use of `Delay` as a monad to model general recursion, with the `>>=` operator concatenating sequences of `later`s. By way of example, he gives an interpretation of the classic language with an operator seeking the minimum number satisfying a test. Let us therefore equip `Delay` with a `>>=` operator. It can be given as an `unfold`, but the direct definition with sized types is more straightforward. Abel and Chapman give us the following definition.

**mutual**

```

->>=_{D}- : ∀ {i A B} →
  Delay i A → (A → Delay i B) → Delay i B
now a >>=_{D} f = f a
later a' >>=_{D} f = later (a' >>=_{D} f)
->>=_{D}^∞- : ∀ {i A B} →
  Delay∞ i A → (A → Delay i B) → Delay∞ i B
force (a' >>=_{D}^∞ f) = force a' >>=_{D} f

```

and hence our purpose will be served by taking

```

DelayK : {i : Size} → Kleisli (Delay i)
DelayK = record {return = now; ->>=- = ->>=_{D}-}

```

Abel and Chapman go further and demonstrate that these definitions satisfy the monad laws up to strong bisimilarity, which is the appropriate notion of equality for coinductive data but sadly not the propositional equality which Agda makes available. I shall not recapitulate their proof.

It is worth noting that the `Delay` monad is an example of a *completely iterative* monad, a final coalgebra  $\nu Y. X + F Y$ , where the free monad, `General`, is an initial algebra [14]. For `Delay`, take  $F Y = Y$ , or isomorphically,  $F Y = \mathbf{1} \times \mathbf{1} \rightarrow Y$ , representing a trivial request-response interaction. That is `Delay` represents processes which must always eventually *yield*, allowing their environment the choice of whether or not to resume them. We have at least promised to obey control-C!

By way of connecting the Capretta semantics with the petrol-driven variety, we may equip every `Delay` process with a monotonic `engine`.

```

engine : Nat → ∀ {X} → Delay _ X → Maybe X
engine _ (now x) = yes x
engine zero (later _) = no
engine (suc n) (later d) = engine n (force d)

```

Note that `engine n` is not a monad morphism unless `n` is `zero`.

```

engine 1 (later ⟨now tt⟩ >>= λ v → later ⟨now v⟩) = no
engine 1 (later ⟨now tt⟩) >>= λ v → engine 1 (later ⟨now v⟩) = yes tt

```

Meanwhile, given a petrol-driven process, we can just keep trying more and more fuel. This is one easy way to write the minimization operator.

```

tryMorePetrol : ∀ {i X} → (Nat → Maybe X) → Delay i X
tryMorePetrol {-} {X} f = unfold try zero where
  try : Nat → X + Nat
  try n with f n
  ... | yes x = inl x
  ... | no    = inr (suc n)
minimize : (Nat → Bool) → Delay _ Nat
minimize test = tryMorePetrol λ n → if test n then yes n else no

```

Our request-response characterization of general recursion is readily mapped onto `Delay`. Sized types allow us to give the monad morphism directly, corecursively interpreting each recursive `call`.

```

mutual
  delay : ∀ {i S T} (f : PiG S T) {X} → General S T X → Delay i X
  delay f = morph DelayK λ s → later (delay∞ f (f s))
  delay∞ : ∀ {i S T} (f : PiG S T) {X} → General S T X → Delay∞ i X
  force (delay∞ f g) = delay f g

```

We can now transform our `General` functions into their coinductive counterparts.

```

lazy : ∀ {S T} → PiG S T → (s : S) → Delay _ (T s)
lazy f = delay f · f

```

## 7 A Little $\lambda$ -Calculus

By way of a worked example, let us implement the untyped  $\lambda$ -calculus. We can equip ourselves with de Bruijn-indexed terms in the usual way. I have taken the liberty of parametrizing these terms by a type of inert constants `X`

```

data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)
data  $\wedge$  (X : Set) (n : Nat) : Set where
   $\kappa$  : X →  $\wedge$  X n
  #    : Fin n →  $\wedge$  X n
   $\lambda$  :  $\wedge$  X (suc n) →  $\wedge$  X n
   $\_$ $ $\_$  :  $\wedge$  X n →  $\wedge$  X n →  $\wedge$  X n
infixl 5  $\_$ $ $\_$ 

```

In order to evaluate terms, we shall need a suitable notion of environment. Let us make sure they have the correct size to enable projection.

```

data Vec (X : Set) : Nat → Set where
   $\langle$  : Vec X zero
   $\_$ ← : {n : Nat} → Vec X n → X → Vec X (suc n)
proj : ∀ {X n} → Vec X n → Fin n → X
proj ( $\_$ ← x) zero = x
proj ( $\gamma$ ←  $\_$ ) (suc n) = proj  $\gamma$  n

```

Correspondingly, a *value* is either a constant applied to other values, or a function which has got stuck for want of its argument.

```
data Val (X : Set) : Set where
  κ : X → {n : Nat} → Vec (Val X) n → Val X
  λ : {n : Nat} → Vec (Val X) n → Λ X (suc n) → Val X
```

Now, in general, we will need to evaluate *closures*—open terms in environments.

```
data Closure (X : Set) : Set where
  _⊢_ : {n : Nat} → Vec (Val X) n → Λ X n → Closure X
infix 4 _⊢_
```

We can now give the evaluator,  $\llbracket \_ \rrbracket$  as a **General** recursive strategy to compute a value from a closure. Application is the fun case. When evaluating the argument and the function—subterms of the application—we may use  $\llbracket \_ \rrbracket$  itself, rather than **call**. However, when a  $\beta$ -redex starts a further evaluation, **call** is called for.

```
 $\llbracket \_ \rrbracket$  : {X : Set} → PiG (Closure X) λ _ → Val X
 $\llbracket \gamma \vdash \kappa x \rrbracket$  = !! (κ x ⟨⟩)
 $\llbracket \gamma \vdash \# i \rrbracket$  = !! (proj γ i)
 $\llbracket \gamma \vdash \lambda b \rrbracket$  = !! (λ γ b)
 $\llbracket \gamma \vdash f \$ s \rrbracket$  =
  {  $\llbracket \gamma \vdash s \rrbracket \ggg_{\mathbf{G}} \lambda v \rightarrow \llbracket \gamma \vdash f \rrbracket \ggg_{\mathbf{G}} \lambda \{$ 
    (κ x vs) → !! (κ x (vs, v)) ;
    (λ δ b) → call (δ, v ⊢ b) } }
```

Thus equipped, **lazy**  $\llbracket \_ \rrbracket$  is the **Delayed** version. Abel and Chapman give a **Delayed** interpreter (for typed terms) directly, exercising some craft in negotiating size and mutual recursion [2]. The **General** construction makes that craft systematic.

## 8 An Introduction or Reimmersion in Induction-Recursion

I have one more semantics for general recursion to show you, constructing for any given  $f : \text{PiG } S \ T$  its *domain*. The domain is an inductively defined predicate, classifying the arguments which give rise to call trees whose paths are finite. As Ana Bove observed, the fact that a function is defined on its domain is a structural recursion—the tricky part is to show that the domain predicate holds [7]. However, to support nested recursion, we need to define the domain predicate and the resulting output *mutually*. Bove and Capretta realised that such mutual definitions are just what we get from Dybjer and Setzer’s notion of *induction-recursion* [8, 12], giving rise to the ‘Bove-Capretta method’ of modelling general recursion and generating termination proof obligations.

We can make the Bove-Capretta method generic, via the universe encoding for (indexed) inductive-recursive sets presented by Dybjer and Setzer. The idea is that each node of data is a record with some ordinary fields coded by  $\sigma$ , and some places for recursive substructures coded by  $\delta$ , with  $\iota$  coding the end.

```
data IR {I} {S : Set} (I : S → Set I) (O : Set I) : Set (I ⊔ Isuc Izero) where
  ι : (o : O) → IR I O
  σ : (A : Set) (T : A → IR I O) → IR I O
  δ : (B : Set) (s : B → S)
      (T : (i : (b : B) → I (s b)) → IR I O) → IR I O
```

Now, in the indexed setting, we have  $S$  sorts of recursive substructure, and for each  $s : S$ , we know that an ‘input’ substructure can be interpreted as a value

of type  $I\ s$ . Meanwhile,  $O$  is the ‘output’ type in which we must interpret the whole node. I separate inputs and outputs when specifying individual nodes, but the connection between them will appear when we tie the recursive knot. When we ask for substructures with  $\delta$  branching over  $B$ , we must say which sort each must take via  $s : B \rightarrow S$ , and then we will learn the interpretations of those substructures before we continue. Eventually, we must signal ‘end of node’ with  $\iota$  and specify the output. As you can see,  $\sigma$  and  $\delta$  pack up **Set**s, so **IR** codes are certainly large: the interpretation types  $I$  and  $O$  can be still larger.

Now, to interpret these codes as record types, we shall need the usual notion of dependent pair types. We shall need  $\Sigma$  for nothing larger than **Set**, because although **IR** types can have large interpretations, the types themselves are small.

```

record  $\Sigma$  ( $S : \mathbf{Set}$ ) ( $T : S \rightarrow \mathbf{Set}$ ) : Set where
  constructor  $\rightarrow, -$ 
  field  $\text{fst} : S; \text{snd} : T\ \text{fst}$ 
open  $\Sigma$ 

```

By way of abbreviation, let me also introduce the notion of a sort-indexed family of maps, between sort-indexed families of sets.

```

 $\rightarrow_{\text{Set}} : \forall \{l\} \{S : \mathbf{Set}\} (X : S \rightarrow \mathbf{Set}) (I : S \rightarrow \mathbf{Set}\ l) \rightarrow \mathbf{Set}\ l$ 
 $X \rightarrow_{\text{Set}} I = \forall \{s\} \rightarrow X\ s \rightarrow I\ s$ 

```

If we know what the recursive substructures are and how to interpret them, we can say what nodes consist of.

```

 $\llbracket - \rrbracket_{\mathbf{Set}} : \forall \{l\ S\ I\ O\} (T : \mathbf{IR}\ \{l\}\ I\ O) (X : S \rightarrow \mathbf{Set}) (i : X \rightarrow_{\text{Set}} I) \rightarrow \mathbf{Set}$ 
 $\llbracket \iota\ o \rrbracket_{\mathbf{Set}} X\ i = 1$ 
 $\llbracket \sigma\ A\ T \rrbracket_{\mathbf{Set}} X\ i = \Sigma\ A\ \lambda\ a \rightarrow \llbracket T\ a \rrbracket_{\mathbf{Set}} X\ i$ 
 $\llbracket \delta\ B\ s\ T \rrbracket_{\mathbf{Set}} X\ i = \Sigma\ ((b : B) \rightarrow X\ (s\ b))\ \lambda\ r \rightarrow \llbracket T\ (i \cdot r) \rrbracket_{\mathbf{Set}} X\ i$ 

```

Moreover, we can read off their output.

```

 $\llbracket - \rrbracket_{\text{out}} : \forall \{l\ S\ I\ O\} (T : \mathbf{IR}\ \{l\}\ I\ O) (X : S \rightarrow \mathbf{Set}) (i : X \rightarrow_{\text{Set}} I) \rightarrow \llbracket T \rrbracket_{\mathbf{Set}} X\ i \rightarrow O$ 
 $\llbracket \iota\ o \rrbracket_{\text{out}} X\ i\ \langle \rangle = o$ 
 $\llbracket \sigma\ A\ T \rrbracket_{\text{out}} X\ i\ (a, t) = \llbracket T\ a \rrbracket_{\text{out}} X\ i\ t$ 
 $\llbracket \delta\ B\ s\ T \rrbracket_{\text{out}} X\ i\ (r, t) = \llbracket T\ (i \cdot r) \rrbracket_{\text{out}} X\ i\ t$ 

```

Now we can tie the recursive knot. Again, I make use of Abel’s sized types to be precise about why **decode** terminates.

```

mutual
data  $\mu$   $\{l\} \{S\} \{I\} (F : (s : S) \rightarrow \mathbf{IR}\ \{l\}\ I\ (I\ s)) (j : \mathbf{Size}) (s : S) : \mathbf{Set}$ 
  where  $\langle - \rangle : \{k : \mathbf{Size} < j\} \rightarrow \llbracket F\ s \rrbracket_{\mathbf{Set}} (\mu\ F\ k)\ \mathbf{decode} \rightarrow \mu\ F\ j\ s$ 
 $\mathbf{decode} : \forall \{l\} \{S\} \{I\} \{F\} \{j\} \rightarrow \mu\ \{l\} \{S\} \{I\}\ F\ j \rightarrow I$ 
 $\mathbf{decode}\ \{F = F\}\ \{s = s\}\ \langle n \rangle = \llbracket F\ s \rrbracket_{\text{out}} (\mu\ F\ -)\ \mathbf{decode}\ n$ 

```

Of course, you and I can see from the definition of  $\llbracket - \rrbracket_{\text{out}}$  that the recursive uses of **decode** will occur only at substructures, but without sized types, we should need to inline  $\llbracket - \rrbracket_{\text{out}}$  to expose that guardedness to Agda.

Now, as Ghani and Hancock observe,  $\mathbf{IR}\ I$  is a (relative) monad [13].<sup>2</sup> Indeed, it is the free monad generated by  $\sigma$  and  $\delta$ . Its  $\gg=$  operator is perfectly standard, concatenating dependent record types. I omit the unremarkable proofs of the laws.

<sup>2</sup> They observe also that  $\llbracket - \rrbracket_{\mathbf{Set}}$  and  $\llbracket - \rrbracket_{\text{out}}$  form a monad morphism.

$$\begin{aligned}
\text{IRK} &: \forall \{l\} \{S\} \{I : S \rightarrow \text{Set } l\} \rightarrow \text{Kleisli } (\text{IR } I) \\
\text{IRK } \{l\} \{S\} \{I\} &= \text{record } \{\text{return} = \iota; \_ \gg\!\! = \_ = \_ \gg\!\! =_1\} \text{ where} \\
\_ \gg\!\! =_1 &: \forall \{X Y\} \rightarrow \text{IR } I X \rightarrow (X \rightarrow \text{IR } I Y) \rightarrow \text{IR } I Y \\
\iota x \quad \gg\!\! =_1 K &= K x \\
\sigma A T \quad \gg\!\! =_1 K &= \sigma A \lambda a \rightarrow T a \gg\!\! =_1 K \\
\delta B s T \gg\!\! =_1 K &= \delta B s \lambda f \rightarrow T f \gg\!\! =_1 K
\end{aligned}$$

Now, the Bove-Capretta method amounts to a monad morphism from **General**  $S T$  to  $\text{IR } T$ . That is, the domain predicate is indexed over  $S$ , with domain evidence for a given  $s$  **decoded** in  $T s$ . We may generate the morphism as usual from the treatment of a typical **call**  $s$ , demanding the single piece of evidence that  $s$  is also in the domain, then returning at once its decoding.

$$\begin{aligned}
\text{callInDom} &: \forall \{l S T\} \rightarrow (s : S) \rightarrow \text{IR } \{l\} T (T s) \\
\text{callInDom } s &= \delta 1 (\text{const } s) \lambda t \rightarrow \iota (t \langle \rangle) \\
\text{DOM} &: \forall \{S T\} \rightarrow \text{PiG } S T \rightarrow (s : S) \rightarrow \text{IR } T (T s) \\
\text{DOM } f s &= \text{morph IRK callInDom } (f s)
\end{aligned}$$

Now, to make a given  $f : \text{PiG } S T$  total, it is sufficient to show that its domain predicate holds for all  $s : S$ .

$$\begin{aligned}
\text{total} &: \forall \{S T\} (f : \text{PiG } S T) (\text{allInDom} : (s : S) \rightarrow \mu (\text{DOM } f) \_ s) \rightarrow \\
&\quad (s : S) \rightarrow T s \\
\text{total } f \text{ allInDom} &= \text{decode} \cdot \text{allInDom}
\end{aligned}$$

The absence of  $\sigma$  from **callInDom** tells us that domain evidence contains at most zero bits of data and is thus ‘collapsible’ in Edwin Brady’s sense [9], thus enabling **total**  $f$  to be compiled for run time execution exactly as the naïve recursive definition of  $f$ .

## 9 Discussion

We have seen how to separate the business of saying what it is to *be* a recursive definition from the details of what it means to *run* a recursive program. The former requires only that we work in the appropriate free monad to give us an interface permitting the recursive calls we need to make. Here, I have considered only recursion at a fixed arity, but the method should readily extend to partially applied recursive calls, given that we need only account for their *syntax* in the first instance. It does not seem like a big stretch to expect that the familiar equational style of recursive definition could be translated monadically, much as we see in the work on algebraic effects.

The question, then, is not what is *the* semantics for general recursion, but rather how to make use of recursive definitions in diverse ways by giving appropriate monad morphisms—that is, by explaining how each individual call is to be handled. We have seen a number of useful possibilities, not least the Bove-Capretta domain construction, by which we can seek to establish the totality of our function and rescue it from its monadic status.

However, the key message of this paper is that the status of general recursive definitions is readily negotiable within a total framework. There is no need to give up on the ability either to execute potentially nonterminating computations or to be trustably total. There is no difference between what you can *do* with a partial language and what you can *do* with a total language: the difference is in what you can *know*. The time for wilful ignorance is over.

## References

1. Andreas Abel. *Type-based termination: a polymorphic lambda-calculus with sized higher-order types*. PhD thesis, Ludwig Maximilians University Munich, 2007.
2. Andreas Abel and James Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In P. Levy and N. Krishnaswami, editors, *Workshop on Mathematically Structured Functional Programming 2014*, volume 153 of *EPTCS*, pages 51–67, 2014.
3. Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *ACM Symposium on Principles of Programming Languages, POPL '13*, pages 27–38. ACM, 2013.
4. Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *LNCS*, pages 297–311. Springer, 2010.
5. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *LNCS*, pages 453–468. Springer, 1999.
6. Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.
7. Ana Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, 2001.
8. Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In R.J. Boulton and P.B. Jackson, editors, *TPHOLs*, volume 2152 of *LNCS*, pages 121–135. Springer, 2001.
9. Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs 2003*, volume 3085 of *LNCS*, pages 115–129. Springer, 2003.
10. Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.
11. Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications 1999*, volume 1581 of *LNCS*, pages 129–146. Springer, 1999.
12. Peter Dybjer and Anton Setzer. Indexed induction-recursion. In R. Kahle, P. Schroeder-Heister, and R. F. Stärk, editors, *Proof Theory in Computer Science 2001*, volume 2183 of *LNCS*, pages 93–113. Springer, 2001.
13. Neil Ghani and Peter Hancock. Containers, monads and induction recursion. *Mathematical Structures in Computer Science*, FirstView:1–25, 2 2015.
14. Neil Ghani, Christoph Lüth, Federico De Marchi, and John Power. Algebras, coalgebras, monads and comonads. *Electr. Notes Theor. Comput. Sci.*, 44(1):128–145, 2001.
15. Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.
16. Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
17. D.A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.