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 is 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.

This shows that we cannot internalize the meta-theoretical observation that every definable function is continuous. However, we can adapt Brouwer’s ideas to an important class of functions and propose a reformulation of the continuity principle that is internally provable. We note that Brouwer talked about functions on choice sequences, which are described as free progressions of values not necessarily generated by a rule. The functions must produce their results independently of how the sequences are generated.

We formalize them as 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: \( \alpha|_n = a_0 \triangleleft a_1 \triangleleft \cdots \triangleleft a_{n-1} \triangleleft \text{nil} \). (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 \( \triangleleft \) and an empty list base case \( \text{nil} \).

We say that two streams \( \alpha \) and \( \alpha' \) are \( n \)-equal if their first \( n \) elements are the same: \( \alpha' =_n \alpha \) if \( \alpha'|_n = \alpha|_n \). Brouwer’s 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 \( \alpha \) depends only on an initial segment \( \alpha|_n \), so that on any other sequence \( \alpha' \) with the same initial \( n \) elements, \( f \) will produce the same result.

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 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 a 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 [26] and Agda [23], and useful applications.

We may now ask if the theory can be extended with stronger constructive principles, specifically if we can add the continu-
ity principle to it. Unfortunately not: Recently Martín Escardó
discovered that the straightforward addition of the continuity
principle to type theory leads to a contradiction [11]. In the
aftermath of the discovery, discussion focused on analyzing the
source of the problem and investigating alternative formulati-
on 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, which are non-computable, and functions, which
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 func-
tions \( \mathbb{N} \to \mathbb{N} \) with a \textit{coinductive} type of streams \( S_A \). An
element \( \alpha : S_A \) is built by using the constructor \( \alpha \) an infinite
number of times. (\textit{Corecursion} patterns tell us how we can
generate the infinite sequence by a finite process.)

There is a correspondence between \( S_A \) 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 crite-
riion: it could be generated by a coalgebra [18], characterized
as the fixed point of a guarded equation [8], [16], or produced
by a guarded-by-constructors 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 interac-
tive type \( (\text{IO} A) \) 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 \( S_{M,A} \), for a given monad \( M \), has
elements of the form \( \text{mcons} m \), where \( m \) is an \( M \)-action,
m : \( M (A \times S_{M,A}) \). 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
\[
\forall M, S_{M,A} \to MB.
\]
The variable \( M \) ranges over monads. A function \( f \) of this type
should operate in a \textit{uniform} way independently of the monad.
We make this notion precise through a naturality condition. A
different way to characterize uniformity is through parametric-
ity [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.

\textbf{Acknowledgements}

We would like to thank members of the FP Lab at Not-
ttingham and the anonymous referees for useful comments
and suggestions about this work. A special thanks to Paolo
Capiroti for his vital contributions 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.

\textbf{II. Monadic Streams}

\textit{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 sim-
ple characterization by finality, and the formal implementation,
with its syntactic conditions [21]. 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 \textbf{data}
and \textbf{codata} marking inductive and coinductive types. As
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: \(2:3:5:7:.\) We also use typical list notation e.g. the above list can be written \([2,3,5,7].\)

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 \((S, P)\) with \(S : \text{Set}\), a set of shapes, and \(P : S \to \text{Set}\), a family giving a set of positions for every shape. Every container defines a functor:

\[
(S \triangleright P) : \text{Set} \to \text{Set}
\]

\[
(S \triangleright P) X = \Sigma \! S : S. \ P s \to X.
\]

An element of \((S \triangleright P) X\) is a pair \(\langle s, x s \rangle\) where \(s : S\) is a shape and \(x s : P s \to X\) 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 \(\nu(S \triangleright P)\) inhabited by trees with nodes labelled by shapes \(s\) and branches labelled by the positions \(P s\) of the node shape. So every element of \(t : \nu(S \triangleright P)\) is uniquely given by a shape, \(\text{shape} t : S\), and a family of sub-elements, \(\text{subs} t : P (\text{shape} t) \to \nu(S \triangleright P)\).

The actual final coalgebra is the function

\[
\text{out}_\nu : \nu(S \triangleright P) \to (S \triangleright P) (\nu(S \triangleright P)).
\]

Using this terminology and notation, streams can be defined by, \(S_A = \nu(A \triangleright \lambda a.1)\). 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 \(\nu\) 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 \(c : X \to (S \triangleright P) X\) has a unique anamorphism \(\hat{c} : X \to \nu(S \triangleright P)\) such that \(\text{out}_\nu \circ \hat{c} = (S \triangleright P) \circ \hat{c}\).

\[
\nu(S \triangleright P) \xrightarrow{\text{out}_\nu} (S \triangleright P) (\nu(S \triangleright P)) \xrightarrow{(S \triangleright P) \hat{c}} (S \triangleright P) X.
\]

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

\[
\text{shape} (\hat{c} x) = c_S x,
\]

\[
\text{subs} (\hat{c} x) p = \hat{c} (c_P x p).
\]

We will continue to use the more intuitive coinductive 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:

\[
\text{codata } S_{M,A} : \text{Set}
\]

\[
mcons_M : M (A \times S_{M,A}) \to S_{M,A}.
\]

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, \(M = S_M \triangleright P_M\), 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 \(P_F (s, h) = P_M s\). The shapes of \(F\) are shapes of \(M\) with the positions ornamented [22] 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 lax sum types.)

Pure streams are monadic streams for the identity monad \(\text{id}\): the type of \(\text{mcons}_{\text{id}}\) is isomorphic to that of \(\langle \rangle\) by currying:

\[
\text{mcons}_{\text{id}} : (A \times \text{Id}_{A}) \to \text{Id}_{A} \cong A \to \text{Id}_{A} \to \text{Id}_{A}.
\]

Interesting instantiations are obtained by using other monads. Some of them are important in later sections. If we choose the Maybe monad, we obtain co-lists, sequences of elements that may or may not be finite.

The elements of Maybe \(X\) are copies of each element \(x : X\), Just \(x\), and an error element Nothing. When we instantiate the definition of monadic streams with Maybe we obtain the type \(S_{\text{Maybe}, A}\), with two distinct ways to construct streams (although there is only one constructor) according to the monadic action. If the monadic action is Nothing, we get an empty stream object: \(\text{nil} = \text{mcons}_{\text{Maybe}}\text{Nothing}\). If the monadic action is Just, we get a head element and a tail: \(a \triangleleft a = \text{mcons}_{\text{Maybe}} \text{(Just } \langle a, a \rangle\text{)}\). Both finite lists and pure streams can
be injected in $S_{\text{Maybe},A}$. A list $[a_0, \ldots, a_n]$ is represented as $\text{mcons}(\text{Just}(a_0, \ldots, \text{mcons}(\text{Just}(a_n, \text{mcons}(\text{Nothing})\ldots)).$

The list constructor is itself a monad, so it makes sense to consider $S_{\text{List},A}$. This turns out to be the set of non-well-founded finitely branching trees with edges labelled by elements of $A$. An element of it has the form $\text{mcons}([a_0, a_0], [a_1, a_1], [a_2, a_2], \cdots)$ where $a_0, a_1, a_2$ are elements of $A$ and $a_0, a_1, a_2$ are monadic streams in $S_{\text{List},A}$.

Another interesting instantiation uses the state monad. This characterizes computations whose side effects consist in reading and modifying a state value in some type $S$: $\text{State}_S X = S \rightarrow X \times S$. A monadic action of type $S$ reads the present state and produces a result in $X$ and a new state. A monadic stream in $S_{\text{State},S,A}$ 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 $\text{mcons}(h, s)$, where $h : S \rightarrow A \times S_{\text{State},S,A} \times S$. As a simple example, here is the state-monadic stream of Fibonacci numbers.

$\text{fib}_\text{gen} : S_{\text{State},S,N,I} \rightarrow S_A$

$\text{fib}_\text{gen} = \text{mcons}(\langle [a, b], [b, \text{fib}_\text{gen}, (b, a + b)] \rangle)$

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.

$\text{runstr} : S_{\text{State},A} \rightarrow S \rightarrow S_A$

$\text{runstr}(\text{mcons}(h, s)) = \langle h, s' \rangle$

$\text{fib} = \text{runstr} \text{fib}_\text{gen} (0, 1)$

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 $(I, e, *)$: 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, Writer$I X = X \times I$. Elements of $S_{\text{Writer},A}$ are essentially streams of pairs $(a_0, i_0) \triangleright (a_1, i_1) \triangleright (a_2, i_2) \triangleright \cdots$ where $a_0 : A$ and $i_n : I$ for every $n$. Formally, the order of the arguments is different, because of the way the products associate: $\alpha = \text{mcons}(\langle a_0, (\text{mcons}(\langle a_1, (\text{mcons}(\langle a_2, \ldots, a_2 \rangle, i_2) \rangle, i_1) \rangle, i_0) \rangle).$

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, $(\text{return } a) : M A$, and the bind operation takes a monadic action $m : M A$ along with a function $f : A \rightarrow M B$ 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:

$\begin{align*}
\text{do} & \quad x_0 \leftarrow m_0 \\
& \quad x_1 \leftarrow m_1 \\
& \quad \text{return } e
\end{align*}$

means $m_0 \gg= (\lambda x_0. m_0 \gg= (\lambda x_1. \text{return } e))$.

### 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 $\text{return}_0, >>= 0$ and $\text{return}_1, >>= 1$. A monad morphism is a natural transformation, $\phi : M_0 \rightarrow M_1$, that respects the monad operations by satisfying the following laws:

$$\phi_A \circ \text{return}_0 = \text{return}_1$$

$$\phi_B (m >>= f) = (\phi_A m) >>= 1 (\phi_B \circ f)$$

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 $\phi : M_0 \rightarrow M_1$, its lifting to streams is a family of morphisms on monadic streams:

$$\begin{array}{c}
S_{\phi,A} : S_{M_0,A} \rightarrow S_{M_1,A} \\
S_{\phi,A}(\text{mcons}(m)) = \text{mcons}(\hat{\phi}_A m)
\end{array}$$

where $\hat{\phi}_A : M_0 (A \times S_{M_0,A}) \rightarrow M_1 (A \times S_{M_1,A})$.

$$\phi_A = \phi_A \times S_{M_1,A} \circ M_0 (\text{id}_A \times S_{\phi,A})$$

$M_1 (A \times S_{M_1,A})$ (equal by naturality of $\phi$).

This is a typical corecursive definition: we give equations for a function $S_{\phi,A}$ that generates a coinductive object in $S_{M_1,A}$. Circularly, $S_{\phi,A}$ depends on $\phi$ which in turn calls $S_{\phi,A}$. 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.

The definition of lifting is also applicable more generally when $\phi$ is a natural transformation. We use this more general version later but are explicit when we do so.

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 $f : \forall M, S_{M,A} \rightarrow M B$ is natural in $M$ if this diagram commutes for all monads $M_0$, $M_1$ and monad morphisms $\phi : M_0 \rightarrow M_1$:

\[
\begin{array}{ccc}
S_{M_0,A} & \xrightarrow{f_{M_0}} & M_0 B \\
\phi_B \circ f_{M_0} & = & f_{M_1} \circ S_{\phi,A} \\
S_{M_1,A} & \xrightarrow{f_{M_1}} & M_1 B
\end{array}
\]

Notice that naturality of $f$ 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 in this way, changing the monadic actions in the input stream (with $S_{\phi,A}$) can cause a corresponding change in the result (with $\phi_B$).

**Definition 5.** A function $g : S_A \rightarrow B$ is pure if there exists a monadic stream function $f : \forall M, S_{M,A} \rightarrow M B$ natural in $M$, such that $g = f_{\text{id}}$. We say $g$ instantiates $f$.

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

\[
nodec : S_N \rightarrow N
\]

\[
nodec(x_0 \& x_1 \& \alpha) = \left\{ \begin{array}{ll}
1 & \text{if } x_0 \leq x_1 \\
nodec(x_1 \& \alpha) + 1 & \text{otherwise}.
\end{array} \right.
\]

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

\[
nodec : \forall M, S_{M,N} \rightarrow M N
\]

\[
nodec(mcons m_0) = \text{do} \left\{ \begin{array}{l}
(x_0, \alpha) \leftarrow m_0 \\
nodec(x_0 \& \alpha)
\end{array} \right.
\]

\[
nodec' : S_{M,N} \rightarrow \forall M, S_{M,N} \rightarrow M N
\]

\[
nodec'(x_0 \& mcons m_0) = \text{do} \left\{ \begin{array}{l}
(x_1, \alpha_1) \leftarrow m_0 \\
\text{if } x_0 \leq x_1 \text{ then return } 1 \\
\text{else } M(+1)(\text{nodec}'(x_1 \& \alpha_1))
\end{array} \right.
\]

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:

\[
\text{noend} : \forall M, S_{M,N} \rightarrow M N
\]

\[
\text{noend}(mcons m_0) = \text{do} \left\{ \begin{array}{l}
(x_0, mcons m_1) \leftarrow m_0 \\
(x_1, mcons m_2) \leftarrow m_1 \\
\text{if } x_0 \leq x_1 \text{ then return } 1 \\
\text{else } M(+1)(\text{noend}(mcons m_1))
\end{array} \right.
\]

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:

\[
\text{flipper} : S_{\text{State},N} \rightarrow N
\]

\[
\text{flipper} = \text{mcons} (\lambda n. (\langle n, \text{flipper} \rangle, \text{mod } (n + 1) 2))
\]

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.

In the introduction we mentioned a different characterization of pure functions which is defined using parametricity as opposed to naturality [3]. Our characterization is weaker and therefore results in stronger properties.

That naturality of monadic stream functions is weaker than parametricity stems from the relation between parametricity and naturality. Although the exact details of this relation are unclear, it is known that parametricity for certain types, including monadic stream functions, entails strong dinaturality [17]. Strong dinaturality for a monadic stream function is a stronger condition than naturality in $M$. Particularly, strong dinaturality implies that the monadic stream function should behave uniformly across all natural transformations, not just monad morphisms. As this strong dinaturality condition is implied by parametricity, it follows that parametricity is stronger than naturality for monadic stream functions.

**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 $f : \forall M, S_{M,A} \rightarrow M B$ be natural in $M$. If there exists $b : B$ such that $f_{\text{Maybe}}(\text{mcons} \text{Nothing}) = \text{Just } b$, then for every monad $M$ and every $\alpha : S_A$, $f_M \alpha = \text{return}_M \ b$.

**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 for $M$, we use an intermediate instantiation of $f$ with an error monad, $\text{Error}_M$, whose error values are the monadic actions of $M$:

\[
\text{data Error}_M A : \text{Set}
\]

\[
\text{Pure} : A \rightarrow \text{Error}_M A
\]

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

\[
\begin{align*}
\text{forget} : & \text{Error}_M \to \text{Maybe} \\
\text{forget}_A(Pure a) & = \text{Just } a \\
\text{forget}_A(\text{Throw } m) & = \text{Nothing}.
\end{align*}
\]

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

\[
\begin{array}{ccc}
S_{\text{Maybe}, A} & \xrightarrow{f_{\text{Maybe}}} & B \\
S_{\text{forget}, A} & \xrightarrow{f_{\text{Error}_M}} & \text{Error}_M B \\
S_{\text{Error}_M, A} & \xrightarrow{f_{\text{Error}_M}} & \text{Error}_M B \\
S_{\text{merge}, A} & \xrightarrow{f_{\text{merge}}'} & M B \\
S_M & \xrightarrow{f_M} & M B
\end{array}
\]

We can lift a monadic stream into the ErrorM monad with the stream function S_{Throw, A} (note that Throw is a natural transformation but not a monad morphism; however, as mentioned in Section III, it can be lifted to streams in the same way). Observe that since forget(Throw m) = Nothing for every m, then S_{forget, A}(S_{Throw, A} α) = mcons Nothing for every α.

It is immediate that merge ◦ Throw = id and subsequently S_{merge, A} ◦ S_{Throw, A} = id. By commutativity of the upper square we have:

\[
\begin{align*}
\text{forget}_B(f_{\text{Error}_M}(S_{\text{Throw}, A} α)) & = f_{\text{Maybe}}(\text{forget}_A(S_{\text{Throw}, A} α)) \\
& = f_{\text{Maybe}}(\text{mcons Nothing}) \\
& = \text{Just } b
\end{align*}
\]

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

\[
\begin{align*}
f_M α & = f_M(S_{\text{merge}, A}(S_{\text{Throw}, A} α)) \\
& = \text{merge}_B(f_{\text{Error}_M}(S_{\text{Throw}, A} α)) \\
& = \text{merge}_B(Pure b) \\
& = \text{return}_M b
\end{align*}
\]

This gives us an effective test for syntactic constancy of pure functions. A similar idea helps us to prove the main result of the paper that a pure function, 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.

To begin with we define the modulus of continuity:

**Definition 6.** The modulus of continuity of a stream function, \( g : S_A \to B \), is a function, \( \text{mod}_g : S_A \to \mathbb{N} \), such that:

\[
\forall α, α' : S_A, α' \equiv (\text{mod}_g α) α \Rightarrow g α' = g α.
\]

Here the relation \( α' \equiv_n α \) means that α and α' have the same initial n elements.

To calculate the modulus of continuity of a pure function, we use the Writer monad with the monoid \((\mathbb{N}, 0, \text{max})\). A monadic stream for this monad, \( α : S_{\text{Writer}_M, A} \) is essentially a stream of pairs of elements of A and natural numbers:

\[
α \sim \langle a_0, n_0 \rangle \sqcup \langle a_1, n_1 \rangle \sqcup \langle a_2, n_2 \rangle \cdots
\]

When we evaluate the elements of the stream, we **write out** the maximum value of the n_i's of all the read elements.

We lift each pure stream \( α = a_0 \sqcup a_1 \sqcup a_2 \cdots : S_A \) to a Writer-monadic stream that decorates each element with the successor of its index:

\[
\langle a_0, 1 \rangle \sqcup \langle a_1, 2 \rangle \sqcup \langle a_2, 3 \rangle \cdots
\]

Formally we can define it as follows:

\[
\begin{align*}
-1 & : S_A \to S_{\text{Writer}_M, A} \\
α & = \text{index}_1 α \\
\text{where } & \text{index} : \mathbb{N} \to S_A \to S_{\text{Writer}_M, A} \\
\text{index}_n & (a_0 \sqcup a_1 \sqcup \cdots) = \text{mcons} \langle\langle a_0, \text{index}(n+1) α\rangle, n\rangle
\end{align*}
\]

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

\[
\text{modulus} : (\forall M, S_{M, A} \to M B) \to S_A \to \mathbb{N}
\]

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

**Theorem 2.** Let \( g : S_A \to B \) be a pure function that instantiates f : \( \forall M, S_{M, A} \to M B \). Then (modulus f) is the modulus of continuity of g.

**Proof.** We can formulate this continuity conjecture in a slightly different way by extending the claim about Maybe-monadic streams. We use now the Writer_M 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 \( α = a_0 \sqcup a_1 \sqcup a_2 \cdots \) at the m-th element, where \( m = (\text{modulus } f α) = \pi_1(f_{\text{Writer}_M, α}) \).

For a given \( α \) define two monad morphisms from the Writer monad to Maybe and Id:

\[
\begin{align*}
\phi : & \text{Writer}_M \to \text{Maybe} \\
\phi_X(x, n) & = \begin{cases} 
\text{Just } x & \text{if } n \leq m \\
\text{Nothing} & \text{otherwise}
\end{cases} \\
ψ : & \text{Writer}_M \to \text{Id} \\
ψ_X(x, n) & = x
\end{align*}
\]

\( \square \)
Note that $\phi$, which truncates the stream, is dependent on the modulus of continuity of $\alpha$. Naturality of $f$ in the monad tells us that the following two squares commute:

$$
\begin{array}{c}
S_{\text{Maybe},A} \\ \downarrow f_{\text{Maybe}} \\
\phi_B \\
\downarrow \phi_B \\
S_{\phi,A} \\ \downarrow f_{\text{Writer}},A \\
S_{\phi,A} \\
\downarrow f_{\text{id}}=g \\
B
\end{array}
$$

Starting with $\alpha : S_{\text{Writer},A}$, we know that $(f_{\text{Writer}},\alpha_s) = (b, m)$ for some $b : B$. Using the commutativity of the lower rectangle, we discover that

$$
\begin{align*}
\phi_B \circ f_{\text{Writer}} &= f_{\text{Maybe}} \circ S_{\phi,A} \\
\psi_B \circ f_{\text{Writer}} &= f_{\text{id}} \circ S_{\psi,A}.
\end{align*}
$$

Following the upper route, by the definition of $\phi$ and of its lifting to streams, we have that $(S_{\phi,A}(\alpha_s)) = a_0 \circ \ldots \circ a_{m-1} \circ \text{nil}$; so we discover that $(f_{\text{Maybe}}(a_0 \circ \ldots \circ a_{m-1} \circ \text{nil})) = \phi_B(b, m) = \text{Just } b$.

If we now take another pure stream $\alpha'$ such that $\alpha' = m \alpha$, we also have $(S_{\phi,A}(\alpha')) = a_0 \circ \ldots \circ a_{m-1} \circ \text{nil}$. This forces $(f_{\text{Writer}} \alpha) = (b, m')$ for some $m'$ (we can easily prove that $m' = m$) and consequently $(f_{\text{id}} \alpha') = b$.

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? Continuity is achieved by showing that a finite initial segment $\alpha|_n$ of the stream is sufficient to determine the value of $(f_{\text{id}} \alpha)$. 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 a version of Theorem 2 stating that the value of $f_{\text{List}} \alpha$ 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 $\alpha$.

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 $\alpha_s$, 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 $\langle \text{List}(A), \ast, \text{nil} \rangle$, taking as monoid operation the longer of two lists, giving preference to the second:

$$
\begin{align*}
\text{nil} \ast l &= l \\
l \ast \text{nil} &= l \\
(a_0 \ast as) \ast (b_0 \ast bs) &= b_0 \ast (as \ast bs).
\end{align*}
$$

We lift every stream to one inside $S_{\text{Writer},A}$ by decorating every element with the initial segment of the stream leading to it.

$$
\begin{align*}
\alpha_{\ast} &: S_{A} \rightarrow S_{\text{Writer},A} \\
\alpha_{\ast} &= \text{decorate } \alpha \ni 0
\end{align*}
$$

Intuitively we have that

$$
\alpha_{\ast} \sim \langle a_0, [a_0] \rangle \sim \langle a_1, [a_0, a_1] \rangle \sim \langle a_2, [a_0, a_1, a_2] \rangle \sim \ldots .
$$

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

$$
\begin{align*}
\text{approx} &: (\forall M, S_{M,A} \rightarrow M B) \rightarrow S_{A} \\
\text{approx} f \alpha &= \pi_1(f_{\text{Writer}},(\alpha_{\ast})).
\end{align*}
$$

Let us write $\alpha \vdash l$ to state that the list $l$ is an initial segment of the stream $\alpha$, that is, $l$ approximates $\alpha$ (notation inspired by formal topology [9], [25]). An alternative formulation of continuity states that this approximation is sufficient to determine the result.

Theorem 3. Let $f : \forall M, S_{M,A} \rightarrow M B$ be natural in $M$.

$$
\forall \alpha, \alpha' : S_{A}, \alpha' \vdash (\text{approx } \alpha) \Rightarrow f_{\text{id}} \alpha' = f_{\text{id}} \alpha.
$$

Proof. By a modification of the proof of Theorem 2.
Definition 7. Let $f : \forall M, S_{M,A} \rightarrow MB$ be natural in $M$ and let $M$ be a specific monad, the continuity principle for $M$ states that

\[
\forall \alpha : S_{M,A}, \exists l : L_{M,A}, \alpha \vdash l \land
\forall \alpha' : S_{M,A}, \alpha' \vdash l \Rightarrow f_M \alpha' = f_M \alpha.
\]

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.

\[
\text{misTake} : \forall M, S_{M,N} \rightarrow M[N]
\]

\[
\text{misTake}_M (\text{mcons} \, m) = \text{do} \,(n_\perp) \leftarrow m
\]

\[
\text{take}_M \, 0 \alpha = \text{return} [\]

\[
\text{take}_M \, \text{suc} \,(n) \,(\text{mcons} \, m) = \text{do} \,(a, m') \leftarrow m
\]

\[
M (a::) (\text{take}_M \, n \, m)
\]

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.

\[
\text{goleft} : S_{\text{Id},N} \rightarrow S_{\text{State},N}
\]

\[
\text{goleft} \, \alpha = \text{mcons} (\lambda n. \, \text{ifzero} n \,(\text{atleft} \, \alpha) \, \bar{0}, \, 0))
\]

\[
\text{where}\:\: \text{atleft} : S_{\text{Id},N} \rightarrow S_{\text{State},N}
\]

\[
\text{atleft} \,(a :: \alpha) = \text{mcons} \,(\lambda n. \, \text{ifzero} n \,(a, \text{atleft} \, \alpha, \, 0) \,(\bar{0}, \, 0, \, n))
\]

\[
\bar{0} : \forall M, S_{M,N}
\]

\[
\bar{0}_M = \text{mcons} \,(\text{return} \, (\bar{0}, \, \bar{0}_M))
\]

The first monadic state action of goleft $\alpha$ returns the initial state and changes the state to 0. The tail depends on 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 $\alpha$ and recur. If we picture an element of $S_{\text{State},N}$ as a tree in which every node branches according to the state value, then goleft $\alpha$ has the value of the state in the the nodes at depth one and at depths greater than one is zero everywhere except on the leftmost spine, where it contains the elements of $\alpha$.

The counterexample is made by applying misTake to a state goleft $\alpha$. 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 goleft $\alpha$. Equationally:

\[
\text{misTake} \, (\text{goleft} \, \alpha) \, n = 0 :: \alpha|_n
\]

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 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 goleft $\alpha$ and its approximations, before giving a proof of the contradiction. First we define leftof which takes the left spine of an approximation:

\[
\text{leftof} : \forall \text{State}_{\alpha,A} \rightarrow \text{Id}_{A}
\]

\[
\text{leftof} \, (\text{mcons} \, m) = \text{let} \,(a, \alpha, \_ \,) = m \, a :: \text{leftof} \, \alpha
\]

\[
\text{leftof} \, \text{unknown} = \text{unknown}
\]

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

\[
\text{goleft} \, \alpha \vdash l \Rightarrow \exists n. \text{leftof} \, l = 0 :: \alpha|_{n-1}
\]

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

\[
\text{goleft} \, \alpha \vdash l \land \,(0 < \alpha') \vdash \text{leftof} \, l \Rightarrow \text{goleft} \, \alpha' \vdash l
\]

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

\[
\text{leftof} \, l = 0 < \bar{0}|_{n-1} = \bar{0}|_n
\]

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

\[
\text{misTake} \, (\text{goleft} \, \alpha) = \text{misTake} \, (\text{goleft} \, \bar{0})
\]

However using (1) we have

\[
\text{misTake} \, (\text{goleft} \, \bar{0}) \,(n + 1) = 0 :: \bar{0}|_n
\]

but

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

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 [20]. Martín Escardó used them to give a concise proof of continuity of the functionals 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.

\[
\text{data} \: \text{Dialogue}_{A,B} : \text{Set}
\]

\[
\text{Answer} : B \rightarrow \text{Dialogue}_{A,B}
\]

\[
\text{Ask} : (A \rightarrow \text{Dialogue}_{A,B}) \rightarrow \text{Dialogue}_{A,B}
\]

A tree can have two forms: an immediate answer (Answer $b$) corresponding to a constant function returning $b$ without
reading any element of the input stream, or a branching construction \( (\text{Ask } h) \) with \( h : A \rightarrow \text{Dialogue}_{A,B} \) 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:

\[
\begin{align*}
\text{eval} : \text{Dialogue}_{A,B} &\rightarrow S_A \rightarrow B \\
\text{eval}(\text{Answer } b) &\alpha = b \\
\text{eval}(\text{Ask } h) (a \circ \alpha) &= \text{eval}(h \, a) \, \alpha.
\end{align*}
\]

The function \( \text{eval} \) is continuous for every tree \( t \), i.e. it has a modulus of continuity. The converse is challenging. In their paper, Ghani et al. prove it negatively: If a function has no modulus of continuity. The converse is challenging. In their paper, Ghani et al. prove it negatively: If a function has no modulus of continuity.

The evaluation operation is readily generalized to monadic streams:

\[
\begin{align*}
\text{meval} : \text{Dialogue}_{A,B} &\rightarrow \forall M, S_{M,A} \rightarrow M B \\
(\text{meval}(\text{Answer } b))_M &\alpha = \text{return } b \\
(\text{meval}(\text{Ask } h))_M (\text{mcons } m) &= \\
&\quad \quad m \triangleright= \lambda(a, \alpha).\langle(\text{meval}(h \, a))_M \alpha
\end{align*}
\]

For any dialogue tree \( t \), \( (\text{eval} \, t) \) instantiates \( (\text{meval} \, t) \) with the identity monad, which provides an alternative proof for the continuity of \( (\text{eval} \, t) \). To verify this, we need to show that \( (\text{eval} \, t) = (\text{meval} \, t) \circ \text{id} \) and that \( (\text{meval} \, t)_M \) is natural in \( M \); both these facts follow from the definition of \( \text{meval} \) (by substituting \( \text{id} \) and by construction, respectively).

Also in this case, the converse 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 \( \text{Dialogue}_A \) is itself a monad on \( B \):

\[
\begin{align*}
\text{return} &= \text{Answer} \\
(\text{Answer } b) &\triangleright= g = g \, b \\
(\text{Ask } h) &\triangleright= g = \text{Ask } (\lambda a. \, h \, a) \triangleright= g.
\end{align*}
\]

So it makes sense to consider streams on it, \( S_{\text{Dialogue}_A,B} \). 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:

\[
\begin{align*}
\iota_A : S_{\text{Dialogue}_A,B} \\
\iota_A &= \text{mcons } (\text{Ask } (\lambda a. \, \text{Answer } (a, \iota_A))).
\end{align*}
\]

Using \( \iota_A \) we can define a tabulation function, which converts a monadic stream function into a dialogue tree:

\[
\begin{align*}
\text{tabulate} : (\forall M, S_{M,A} \rightarrow M B) &\rightarrow \text{Dialogue}_{A,B} \\
\text{tabulate } f &= f \circ \iota_A \\
\end{align*}
\]

Theorem 4. The function \( \text{tabulate} \) is a left inverse of the evaluation operation, \( \text{meval} \):

\[
\begin{align*}
\forall t : \text{Dialogue}_{A,B} \quad \text{tabulate} \, (\text{meval} \, t) &= t.
\end{align*}
\]

Proof. By induction on the dialogue tree \( t \).

In the base case the dialogue tree is of the form \( \text{Answer } b \), we have:

\[
\begin{align*}
\text{tabulate } (\text{meval } (\text{Answer } b)) &= (\text{meval } (\text{Answer } b)) \circ \iota_A \\
&= \text{return } b = \text{Answer } b
\end{align*}
\]

In the inductive case the dialogue tree is of the form \( \text{Ask } h \), we have:

\[
\begin{align*}
\text{tabulate } (\text{meval } (\text{Ask } h)) &= (\text{meval } (\text{Ask } h)) \circ \iota_A \\
&= (\text{Ask } (\lambda a. \text{Answer } (a, \iota_A))) \circ \iota_A \\
&= \text{Ask } (\lambda a. \text{Answer } (a, \iota_A)) = \text{Ask } (\lambda a. \text{tabulate } (\text{meval } (h \, a))) \\
&= \text{Ask } h \quad \text{(by induction hypothesis)}.
\end{align*}
\]

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

\[
\begin{align*}
\text{asktwice } : \forall M, S_{M,A} &\rightarrow M A \\
\text{asktwice }_M (\text{mcons } m) &= \text{do } \langle a_0, a_0 \rangle \leftarrow m \\
&\quad \langle a_1, a_1 \rangle \leftarrow m \\
&\quad \text{return } a_1.
\end{align*}
\]

This function executes the monadic action of the stream twice, so it reads the head of the stream twice, then returns the second value that it obtains. On pure streams this simply means that it reads the first value twice, \( a_0 \) and \( a_1 \) are the same:

\[
\text{asktwice }_\text{id} (\alpha \circ \alpha) = a_0 \circ a_0
\]

So the behaviour is the same as that of a function that returns the first input it gets:

\[
\begin{align*}
\text{askonce } : \forall M, S_{M,A} &\rightarrow M A \\
\text{askonce }_M (\text{mcons } m) &= \text{do } \langle a_0, a_0 \rangle \leftarrow m \\
&\quad \text{return } a_0.
\end{align*}
\]

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 \text{asktwice} and \text{askonce} with the state transformer monad and apply it to a simple state increment stream, we get different results.

\[
\begin{align*}
\text{incr } : S_{\text{State}_{\mathbb{N}}, \mathbb{N}} \\
\text{incr } = \text{mcons } (\lambda s. \langle s, \text{incr } s + 1 \rangle)
\end{align*}
\]

\[
\begin{align*}
\text{runstr } (\text{askonce } \text{incr}) 0 &= 0 \\
\text{runstr } (\text{asktwice } \text{incr}) 0 &= 1
\end{align*}
\]

Now, if we generate a dialogue tree by applying \text{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.

\[
\text{asktwice}_{\text{Dialogue}_A} \quad \lambda a.\text{mcons}(\text{Ask}\lambda a.\text{Answer}(a,\langle A \rangle))
\]

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.

\[
\text{meval}(\text{asktwice}_{\text{Dialogue}_A} \quad \lambda a.\text{mcons}(\text{mcons}(m))) =
\]

When applied to a pure stream, this function returns the second element, not the first as asktwice does:

\[
\text{eval}(\text{asktwice}_{\text{Dialogue}_A} \quad \lambda a.\text{mcons}(\text{mcons}(m))) = m
\]

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 \( m_0 : M (A \times S_{M,A}) \) 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 [19] show that there is a one-to-one correspondence between dialogue trees in \( \text{Dialogue}_{A,B} \) and monadic functions of type \( \forall M, M A \rightarrow M B \).

VII. RELATED AND FUTURE WORK

Monadic streams are a powerful abstraction that already found concrete applications. Perez, Barenz and Nilsson [24] 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], [19], [10]) studied the closely related topic of pure functions on a type of streams encoded by the type \( \mathbb{N} \rightarrow M A \), its relation to dialogue trees and its use to prove the continuity of functional programs.

Intuitively, the difference with out work is that a function \( \mathbb{N} \rightarrow M A \) gives an “infinite number of channels”, each of which can be consulted to obtain an element of \( A \). In our formalization, a stream of type \( S_{M,A} \) is a single channel that can be consulted to get an element of \( A \) and a second channel (or a modification of the initial channel) that will produce future elements of \( A \).

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 \( f : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \) 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

\[
\text{Func} = \prod_{T \in \text{Monad}} (A \rightarrow T B) \rightarrow T C.
\]

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 ([13], [14]) that we also use in the present paper. The type Tree is inductively defined by two constructors: \( \text{Ans} : C \rightarrow \text{Tree} \), that specifies a constant; \( \text{Que} : A \rightarrow (B \rightarrow \text{Tree}) \rightarrow \text{Tree} \), that specifies an interaction in which the function makes a 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 a pure functional in \( \text{Func} \). Conversely, a pure functional \( F : \text{Func} \) gives a tree by instantiation to the continuation monad with Tree as result type: \( \text{fun}2\text{tree} F = F_{\text{Cont}_\text{Tree}} \text{Que} \text{Ans} \). Notice how the argument of \( F \), when instantiated to the continuation monad, has type

\[
A \rightarrow \text{Cont}_{\text{Tree}} B = A \rightarrow (B \rightarrow \text{Tree}) \rightarrow \text{Tree}
\]

which is exactly the type of \( \text{Que} \). The result type is \( \text{Cont}_{\text{Tree}} C = (C \rightarrow \text{Tree}) \rightarrow \text{Tree} \), so we can apply it to \( \text{Ans} \).

The authors prove that purity implies that the transformations from trees to functionals 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 \( \mathbb{B} = \mathbb{N} \rightarrow \mathbb{N} \). Call such a functional \( F : \mathbb{B} \rightarrow \mathbb{N} \) pure if it is the instantiation of a monadically parametric function to the identity monad: \( F = \prod{T \in \text{Monad}} (\mathbb{N} \rightarrow T \mathbb{N}) \rightarrow T \mathbb{N} \), \( F = F_{\text{Id}} \).

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:

\[
\text{modulus } F f = \max \left( \text{snd} \left( \text{F}_{\text{State}_{\text{List}([a])}} \left( \text{instr } f \right) \text{nil} \right) \right)
\]

where \( \text{instr } f : \mathbb{N} \rightarrow \text{State}_{\text{List}([n])} \mathbb{N} \)

\[
\text{instr } f = \lambda a.\lambda l.\langle f a, l+\langle a \rangle \rangle.
\]

This is similar to our technique: By using the state monad, they force \( f \) to return a list of all the indices on which the input function has been applied during the computation.
Jaskelioff and O'Connor [19] give representation theorems for polymorphic higher order functionals. Their general type is: \( \forall F. (A \to FB) \to FC \) where \( F \) ranges over a class of functors. Depending on 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 \( \eta_X : X \to FX \)) the representation is \( A \times (B \to C) + C \). For applicative functors (having in addition a natural operation \( \star_X,Y : (FX \times FY) \to F(X \times Y) \)) the representation is \( \Sigma_{n \in N} (A^n \times (B^n \to C)) \).

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 contain at least \( R_{A,B} X = A \times (B \to X) \), we have: \( \int F \in \mathcal{F} \ (A \to FB) \to FX \cong R_{A,B} X \) which corresponds to the isomorphism for functors: \( \forall F. (A \to FB) \to FX \cong A \times (B \to X) \). For categories 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 category of functors with structure, \( \mathcal{E} \) a small subcategory 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 \( *(\_)* \to U \). Then the representation theorem is \( \int F \in \mathcal{F} \ (A \to UF B) \to UF X \cong UR_{A,B} X \).

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

\[
\forall m. \text{Monad} \ m \Rightarrow (a \to m b) \to mx \cong \text{Free} \ (\text{PStore} \ ab)x
\]

where \( \text{Free} \) and \( \text{PStore} \) are the Haskell implementation of the free monad construction and of \( R_{A,B} \).

The type \( \text{Free} \ (\text{PStore} \ ab)x \) is once again a presentation of dialogue/strategy trees.

Escardó [10] gives a new short and simple proof of the known result that any function \( f : (N \to N) \to N \) 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 and is essentially the same as in the two previously considered articles. 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 \( X = N \) then we represent functions on streams of elements of \( Y \) (of type \( N \to Y \)) 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 \( Z \):

```haskell
data D (X, Y, Z : Set) : Set
  \eta : Z \to DXYZ
  B : (Y \to DXYZ) \to X \to DXYZ.
```

Of specific interest are dialogues over the Baire type \( N \to N \), represented by the type former \( B = D \times N \times N \).

A system \( T \) higher-order functional \( f : (N \to N) \to N \) is given a non-standard interpretation by replacing the plain natural numbers \( N \) with dialogue trees that return naturals, \( \mathbb{N} = B \times N \times D \times N \times N \). 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 \( \tilde{f} : (\tilde{N} \to \tilde{N}) \to \tilde{N} \). We can then apply it to a generic sequence of type \( \tilde{N} \to \tilde{N} \) 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:

```
\[
\begin{array}{c}
\text{decode } \alpha \\
\tilde{N}
\end{array}
\quad \begin{array}{c}
\text{generic} \\
N
\end{array}
\quad \begin{array}{c}
\text{decode } \alpha \\
\tilde{N}
\end{array}
\]
```

This is achieved simply by adding an extra step of interaction to the dialogue: generic replaces every leaf of the form \( (\eta m) \) with \( B (\lambda n. \eta n) m \). 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 \( t_A \).

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 [24] 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**


