The Continuity of Monadic Stream Functions

Venanzio Capretta and Jonathan Fowler
School of Computer Science
University of Nottingham, UK
Email : {venanzio.capretta,jonathan.fowler}@nottingham.ac.uk

***Abstract*—Brouwer's continuity principle states that all functions from infinite sequences of naturals to naturals are continuous, that is, for every sequence the result depends only on a finite initial segment. It is an intuitionistic axiom that is incompatible with classical mathematics. Recently Martín Escardó proved that it is also inconsistent in type theory. We propose a reformulation of the continuity principle that may be more faithful to the original meaning by Brouwer. It applies to monadic streams, potentially unending sequences of values produced by steps triggered by a monadic action, possibly involving side effects. We consider functions on them that are uniform, in the sense that they operate in the same way independently of the particular monad that provides the specific side effects. Formally this is done by requiring a form of naturality in the monad. Functions on monadic streams have not only a foundational importance, but have also practical applications in signal processing and reactive programming. We give algorithms to determine the modulus of continuity of monadic stream functions and to generate dialogue trees for them (trees whose nodes and branches describe the interaction of the process with the environment).**

***Index Terms*—monadic stream function, continuity, type theory, functional programming, stream, monad, dialogue trees, strategy trees**

I. Introduction

Brouwer's continuity principle is a non-standard intuitionistic postulate, incompatible with classical mathematics. It states that every function from infinite sequences of natural numbers to natural numbers is continuous. Continuity, in this setting, means that the value of the function on each specific input depends only on a finite initial segment of the sequence.

We denote an arbitrary sequence by $\alpha = a_0 \triangleleft a_1 \triangleleft a_2 \triangleleft \cdots$. We write $\alpha|_n$ for the initial segment consisting of the first n elements: $\alpha|_n = a_0 :: a_1 :: \ldots :: a_{n-1} :: \mathsf{nil}$. (We use the type theoretic notation that we introduce formally later: infinite sequences, or *streams*, form a coinductive type with constructor $\triangleleft$; finite sequences, or *lists*, form an inductive type with constructor $::$ plus empty list base case, nil.)

We say that two streams α and α' are n-equal if their first n elements are the same: $\alpha' =_n \alpha$ if $\alpha'|_n = \alpha|_n$. The principle is expressed by the following formula:

$$\forall \alpha, \exists n, \forall \alpha', \alpha' =_n \alpha \Rightarrow f\,\alpha' = f\,\alpha.$$

It states that the value of f on any input α depends only on an initial segment $\alpha|_n$, so that on any other sequence α' with the same initial n elements, f will produce the same result.

Paolo Capriotti gave a vital contribution to the ideas in this article. We originally discussed the issue of continuity of monadic stream functions with Paolo and we developed together their application to dialogue trees. He came up with the counterexample asktwice in Section VI.

The justification that Brouwer gave for the principle rests on his philosophy of mathematics, specifically on his interpretation of the meaning of *infinite sequence* and *function*. The objects on which the functions operate are *choice sequences*, progressions of values that are completely free and not governed to a generating rule. They may be produced by a *creative subject* and are not necessarily algorithmic. On the contrary, functions are effective procedures, consisting of precise mental steps. A function can consult its sequence argument one element at a time and must algorithmically compute a result in a finite time. It follows that a function can only obtain a finite number of sequence elements in the time it takes it to produce the result. Hence, it must be continuous.

The relevance of intuitionistic mathematics to modern computer science rests on the parallel between the philosophical apprehension of mathematical objects as mental constructions and their computational realization as data structures and programs. A function is now an computational procedure. Brouwer's choice sequences can be reinterpreted as input streams. These need not be data structures implemented on a computer, but can be progressions of input values read from some device.

The setup of Brouwer's principle can be reformulated thus: We have an interactive program that can ask the user to insert a value at any point of its computation; after a finite number of steps, the program must end and produce a result. It is not essential to think that the sequence is provided by a user; in scientific and real-world applications we may think of the sequence as produced by a measuring device or by any other signalling process. There is no predefined limit to the number of input values that the program will ask for, but it can only get a finite number of them if it needs to terminate. Therefore the program is the realization of a continuous function in Brouwer's sense.

The most coherent and complete realization of the correspondence between intuitionistic mathematics and computer science is in Martin-Löf's Type Theory. It is, at the same time, a programming language and a formal system for the foundations of mathematics. It is compatible with both intuitionistic and classical mathematics. It has been very successful and led to concrete implementations, notably the systems Coq [25] and Agda [22], and useful applications.

We may now ask if the theory can be extended with stronger constructive principles, specifically if we can add the continuity principle to it. Unfortunately not: Recently Martín Escardó discovered that the straightforward addition of the continuity principle to type theory leads to contradiction

[11]. In the aftermath of the discovery, discussion focused on analyzing the source of the problem and investigating alternative formulations of the principle that are not lethal. One solution, proposed by Escardó himself, is to adopt a weaker notion of existential quantifier. One crucial point in his paradox was that we can use the constructive content of the existential quantification on the length n of the initial segment to construct a function that turns out not to be continuous. By weakening the existential quantifier, so that no extraction of a witness is allowed, we prevent the construction of such an evil function.

Here we propose an alternative formulation, based on a different diagnosis of the paradox. The construction of the evil function has an input sequence as a parameter. But in Brouwer's conception there is a clear distinction between sequences, that are non-computable, and functions, that must be effectively computable. Therefore we should not allow the definition of a function to depend on a sequence. However, in type theory all objects are intended to be internal to the theory itself and we are authorized to use any object in the definition of another. Specifically, infinite sequences are realized as functions on the natural numbers. A sequence of natural numbers is just a function $\alpha : \mathbb{N} \to \mathbb{N}$. If we construe functions as computable, which is essential for the justification of the continuity principle, then so are infinite sequences, contrary to the spirit of the principle.

We may substitute the representation of sequences as functions $\mathbb{N} \to \mathbb{N}$ with a *coinductive* type of streams $\mathbb{S}_A$. An element $\alpha : \mathbb{S}_A$ is built by using the constructor $\triangleleft$ an infinite number of times. (*Corecursion* patterns tell us how we can generate the infinite sequence by a finite process.)

There is a correspondence between $\mathbb{S}_A$ and $\mathbb{N} \to A$, which is one-to-one if we assume extensionality of functions and bisimilarity of streams: functions are equal if they are pointwise equal, streams are equal if they can *simulate* each other. Therefore, the change of data type does not in itself solve the issue. But it affords a way to generalize the notions.

A stream, as defined above, is still an internal object of type theory. It is meant to be defined by some computational criterion: it could be generated by a coalgebra [17], characterized as the fixed point of a guarded equation [8], [16], or produced by a guarded-by-destructors pattern [1].

We look for a notion of stream that comprises other ways of generating the sequence of elements, in particular allowing them to be given interactively. In functional programming, specifically in the language Haskell, interactive programming is realized by using the IO monad. While a basic type A contains *pure* elements which are immutable data structures, when we put it inside the IO monad we obtain an interactive type $(\mathsf{IO}\, A)$ whose elements are values of A produced through interaction and producing side effects. Other monads characterize different kinds of side effects.

We define a notion of stream that embodies the possibility of its elements being produced through monadic actions. The coinductive type $\mathbb{S}_{M,A}$, for a given monad M, has elements of the form $\mathsf{mcons}\, m$, where m is an M-action, $m : M\,(A \times \mathbb{S}_{M,A})$. This action, when executed, will produce some side effects and give results consisting of an element of A and a new monadic stream. For example, if M is the IO monad, there will be some interaction with the user to obtain the first element of the stream and the tail, which is again a monadic stream that results in a new IO action. Other monads result in different stream behaviours: the Maybe monad admits the possibility of the action not giving a value, thus allowing the sequence to terminate; the list monad allows the stream to branch into many possible continuations; the state monad allows the elements to depend on and modify a changing state, the writer monad gives streams that, when read, produce some output; and so on.

Now we come to the characterization of functions on streams. We want a function to operate independently of how the stream is produced. It shouldn't matter if the stream is a pure internal procedure or if it is an interactive process or if it produces any other side effects. In other words, functions should apply to monadic streams and be polymorphic on the monad. Therefore, the type of functions we are interested in is

$$\forall M, \mathbb{S}_{M,A} \to M\,B.$$

The variable M ranges over monads. A function f of this type should operate in a *uniform* way independently of the monad. We make this notion precise through a naturality condition. A different way to characterize uniformity is through parametricity [3]. Naturality is weaker. Its advantages are that it has a more abstract and clear formulation and results in stronger properties about the function.

Every natural monadic stream function is continuous: Brouwer's Principle becomes a theorem.

II. Monadic Streams

Coinductive types are data structures that contain non-well-founded elements (See Chapter 13 of the book by Bertot and Casteran [4] for a good introduction). They have their roots in the categorical theory of final coalgebras and have been implemented in the major type theoretic systems, Coq, Agda and Idris [5]. At present, their understanding still suffers from tension between the abstract theory of coalgebras, with its simple characterization by finality, and the formal implementation, with its syntactic conditions [20]. The best synthesis so far is probably in the technique of copatterns [1], which offers an easy syntactic format that transparently mirrors coalgebraic definitions. For our purposes, we skip over syntactic details and issues of unicity, decidability and extensionality.

We use an Agda-style notation, with the key words **data** and **codata** marking inductive and coinductive types. As simple examples of both, here are the definitions of the types of lists and of pure streams.

$$\begin{array}{ll}
\textbf{data}\ \mathsf{List}(A) : \mathsf{Set} & \qquad \textbf{codata}\ \mathbb{S}_A : \mathsf{Set} \\
\quad \mathsf{nil} : \mathsf{List}(A) & \qquad\quad (\triangleleft) : A \to \mathbb{S}_A \to \mathbb{S}_A \\
\quad (::) : A \to \mathsf{List}(A) \to \mathsf{List}(A) &
\end{array}$$

Elements of **data** types are built bottom-up using constructors in a well-founded way. It is necessary to have a non-recursive constructor, nil in this case, to provide a basis for the manufacture of lists. We can define them by directly giving their structure, for example $2 :: 3 :: 5 :: 7 :: 11 :: \mathsf{nil}$.

Elements of **codata** types are built top-down, and there may not be a bottom. We can apply the constructors in a non-well-founded way. We do not need a non-recursive base constructor (but we may have one). Since the structure of coinductive objects can be infinite, we cannot usually define them by directly giving their components. Instead, we use recursive definitions that generate the streams step by step when we unfold them.

Both inductive and coinductive types are fixed points of functors. For the definition to make constructive sense in type theory, the functor must be *strictly positive*, that is, in its syntactic form the argument type must occur only on the right-hand side of functional type formers. A more elegant, less syntax-bound characterization is the notion of *container* [2] (or *dependent polynomial functor* [12] in the categorical literature).

Definition 1: A *container* is a pair $\langle S, P\rangle$ with $S : \mathsf{Set}$, a set of *shapes*, and $P : S \to \mathsf{Set}$, a family giving a set of *positions* for every shape. Every container defines a functor:

$$\begin{aligned}&(S \rhd P) : \mathsf{Set} \to \mathsf{Set}\\&(S \rhd P)\, X = \Sigma s : S.\, P\, s \to X.\end{aligned}$$

An element of $(S \rhd P)\, X$ is a pair $\langle s, xs\rangle$ where $s : S$ is a shape and $xs : P\, s \to X$ is a function assigning an element of X to every position in the shape s.

The carrier of the final coalgebra of a container is a type $\nu(S \rhd P)$ inhabited by trees with nodes labelled by shapes $s : S$ and branches labelled by the positions $(P\, s)$ of the node shape. So every element of $t : \nu(S \rhd P)$ is uniquely given by a shape, $\mathsf{shape}\, t : S$, and a family of sub-elements, $\mathsf{subs}\, t : P\, (\mathsf{shape}\, t) \to \nu(S \rhd P)$.

The actual final coalgebra is the function

$$\begin{aligned}&\mathsf{out}_\nu : \nu(S \rhd P) \to (S \rhd P)\, (\nu(S \rhd P))\\&\mathsf{out}_\nu\, t = \langle \mathsf{shape}\, t, \mathsf{subs}\, t\rangle.\end{aligned}$$

Using this terminology and notation, streams can be defined by, $\mathbb{S}_A = \nu(A \rhd \lambda a.1)$. So streams are the final coalgebra of the container with a shape for every element of A and a single position in each shape. We will continue to use the more intuitive **codata** formalism. The correspondence with the ν formalism should be evident in every case.

Once we defined the coinductive type, we need a formalism to program with it. Categorically, coinductive types are final coalgebras. We can use their universal property as a definitional scheme: Every coalgebra $c : X \to (S \rhd P)\, X$ has a unique *anamorphism* $\hat{c} : X \to \nu(S \rhd P)$ such that $\mathsf{out}_\nu \circ \hat{c} = (S \rhd P)\, \hat{c} \circ c$.

$$\begin{tikzcd}
\nu(S \rhd P) \arrow[r, "\mathsf{out}_\nu"] & (S \rhd P)\, (\nu(S \rhd P)) \\
X \arrow[u, "\hat{c}"] \arrow[r, "c"'] & (S \rhd P)\, X \arrow[u, "(S \rhd P)\, \hat{c}"']
\end{tikzcd}$$

Since $(S \rhd P)\, X = \Sigma s : S.\, P\, s \to X$, the coalgebra c has two components $c = \langle c_S, c_P\rangle$ with $c_s : X \to S$ and $c_P : (x : X) \to (P\, s) \to X$. The commutativity of the diagram can then be expressed by the two equations

$$\begin{aligned}&\mathsf{shape}\, (\hat{c}\, x) = c_S\, x\\&\mathsf{subs}\, (\hat{c}\, x)\, p = \hat{c}\, (c_P\, x\, p).\end{aligned}$$

We will continue to use the more intuitive recursive formalism. The correspondence with coalgebraic definitions should be evident in every case.

The only coinductive structure we are interested in here is that of streams. However, we want to generalize it to monadic streams, in which the constructor shields the head and tail behind a monadic action. The justification of such data formation requires the full range of final coalgebras for containers. Given a monad M, the type of monadic streams on M is defined as follows:

$$\begin{aligned}&\mathbf{codata}\ \ \mathbb{S}_{M,A} : \mathsf{Set}\\&\qquad\quad \mathsf{mcons}_M : M\, (A \times \mathbb{S}_{M,A}) \to \mathbb{S}_{M,A}.\end{aligned}$$

Categorically, we can see this type as the final coalgebra of the functor $F\, X = M\, (A \times X)$. For it to make constructive sense, it should be strictly positive, so we must put it in container form. This is possible only if M itself is a container. Not all monads are; for example the continuation monad is not strictly positive. If M is itself a container, $M = S_M \rhd P_M$, the above functor F can also be presented as a container with shapes $S_F = \Sigma s : S_M.P_M \to A$ and positions $P_F\, \langle s, h\rangle = P_M\, s$. The shapes of F are shapes of M with the positions ornamented [21] by elements of A. From now on we always assume that the monad M is a container. (Thorsten Altenkirch, in a personal communication, showed that monad containers are exactly type universes closed under sum types.)

Pure streams are monadic streams for the identity monad Id: the type of $\mathsf{mcons}_{\mathsf{Id}}$ is isomorphic to that of $(\lhd)$ by currying:

$$\mathsf{mcons}_{\mathsf{Id}} : (A \times \mathbb{S}_{\mathsf{Id},A}) \to \mathbb{S}_{\mathsf{Id},A} \cong A \to \mathbb{S}_{\mathsf{Id},A} \to \mathbb{S}_{\mathsf{Id},A}.$$

Interesting instantiations are obtained by using other monads. Some of them are important in later chapters. If we choose the Maybe monad, we obtain *co-lists*, sequences of elements that may or may not be finite. The elements of $\mathsf{Maybe}\, X$ are copies of each element $x : X$, $\mathsf{Just}\, x$, and an *error* element $\mathsf{Nothing}$.

When we instantiate the definition of monadic streams with Maybe we obtain the type $\mathbb{S}_{\mathsf{Maybe},A}$, with two distinct ways to construct streams (although there is only one constructor) according to the monadic action. If the monadic action is $\mathsf{Nothing}$, we get an *empty stream* object: $\mathsf{nil} = \mathsf{mcons}_{\mathsf{Maybe}}\, \mathsf{Nothing}$. If the monadic action is Just, we get a head element and a tail: $a \lhd \alpha = \mathsf{mcons}_{\mathsf{Maybe}}\, (\mathsf{Just}\, \langle a, \alpha\rangle)$. Both finite lists and pure streams can be injected in $\mathbb{S}_{\mathsf{Maybe},A}$. A list $a_0 :: \cdots :: a_n :: \mathsf{nil}$ is represented as $\mathsf{mcons}\, (\mathsf{Just}\, \langle a_0, \ldots \mathsf{mcons}\, (\mathsf{Just}\, \langle a_n, \mathsf{mcons}\, \mathsf{Nothing}\rangle) \ldots \rangle)$.

The list constructor is itself a monad, so it makes sense to consider $\mathbb{S}_{\mathsf{List},A}$. This turns out to be the set of finitely branching trees with edges labelled by elements of A. An element of it has the form $\mathsf{mcons}\, (\langle a_0, \alpha_0\rangle :: \langle a_1, \alpha_1\rangle :: \langle a_2, \alpha_2\rangle :: \cdots :: \mathsf{nil})$ where a_0, a_1, a_2 are elements of A and α_0, α_1, α_2 are monadic streams in $\mathbb{S}_{\mathsf{List},A}$.

Another interesting instantiation uses the *state monad*. This characterizes computations whose side effects consist in reading and modifying a state value in some type S: $\mathsf{State}_S\, X = S \to X \times S$. A monadic action of type X reads the present state and produces a result in X and a new state. A monadic stream in $\mathbb{S}_{\mathsf{State}_S,A}$ is an infinite sequence of values such that the evaluation of each component depends on and modifies the current state. An element of it has the form $\mathsf{mcons}\, h$, where $h : S \to A \times \mathbb{S}_{\mathsf{State},A} \times S$. As a simple example, here is the state-monadic stream of Fibonacci numbers.

$$\begin{array}{l} \mathsf{fib\_gen} : \mathbb{S}_{\mathsf{State}_{\mathbb{N}\times\mathbb{N}},\mathbb{N}} \\ \mathsf{fib\_gen} = \mathsf{mcons}\,(\lambda\langle a,b\rangle.\langle b, \mathsf{fib\_gen}, \langle b, a+b\rangle\rangle) \end{array}$$

This is in fact a constant stream, in the sense that it recursively calls itself with no variation. It generates a dynamically changing stream when we execute it with a varying state containing the pair of the last two Fibonacci numbers we computed.

$$\begin{array}{ll} \mathsf{runstr} : \mathbb{S}_{\mathsf{State}_S,A} \to S \to \mathbb{S}_A & \\ \mathsf{runstr}\,(\mathsf{mcons}\,h)\,s = & \mathsf{let}\ \langle a, \alpha, s'\rangle = (h\,s) \\ & \mathsf{in}\ a \triangleleft (\mathsf{runstr}\,\alpha\,s') \\ \mathsf{fib} : \mathbb{S}_{\mathbb{N}} & \\ \mathsf{fib} = \mathsf{runstr}\,\mathsf{fib\_gen}\,\langle 0, 1\rangle & \end{array}$$

A special case of the state monad is the *writer* monad. It is a state monad in which the monadic actions only write into the state, they never read it. The state space itself is a monoid $\langle I, e, *\rangle$: the initial state is the unit e and each action produces a value in I that is inserted into the state by the operation $*$. Formally, $\mathsf{Writer}_I\, X = X \times I$. Elements of $\mathbb{S}_{\mathsf{Writer}_I,A}$ are essentially streams of pairs $\langle a_0, i_0\rangle \triangleleft \langle a_1, i_1\rangle \triangleleft \langle a_2, i_2\rangle \triangleleft \cdots$ where $a_n : A$ and $i_n : I$ for every n. (Formally, the order of the arguments is different, because of the way the products associate: $\alpha = \mathsf{mcons}\,\langle\langle a_0, (\mathsf{mcons}\,\langle\langle a_1, (\mathsf{mcons}\,\langle\langle a_2, \ldots\rangle, i_2\rangle)\rangle, i_1\rangle)\rangle, i_0\rangle)$. The intuitive idea is that the evaluation of consecutive elements of the stream will generate successive states i_0, $i_0 * i_1$, $i_0 * i_1 * i_2$, and so on.

Remark on Monad Notation: We use return/bind notation for monads. The operation return lifts a pure value $a : A$ into the monad, $(\mathsf{return}\, a) : M\, A$, and the bind operation takes a monadic action $m : M\, A$ along with a function $f : A \to M\, B$ and binds the results of the action to the function $(m \gg\!= f) : M\, B$. We also use do notation which is a convenient syntax for expressing bind operations. A do block contains a sequence of bindings and expressions, resulting in a monadic value. So, for example:

$$\left(\begin{array}{ll} \mathsf{do} & x_0 \leftarrow m_0 \\ & x_1 \leftarrow m_1 \\ & \mathsf{return}\, e \end{array}\right) \quad \text{means} \quad \begin{array}{l} m_0 \gg\!= (\lambda x_0. \\ m_1 \gg\!= (\lambda x_1. \\ \mathsf{return}\, e)). \end{array}$$

III. Pure Functions

The main subject of this article is the study of pure functions on streams, where *pure* means that the operations of the function do not depend on how the stream is produced. Therefore we require these functions to operate on monadic streams and be polymorphic on the monad. We impose a naturality condition that specifies that the function works in the same way independently of the monad: it must be the same function instantiated to each monad, rather than a collection of different functions, each specifically defined for a particular monad.

Definition 2: Let M_0, M_1 be two monads with respective operators return_0, $\gg\!=_0$ and return_1 and $\gg\!=_1$. A monad morphism is a natural transformation, $\phi : M_0 \dot{\to} M_1$, that respects the monad operations by satisfying the following laws:

$$\begin{array}{l} \phi_A \circ \mathsf{return}_0 = \mathsf{return}_1 \\ \phi_B\,(m \gg\!=_0 f) = (\phi_A\, m) \gg\!=_1 (\phi_B \circ f) \end{array}$$

We want to extend the notion of naturality to monadic stream functions. In order to do this, we must lift monad morphisms to streams, by applying them to every action in the stream.

Definition 3: Given a monad morphism $\phi : M_0 \dot{\to} M_1$, its *lifting to streams* is a family of morphisms on monadic streams

$$\begin{array}{l} \mathbb{S}_{\phi,A} : \mathbb{S}_{M_0,A} \to \mathbb{S}_{M_1,A} \\ \mathbb{S}_{\phi,A}\,(\mathsf{mcons}\, m) = \mathsf{mcons}\,(\hat{\phi}_A\, m) \\ \quad \text{where} \quad \hat{\phi}_A : M_0\,(A \times \mathbb{S}_{M_0,A}) \to M_1\,(A \times \mathbb{S}_{M_1,A}) \\ \qquad\qquad \hat{\phi}_A \quad = \phi_{A\times\mathbb{S}_{M_1,A}} \circ M_0\,(\mathsf{id}_A \times \mathbb{S}_{\phi,A}) \\ \qquad\qquad\qquad = M_1\,(\mathsf{id}_A \times \mathbb{S}_{\phi,A}) \circ \phi_{A\times\mathbb{S}_{M_0,A}}. \\ \qquad\qquad\qquad \text{(equal by naturality of } \phi) \end{array}$$

$$\begin{tikzcd} M_0\,(A \times \mathbb{S}_{M_0,A}) \arrow[r, "M_0\,(\mathsf{id}_A\times\mathbb{S}_{\phi,A})"] \arrow[d, "\phi_{A\times\mathbb{S}_{M_0,A}}"'] \arrow[dr, "\hat{\phi}_A"] & M_0\,(A \times \mathbb{S}_{M_1,A}) \arrow[d, "\phi_{A\times\mathbb{S}_{M_1,A}}"] \\ M_1\,(A \times \mathbb{S}_{M_0,A}) \arrow[r, "M_1\,(\mathsf{id}_A\times\mathbb{S}_{\phi,A})"'] & M_1\,(A \times \mathbb{S}_{M_1,A}) \end{tikzcd}$$

This is a typical corecursive definition: we give equations for a function $\mathbb{S}_{\phi,A}$ that generates a coinductive object in $\mathbb{S}_{M_1,A}$. Circularly, $\mathbb{S}_{\phi,A}$ depends on $\hat{\phi}$ which in turns calls $\mathbb{S}_{\phi,A}$. Careful analysis of the structure of the equations shows that the recursive call generates subobjects of the output streams, while the top structure is given directly. This guarantees that the equation is guarded and the definition productive.

We have now the conceptual framework to specify exactly what it means for a polymorphic function to be uniform on all monads.

Definition 4: A monadic stream function $f : \forall M, \mathbb{S}_{M,A} \to M\, B$ is *natural in* M if this diagram commutes for all monads M_0, M_1 and morphisms $\phi : M_0 \dot{\to} M_1$:

$$\begin{tikzcd} \mathbb{S}_{M_0,A} \arrow[r, "f_{M_0}"] \arrow[d, "\mathbb{S}_{\phi,A}"'] & M_0\, B \arrow[d, "\phi_B"] \\ \mathbb{S}_{M_1,A} \arrow[r, "f_{M_1}"'] & M_1\, B \end{tikzcd} \qquad \phi_B \circ f_{M_0} = f_{M_1} \circ \mathbb{S}_{\phi,A}.$$

Notice that naturality for f implicitly states that all the monadic side effects returned in the output must have been generated by evaluating the monadic actions in the input stream. Only thus changing the effect in the input by a monad morphism ϕ can be equivalent to changing the monadic actions in the input stream by $\mathbb{S}_{\phi,A}$.

Definition 5: A function $g : \mathbb{S}_A \to B$ is *pure* if there exists a monadic stream function $f : \forall M, \mathbb{S}_{M,A} \to M\, B$ natural in M, such that $g = f_{\mathsf{Id}}$.

As an example, we define the function on streams that returns the index of the first non-decreasing element:

$$\begin{array}{l} \mathsf{nodec} : \mathbb{S}_{\mathbb{N}} \to \mathbb{N} \\ \mathsf{nodec}\,(x_0 \triangleleft x_1 \triangleleft \alpha) = \begin{cases} 1 & \text{if } x_0 \leq x_1 \\ \mathsf{nodec}\,(x_1 \triangleleft \alpha) + 1 & \text{otherwise.} \end{cases} \end{array}$$

This function is pure: we can easily generalize it to all monadic streams:

$$
\begin{aligned}
&\mathsf{nodec} : \forall M, \mathbb{S}_{M,\mathbb{N}} \to M\,\mathbb{N} \\
&\mathsf{nodec}\,(\mathsf{mcons}\,m_0) = \mathsf{do}\ \ \langle x, \alpha\rangle \leftarrow m_0 \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{nodec}'\ x\ \alpha \\
&\mathsf{nodec}' : \mathbb{N} \to \forall M, \mathbb{S}_{M,\mathbb{N}} \to M\,\mathbb{N} \\
&\mathsf{nodec}'\,x_0\,(\mathsf{mcons}\,m_0) = \mathsf{do} \\
&\qquad \langle x_1, \alpha_1\rangle \leftarrow m_0 \\
&\qquad \mathsf{if}\ x_0 \le x_1\ \ \mathsf{then\ return}\ 1 \\
&\qquad\qquad\qquad\quad\ \mathsf{else}\ M\,(+1)\,(\mathsf{nodec}'\,x_1\,\alpha_1)
\end{aligned}
$$

When instantiated with the identity monad, this function is equal to the previous definition on pure streams. If we instantiate it to the Maybe monad, it becomes a function on colists: it returns Just of the index of the first non-decreasing element, if such element exists; if the colist is finite and decreasing, it returns Nothing.

If we instantiate nodec with the list monad, it becomes a function on (non-well-founded) finitely branching trees, returning a list of values. The function follows all the paths in the tree in a left-to-right lexicographic order, it returns the indices of the first non-decreasing element of each path, when they exist (some paths may be finite and decreasing).

One might ask why we didn't define nodec as follows:

$$
\begin{aligned}
&\mathsf{noend} : \forall M, \mathbb{S}_{M,\mathbb{N}} \to M\,\mathbb{N} \\
&\mathsf{noend}\,(\mathsf{mcons}\,m_0) = \mathsf{do} \\
&\quad \langle x_0, \mathsf{mcons}\,m_1\rangle \leftarrow m_0 \\
&\quad \langle x_1, \mathsf{mcons}\,m_2\rangle \leftarrow m_1 \\
&\quad \mathsf{if}\ x_0 \le x_1\ \ \mathsf{then\ return}\ 1 \\
&\qquad\qquad\qquad\ \mathsf{else}\ M\,(+1)\,(\mathsf{noend}\,(\mathsf{mcons}\,m_1))
\end{aligned}
$$

This new function noend is directly recursive, taking the tail of the monadic stream as its argument. If we instantiate noend with the Id monad it behaves as the original nodec. However, noend is actually not well-founded: while the definition of nodec' is clearly recursive on its first argument, no such recursion justifies noend. To see this consider the following stream on the State monad:

$$
\begin{aligned}
&\mathsf{flipper} : \mathbb{S}_{\mathsf{State}_{\mathbb{N}},\mathbb{N}} \\
&\mathsf{flipper} = \mathsf{mcons}\,(\lambda n.\langle\langle n, \mathsf{flipper}\rangle, \mathsf{mod}\,(n+1)\,2\rangle)
\end{aligned}
$$

The flipper stream returns the current state, irrespective of the position in the stream, and then repeatedly flips the state between 0 and 1. If we consider (noend flipper 1), we find it does not terminate. Each call to noend reads two decreasing values, 1 and 0, and the state returns to its initial value of 1. The recursive call is applied to the same stream, flipper, and therefore the evaluation loops.

The problem arises because we evaluate a monadic action twice: while this returns the same value for some monads, for example Id and List, for others, for example State, we may get different results. The monadic nodec avoids this problem by only reading each monadic action once and then passing the value as an argument for further comparisons.

IV. Monadic Continuity

Now we would like to prove that pure functions are necessarily continuous.

We begin with a more modest task: Can we test when a pure function is (syntactically) constant? Deciding the constancy of a stream function is in general undecidable, but we are interested in detecting functions that return a value without even reading any of their input.

Theorem 1: Let $f : \forall M, \mathbb{S}_{M,A} \to M\,B$ be natural in M. If there exists $b : B$ such that $f_{\mathsf{Maybe}}\,(\mathsf{mcons\,Nothing}) = \mathsf{Just}\,b$, then for every monad M and every $\alpha : \mathbb{S}_A$, $f_M\,\alpha = \mathsf{return}_M\,b$.

Proof: There is no general monad morphism between Maybe and a generic monad M. To bridge the gap between the instantiation of f for Maybe and form M, we use an intermediate instantiation of f with an error monad, Error_M, whose error values are the monadic actions of M:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Error}_M\ A : \mathsf{Set} \\
&\quad \mathsf{Pure} : A \to \mathsf{Error}_M\ A \\
&\quad \mathsf{Throw} : M\ A \to \mathsf{Error}_M\ A
\end{aligned}
$$

with the following return and $\gg\!=$ operators:

$$
\begin{aligned}
&\mathsf{return} = \mathsf{Pure} \\
&\mathsf{Pure}\ a \gg\!= f = f\ a \\
&\mathsf{Throw}\ m \gg\!= f = \mathsf{Throw}\ (m \gg\!=_M (\mathsf{merge} \circ f)) \\
&\text{where}\ \ \mathsf{merge}_M : \mathsf{Error}_M \stackrel{\cdot}{\to} M \\
&\qquad\quad\ \mathsf{merge}_{M,A}(\mathsf{Pure}\,a) = \mathsf{return}\,a \\
&\qquad\quad\ \mathsf{merge}_{M,A}(\mathsf{Throw}\,m) = m.
\end{aligned}
$$

The $\mathsf{merge}_{M,A}$ function is a monad morphism between Error_M and M that merges the Throw and Pure values into the monad M. This is used in the bind operation for Error_M but also provides the link in our proof between the bridging monad Error_M and the monad M. The second link, between Error_M and Maybe, is provided by a second monad morphism that forgets the monadic value in Throw:

$$
\begin{aligned}
&\mathsf{forget} : \mathsf{Error}_M \stackrel{\cdot}{\to} \mathsf{Maybe} \\
&\mathsf{forget}_A\ (\mathsf{Pure}\,a) = \mathsf{Just}\,a \\
&\mathsf{forget}_A\ (\mathsf{Throw}\,m) = \mathsf{Nothing}.
\end{aligned}
$$

Naturality of f in the monad tells us that the following two squares commute:

$$
\begin{tikzcd}
\mathbb{S}_{\mathsf{Maybe},A} \arrow[r, "f_{\mathsf{Maybe}}"] & \mathsf{Maybe}\,B \\
\mathbb{S}_{\mathsf{Error}_M,A} \arrow[u, "\mathbb{S}_{\mathsf{forget},A}"] \arrow[r, "f_{\mathsf{Error}_M}"] \arrow[d, "\mathbb{S}_{\mathsf{merge},A}"'] & \mathsf{Error}_M\,B \arrow[u, "\mathsf{forget}_B"'] \arrow[d, "\mathsf{merge}_B"] \\
\mathbb{S}_{M,A} \arrow[r, "f_M"] & M\,B
\end{tikzcd}
$$

$$
\begin{aligned}
\mathsf{forget}_B \circ f_{\mathsf{Error}_M} &= f_{\mathsf{Maybe}} \circ \mathbb{S}_{\mathsf{forget},A} \\
\mathsf{merge}_B \circ f_{\mathsf{Error}_M} &= f_M \circ \mathbb{S}_{\mathsf{merge},A}.
\end{aligned}
$$

We can lift a monadic stream into the Error_M monad with the stream function $\mathbb{S}_{\mathsf{Throw},A}$ (note that Throw is not a monad morphism; however, it can be lifted to streams in the same way). Observe that since $\mathsf{forget}\,(\mathsf{Throw}\,m) = \mathsf{Nothing}$ for every m, then $\mathbb{S}_{\mathsf{forget},A}\,(\mathbb{S}_{\mathsf{Throw},A}\,\alpha) = \mathsf{mcons\,Nothing}$ for every α.

It is immediate that $\mathsf{merge} \circ \mathsf{Throw} = \mathsf{id}$ and subsequently $\mathbb{S}_{\mathsf{merge},A} \circ \mathbb{S}_{\mathsf{Throw},A} = \mathsf{id}$. By commutativity of the upper square we have:

$$\begin{aligned} \mathsf{forget}_B\,(f_{\mathsf{Error}_M}(\mathbb{S}_{\mathsf{Throw},A}\,\alpha)) &= f_{\mathsf{Maybe}}\,(\mathbb{S}_{\mathsf{forget},A}\,(\mathbb{S}_{\mathsf{Throw},A}\,\alpha)) \\ &= f_{\mathsf{Maybe}}\,(\mathsf{mcons}\,\mathsf{Nothing}) \\ &= \mathsf{Just}\,b \end{aligned}$$

where the last step is the assumption. Since forget_B only returns a Just value for Pure inputs, it follows that $f_{\mathsf{Error}_M}\ (\mathbb{S}_{\mathsf{Throw},A}\,\alpha) = \mathsf{Pure}\,b$. Commutativity of the lower square then gives:

$$\begin{aligned} f_M\,\alpha &= f_M(\mathbb{S}_{\mathsf{merge},A}\,(\mathbb{S}_{\mathsf{Throw},A}\,\alpha)) \\ &= \mathsf{merge}_B\,(f_{\mathsf{Error}_M}\,(\mathbb{S}_{\mathsf{Throw},A}\,\alpha)) \\ &= \mathsf{merge}_B\,(\mathsf{Pure}\,b) \\ &= \mathsf{return}_M\,b \end{aligned}$$

■

This gives us an effective test for syntactic constancy of pure functions. A similar idea helps us to prove that f must be continuous. (In fact, continuity can probably be obtained as a direct consequence of decidability of syntactic constancy [6, Section 7].) We can find a monadic instantiation that forces f to compute the modulus of continuity at every stream.

We use the writer monad with the monoid $\langle \mathbb{N}, 0, \max\rangle$. A monadic stream for this monad, $\alpha : \mathbb{S}_{\mathsf{Writer}_{\mathbb{N}},A}$ is essentially a stream of pairs of elements of A and natural numbers: $\alpha \sim \langle a_0, n_0\rangle \lhd \langle a_1, n_1\rangle \lhd \langle a_2, n_2\rangle \lhd \cdots$. When we evaluate the elements of the stream, we *write out* the maximum value of the n_is of all the read elements.

We lift each pure stream $\alpha = a_0 \lhd a_1 \lhd a_2 \cdots : \mathbb{S}_A$ to a writer-monadic stream that decorates each element with its index: $\alpha_\iota \sim \langle a_0, 0\rangle \lhd \langle a_1, 1\rangle \lhd \langle a_2, 2\rangle \lhd \cdots$. Formally we can define it as follows:

$$\begin{aligned} &-_\iota : \mathbb{S}_A \to \mathbb{S}_{\mathsf{Writer}_{\mathbb{N}},A} \\ &\alpha_\iota = \mathsf{index}\,0\,\alpha \\ &\quad \text{where } \mathsf{index} : \mathbb{N} \to \mathbb{S}_A \to \mathbb{S}_{\mathsf{Writer}_{\mathbb{N}},A} \\ &\qquad\quad \mathsf{index}\,n\,(a_0 \lhd \alpha) = \mathsf{mcons}\,\langle\langle a_0, \mathsf{index}\,(n+1)\,\alpha\rangle, n\rangle \end{aligned}$$

According to the intuitive explanation, when we evaluate a monadic stream inside this Writer monad, we should obtain the maximum index of the elements that were actually read. This gives an immediate suggestion for the computation of the continuity modulus.

$$\begin{aligned} &\mathsf{modulus} :: (\forall M, \mathbb{S}_{M,A} \to M\,B) \to \mathbb{S}_A \to \mathbb{N} \\ &\mathsf{modulus}\,f\,\alpha = \pi_1\,(f_{\mathsf{Writer}\mathbb{N}}\,\alpha_\iota) + 1 \end{aligned}$$

A variant of the proof of Theorem 1 shows that this function indeed computes a correct modulus of continuity.

Theorem 2: Let $f : \forall M, \mathbb{S}_{M,A} \to M\,B$ be natural in M. The function $(\mathsf{modulus}\,f)$ computes a correct modulus of continuity for f_{Id}, that is

$$\forall \alpha, \alpha' : \mathbb{S}_A, \alpha' =_{(\mathsf{modulus}\,f\,\alpha)} \alpha \Rightarrow f_{\mathsf{Id}}\,\alpha' = f_{\mathsf{Id}}\,\alpha.$$

Proof: We can formulate this continuity conjecture in a slightly different way by extending the claim about Maybe-monadic streams. We use now the $\mathsf{Writer}_{\mathbb{N}}$ monad in place Error_M in Theorem 1. There we used it as a mediation step from M-streams to Maybe-streams, injecting all M-actions into Throw elements to force its truncation to the empty list. Now we can do a similar operation, but truncating a stream $\alpha = a_0 \lhd a_1 \lhd a_2 \lhd \cdots$ after the m-th element, where $m+1 = (\mathsf{modulus}\,f\,\alpha)$. Define, as before, two monad morphisms from the writer monad to Maybe and Id:

$$\phi : \mathsf{Writer}_{\mathbb{N}} \dot{\to} \mathsf{Maybe} \qquad\qquad \psi : \mathsf{Writer}_{\mathbb{N}} \dot{\to} \mathsf{Id}$$

$$\phi_X\,\langle x, n\rangle = \begin{cases} \mathsf{Just}\,x & \text{if } n \le m \\ \mathsf{Nothing} & \text{otherwise} \end{cases} \qquad \psi_X\,\langle x, n\rangle = x$$

Naturality of f in the monad tells us that the following two squares commute:

$$\begin{tikzcd} \mathbb{S}_{\mathsf{Maybe},A} \arrow[r, "f_{\mathsf{Maybe}}"] & \mathsf{Maybe}\,B \\ \mathbb{S}_{\mathsf{Writer}_{\mathbb{N}},A} \arrow[u, "\mathbb{S}_{\phi,A}"] \arrow[r, "f_{\mathsf{Writer}_{\mathbb{N}}}"] \arrow[d, "\mathbb{S}_{\psi,A}"'] & \mathsf{Writer}_{\mathbb{N}}\,B \arrow[u, "\phi_B"'] \arrow[d, "\psi_B"] \\ \mathbb{S}_A \arrow[r, "f_{\mathsf{Id}}"'] & B \end{tikzcd}$$

$$\begin{aligned} \phi_B \circ f_{\mathsf{Writer}_{\mathbb{N}}} &= f_{\mathsf{Maybe}} \circ \mathbb{S}_{\phi,A} \\ \psi_B \circ f_{\mathsf{Writer}_{\mathbb{N}}} &= f_{\mathsf{Id}} \circ \mathbb{S}_{\psi,A}. \end{aligned}$$

Starting with $\alpha_\iota : \mathbb{S}_{\mathsf{Writer}_{\mathbb{N}},A}$, we know that $(f_{\mathsf{Writer}_{\mathbb{N}}}\,\alpha_\iota) = \langle b, m\rangle$ for some $b : B$. Using the commutativity of the lower rectangle, we discover that

$$\begin{aligned} f_{\mathsf{Id}}\,\alpha &= f_{\mathsf{Id}}\,(\mathbb{S}_{\psi,A}\,\alpha_\iota) \\ &= \psi_B\,(f_{\mathsf{Writer}_{\mathbb{N}}}\,\alpha_\iota) \\ &= \psi_B\,\langle b, m\rangle \\ &= b. \end{aligned}$$

Following the upper route, by the definition of ϕ and of its lifting to streams, we have that $(\mathbb{S}_{\phi,A}\,\alpha_\iota) = [a_0, \ldots, a_m]$; so we discover that $(f_{\mathsf{Maybe}}\,[a_0, \ldots, a_m]) = \phi_B\,\langle b, m\rangle = \mathsf{Just}\,b$.

If we now take another pure stream α' such that $\alpha' =_m \alpha$, we also have $(\mathbb{S}_{\phi,A}\,\alpha'_\iota) = [a_0, \ldots, a_m]$. This forces $(f_{\mathsf{Writer}_{\mathbb{N}}}\,\alpha'_\iota) = \langle b, m'\rangle$ for some m' and consequently $(f_{\mathsf{Id}}\,\alpha') = b$. ■

V. Monadic Approximations

Theorem 2 shows that a pure function has a modulus of continuity. We may ask if this is true at a more abstract level: is there a general notion of "modulus" that makes sense for any monad and can we construct it for every polymorphic function on monadic streams? Continuity is achieved by showing that a finite initial segment $\alpha|_n$ of the stream is sufficient to determine the value of $(f_{\mathsf{Id}}\,\alpha)$. For a general monad, we may look at well-founded approximations of the monadic stream. For example, streams with respect to the list monad are finitely branching non-well-founded trees; an approximation is obtained by cutting each branch at a finite depth. This, however, does not work in general: for the list monad we get indeed an version of Theorem 2 stating that the value of $f_{\mathsf{List}}\,\alpha$ depends only on a finite approximation; however, for other non-discrete monads, for example the state monad, this is not true: the result of f may indeed depend on an infinite amount of information from α.

The definition of the modulus function uses the writer monad to produce the index of the furthest read element. The first step in that direction was α_ι, where we decorated each entry in the stream with its index. With a small variation of the definition, we can decorate each entry with the list of elements of the stream up to that point. We use the writer monad with the list monoid $\langle \mathsf{List}(A), *, \mathsf{nil}\rangle$, taking as monoid operation the longer of two lists, giving preference to the second:

$$\begin{aligned}&\mathsf{nil} * l = l\\ &l * \mathsf{nil} = l\\ &(a_0 :: as) * (b_0 :: bs) = b_0 :: (as * bs).\end{aligned}$$

We lift every stream to one inside $\mathbb{S}_{\mathsf{Writer}_{\mathsf{List}(A)},A}$ by decorating every element with the initial segment of the stream leading to it.

$$\begin{aligned}&-_\kappa : \mathbb{S}_A \to \mathbb{S}_{\mathsf{Writer}_{\mathsf{List}(A)},A}\\ &\alpha_\kappa = \mathsf{decorate}\,\mathsf{nil}\,\alpha\\ &\quad \text{where } \mathsf{decorate} : \mathsf{List}(A) \to \mathbb{S}_A \to \mathbb{S}_{\mathsf{Writer}_{\mathbb{N}},A}\\ &\qquad\quad \mathsf{decorate}\, l\, (a_0 \lhd \alpha) =\\ &\qquad\qquad \mathsf{mcons}\,\langle\langle a_0, \mathsf{decorate}\,(l +\!\!+ [a_0])\,\alpha\rangle, (l +\!\!+ [a_0])\rangle\end{aligned}$$

Intuitively we have that

$$\alpha_\kappa \sim \langle a_0, [a_0]\rangle \lhd \langle a_1, [a_0, a_1]\rangle \lhd \langle a_2, [a_0, a_1, a_2]\rangle \lhd \cdots .$$

We can directly extract the prefix sufficient for the computation of a monadic function with a modification of the modulus:

$$\begin{aligned}&\mathsf{approx} :: (\forall M, \mathbb{S}_{M,A} \to M\,B) \to \mathbb{S}_A \to \mathsf{List}(A)\\ &\mathsf{approx}\, f\, \alpha = \pi_1\,(f_{\mathsf{WriterList}(A)}\,\alpha_\kappa).\end{aligned}$$

Let us write $\alpha \Vdash l$ to state that the list l is an initial segment of the stream α, that is, l approximates α (notation inspired by formal topology [9], [24]). An alternative formulation of continuity states that this approximation is sufficient to determine the result.

Theorem 3: Let $f : \forall M, \mathbb{S}_{M,A} \to M\,B$ be natural in M.

$$\forall \alpha, \alpha' : \mathbb{S}_A, \alpha' \Vdash (\mathsf{approx}\,\alpha) \Rightarrow f_{\mathsf{Id}}\,\alpha' = f_{\mathsf{Id}}\,\alpha.$$

Proof: By a modification of the proof of Theorem 2. ■

Let us generalize the definition of approximations to any monad. For container-monad, $\mathbb{S}_{M,A}$ is a set of non-well-founded trees. An approximation consists of a well-founded trees with leaves representing unknown information.

$$\begin{aligned}&\mathbf{data}\ \ \mathbb{L}_{M,A} : \mathsf{Set}\\ &\qquad \mathsf{unknown}_M : \mathbb{L}_{M,A}\\ &\qquad \mathsf{lcons}_M : M\,(A \times \mathbb{L}_{M,A}) \to \mathbb{L}_{M,A}.\end{aligned}$$

The approximation relation between $\mathbb{S}_{M.A}$ and $\mathbb{L}_{M,A}$ is defined inductively by the following rules.

$$\frac{}{\alpha \Vdash \mathsf{unknown}} \qquad \frac{m \Vdash_M l}{\mathsf{mcons}\, m \Vdash \mathsf{lcons}\, l}$$

where $\Vdash_M$ is the *lifting* of $\Vdash$ to the monad, a relation between $M\,(A \times \mathbb{S}_{M,A})$ and $M\,(A \times \mathbb{L}_{M,A}$. If M is a container, $M = \langle S, P\rangle$, the relation states that the elements must have the same shape and related components in corresponding positions (with the same A-elements): if $m = \langle s_m, h_m$ with $s_m : A$ and $h_m : P\,s_m \to A \times \mathbb{S}_{M,A}$ and if $l = \langle s_l, h_l$ with $s_l : A$ and $h_l : P\,s_l \to A \times \mathbb{L}_{M,A}$, then

$$\begin{aligned}m \Vdash_M l \quad \text{iff} \quad & s_m = s_l \land\\ & \forall p : P\,s_m,\ \ \pi_0\,(h_m\,p) = \pi_0\,(h_l\,p) \land\\ &\qquad\qquad\quad \pi_1\,(h_m\,p) \Vdash \pi_1\,(h_l\,p).\end{aligned}$$

This allows us to formulate a continuity principle for every monad.

Definition 6: Let $f : \forall M, \mathbb{S}_{M,A} \to M\,B$ be natural in M and let M be a specific monad, the *continuity principle for* M states that

$$\begin{aligned}&\forall \alpha : \mathbb{S}_{M,A}, \exists l : \mathbb{L}_{M,A}, \alpha \Vdash l \land\\ &\quad \forall \alpha' : \mathbb{S}_{M,A}, \alpha' \Vdash l \Rightarrow f_M\,\alpha' = f_M\,\alpha.\end{aligned}$$

Although it may be possible, for finitary monads (containers for which every shape has a finite number of positions), to generalize the definition of approx and the proof of Theorem 2, the continuity principle is not true in general. As a counterexample, consider the following function and observe what happens when we apply it to a stream in the state monad.

$$\begin{aligned}&\mathsf{misTake} : \forall M, \mathbb{S}_{M,\mathbb{N}} \to M[\mathbb{N}]\\ &\mathsf{misTake}_M\,(\mathsf{mcons}\, m) = \mathsf{do}\ \ (n, \_) \leftarrow m\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{take}_M\, n\,(\mathsf{mcons}\, m)\end{aligned}$$

$$\begin{aligned}&\mathsf{take} : \forall M, \mathbb{N} \to \mathbb{S}_{M,A} \to M[A]\\ &\mathsf{take}_M\, 0\, \alpha = \mathsf{return}\, []\\ &\mathsf{take}_M\,(\mathsf{suc}\, n)\,(\mathsf{mcons}\, m) = \mathsf{do}\ \ (a, m') \leftarrow m\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad M\,(a ::)\,(\mathsf{take}_M\, n\, m)\end{aligned}$$

The function misTake reads the first element of the stream, n, and returns the first n elements of the stream, including the head n itself.

Now consider the following set of streams in the state monad.

$$\begin{aligned}&\mathsf{goleft} : \mathbb{S}_{\mathsf{Id},\mathbb{N}} \to \mathbb{S}_{\mathsf{State}_{\mathbb{N}},\mathbb{N}}\\ &\mathsf{goleft}\,\alpha = \mathsf{mcons}\,(\lambda n. \langle n, \mathsf{ifzero}\, n\,(\mathsf{atleft}\,\alpha)\,\bar{0}, 0\rangle)\\ &\quad \text{where } \mathsf{atleft} : \mathbb{S}_{\mathsf{Id},\mathbb{N}} \to \mathbb{S}_{\mathsf{State}_{\mathbb{N}},\mathbb{N}}\\ &\qquad\qquad \mathsf{atleft}\,(a :: \alpha) = \mathsf{mcons}\,(\lambda n.\\ &\qquad\qquad\qquad \mathsf{ifzero}\, n\,\langle a, \mathsf{atleft}\,\alpha, 0\rangle\,\langle 0, \bar{0}, n\rangle)\\ &\bar{0} : \forall M. \mathbb{S}_{M,\mathbb{N}}\\ &\bar{0}_M = \mathsf{mcons}\,(\mathsf{return}\,\langle 0, \bar{0}_M\rangle)\end{aligned}$$

The first monadic state action of $\mathsf{goleft}\,\alpha$ returns the initial state and changes the state to 0. The tail depends of the value of the state: if it is non-zero, we return the zero stream; if it is zero, we return the first element of α and recurse. If we picture an element of $\mathbb{S}_{\mathsf{State}_{\mathbb{N}},\mathbb{N}}$ as a tree in which every node branches according to the state value, then $\mathsf{goleft}\,\alpha$ has the value of the state in the the nodes at depth one, then it is zero everywhere except on the leftmost spine, where it contains the elements of α.

The counterexample is made by applying a stream $\mathsf{goleft}\,\alpha$ to misTake. The result of an application is a function which when applied to initial state n returns the first n elements of the left spine of $\mathsf{goleft}\,\alpha$. Equationally:

$$\mathsf{misTake}\,(\mathsf{goleft}\,\alpha)\, n = 0 :: \alpha|_{n-1} \tag{1}$$

This contradicts the continuity principle because an arbitrarily large amount of the left spine can be read depending on the initial state - any well-founded approximation of $\mathsf{goleft}\,\alpha$ would have to truncate this spine. We note that it is important for the counterexample that misTake re-evaluates the first monadic action, as otherwise the left-most spine would not be returned.

Next we give two important relations between the monadic stream $\mathsf{goleft}\,\alpha$ and its approximations, before giving a proof of the contradiction. . First we define leftof which takes the left spine of an approximation:

$$\begin{aligned}
&\mathsf{leftof} :: \mathbb{L}_{\mathsf{State}_{\mathbb{N}},A} \to \mathbb{L}_{\mathsf{Id},A} \\
&\mathsf{leftof}\,(\mathsf{lcons}\,m) = \mathsf{let}\ \langle a, \alpha, _\rangle = m\,0\ \mathsf{in}\ a :: \mathsf{leftof}\,\alpha \\
&\mathsf{leftof}\ \mathsf{unknown} = \mathsf{unknown}
\end{aligned}$$

Firstly, for any approximation of a goleft stream, leftof gives us a truncation of the left-most spine:

$$\mathsf{goleft}\,\alpha \Vdash l \implies \exists n.\mathsf{leftof}\,l = 0 :: \alpha|_{n-1} \tag{2}$$

Secondly, two goleft streams only differ on their left spine. This can be stated using approximations as so:

$$\mathsf{goleft}\,\alpha \Vdash l \wedge (0 \lhd \alpha') \Vdash \mathsf{leftof} l \implies \mathsf{goleft}\,\alpha' \Vdash l \tag{3}$$

We are now ready to prove that the continuity principle does not hold for $\mathsf{State}_{\mathbb{N}}$. Suppose, towards a contradiction, that l is an approximation to $\mathsf{goleft}\,\bar{0}$ which satisfies the continuity principle for misTake. Using equation 2 we have n such that:

$$\mathsf{leftof}\,l = 0 \lhd \bar{0}|_{n-1} = \bar{0}|_n$$

Take $\alpha = \bar{0}|_{n-1} +\!\!+\ \bar{1}$ then it follows directly that $0 \lhd \alpha \Vdash \mathsf{leftof}\,l$. Using equation 3 we have that $\mathsf{goleft}\,\alpha$ is approximated by l, i.e. $\mathsf{goleft}\alpha \Vdash l$. We can now apply the continuity principle to give:

$$\mathsf{misTake}\,(\mathsf{goleft}\,\alpha) = \mathsf{misTake}\,(\mathsf{goleft}\,\bar{0})$$

However using (1) we have

$$\mathsf{misTake}\,(\mathsf{goleft}\,\bar{0})(n+1) = 0 :: \bar{0}|_n$$

but

$$\mathsf{misTake}\,(\mathsf{goleft}\,\alpha)(n+1) = 0 :: \alpha|_n = 0 :: \bar{0}|_{n-1} +\!\!+\ [1]$$

which contradicts the equality given by the continuity principle.

VI. Dialogue Trees

In their characterization of continuous functions on streams Ghani, Hancock and Pattinson [13], [14] define a type of *dialogue trees* whose paths represent the successive calls that an effective procedure makes on the elements of a stream. The idea of a data structure representing interaction goes back to Brouwer and was used by Kleene in his article about higher-order functionals [19]. Martín Escardó used them to give a concise proof of continuity of the functional of system T [10]. They are called *strategy trees* by Bauer et al. [3], who use them to characterize pure stream programs in ML-like languages.

A dialogue tree has nodes representing a single interaction with the input stream or the return of an output result. Since the procedure must be terminating, the tree must be well-founded.

$$\begin{aligned}
\mathbf{data}\ \ &\mathsf{Dialogue}_{A,B} : \mathsf{Set} \\
&\quad \mathsf{Answer} : B \to \mathsf{Dialogue}_{A,B} \\
&\quad \mathsf{Ask} : (A \to \mathsf{Dialogue}_{A,B}) \to \mathsf{Dialogue}_{A,B}
\end{aligned}$$

A tree can have two forms: an immediate answer $(\mathsf{Answer}\,b)$ corresponding to a constant function returning b without reading any element of the input stream, or a branching construction $(\mathsf{Ask}\,h)$ with $h : A \to \mathsf{Dialogue}_{A,B}$ corresponding to a function that reads the next element of input stream, a, and then continues as $(h\,a)$. This leads to the definition of the evaluation function that associates a continuous function to a tree, by structural recursion on the tree:

$$\begin{aligned}
&\mathsf{eval} : \mathsf{Dialogue}_{A,B} \to \mathbb{S}_A \to B \\
&\mathsf{eval}\,(\mathsf{Answer}\,b)\,\alpha = b \\
&\mathsf{eval}\,(\mathsf{Ask}\,h)\,(a \lhd \alpha) = \mathsf{eval}\,(h\,a)\,\alpha.
\end{aligned}$$

The function $(\mathsf{eval}\,t)$ is continuous for every tree t. The vice versa is challenging. In their paper, Ghani et al. prove it negatively: If a function has no representation as a dialogue tree then it is not continuous. In recent work, Tarmo Uustalu and one of us [7] gave a constructive proof based on Brouwer's Bar Induction Principle.

The evaluation operation is readily generalized to monadic streams:

$$\begin{aligned}
&\mathsf{meval} : \mathsf{Dialogue}_{A,B} \to \forall M, \mathbb{S}_{M,A} \to M\,B \\
&(\mathsf{meval}\,(\mathsf{Answer}\,b))_M\ \alpha = \mathsf{return}\,b \\
&(\mathsf{meval}\,(\mathsf{Ask}\,h))_M\ (\mathsf{mcons}\,m) = \\
&\qquad\qquad m \gg\!\!= \lambda\langle a, \alpha\rangle.(\mathsf{meval}\,(h\,a))_M\,\alpha
\end{aligned}$$

Also in this case, the vice versa is the challenging direction of the correspondence. How can we construct a dialogue tree from a monadic stream function? We can observe that the dialogue tree type former $\mathsf{Dialogue}_A$ is itself a monad on B:

$$\begin{aligned}
&\mathsf{return} = \mathsf{Answer} \\
&(\mathsf{Answer}\,b) \gg\!\!= g = g\ b \qquad\qquad l \\
&(\mathsf{Ask}\,h) \gg\!\!= g = \mathsf{Ask}\,(\lambda a.h\,a \gg\!\!= g).
\end{aligned}$$

So it makes sense to consider streams on it, $\mathbb{S}_{\mathsf{Dialogue}_A,A}$. These streams actually correspond to the representation of *stream processors* in another work by Ghani, Hancock and Pattinson [15]. In particular, we can represent the *identity processor* that repeatedly reads an element from the input stream and immediately returns it to the output streams:

$$\begin{aligned}
&\iota_A : \mathbb{S}_{\mathsf{Dialogue}_A,A} \\
&\iota_A = \mathsf{mcons}\,(\mathsf{Ask}\,(\lambda a.\mathsf{Answer}\,\langle a, \iota_A\rangle)).
\end{aligned}$$

Using ι_A we can define a tabulation function, which converts a monadic stream function into a dialogue tree:

$$\begin{aligned}
&\mathsf{tabulate} : (\forall M,\ \mathbb{S}_{M,A} \to M\,B) \to \mathsf{Dialogue}_{A,B} \\
&\mathsf{tabulate}\,f = f_{\mathsf{Dialogue}_A}\,\iota_A
\end{aligned}$$

Theorem 4: The function tabulate is a left inverse of the evaluation operation, meval:

$$\forall t : \mathsf{Dialogue}_{A,B}, \mathsf{tabulate}\,(\mathsf{meval}\,t) = t.$$

Proof: By induction on the dialogue tree t.

In the base case the dialogue tree is of the form $\mathsf{Answer}\,b$, we have:

$$\begin{aligned}
\mathsf{tabulate}\,(\mathsf{meval}\,(\mathsf{Answer}\,b)) &= (\mathsf{meval}\,(\mathsf{Answer}\,b))_{\mathsf{Dialogue}_A}\ \iota_A \\
&= \mathsf{return}\,b = \mathsf{Answer}\,b
\end{aligned}$$

In the inductive case the dialogue tree is of the form $\mathsf{Ask}\,h$, we have:

$$\begin{aligned}
&\mathsf{tabulate}\,(\mathsf{meval}\,(\mathsf{Ask}\,h))\\
&\quad = (\mathsf{meval}\,(\mathsf{Ask}\,h))_{\mathsf{Dialogue}_A}\ \iota_A\\
&\quad = (\mathsf{Ask}\,(\lambda a.\mathsf{Answer}\,\langle a,\iota_A\rangle)) \gg\!=\\
&\qquad\qquad\qquad \lambda\langle a,\alpha\rangle.(\mathsf{meval}\,(h\,a))_M\,\alpha\\
&\quad = \mathsf{Ask}\,(\lambda a.\mathsf{Answer}\,\langle a,\iota_A\rangle \gg\!=\\
&\qquad\qquad\qquad \lambda\langle a,\alpha\rangle.(\mathsf{meval}\,(h\,a))_M\,\alpha)\\
&\quad = \mathsf{Ask}\,(\lambda a.(\mathsf{meval}\,(h\,a))_{\mathsf{Dialogue}_A}\,\iota_A)\\
&\quad = \mathsf{Ask}\,(\lambda a.\mathsf{tabulate}(\mathsf{meval}\,(h\,a)))\\
&\quad = \mathsf{Ask}\,h \quad \text{(by induction hypothesis).}
\end{aligned}$$

■

The tabulate function however is not the right inverse of meval, as $\mathsf{meval}(\mathsf{tabulate}\,f)$ is not necessarily the same function as f. This can be seen already on pure streams α, $\mathsf{eval}\,(f_{\mathsf{Dialogue}_A}\,\iota_A)\,\alpha$ is not always equal to $f_{\mathsf{Id}}\,\alpha$, as we can see from the following counterexample by Paolo Capriotti:

$$\begin{aligned}
&\mathsf{asktwice} : \forall M, \mathbb{S}_{M,A} \to M\,A\\
&\mathsf{asktwice}_M\,(\mathsf{mcons}\,m) = \mathsf{do}\ \ \langle a_0,\alpha_0\rangle \leftarrow m\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \langle a_1,\alpha_1\rangle \leftarrow m\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathsf{return}\,a_1.
\end{aligned}$$

This function executes the monadic action of the stream twice, so reading the head of the stream twice, then returns the second value that it obtained. On pure streams this simply means that it reads the first value twice, a_0 and a_1 are the same: $\mathsf{asktwice}_{\mathsf{Id}}\,(a_0 \triangleleft \alpha) = a_0$. So the behaviour is the same as that of a function that returns the first input it gets:

$$\begin{aligned}
&\mathsf{askonce} : \forall M, \mathbb{S}_{M,A} \to M\,A\\
&\mathsf{askonce}_M\,(\mathsf{mcons}\,m) = \mathsf{do}\ \ \langle a_0,\alpha_0\rangle \leftarrow m\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathsf{return}\,a_0
\end{aligned}$$

But if we instantiate it with monads that have side effects, the head of the stream might have changed after the first reading. For example, if we instantiate asktwice and askonce with the state transformer monad and apply it to a simple state increment stream, we get different results.

$$\begin{aligned}
&\mathsf{incr} : \mathbb{S}_{\mathsf{State}_{\mathbb{N}},\mathbb{N}}\\
&\mathsf{incr} = \mathsf{mcons}\,(\lambda s.\langle s, \mathsf{incr}, s+1\rangle)\\[1ex]
&\mathsf{runstr}\,(\mathsf{askonce}\,\mathsf{incr})\,0 = 0\\
&\mathsf{runstr}\,(\mathsf{asktwice}\,\mathsf{incr})\,0 = 1
\end{aligned}$$

Now, if we generate a dialogue tree by applying asktwice to the identity stream processor and then we evaluate this tree, we obtain a function that reads the first two element of the stream and returns the second.

$$\begin{aligned}
&\mathsf{asktwice}_{\mathsf{Dialogue}_A}\,\iota_A\\
&\quad = \mathsf{asktwice}\,(\mathsf{mcons}\,(\mathsf{Ask}\,\lambda a.\mathsf{Answer}\langle a,\iota_A\rangle))\\
&\quad = \mathsf{do}\ \ \langle a_0,\alpha_0\rangle \leftarrow \mathsf{Ask}\,\lambda a.\mathsf{Answer}\langle a,\iota_A\rangle\\
&\qquad\qquad \langle a_1,\alpha_1\rangle \leftarrow \mathsf{Ask}\,\lambda a.\mathsf{Answer}\langle a,\iota_A\rangle\\
&\qquad\qquad \mathsf{return}\,a_1\\
&\quad = \mathsf{Ask}\,\lambda a_0.\mathsf{do}\ \ \langle a_1,\alpha_1\rangle \leftarrow \mathsf{Ask}\,\lambda a.\mathsf{Answer}\langle a,\iota_A\rangle\\
&\qquad\qquad\qquad\qquad \mathsf{return}\,a_1\\
&\quad = \mathsf{Ask}\,\lambda a_0.\mathsf{Ask}\,\lambda a_1.\mathsf{Answer}\,a_1
\end{aligned}$$

This leads to a slightly different monadic stream function: instead of evaluating the original input action twice, it evaluates it once and then uses the monadic action of the tail.

$$\begin{aligned}
&\mathsf{meval}\,(\mathsf{asktwice}_{\mathsf{Dialogue}_A}\,\iota_A)\,(\mathsf{mcons}\,m) =\\
&\quad \mathsf{do}\ \ \langle a_0,(\mathsf{mcons}\,m_0)\rangle \leftarrow m\\
&\qquad\quad \langle a_1,(\mathsf{mcons}\,m_1)\rangle \leftarrow m_0\\
&\qquad\quad \mathsf{return}\,a_1
\end{aligned}$$

When applied to a pure stream, this function returns the second element, not the first as asktwice does: $\mathsf{eval}\,(\mathsf{asktwice}_{\mathsf{Dialogue}_A}\,\iota_A)\,(a_0 \triangleleft a_1 \triangleleft \alpha) = a_1$. We can note that this non-correspondence is caused by the fact that asktwice evaluates the input monadic action m twice. In the original function, this means that we read the same stream element twice, but in the dialogue tree it results in two consecutive requests for input.

We see that there is a discrepancy between two different ways of obtaining the next element of the stream, which the evaluation to a dialogue tree conflates. A monadic action $m : M\,A$ can be seen as a channel through which values of type A are transmitted; every time we evaluate it, we request a new transmission. On the other hand, evaluating a richer action $m_0 : M\,(A \times \mathbb{S}_{M,A})$ returns a value of type A and a new channel m_1; at this point we have the choice of which channel we use next.

This observation tells us that a simple action $M\,A$ can already be used to produce a sequence of values of type A. Indeed Jaskelioff and O'Connor [18] show that there is a one-to-one correspondence between dialogue trees in $\mathsf{Dialogue}_{A,B}$ and monadic functions of type $\forall M, M\,A \to M\,B$.

VII. Related and Future Work

Monadic streams are a powerful abstraction that already found concrete applications. Perez, Bärenz and Nilsson [23] used them to give a mathematically coherent and practical implementation of Functional Reactive Programming. Thus, aside from the theoretical interest in the foundations of mathematics and computer science, the study of monadic stream functions has important applications.

Three recent articles ([3], [18], [10]) studied the closely related topic of pure functions on a type of streams encoded by the type $\mathbb{N} \to M\,A$, its relation to dialogue trees and its use to prove the continuity of functional programs.

Bauer, Hofmann and Karbyshev [3] give a precise definition of *purity* for higher-order functionals in languages with side effects, for example ML. A function $f : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ is called *pure* if the only side effects that it generates are due to the evaluation of its argument.

This is done by interpreting such functions as polymorphic monadic functionals of type $\mathsf{Func} = \prod_{T \in \mathsf{Monad}} (A \to T\,B) \to T\,C$. The formal definition of purity is formulated on the basis of a relational semantics. A functional is pure (or *monadically parametric*) if its instantiations by different monads are related by this semantics. This is an abstract way of specifying that the functional behaves in the same way on all monads.

They use *dialogue trees*, which they call strategy trees, as a representation of functionals. Their representation is a slight generalization of the dialogue trees used by Ghani et al.'s in their paper[13], [14] that we also used in the present paper. The type Tree is inductively defined by two constructors: $\mathsf{Ans} : C \to \mathsf{Tree}$, that specifies a constant; $\mathsf{Que} : A \to (B \to \mathsf{Tree}) \to \mathsf{Tree}$, that specifies an interaction in which the function makes an query of type A, receives a response of type B, and produces a new tree to continue the computation. It is easy to show how every tree can be interpreted as pure functional in Func. Vice versa, a pure functional $F : \mathsf{Func}$ gives a tree by instantiation to the continuation monad with Tree as result type: $\mathsf{fun2tree}\, F = F_{\mathsf{Cont}_{\mathsf{Tree}}}\, \mathsf{Que}\, \mathsf{Ans}$. The authors prove that purity implies that the transformations from trees to functional and vice versa are inverse of each other (Theorem 12).

Among the applications, the authors show how to compute a modulus of continuity for functionals on the Baire space $\mathbb{B} = \mathbb{N} \to \mathbb{N}$. Call such a functional $F : \mathbb{B} \to \mathbb{N}$ *pure* if it is the instantiation of a monadically parametric function to the identity monad: $\bar{F} : \prod_{T \in \mathsf{Monad}} (\mathbb{N} \to T\,\mathbb{N}) \to T\,\mathbb{N}$, $F = \bar{F}_{\mathsf{Id}}$. The authors prove that such a functional is continuous (for every $f : \mathbb{B}$, the computation of $F\,f$ depends only on an initial segment of f). They give an explicit calculation of the modulus of continuity:

$$\begin{aligned} \mathsf{modulus}\, F\, f = \quad & \max\,(\mathsf{snd}\,(\bar{F}_{\mathsf{State}_{\mathsf{List}(\,)\mathbb{N}}}\,(\mathsf{instr}\, f)\,\mathsf{nil})) \\ & \text{where } \mathsf{instr}\, f : \mathbb{N} \to \mathsf{State}_{\mathsf{List}(\,)\mathbb{N}}\mathbb{N} \\ & \qquad\quad \mathsf{instr}\, f = \lambda a.\lambda l.\langle f\,a, l +\!\!\!+ [a]\rangle. \end{aligned}$$

Jaskelioff and O'Connor [18] give representation theorems for polymorphic higher order functionals. Their general type is: $\forall F.(A \to F\,B) \to F\,C$ where F ranges over a class of functors. Depending to the class of functors considered, we get different concrete representations of the functionals. For example, if F ranges over all functors, then the functionals are represented exactly by the type $A \times (B \to C)$: a functional h is necessarily of the form $h = \lambda g.F\,(k)\,(g\,a)$ for $a : A$ and $k : B \to C$. For pointed functors (with a natural transformation $\eta_X : X \to F\,X$) the representation is $A \times (B \to C) + C$. For applicative functors (having in addition a natural operation $\star_{X,Y} : (F\,X \times F\,Y) \to F\,(X \times Y)$) the representation is $\Sigma_{n:\mathbb{N}}(A^n \times (B^n \to C))$.

These representation results have a common structure: a general Representation Theorem formulated in categorical language. First they establish that for any small category $\mathcal{F}$ of endofunctors on sets that contains at least $R_{A,B}X = A \times (B \to X)$, we have: $\int_{F \in \mathcal{F}} (A \to F\,B) \to F\,X \cong R_{A,B}\,X$ which corresponds to the isomorphism for functors: $\forall F.(A \to F\,B) \to F\,X \cong A \times (B \to X)$. For classes of functors with extra structure (pointed functors, applicatives, monads), the representation theorem can be applied through an adjunction. Suppose that $\mathcal{F}$ is such a class of functors with structure, $\mathcal{E}$ a small class of endofunctors on sets. Assume that there is an adjunction between a forgetful functor $\mathcal{F} \to \mathcal{E}$ and a free functor $\_^* : \mathcal{E} \to \mathcal{F}$, so that $(\_^*) \dashv U$. Then the representation theorem is $\int_{F \in \mathcal{F}} (A \to U F\,B) \to U F\,X \cong U R^*_{A,B}\,X$.

When we instantiate the representation theorem to monads, we obtain that the type of polymorphic monadic higher order functions is isomorphic to the free monad on $R_{A,B}$. In Haskell notation this becomes

$$\forall m.\mathsf{Monad}\, m \Rightarrow (a \to m\,b) \to m\,x \cong \mathsf{Free}\,(\mathsf{PStore}\,a\,b)\,x$$

where Free and PStore are the Haskell implementation of the free monad construction and of $R_{A,B}$.

Escardó [10] gives a new short and simple proof of the know result that any function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ definable in Gödel's system T is continuous. The proof utilizes an auxiliary interpretation of system T in which natural numbers are interpreted as well-founded dialogue trees.

His definition of dialogue trees is more general than ours. A dialogue has three type parameters X, Y, Z: X is the type of queries that can be asked by a function, Y is the type of possible answers to a query, Z is the type of results that the function can return. If $X = \mathbb{N}$ then we represent functions on streams of elements of Y (of type $\mathbb{N} \to Y$) which return a result of type Z. The dialogue trees have internal nodes decorated with a query of type X, branching according to the response type Y, and leaves of result type Z:

$$\begin{aligned} \textbf{data}\ & \mathsf{D}\,(X, Y, Z : \mathsf{Set}) : \mathsf{Set} \\ & \eta : Z \to \mathsf{D}\,X\,Y\,Z \\ & \mathsf{B} : (Y \to \mathsf{D}\,X\,Y\,Z) \to X \to \mathsf{D}\,X\,Y\,Z. \end{aligned}$$

Of specific interest are dialogues over the Baire type $\mathbb{N} \to \mathbb{N}$, represented by the type former $\mathsf{B} = \mathsf{D}\,\mathbb{N}\,\mathbb{N}$.

A system T higher-order functional $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is given a non-standard interpretation by replacing the plain natural numbers $\mathbb{N}$ with dialogue trees the return naturals, $\tilde{\mathbb{N}} = \mathsf{B}\,\mathbb{N} = \mathsf{D}\,\mathbb{N}\,\mathbb{N}\,\mathbb{N}$. The interpretation utilizes the monadic operations of dialogue trees: zero becomes just returning 0, the successor operator is lifted by the monadic functor; primitive recursion is interpreted by the use of the bind operation. The continuity of the dialogue tree interpretation is established.

We obtain an operator on functions between dialogue trees $\tilde{f} : (\tilde{\mathbb{N}} \to \tilde{\mathbb{N}}) \to \tilde{\mathbb{N}}$. We can the apply it to a *generic sequence* of type $\tilde{\mathbb{N}} \to \tilde{\mathbb{N}}$ to obtain a dialogue tree that will turn out to be exactly a representation of the original f. The generic sequence is constructed so as to make the following diagram commute:

$$\begin{tikzcd} \tilde{\mathbb{N}} \arrow[r, "\mathsf{generic}"] \arrow[d, "\mathsf{decode}\,\alpha"'] & \tilde{\mathbb{N}} \arrow[d, "\mathsf{decode}\,\alpha"] \\ \mathbb{N} \arrow[r, "\alpha"'] & \mathbb{N} \end{tikzcd}$$

This is achieved simply by adding an extra step of interaction to the dialogue: generic replaces every leaf of the form $(\eta\, m)$ with $\mathsf{B}\,(\lambda n.\eta\, n)\, m$. This idea is analogous, but not the same, as our definition of tabulate, which also instantiates the monadic function with the dialogue tree monad and applies it to the identity processor ι_A.

Finally, the interpretation is shown to be equivalent to the standard interpretation of system T.

Future Work. An important next step in this line of work is to extend Escardó's characterization of functionals of system T to richer type systems. Does every function definable in some form of (dependent) type theory have a representation as a monadic function or dialogue tree?

Another avenue of exploration is the application to concrete functional programming. Already the work of Perez, Bärenz and Nilsson [23] shows that programming with monadic streams is very effective to create functional interactive applications. A complete understanding of the computation power and logical properties of these functions will be useful in practice.

REFERENCES

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming Infinite Structures by Observations. In *Symposium on Principles of Programming Languages*, 2013.

[2] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[3] Andrej Bauer, Martin Hofmann, and Aleksandr Karbyshev. On monadic parametricity of second-order functionals. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7794 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2013.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[5] Edwin Brady. Idris: A language with dependent types. http://www.idris-lang.org/. Accessed: 2016-10-10.

[6] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

[7] Venanzio Capretta and Tarmo Uustalu. A coalgebraic view of bar recursion and bar induction. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9634 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2016.

[8] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[9] Thierry Coquand, Giovanni Sambin, Jan M. Smith, and Silvio Valentini. Inductively generated formal topologies. *Ann. Pure Appl. Logic*, 124(1-3):71–106, 2003.

[10] Martín Hötzel Escardó. Continuity of Godel's system T functionals via effectful forcing. *Electr. Notes Theor. Comput. Sci.*, 298:119–141, 2013. Proceedings of the 29th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2013).

[11] Martín Hötzel Escardó and Chuangjie Xu. The inconsistency of a brouwerian continuity principle with the curry-howard interpretation. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 153–164. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[12] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003.

[13] Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 164(1):141–155, 2006. Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006).

[14] Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 249:3–18, 2009. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009).

[15] Neil Ghani, Peter Hancock, and Dirk Pattinson. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3), 2009.

[16] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[17] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[18] Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *J. Funct. Program.*, 25, 2015.

[19] Stephen Cole Kleene. Recursive functionals and quantifiers of finite types I. *Transactions of the American Mathematical Society*, 91(1):1–52, 1959.

[20] Conor McBride. Let's see how things unfold: Reconciling the infinite with the intensional (extended abstract). In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *CALCO*, volume 5728 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2009.

[21] Conor McBride. rnamental algebras, algebraic ornaments. To appear in Journal of Functional Programming, 2010.

[22] Ulf Norell, Andreas Abel, Nils Anders Danielsson, Makoto Takeyama, and Catarina Coquand. *The Agda User Manual*, 2016. `http://agda.readthedocs.io/en/latest/`.

[23] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 33–44. ACM, 2016.

[24] Giovanni Sambin. Some points in formal topology. *Theor. Comput. Sci.*, 305(1-3):347–408, 2003.

[25] The Coq Development Team. *The Coq Proof Assistant. Reference Manual. Version 8.5*. INRIA, 2016. `http://coq.inria.fr/refman/index.html`.