REPRESENTATIONS OF STREAM PROCESSORS USING NESTED FIXED POINTS

PETER HANCOCK, DIRK PATTINSON, AND NEIL GHANI

School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB
e-mail address: hancock@spamcop.net

Department of Computing, Imperial College London, 180 Queen’s Gate, London SW7 2AZ
e-mail address: dirk@doc.ic.ac.uk

Computer and Information Sciences, University of Strathclyde, Livingstone Tower, 26 Richmond
St, Glasgow G1 1XH
e-mail address: neil.ghani@cis.strath.ac.uk

Abstract. We define representations of continuous functions on infinite streams of discrete values, both in the case of discrete-valued functions, and in the case of stream-valued functions. We define also an operation on the representations of two continuous functions between streams that yields a representation of their composite.

In the case of discrete-valued functions, the representatives are well-founded (finite-path) trees of a certain kind. The underlying idea can be traced back to Brouwer’s justification of bar-induction, or to Kreisel and Troelstra’s elimination of choice-sequences. In the case of stream-valued functions, the representatives are non-wellfounded trees pieced together in a coinductive fashion from well-founded trees. The definition requires an alternating fixpoint construction of some ubiquity.

INTRODUCTION

This paper is concerned with the representation and implementation of continuous functions on spaces of infinite sequences, or streams of discrete values, such as binary digits (Cantor space), or natural numbers (Baire space). That is to say, we will look at functions of type

\[ f : A^\omega \Rightarrow X \]

where \( A \) is a discrete space, \( A^\omega \) is the space of streams of elements of \( A \) with the product topology, and \( X \) is either a discrete space \( B \), or itself a space of streams \( B^\omega \). We use \( \Rightarrow \) for the continuous function space. Functions of this kind and closely related kinds arise in many contexts in mathematics and are pervasive in programming, as with pipes, stream IO, and coroutines.

2000 ACM Subject Classification: 68P05, 68N18, 54C35.

Key words and phrases: Streams, continuous functions, initial algebras, final coalgebras.

Hancock’s work was supported by EPSRC grant EP/C511964/1.
If one is to implement in any practical sense such a function by means of a program or machine that consumes successive values in an input stream, and produces a value (all at once in the discrete case, or in a stream of successive values in the stream-valued case), it seems necessary that the function be continuous. Otherwise, the whole input stream would be needed at once: an output would be forthcoming only ‘at the end of time’. Continuity means that finite information concerning the output of the function is determined by finite information concerning its input. In the simpler, discrete-valued case, this amounts to the requirement that the value \( b = f(\alpha) \) of the function at argument \( \alpha \) is determined (or ‘secured’) by some finite prefix \( \pi_n = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{n-1}) \) of \( \alpha \).

It is fairly clear how to represent a continuous function on \( A^\omega \) with discrete values in \( B \): take a well-founded tree branching over \( A \), with \( B \)'s at the leaves. Such a tree represents a continuous function. Start at the root, then use successive entries in the argument stream to steer your way along some path to a leaf. When you arrive at the leaf, there is your value for that argument. We can visualise the representation as follows.

\[
\begin{align*}
  f : 2^\omega &\to 2 \\
  f(0, \ldots) &= 0 \\
  f(1,0,\ldots) &= 1 \\
  f(1,1,0,\ldots) &= 0 \\
  f(1,1,1,\ldots) &= 1
\end{align*}
\]

At the black inner nodes, the representation ‘eats’ the next entry in the argument stream, and goes left or right according to whether it’s 0 or 1; at leaves, it ‘spits’ the (boxed) value for that argument.

It is worth noting that there will be several (actually, infinitely many) representations of the same function. For example, the tree below represents the same function as the one above.

\[
\begin{align*}
  f : 2^\omega &\to 2 \\
  f(0, \ldots) &= 0 \\
  f(1,0,\ldots) &= 1 \\
  f(1,1,0,\ldots) &= 0 \\
  f(1,1,1,\ldots) &= 1
\end{align*}
\]

It is a little less obvious that any continuous function is representable in this way. The most straightforward argument is irredeemably classical: suppose the function has no representation, and derive a stream at which it is not continuous. A fascinating but fallacious argument was given by Brouwer, analysed by Dummett in [4, pp 68–75].
Now what about stream-valued continuous functions on \(A^\omega\) with values in \(B^\omega\)? The idea is again quite simple, though as far as we know, new. It is also difficult to depict. What we want is a non-wellfounded tree, branching over \(A\), along every path of which there are infinitely many nodes labelled with an element of \(B\). Start at the root, then use successive entries in the argument stream to steer your way along some path. When you arrive at a node labelled with an element of \(B\), emit that element as the next entry in the output stream. It turns out to be straightforward to express the type of trees we need as a nested fixed point, in which one forms the final coalgebra of a functor that is defined using an initial algebra construction.

It is more difficult to see is that every stream-valued function on streams is representable by a non-wellfounded tree of the kind we have described.

The datatype of representations provides a convenient basis for writing stream processing components in a functional programming language such as Haskell. Nevertheless, the coding in Haskell is imperfect. The foundations of Haskell are in an algebraically compact category in which initial and final coalgebras coincide. From that perspective, programs are partial functions, and not functions in the standard mathematical sense. Our approach would be better expressed in a language for total functional programming, as advocated by Turner [17], and approached in systems such as Epigram and Agda. This means that evaluating the constructor form of the value of a function at an argument in its domain necessarily terminates. (Such a language is necessarily not Turing incomplete.)

The paper is organised as follows.

- **preliminaries**
- **section 2:** we define the representation of the continuous function space \(A^\omega \to B\) by the datatype \(T_A(B) = (\mu X) [B + X A]\) of wellfounded trees branching over \(A\) and terminating in \(B\), and show it is complete in the sense that each such continuous function has a representative (in fact many) in \(T_A(B)\). This part of the paper is in essence fairly well known.
- **section 3:** we define the representation of the continuous function space \(A^\omega \to B^\omega\) by \(P_A(B) = (\nu Y) T_A(B \times Y)\). The main contribution here is the proof of completeness, which is not completely straightforward. The proof is constructive, given completeness in the discrete-valued case.
- **section 4:** we define two representations of composition, as operators of type \(P_B(C) \times P_A(B) \to P_A(C)\), and show their correctness. As far as we have been able to discover, this representation is new.
- in conclusion, we summarise what has been done, point out related work, and indicate some further work extending these ideas to continuous functions on more general final coalgebras than streams.

### 1. Preliminaries

We assume the reader is familiar with the categorical notions of product, coproduct, and exponential, and standard notations associated with these.

#### 1.1. Streams.

If \(A\) is a set, we write \(A^\omega\) for the set of countably infinite streams (\(\omega\)-sequences) of elements of \(A\), and \(A^*\) for the set of finite sequences (lists) of elements of \(A\).

We use Greek letters \(\alpha, \beta, \ldots\) as variables over stream types.
We overload the infix operator \( (\_\_\_\_ ) \) (with section notation) both as our basic means of constructing streams and non-empty lists. Thus if \( a : A \), then
\[
(a\_\_\_\_ ) : A^\omega \Rightarrow A^\omega \quad (a\_\_\_\_ ) : A^* \Rightarrow A^*
\]
We also have the empty list \( \_ : A^* \).

As destructors of streams we use \( \text{hd} \) and \( \text{tl} \).
\[
\text{hd} : A^\omega \rightarrow A \\
\text{tl} : A^\omega \rightarrow A^\omega
\]
For all \( a : A \) and \( \alpha : A^\omega \) we have
\[
\text{hd}(a\_\_\_\_ ) = a : A \\
\text{tl}(\_\_\_\_ \alpha) = \alpha : A^\omega \\
\alpha = \text{hd}(\alpha) ; \text{tl}(\alpha) : A^\omega
\]
The destructors are used implicitly in pattern-matching definitions.

We sometimes write \( \alpha_0 \) for \( \text{hd}(\alpha) \), and \( \alpha' \) for \( \text{tl}(\alpha) \).

We use the function \( (\_ ) : A^\omega \rightarrow (A^*)^\omega \) which returns the stream of finite prefixes of its argument. It is defined by \( \overline{\omega}_0 = \_ \) and \( \overline{\omega}_{n+1} = \alpha_0 \overline{(\_\_\_\_ )}_n \).

Streams are endowed with a topology in which the neighbourhoods are given by finite sequences \( c : A^* \). Each such represents the predicate \( N(c) = \{ \alpha \mid c = \overline{\alpha}(\text{len}(c)) \} \) of streams sharing prefix \( c \). We usually suppress the distinction between \( c : A^* \) and \( N(c) \subseteq A^\omega \). The relation \( \alpha \in N(c) \) can be defined by recursion on list \( c \).

We use \( \Rightarrow \) for the continuous function space. Thus \( A^\omega \Rightarrow X \) consists of the continuous functions from \( A^\omega \) to \( X \), where \( X \) is either a discrete space \( D \), or a space \( D^\omega \) where \( D \) is discrete.

- A discrete valued continuous function \( f : A^\omega \rightarrow D \) is continuous at \( \alpha : A^\omega \) if there is some neighbourhood of \( \alpha \) throughout which \( f \) is constant. In other words, there exists \( n \in \omega \) such that \( f \) has the same value throughout the neighbourhood \( \overline{\alpha}(n) \).

\[
image(f, \overline{\alpha}(n)) = \{ f(\alpha) \}
\]

\( A^\omega \Rightarrow D \) consists of functions that are continuous throughout \( A^\omega \).

- A stream-valued continuous function \( f : A^\omega \rightarrow B^\omega \) is continuous at \( \alpha : A^\omega \) if
\[
(\forall n \in \omega)(\exists m \in \omega) f(\overline{\alpha}(m)) \subseteq f(\overline{\alpha})(n),
\]
or in other words to find out a finite amount of information about the value, one need only provide a finite amount of information about the argument. \( A^\omega \Rightarrow B^\omega \) consists of functions that are continuous throughout \( A^\omega \). If \( m \) does not depend on \( \alpha \), the function is uniformly continuous. Such a function \( f \) is contractive if it decreases the distance between streams. Prime examples of contractors are the functions \( (a\_\_\_\_ ) : A^\omega \Rightarrow A^\omega \), indexed by \( a : A \).

1.2. Initial algebras and final coalgebras. We use \( (\mu X) F(X) = \mu F \) and \( (\nu X) F(X) = \nu F \) to denote initial and final coalgebras for an endofunctor \( F \), typically an endofunctor on the category of sets.

Initial algebras In general we use \( \text{in} \) for the structure map into the carrier of an initial algebra. Thus \( \text{in} : F(\mu(F)) \rightarrow \mu(F) \). Given an algebra \( C, \gamma : FC \rightarrow C \), we let \( \text{fold}(C; \gamma) \), or simply \( \text{fold}(\gamma) \) to denote the unique morphism \( \delta : \mu(F) \rightarrow C \) such that
\[
\delta \cdot \text{in} = \gamma \cdot F(\delta).
\]
We use \( \text{in}^{-1} \) for the inverse of the structure map, namely \( \text{fold}(F^\text{Fin}) \).
Example: finite sequences \( A^* \triangleq (\mu X) 1 + A \times X \). We use \( \circ \) and \( (\_ ; \_ ) \) as constructors associated with \( \_ \), so

\[
\begin{array}{ccc}
1 & \circ & A^* \\
A \times A^* & (\_ ; \_ ) & A^*
\end{array}
\]

\( in = [\circ, (\_ ; \_ )] : 1 + A \times A^* \rightarrow A^* \)

Example: \( T_A(B) \triangleq (\mu X) B + X A \), defined in section 2. The bifunctor \( T_A(B) \) is covariant in \( B \), and contravariant in \( A \). For fixed \( A \), \( T_A : \text{Set} \rightarrow \text{Set} \) is actually the free monad over the functor \( (\_ ; A) \) (alias \( A \rightarrow \), known as the reader monad). Intriguingly, our constructions all pivot on the freeness of this monad. \( T_A \) is also known as the tree monad. We use \( \text{Ret} \) and \( \text{Get} \) for the constructors associated with \( T_A \). Thus

\[
\begin{array}{ccc}
B & \text{Ret} & T_A(B) \\
(T_A(B))^A & \text{Get} & T_A(B)
\end{array}
\]

\( in = [\text{Ret} | \text{Get}] : B + (T_A(B))^A \rightarrow T_AB \)

**Final coalgebras** In general we use \( out \) for the structure map from the carrier of a final coalgebra. Thus \( out : \nu F \rightarrow F(\nu F) \). Given a coalgebra \( C, \gamma : C \rightarrow FC \), we use \( \text{unfold}(C;\gamma) \), or simply \( \text{unfold}(\gamma) \) (also called the coiteration of \( \gamma \)) to denote the unique coalgebra morphism \( \delta : C \rightarrow \nu F \) such that

\[
out \cdot \delta = F(\delta) \cdot \gamma
\]

We use \( out^{-1} \) for the inverse of the structure map, namely \( \text{unfold}(Fout) \).

Example: streams \( A^\omega \). We use \( \text{hd} \) and \( \text{tl} \) to access components of a stream. \( out = (\text{hd}, \text{tl}) : A^\omega \rightarrow A \times A^\omega \), while \( out^{-1}(a, \alpha) = a; \alpha \).

Example: \( P_A(B) \triangleq (\nu X)T_A(B \times X) \), defined in section 3.

2. **Discrete codomain**

In this section we define a function \( \text{eat} \) of type \( T_AB : A^\omega \rightarrow B \times A^\omega \) that allows us to represent continuous functions in \( A^\omega \Rightarrow B \) using elements of \( T_AB \). Then we give a non-constructive argument that this representation is complete.

2.1. **Definition of eat.** Let \( M_A(B) \triangleq A^\omega \rightarrow (B \times A^\omega) \) be the state monad, with state set \( A^\omega \) (the state is the suffix of the input stream that remains unread. The unit and bind (infix \( \gg= \)) operators of the state monad are

\[
\eta : B \rightarrow M_A(B)
\]

\( (\gg=) : M_A(B) \rightarrow (B \rightarrow M_A(C)) \rightarrow M_A(C) \)

\[
\eta(b) \triangleq (\lambda \alpha) \langle b, \alpha \rangle
\]

\[
(m \gg= f) \triangleq (\lambda \alpha) \text{let } \langle i, \alpha' \rangle = m \alpha \text{ in } f(i, \alpha')
\]

Note that \( M_A \) supports the operation of reading one input:

\[
\text{get} : M_A(A)
\]

\[
\text{get}(\alpha) = \langle \alpha_0, \alpha' \rangle
\]

This function plays an important rôle below in the guise of \( \langle \text{hd}, \text{tl} \rangle : A^\omega \cong A \times A^\omega \).
The function \textit{eat} can be defined in two ways: by straightforward code, and by a more abstract argument. The straightforward definition of \textit{eat} is by structural recursion.

\[
eat : \mathcal{T}_A(B) \to \mathcal{M}_A(B)
\]

\[
eat(\text{Ret } b) = \eta b
\]

\[
eat(\text{Get } \phi) = \text{get } \triangleright= (\text{eat } \cdot \phi)
\]

This definition can be more compactly, if less comprehensibly, using a fold

\[
eat = \text{fold} [\eta | \text{(get } \triangleright=)]
\]

The more abstract form of the definition reveals that \textit{eat} is actually a morphism between monads. It is the unique such from \(\mathcal{T}_A\) (the free monad over \((A \to )\)) to the state monad \(\mathcal{M}_A\) determined by the natural transformation \(n_X : (A \to ) \to X\ar{\omega} \to (X \times A\omega)\)

\[
n_X(\phi, \alpha) = \langle \phi(\alpha_0), \alpha' \rangle
\]

In other words \(n(\phi) = (\phi \times 1) \cdot \text{get}\).

2.2. \textbf{Completeness}. The following result is in essence fairly well known.

\textbf{Theorem 2.1.} (Completeness of representation of \(A\ar{\omega} \Rightarrow B\) by \(T_A B\).) There is a function \(\text{rep} : (A\ar{\omega} \Rightarrow B) \to \mathcal{T}_A(B)\) such that if \(f : A\ar{\omega} \Rightarrow B\) then \(\pi_0 \cdot (\text{eat}(\text{rep } f)) = f\)

Note that \(\text{rep}\) picks a representative for a continuous function from those that give rise to extensionally the same function. When \(A\) is infinite, there are uncountably many such representatives.

\textit{Proof.} (Classical) Suppose that some function \(f : A\ar{\omega} \Rightarrow B\) has no representative. We ‘construct’ an argument \(\alpha : A\ar{\omega}\) at which \(f\) is not continuous. Thence if \(f\) is continuous at all arguments, there exists some \(r : \mathcal{T}_A(B)\) such that \(\text{eat}(r)\) equals \(f\).

Starting with \(f : A\ar{\omega} \to B\), we ‘construct’ an infinite sequence of functions without representatives. In the first place, for some \(\alpha : A\), the function \(f \cdot (\alpha i)\) has no representative. (Else \(f\) itself would have a representation.) By a form of the axiom of dependent choices, if \(f : A\ar{\omega} \Rightarrow B\) has no representative, then for some \(\alpha : A\ar{\omega}\), none of the functions

\[
f_0 = f, f_1 = f_0 \cdot (\alpha(0) i), f_2 = f_1 \cdot (\alpha(1) i) = f \cdot (\alpha(0) i) \cdot (\alpha(1) i), \ldots
\]

have representatives. In particular, none of these functions can be constant. It follows that \(f\) is not constant in any neighbourhood of \(\alpha\), and so \(f\) is not continuous at \(\alpha\).

The structure of this proof is discussed in Dummett [4, pp 49–55], and Troelstra and van Dalen [16, 8.7, p227].

3. \textbf{Stream codomain}

The previous section gave a complete representation of discrete valued continuous functions \(A\ar{\omega} \Rightarrow B\), where \(A\) and \(B\) are discrete. We turn now to stream-valued functions. First we define a function \(\text{eat}_{\infty}\) with type \(P_A B \to A\ar{\omega} \Rightarrow B\ar{\omega}\). We have not been able to find a similar representation in the literature. We have come across the following: if \(k_f\) represents a function \(f : \omega\ar{\omega} \to B\), then \(\phi : \omega\ar{\omega} \to B\ar{\omega}\) is represented by \(k_f\) where \(f(n; \alpha) = \phi(\alpha, n)\). This manoeuvre, which perhaps can be criticised as a ‘hack’, works only when \(\alpha\) is a stream of natural numbers, or encodable as such.
3.1. Definition of $eat_\infty$. We define $eat_\infty$ to be the curried form of a function $e$ of type $(P_A B) \times A^\omega \to B^\omega$ that is continuous in its second argument. Since $B^\omega$ is a final coalgebra, to define a function into it, it is enough to define a coalgebra for $(B \times \_)$ with carrier $P_A(B) \times A^\omega$.

$$
P_A(B) \times A^\omega \xrightarrow{e} B^\omega
$$

$$
\begin{array}{c}
T_A(B \times P_A(B)) \times A^\omega \\
\downarrow \text{app}(eat \times 1) \\
(B \times P_A(B)) \times A^\omega \\
\downarrow \text{assoc} \\
B \times (P_A(B) \times A^\omega) \\
\downarrow 1 \times e \\
B \times B^\omega
\end{array}
$$

So

$$
e : P_A(B) \times A^\omega \to B^\omega
$$

$$
e = \text{unfold}(\text{assoc} \cdot \text{app} \cdot ((eat \cdot out) \times 1))
$$

Then

$$
eat_\infty : P_A(B) \to A^\omega \Rightarrow B^\omega
$$

$$
eat_\infty = \text{curry}(e)
$$

Remark: this definition generalises effortlessly to the case when the codomain is an arbitrary final coalgebra for a strong functor $F$. Let $R \doteq \nu(T_A \cdot F)$.

$$
R \times A^\omega \xrightarrow{e'} \nu F
$$

$$
\begin{array}{c}
T_A(FR) \times A^\omega \\
\downarrow \text{app}(eat \times 1) \\
(FR) \times A^\omega \\
\downarrow \text{strength} \\
F(R \times A^\omega)
\end{array} \xrightarrow{F(e')} F(\nu F)
$$

Though one can thus represent functions from streams into arbitrary final coalgebras, it is not clear what a completeness result for this general representation would be. Without some serious restriction on the functor $F$ it does not seem possible to conjure up a useful topology on $\nu F$. In fact, this is possible for functors which represent terms in a single sorted signature of finite arity operators. One needs to know what a basic neighbourhood is in the codomain in order to give meaning to the notion of a continuous map. We hope to substantiate these remarks in a subsequent publication.

3.2. Definition of $rep_\infty$. The function $eat_\infty$ allows us to interpret an element of the datatype $P_A B$ as a continuous function in $A^\omega \Rightarrow B^\omega$. Now we define a function $rep_\infty$ that allows us to pick a representative for any such continuous function. In the following subsection, we are going to show that $rep_\infty$ is a right-inverse to $eat_\infty$. 
As the codomain of $\text{rep}_\infty$ is to be the carrier of a final coalgebra for the functor $T_A(B \times \cdot)$, we define $\text{rep}_\infty$ as the (unique) coalgebra morphism from a coalgebra for the same functor with carrier $A^\omega \Rightarrow B^\omega$, namely $\rho \cdot \tau$ in the following diagram.

\[
\begin{array}{ccc}
(A^\omega \Rightarrow B^\omega) & \xrightarrow{\text{rep}_\infty} & PA(B) \\
\downarrow \tau & & \downarrow \text{out} \\
T_A(B) \times (A^\omega \Rightarrow B^\omega) & \xrightarrow{\rho} & T_A(B \times PA(B))
\end{array}
\]

So $\text{rep}_\infty = \text{unfold}(\rho \cdot \tau)$. Here

\[
\tau : (A^\omega \Rightarrow B^\omega) \rightarrow T_A(B) \times (A^\omega \Rightarrow B^\omega) \quad \tau(f) = (\text{rep}(\text{hd} \cdot f), \text{tl} \cdot f)
\]

The other component $\rho$ of the structure map of our $T_A \cdot (B \times \cdot)$-coalgebra is a fold. (For clarity, we give it a more general type than we need.) It is in some sense a ‘fast-forward’ operation.

\[
\rho : T_A(B) \times (A^\omega \Rightarrow C) \rightarrow T_A(B \times (A^\omega \Rightarrow C)) \\
\rho(\text{Ret} b, f) = \text{Ret}(b, f) \\
\rho(\text{Get} \phi, f) = \text{Get}(\lambda a. \rho(\phi(a), f \cdot (a : \omega)))
\]

Remarks: $\rho$ is actually an isomorphism. It does not change the shape of a tree, but only decorates the data stored at its leaves. So, for example, $(\pi_0 \cdot \text{eat})(t, \alpha) = (\pi_0 \cdot \text{assoc} \cdot \text{eat})(\rho(t, f), \alpha)$ for any $t : T_A(B)$ and $f : A^\omega \Rightarrow C$.

Unlike the definition of $\text{rep}$, the construction of $\text{rep}_\infty$ from $\text{rep}$ does not involve classical reasoning.

### 3.3. Completeness of $\text{eat}_\infty$.

Now we want to show that the function $\text{eat}_\infty$ is surjective. It is enough to show that $\text{rep}_\infty$ is a right inverse for $\text{eat}_\infty$.

**Theorem 3.1.** (Completeness of representation of $A^\omega \Rightarrow B^\omega$ by $PA(B)$.)

\[ (\text{eat}_\infty \cdot \text{rep}_\infty) = 1_{A^\omega \Rightarrow B^\omega}. \]

**Proof.** We show that the following relation $R$ is a bisimulation on $B^\omega$, and therefore included in the equality relation.

\[ R = \{(f(\alpha), \text{eat}_\infty(\text{rep}_\infty(f, \alpha))) \mid \alpha : A^\omega, f : A^\omega \Rightarrow B^\omega \} \]

It is enough to prove that if $f : A^\omega \Rightarrow B^\omega$ and $\alpha : A^\omega$, then

1. $\text{hd}(f(\alpha)) = \text{hd}(\text{eat}_\infty(\text{rep}_\infty(f, \alpha)))$, and
2. $\text{tl}(f(\alpha)) = R(\text{tl}(\text{eat}_\infty(\text{rep}_\infty(f, \alpha))))$. 


As for (1),

\[
hd(\text{eat}_\infty(\text{rep}_\infty(f), \alpha)) \\
= (\pi_0 \cdot \text{assoc} \cdot \text{eat})(\text{out}(\text{rep}_\infty(f)), \alpha) \\
= (\pi_0 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho \cdot (\text{rep} \times 1) \cdot (\text{hd}, \text{tl} \cdot) f, \alpha) \\
= (\pi_0 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(\text{rep}(\text{hd} \cdot f), \text{tl} \cdot f), \alpha) \\
= \{(T_A \cdot (B \times))\text{rep}_\infty \cdot \rho \text{ doesn't affect shape, or first coordinate of data }\} \\
(\pi_0 \cdot \text{eat})(\text{rep}(\text{hd} \cdot f), \alpha) \\
= \{\text{Completeness in the discrete-valued case}\} \\
hd(f(\alpha))
\]

As for (2), we start by expanding definitions.

\[
tl(\text{eat}_\infty(\text{rep}_\infty(f), \alpha)) \\
= (\text{eat}_\infty \cdot \pi_1 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(\text{rep}(\text{hd} \cdot f), \text{tl} \cdot f), \alpha)
\]

We have to show that that for all \( f : A^\omega \Rightarrow B^\omega \) and \( \alpha : A^\omega \),

\[
tl(f(\alpha)) R (\text{eat}_\infty \cdot \pi_1 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(\text{rep}(\text{hd} \cdot f), \text{tl} \cdot f), \alpha).
\]

By completeness in the discrete-valued case, it is enough to show that for all \( t \in T_A(b) \),

\[
f'(\alpha) R (\text{eat}_\infty \cdot \pi_1 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(t, f'), \alpha).
\]

We argue by induction on the wellfounded structure \( t \).

- In the base case that \( t \) has the form \textbf{Ret} \( b \), calculation shows that

\[
(\text{eat}_\infty \cdot \pi_1 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(t, f'), \alpha)
\]

But \( f'(\alpha) \text{ Ret} (\text{rep}_\infty(f'), \alpha) \), so we are done with this case.

- In the step case that \( t \) has the form \textbf{Get} \( \phi \), calculation shows that

\[
(\text{eat}_\infty \cdot \pi_1 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(t, f'), \alpha)
\]

But by induction hypothesis,

\[
(\text{eat}_\infty \cdot \pi_1 \cdot \text{assoc} \cdot \text{eat})((T_A \cdot (B \times))\text{rep}_\infty \cdot \rho)(\phi(\alpha_0), f' \cdot (\alpha_0 \cdot), \alpha')
\]

\[
\text{ and moreover } (f' \cdot (\alpha_0 \cdot))(\alpha') = f'(\alpha).
\]

So we are done with this case too. \( \square \)

4. Composition

In the previous section we defined a complete representation for continuous functions in \( A^\omega \Rightarrow B^\omega \), using elements of \( P_A B \). As continuous functions are closed under composition, if \( p : P_B C \) represents \( \phi : B^\omega \Rightarrow C^\omega \), and \( q : P_A B \) represents \( \psi : A^\omega \Rightarrow B^\omega \), then there’s some \( r : P_A C \) that represents \( \phi \cdot \psi \). But can we directly compute such an \( r \) from \( p \) and \( q \)? The answer is yes. The computation is reminiscent of cut-elimination in proof theory, though in this case the objects being ‘cut’ together are infinite, non-wellfounded trees, rather than wellfounded derivation trees.
4.1. Definition of composition as an operation on representatives. We define (using coiteration) an operation (‘⊗’) on representations of stream functions that represents the composition of those functions, in the sense

$$\text{eat}_\infty(p \otimes q) = \text{eat}_\infty p \cdot \text{eat}_\infty q$$

for $p : P_B C$ and $q : P_A B$.

First, we define a coalgebra $\chi$ for the functor $T_A \cdot (C \times)$. The carrier will be the product $S \triangleq T_B (C \times P_B C) \times T_A (B \times P_A B)$. The definition is by nested structural recursion, with the outermost recursion on the postponent (the first coordinate), and the inner recursion on the preponent (the second). We present the defining equations in pattern-matching format, to be read from top to bottom.

$$
\begin{align*}
\chi : & T_B (C \times P_B C) \times T_A (B \times P_A B) \to T_A (C \times (T_B (C \times P_B C) \times T_A (B \times P_A B))) \\
\chi(\text{Ret} \langle c, p_{bc} \rangle, t_{ab}) &= \text{Ret} \langle c, (\text{out} p_{bc}, t_{ab}) \rangle \\
\chi(\text{Get} \phi, \text{Ret} \langle b, p_{ab} \rangle) &= \chi'(\phi b, \text{out} p_{ab}) \\
\chi(\text{Get} \psi, \text{Ret} \langle c, p_{bc} \rangle, t_{ab}) &= \text{Get} \langle \lambda a \rangle \chi'(t_{bc}, \psi a)
\end{align*}
$$

It is routine to tease the recursion into nested folds. Note that – due to the outer recursion on the postponent – the last clause in the above definition is needed only for $t_{bc} = \text{Get} \phi$. We prefer to keep it in this general format to facilitate comparison with data-driven composition, defined below.

In this form of composition, priority is given to the postponent’s desire to produce output. No input is consumed until both the postponent and preponent are reading.

$\chi$ gives rise to a composition combinator $\otimes$ as follows. First, $\text{unfold}_A : S \to P_A C$. We define $\otimes$ by precomposition with this unfold.

$$\otimes : P_B C \times P_A B \to P_A C$$

$$p \otimes q \triangleq (\text{unfold}_A)(\text{out} p, \text{out} q)$$

We call $\otimes$ Demand Driven composition.

Altenkirch and Swierstra noticed that another such coalgebra can be defined. The definition is again by nested recursion, but this time with the outermost recursion on the preponent, and the inner recursion on the postponent.

$$
\begin{align*}
\chi' : & T_B (C \times P_B C) \times T_A (B \times P_A B) \to T_A (C \times (T_B (C \times P_B C) \times T_A (B \times P_A B))) \\
\chi'(t_{bc}, \text{Get} \psi) &= \text{Get} \langle \lambda a \rangle \chi'(t_{bc}, \psi a) \\
\chi'(\text{Get} \phi, \text{Ret} \langle b, p_{ab} \rangle) &= \chi'(\phi b, \text{out} p_{ab}) \\
\chi'(\text{Ret} \langle c, p_{bc} \rangle, t_{ab}) &= \text{Ret} \langle c, (\text{out} p_{bc}, t_{ab}) \rangle
\end{align*}
$$

Similar to demand driven composition, the last clause implicitly requires that $t_{ab}$ has the form $\text{Ret} \langle b, p_{ab} \rangle$. Anthropomorphically, this form of composition gives priority to the preponent’s desire to read input. No output is produced until both the postponent and preponent are writing. We call the composition combinator $\otimes'$ to which $\chi'$ gives rise Data Driven composition.

We confess that we find something intensely ‘fishy’ about Data Driven composition. So far though, we haven’t been able to locate anything wrong with it. It seems to be correct, as we shall now see.
4.2. Correctness of composition. It remains to prove that the two operations that we defined above really represent composition. This pivots on the uniqueness property of unfold $\chi$. Exploiting the similarity of the definitions for $\otimes$ and $\otimes'$ we can state the following basic lemma that applies to both. For the sake of readability, the isomorphism $\text{out} : P_A(B) \cong T_A(B \times P_A(B))$ is left implicit.

**Lemma 4.1.** Both composition operators $\otimes \in \{\otimes, \otimes'\}$ satisfy the following laws:

1. $\text{Ret}(c, p_{bc}) \otimes t_{ab} = \text{Ret}(c, p_{bc} \otimes t_{ab})$ (where $t_{ab} = \text{Ret}(b, p_{ab})$ in case $\otimes = \otimes'$)
2. $(\text{Get } \phi) \otimes \text{Ret}(b, p_{ab}) = \phi(b) \otimes p_{ab}$
3. $p_{bc} \otimes (\text{Get } \psi) = \text{Get}(\lambda a. p_{bc} \otimes \psi(a))$ (where $p_{bc} = \text{Get } \phi$ in case $\otimes = \otimes$)

**Proof.** By unfolding the definitions. Actually, it is the desired effect of the definitions that we have these properties. $\square$

We now set up a bisimulation that shows that $t_{bc} \otimes p_{ab}$ and $t_{bc} \otimes' p_{ab}$ really represents the composite $\text{eat}_\omega(t_{bc}) \cdot \text{eat}_\omega(p_{ab})$. Again, the isomorphism $P_A(B) \cong T_A(B \times P_A(B))$ is left implicit.

**Lemma 4.2.**

$$R = \{(\text{eat}_\omega(p_{bc} \otimes t_{ab}, \alpha), \text{eat}_\omega(p_{bc}, \text{eat}_\omega(t_{ab}, \alpha))) | \alpha \in A^\omega\}$$

is a bisimulation on $C^\omega$ if $\otimes \in \{\otimes, \otimes'\}$.

**Proof.** It is enough to prove that

1. $\text{hd}(\text{eat}_\omega(p_{bc} \otimes t_{ab}, \alpha)) = \text{hd}(\text{eat}_\omega(p_{bc}, \text{eat}_\omega(t_{ab}, \alpha)))$
2. $\text{tl}(\text{eat}_\omega(p_{bc} \otimes t_{ab}, \alpha)), \text{tl}(\text{eat}_\omega(p_{bc}, \text{eat}_\omega(t_{ab}, \alpha))) \in R$

for all $p_{bc} \in P_B(C)$ and all $t_{ab} \in P_A(B)$ and all $\alpha \in A^\omega$. The proof relies on the following identities, which are readily derived using Lemma 4.1:

**Case** $p_{bc} = \text{Ret}(c, q_{bc})$.

$$\begin{align*}
\text{eat}_\omega(p_{bc} \otimes t_{ab}, \alpha) &= c \cdot \text{eat}_\omega(q_{bc} \otimes t_{ab}, \alpha) \\
\text{eat}_\omega(p_{bc}, \text{eat}_\omega(t_{ab}, \alpha)) &= c \cdot \text{eat}_\omega(q_{bc}, \text{eat}_\omega(t_{ab}, \alpha))
\end{align*}$$

**Case** $p_{bc} = \text{Get } \phi$ and $t_{ab} = \text{Ret}(b, t_{ab})$.

$$\begin{align*}
\text{eat}_\omega(p_{bc} \otimes t_{ab}, \alpha) &= \text{eat}_\omega((\phi b) \otimes t_{ab}, \alpha) \\
\text{eat}_\omega(p_{bc}, \text{eat}_\omega(t_{ab}, \alpha)) &= \text{eat}_\omega(\phi b, \text{eat}_\omega(t_{ab}, \alpha)).
\end{align*}$$

**Case** $t_{ab} = \text{Get } \psi$.

$$\begin{align*}
\text{eat}_\omega(p_{bc} \otimes t_{ab}, \alpha) &= \text{eat}_\omega(p_{bc} \otimes \psi(a), \alpha) \\
\text{eat}_\omega(p_{bc}, \text{eat}_\omega(t_{ab}, \alpha)) &= \text{eat}_\omega(p_{bc}, \text{eat}_\omega(\psi(a), \alpha)).
\end{align*}$$

The claim for $\otimes = \otimes$ now follows by nested structural recursion, the outer induction on the postponement, the inner induction on the preponent; for $\otimes'$ the nesting is reversed. $\square$

**Corollary 4.3.** Both $\otimes$ and $\otimes'$ represent composition, i.e. for all $p_{bc}$ and all $t_{ab} \in P_A(B)$ we have

$$\text{eat}_\omega(p_{bc} \otimes t_{ab}) = \text{eat}_\omega t_{bc} \cdot \text{eat}_\omega p_{ab}$$

for $\otimes \in \{\otimes, \otimes'\}$.

**Proof.** Immediate from the fact that $R$, defined above, is a bisimulation and the fact that all bisimulations on a final coalgebra are contained in the diagonal. $\square$
5. Conclusion, related work

We have defined computationally natural representations of continuous functions on streams, and proved completeness of these representations for the classically understood notion of continuity. This involved teasing apart the fixed points involved into those that are initial and those that are final. We also defined combinators on representations that represent the composition of the functions they represent.

Our representations are not unique. Interesting further work might be to investigate the equivalence relation between representations corresponding to (extensional) equality between the represented functions. The relation is clearly not decidable, and may be hyper-arithmetic or worse (when the data items consumed and produced are natural numbers).

The set $A^\omega$ of streams of values in a set $A$ is perhaps the simplest example of a final coalgebra, namely for the functor $(\times A)$, a close relative of the set of natural numbers that is an initial algebra for the functor $(+1)$. Final coalgebras are sets of ‘infinite’ values, that can model storage, communication and other evolving devices. In other work that we hope to publish in due course, we have generalised Brouwer’s representations, so as to cover continuous functions between structures of other coinductive types than streams, that is to final coalgebras for a useful class of functors beyond $(A\times)$. Broadly the same results can be obtained as for the stream case, though the generalisation involves more mathematical machinery. The mathematical techniques involve working with indexed families of sets, using an inductive-recursive definition (of such an indexed family) in a crucial way.

It may be possible to extend these techniques yet further to explore representations of continuous functions on final coalgebras for finitary indexed containers, that are endofunctors on slice categories. Some preliminary investigations suggest that this might be rather laborious. On the other hand, it could well be worthwhile. Endofunctors of that kind would allow us to model non-wellfounded proofs, and so connect our work with Mints’ continuous cut-elimination [11], analysed by Buchholz in [2]. Another connection that might be made is with Brotherston and Simpson’s non-wellfounded proof systems in [1]. Yet another is with Niwiński and Igor Walukiewicz’s infinitary proof trees in [13].

Stream processing is a very venerable approach to systems design. Streams were used in a central way in the OS6 operating system of Stoy and Strachey [14], as well as in commercial operating systems. The Unix piping system, introduced by McIlroy, is stream based, with buffering handled by the system. In practical programming, a stream facility is often based on something more complicated than a mathematical stream (involving perhaps EOF, length, buffering, bounds, putback, ...). These more feature-full streams inhabit coinductive types for more elaborate functors than $(A\times)$, but they are not substantially different.

The earliest form of IO in functional programming languages was stream based [10]: a executable program was a (possibly asynchronous) stream processor. Experience quickly showed it is easy to make mistakes in programs using asynchronous interfaces. Mature implementations of IO interfaces are therefore based on synchronous processing, consuming response streams to produce request streams, in a productive or contractive fashion. Some early functional operating systems [9] also used streams (sometimes in a ring) for communication among system processes.

The programming system Fudgets [12] is based on a representation of stream processors similar to the one in this paper, but without our separation of final from initial fixed points. Fudgets are a language for asynchronous stream processing. Various combinators
are available for building up stream processors. Implementations of Fudgets with Haskell have been used to build powerful user interaction (mouse, keyboard, display) interfaces. The programming system Yampa [7], which has been used to produce code for robots (among other things) uses a synchronous dataflow metaphor, that is well aligned with classical control theory, with its signal processors and feedback loops.

It seems obvious that the semantics of feedback loops involves fixpoints, so it is natural (or at least, productive) to focus on contractive functions, because of Banach’s fixed point theorem (see the references in the paper [2]). This states that contractive functions have unique fixed points. In their paper “Ensuring streams flow” [15] Turner and Telford have analysed a productivity requirement for ensuring unique solutions of recursion equations. Productivity seems to be closely related to contractive functions. From another perspective, Bucholtz has designed a calculus for writing (recursive) stream processing functions, (and even functions processing certain not-well-founded trees) which ensures that function are contractive where required [2]. We have not specifically examined the representation of contractive functions, though they are prominent in the form of the functions \((\alpha_0^1)\) in our constructions. Nor have we yet considered representations of uniformly continuous functions.

The notion of arrow, introduced to functional programming by Hughes [8] was developed to express a generalisation of Kleisli arrows, interacting with a tensor combinator according to some reasonable laws. The reference [6] provides a useful perspective. Carlsson [3] and others have considered interaction with the coproduct monoid. Our stream processors behave quite well with respect to composition (\(\cdot\)), but it is not clear to us how nicely they play with operators such as +, × and other multi-input combinators. It seems one needs to model reading from and writing to distinct channels, perhaps with some datatype such as \((\mu X) A + B + X^C + X^D\).

Acknowledgments. Our colleagues Altenkirch and Swierstra have in unpublished work considered the broad topic of modelling impure (effectful) phenomena such as teletype IO [5], mutable heap variables and multithreading. We are grateful to them for interesting conversations on the topic of stream IO, and in particular the Data Driven form of composition mentioned in section 4. Their model of teletype IO, while close to that expounded in this paper, does not address productivity and continuity.

REFERENCES