Turing-Completeness Totally Free

Conor McBride

University of Strathclyde
conor@strictlypositive.org

Abstract. In this 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 calls should be executed. Diverse semantics can be given within a total framework 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. The paper is literate Agda, but its key ideas are more broadly transferable.

1 Introduction

Advocates of Total Functional Programming [21], such as myself, can prove prone to a false confession, namely that the price of functions which provably function is the loss of Turing-completeness. In a total language, to construct $f : S \to T$ 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 $S \to G\,T$ where G is a suitable monad.

The literature and lore of our discipline are littered with candidates for G [19, 10, 22], and this article will contribute another—the *free* monad with one operation $f : S \to T$. 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 $yield : 1 \to 1$, 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 in the domain [8]. When recursive calls are nested, 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 $f : S \to GT$ to the total $f : S \to T$.

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 a carrier of checkable *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[1], in Agda, we may define a free monad which is both general purpose, in the sense of being generated by the strictly positive functor of your choice, but also suited to the purpose of supporting 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 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. General is a standard inductive definition, available in many dialects of type theory. Indeed, it closely resembles the archetypal such definition: Martin-Löf's 'W-type' [17], W $S\ T$, of well-ordered trees is the leafless special case, General $S\ T$ 0; the reverse construction, General $S\ T\ X\ =$ W $(X + S)$ [const 0, T][2], just attaches value leaves as an extra sort of childless node.

General datatypes come with a catamorphism, or 'fold' operator.[3]

```
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)
```

Agda uses curly braces in types to declare arguments which are suppressed by default; corresponding curly braces are used in abstractions and applications to override that default. Note that unlike in Haskell, Agda's λ-abstractions scope as far to the right as possible and may thus stand without parentheses as the last argument to a function such as c. It is idiomatic Agda to drop those parentheses, especially when we think of the function as a defined form of quantification.

The 'bind' operation for the monad General $S\ T$ substitutes computations for values to build larger computations. It is, of course, a fold.

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

We then acquire what Gordon Plotkin and John Power refer to as a *generic effect* [20]—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 Π-type',

[1] http://github.com/pigworker/Totality
[2] I write brackets for case analysis over a sum
[3] Whenever I intend a monoidal accumulation, I say 'crush', not 'fold'.

```
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.

```
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)
```

Firstly, note that PiG $S\ \lambda\ s\ \rightarrow\ T$ is a defined form of quantification, needing no parentheses. Secondly, observe that each call is but a *placeholder* for a recursive call to fusc. Our code tells us just how to expand the recursion *once*. Note that fusc's *nested* recursive calls make use of the way $>\!\!>\!\!=_{\mathsf{G}}$ allows values from earlier effects to influence the choice of later effects: nested recursion demands the full monadic power of General.

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! Haskell may reserve direct style for the particular monad of nontermination and finite failure, necessitating a more explicit notation for other monads, but languages like Bauer and Pretnar's *Eff* [6] give us grounds to hope for better. They let us write in direct style for whatever effectful interface is locally available, then obtain the computation delivered by the appropriate Moggi-style translation into an explicitly monadic kernel [18]. Just such a translation would give us our monadic fusc from its direct presentation.

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. To save space and concentrate on essentials, I have suppressed some proofs. The online sources for this paper omit nothing.

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]. Where Haskell programmers might expect a type class and ML programmers a module signature, Agda requires me to be more specific and give the type of records which pack up the necessary equipment. Although the 'notion of computation' is given by a functor taking sets of values to sets of computations, that functor need not preserve the level in the predicative set-theoretic hierarchy[4] at which we work, and we shall need this flexibility for the Bove-Capretta construction which represents computations as descriptions of datatypes. The upshot is that we are obliged to work polymorphically in *levels*, i for values and j for computations.

```
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 _>>=_ _◇_
```

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. The Kleisli record type thus packs up operations which quantify over all level i value sets and target specific level j computation sets, so the lowest level we can assign is the maximum of lsuc i and j. Of course, we have

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

The 'Monad laws' amount to requiring that return and $\diamond$ give us a category.

```
record KleisliLaws {i j} {M : Set i → Set j} (KM : Kleisli M)
  : Set (lsuc 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. Indeed, any declaration marked with a dot introduces an entity which exists for reasoning only. Correspondingly, it will not interfere with the effectiveness of computation if we postulate an extensional equality and calculate with 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
```

Indeed, it is clear that the above laws, expressing equations between functions rather than between the concrete results of applying them, can hold only in an extensional setting.

Extensionality gives us, for example, that anything satisfying the defining equations of a fold is a fold.

[4] Agda, Coq, etc., stratify types by 'size' into sets-of-values, sets-of-sets-of-values, and so on, ensuring consistency by forbidding the quantification over 'large' types by 'small'.

```
.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
```

Specifically, the identity function is a fold, giving idLeft for General S T.

Meanwhile, computations are often constructed using $\gg\!=_{\mathsf{G}}$ (defined by specialising fold to the constructor ??) and interpreted recursively by some other fold. Correspondingly, a crucial lemma in the development is the following *fusion* law which eliminates an intermediate $\gg\!=_{\mathsf{G}}$ in favour of its direct interpretation.

```
.foldFusion : ∀ {l S T X Y} {Z : Set l}
  (r : Y → Z) (c : (s : S) → (T s → Z) → Z)
  (f : X → General S T Y) →
  (fold r c · fold f _??_) ≡ fold (fold r c · f) c
```

Indeed, the associativity of Kleisli composition amounts to the fusion of two layers of $\gg\!=_{\mathsf{G}}$, and thus comes out as a special case. The above two results readily give us the KleisliLaws for the GeneralK operations on General S T.

```
.GeneralKLaws : ∀ {S T} → KleisliLaws (GeneralK {S} {T})
```

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

```
record Morphism {i j k} {M : Set i → Set j} {N : Set i → Set k}
                (KM : Kleisli M) (KN : Kleisli N)
  (m : ∀ {X} → M X → N X) : Set (lsuc i ⊔ j ⊔ k) where
  module −M = Kleisli KM; module −N = Kleisli KN
  field
    .respectI : {X : Set i} →
                  m · return_M {X} ≡ return_N {X}
    .respectC : {A B C : Set i} (f : B → M C) (g : A → M B) →
                  m · (f ⋄_M g) ≡ (m · f) ⋄_N (m · g)
```

The proofs, idMorph and compMorph, that monad morphisms are closed under identity and composition, are straightforward and omitted.

Now, General S T is the free monad on the functor $\Sigma\ S\ \lambda\ s \to T\ s \to -$ which captures a single request-response interaction. The fact that it is the free construction, turning functors into monads, tells us something valuable about the ways in which its monad morphisms arise. Categorically, what earns the construction the designation 'free' is that it is left adjoint to the 'forgetful' map which turns monads back into functors by ignoring the additional structure given by return and $\gg\!=$. We seek monad morphisms from General S T to some other M, i.e., some

$$m : \forall\,\{X\} \to \mathsf{General}\ S\ T\ X \to M\ X$$

satisfying the above demands for respect. The free-forgetful adjunction amounts to the fact that such monad morphisms from General S T are in one-to-one correspondence with the functor morphisms (i.e.,polymorphic functions) from the underlying request-response functor to M considered only as a functor. That is, every such m is given by and gives a function of type

$$\forall\,\{X\} \to (\Sigma\ S\ \lambda\ s \to T\ s \to X) \to M\ X$$

with no conditions to obey. Uncurrying the dependent pair and reordering gives us the equivalent type

$$(s : S) \to \forall\,\{X\} \to (T\ s \to X) \to M\ X$$

which the Yoneda lemma tells us amounts to

$$(s : S) \to M\ (T\ s)$$

That is, the monad morphisms from General $S\ T$ to M are exactly given by the 'M-acting versions' of our function. Every such morphism is given by instantiating the parameters of the following definition.

```
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 (_>>=_ · h) where open Kleisli KM
```

We may 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)
```

Moreover, just as the categorical presentation would have us expect, morph give us the *only* monad morphisms from General $S\ T$, by 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}
```

I do not think like, or of myself as, a category theorist. I do, by instinct and training, look to inductive definitions as the basis of a general treatment: syntax as the precursor to any semantics, in this case any monad semantics for programs which can make recursive calls. The categorical question 'What is the *free* monad for call?' poses that problem and tells us to look no further for a solution than General $S\ T$ and morph.

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 $f = \lambda\, s \to$ call s amounts to the often valid but seldom helpful 'definition':

$$\mathsf{f}\ s = \mathsf{f}\ s$$

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

```
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, or when.

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.

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 gain 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? Barely this: there is more to delivering the run time execution semantics of programs than the pure evaluation of expressions. A pedant might quibble that the *language* is Turing-incomplete because it takes the *system* in which you use it to execute arbitrary recursive computations for as long as you are willing to tolerate. Such an objection has merit only in that it speaks against casually classifying a *language* as Turing-complete or otherwise, without clarifying the variety of its semanticses and the relationships between them.

Whilst we are discussing the semantics of total languages, emphatically in the plural, 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 and indeed to make use of restricted evaluation in the course of code generation.

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 λ-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$^\infty$, of those lazy computations themselves. We gain a useful basis for reasoning about infinite behaviour.

```
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
```

Abel explains that Size, here, is a special type which characterizes the *observation depth* to which one may iteratively force the lazy computation. Values of type Size

cannot be inspected by programs, but corecursive calls must reduce size, so cannot be used to satisfy the topmost observation. That is, we must deliver the outermost now or later without self-invocation. Pleasingly, corecursive need not be *syntactically* 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
```

Syntactically, the corecursive call $\mathsf{unfold}^\infty\ f$ is inside a composition inside a defined case analysis operator, but the type of later ensures that the recursive unfold^∞ has a smaller size than that being forced.

Based on projection, copatterns favour 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 inevitable as we alternate between the projection and case analysis. However, 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 laters. 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, $F\ Y\ =\ 1 \times 1 \to Y$, 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! Of course, naïve $\gg\!=_{\mathsf{D}}$ is expensive, especially when left-nested, but the usual efficient semantics, based on interruption rather than yielding, can be used at run time.

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
```

Although my definition of delay is a monad morphism by construction, it is quite possible to give extensionally the same operation as an unfold, thus removing the reliance on sized types but incurring the obligation to show that delay respects return and $\gg\!=_{\mathsf{G}}$ upto strong bisimulation. Some such manoeuvre would be necessary to port this section's work to Coq's treatment of coinduction [15]. There is certainly no deep obstacle to the treatment of general recursion via coinduction in Coq.

7 A Little λ-Calculus

By way of a worked example, let us implement the untyped λ-calculus. We can equip ourselves with de Bruijn-indexed terms, using finite sets to police scope, as

did Altenkirch and Reus [5]. It is sensible to think of Fin n as the type of natural numbers strictly less than n.

```
data Fin : Nat → Set where
  zero : {n : Nat}           → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)
```

I have taken the liberty of parametrizing terms by a type X of inert constants.

```
data Λ (X : Set) (n : Nat) : Set where
  κ    : X                 → Λ X n
  #    : Fin n             → Λ X n
  λ    : Λ X (suc n)       → Λ X n
  _$_ : Λ X n → Λ X n → Λ X n
infixl 5 _$_
```

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
  ⟨⟩  : 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 (γ ‚ _) (suc n) = proj γ 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
infixr 4 _⊢_
```

We can now give the evaluator, ⟦_⟧ as a General recursive strategy to compute a value from a closure. Application is the fun case: a carefully placed **let** shadows the evaluator with an appeal to call, so the rest of the code looks familiar, yet it is no problem to invoke the non-structural recursion demanded by a β-redex.

```
⟦_⟧ : {X : Set} → PiG (Closure X) λ _ → Val X
⟦ γ ⊢ κ x  ⟧     = !! (κ x ⟨⟩)     -- Constants are inert.
⟦ γ ⊢ # i  ⟧     = !! (proj γ i)   -- Variables index the environment.
⟦ γ ⊢ λ b  ⟧     = !! (λ γ b)      -- Unapplied functions get stuck.
⟦ γ ⊢ f $ s ⟧    =
  let ⟦_⟧ : PiG (Closure _) λ _ → Val _; ⟦_⟧ = call in    -- shadow ⟦_⟧
  ⟦ γ ⊢ s ⟧  >>=G λ v →    -- evaluate the argument, then
  ⟦ γ ⊢ f ⟧  >>=G λ {       -- evaluate the function, then, inspecting its value,
    (κ x vs) → !! (κ x (vs ‚ v));   -- either grow an inert constant application,
    (λ δ b)  → ⟦ δ ‚ v ⊢ b ⟧    }   -- or grow a closure from a stuck function.
```

Thus equipped, lazy $[\![\_]\!]$ 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 method makes that craft rather more systematic.

There is still a little room for programmer choice, however. Where a recursive call happens to be structurally decreasing, e.g. when evaluating the function and its argument, we are not *forced* to appeal to the call oracle, but could instead compute by structural recursion the expansion of an evaluation to those of its redexes. Indeed, that is the choice which Abel and Chapman make. It is not yet clear which course is preferable in practice.

8 An Introduction or Reimmersion in Induction-Recursion

I have one more semantics for general recursion to show you, constructing for any given $f : \mathsf{PiG}\ S\ T$ 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 shown consistent by Dybjer and Setzer. The idea is that each node of data is a record with some ordinary fields coded by σ, and some places for recursive substructures coded by δ, with ι coding the end.

```
data IR {l} {S : Set} (I : S → Set l) (O : Set l) : Set (l ⊔ lsuc lzero) where
  ι : (o : O)                                  → IR I O
  σ : (A : Set) (K : A → IR I O)               → IR I O
  δ : (B : Set) (s : B → S)
      (K : (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 δ branching over B, we must say which sort each must take via $s : B \to S$, and then K learns the interpretations of those substructures *before* we continue. It is that early availability of the interpretation which allows Bove and Capretta to define the domain predicate for nested recursions: the interpretation is exactly the value of the recursive call that is needed to determine the argument of the enclosing recursive call.

Eventually, we must signal 'end of node' with ι and specify the output. As you can see, σ and δ pack up Sets, so IR codes are certainly large: the interpretation types I and O can be still larger. Induction-recursive definitions add convenience rather than expressive strength when I and O happen to be small and can be translated to ordinary datatype families indexed by the interpretation under just those circumstances [16]. For our application, they need only be as large as the return type of the function whose domain we define, so the translation applies, yielding not the *domain* of the function but the relational presentation of its *graph*. The latter is less convenient than the domain for termination proofs, but is at least expressible in systems without induction-recursion, such as Coq.

Now, to interpret these codes as record types, we need the usual notion of *small* dependent pair types, for IR gives small types with possibly large interpretations.

```
record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field fst : S; snd : T fst
```

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

```
_⇢_ : ∀ {l} {S : Set} (X : S → Set) (I : S → Set l) → Set l
X ⇢ I = ∀ {s} → X s → I s
```

If we know what the recursive substructures are and how to interpret them, we can say what nodes consist of, namely tuples made with Σ and 1.

```
⟦_⟧Set : ∀ {l S I O} (C : IR {l} I O) (X : S → Set) (i : X ⇢ I)
                    → Set
⟦ ι o     ⟧Set X i = 1
⟦ σ A K   ⟧Set X i = Σ A λ a → ⟦ K a ⟧Set X i
⟦ δ B s K ⟧Set X i = Σ ((b : B) → X (s b)) λ r → ⟦ K (i · r) ⟧Set X i
```

Moreover, we can read off their output by applying i at each δ until we reach $\iota\ o$.

```
⟦_⟧out : ∀ {l S I O} (C : IR {l} I O) (X : S → Set) (i : X ⇢ I)
                    → ⟦ C ⟧Set X i → O
⟦ ι o     ⟧out X i ⟨⟩     = o
⟦ σ A K   ⟧out X i (a , t) = ⟦ K a ⟧out X i t
⟦ δ B s K ⟧out X i (r , t) = ⟦ K (i · r) ⟧out X i t
```

Now we can tie the recursive knot, following Dybjer and Setzer's recipe. Again, I make use of Abel's sized types to convince Agda that decode terminates.

```
mutual
  data μ {l} {S} {I} (F : (s : S) → IR {l} I (I s)) (j : Size) (s : S) : Set
    where ⟨_⟩ : {k : Size< j} → ⟦ F s ⟧Set (μ F k) decode → μ F j s
  decode : ∀ {l} {S} {I} {F} {j} → μ {l} {S} {I} F j ⇢ I
  decode {F = F} {s = s} ⟨ n ⟩ = ⟦ F s ⟧out (μ F _) decode n
```

Of course, you and I can see from the definition of $[\![\_]\!]_{\mathsf{out}}$ that the recursive uses of decode will occur only at substructures, but without sized types, we should need to inline $[\![\_]\!]_{\mathsf{out}}$ to expose that guardedness to Agda.

Now, as Ghani and Hancock observe, IR I is a (relative) monad [13].[5] Indeed, it is the free monad generated by σ and δ. Its $\gg\!=$ operator is perfectly standard, concatenating dependent record types. I omit the unremarkable proofs of the laws.

```
IRK : ∀ {l} {S} {I : S → Set l} → Kleisli (IR I)
IRK {l} {S} {I} = record {return = ι; _>>=_ = _>>=ι_} where
  _>>=ι_ : ∀ {X Y} → IR I X → (X → IR I Y) → IR I Y
  ι x       >>=ι K′ = K′ x
  σ A K     >>=ι K′ = σ A λ a → K a >>=ι K′
  δ B s K   >>=ι K′ = δ B s λ f → K f >>=ι K′
```

Now, the Bove-Capretta method amounts to a monad morphism from General $S\ T$ to IR T. That is, the domain predicate is indexed over S, with domain evidence

[5] They observe also that $[\![\_]\!]_{\mathsf{Set}}$ and $[\![\_]\!]_{\mathsf{out}}$ form a monad morphism.

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.

```
callInDom : ∀ {l S T} → (s : S) → IR {l} T (T s)
callInDom s = δ 1 (const s) λ t → ι (t ⟨⟩)
DOM : ∀ {S T} → PiG S T → (s : S) → IR T (T s)
DOM f s = morph IRK callInDom (f s)
```

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

```
total : ∀ {S T} (f : PiG S T) (allInDom : (s : S) → μ (DOM f) _ s) →
          (s : S) → T s
total f allInDom = decode · allInDom
```

The absence of σ 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* one. 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 *promise*, and it is the partial languages which fall short.

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. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.
16. Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, volume 7941 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013.
17. P. Martin-Löf. Constructive mathematics and computer programming. In L.J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *LMPS 6*, pages 153–175. North-Holland, 1982.
18. 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.
19. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.
20. Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
21. D.A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
22. Tarmo Uustalu. Partiality is an effect. Talk given at Dagstuhl seminar on Dependently Typed Programming http://www.ioc.ee/~tarmo/tday-veskisilla/uustalu-slides.pdf, September 2004.