Strathprints Institutional Repository

Logical Methods in Computer Science
Vol. 5 (3:9) 2009, pp. 1–17
www.lmcs-online.org

Submitted Jul. 14, 2008
Published Sep. 15, 2009

REPRESENTATIONS OF STREAM PROCESSORS USING NESTED FIXED POINTS

PETER HANCOCK [a], DIRK PATTINSON [b], AND NEIL GHANI [c]

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

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

[c] 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

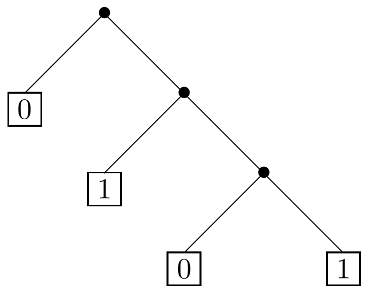
$$f : A^\omega \Rightarrow X$$

where A is a discrete space, A^ω 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^ω. We use the symbol $\Rightarrow$ 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 input-output and coroutines.

1998 ACM Subject Classification: 68P05, 68N18, 54C35.
Key words and phrases: Streams, continuous functions, initial algebras, final coalgebras.
[a] Hancock's work was supported by EPSERC grant EP/C511964/1.

If one is to implement 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 α is determined (or 'secured') by some finite prefix $\overline{\alpha}_n = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{n-1})$ of α.

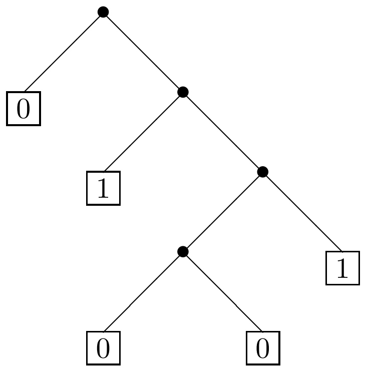
It is fairly clear how to represent continuous functions on A^ω with discrete values in B: take a well-founded tree branching over A, with B's at the leaves. Such a tree represents a continuous function. Start at the root, then use successive entries in the argument stream to steer your way along some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



$$\begin{array}{lcl} f : 2^\omega \to 2 & & \\ f(0,\ldots) & = & 0 \\ f(1,0,\ldots) & = & 1 \\ f(1,1,0,\ldots) & = & 0 \\ f(1,1,1,\ldots) & = & 1 \end{array}$$

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 should be noted 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, where (of course) two functions are the same if their values are equal for all arguments.



It is perhaps a little less obvious that the representation sketched above is complete: *any* continuous function $f : A^\omega \Rightarrow B$ is representable in this way. The most straightforward argument is irredeemably classical: suppose the function has no representation, and derive from this supposition a stream at which it is not continuous. However the completeness of the representation can be established constructively, given only the validity of a certain principle of called 'bar induction', asserting the equivalence of two notions of barred-ness, or covering in Baire space. Here a 'bar' is a monotone subset of A^*, the set of finite lists of A's. One notion of barred-ness is weak, having the form of a quantification over infinite sequences

$$(\forall \alpha : A^\omega)(\exists n : \omega)\, B\langle \alpha_0, \ldots, \alpha_{n-1}\rangle\ .$$

The other notion of barred-ness is strong, being inductively defined, and so having essentially the form of a quantification over subsets of A^*

$$(\forall U \subseteq A^*) B \cup \{\, c : A^* \mid (\forall a : A) U(c \frown a)\,\} \subseteq U \to U\langle\rangle$$

where $\langle\rangle$ denotes the empty list, and $c \frown a$ the list c with a further entry a at the end. Using a variant of this principle, one can show that the two notions of continuity on Baire space (one the usual epsilon-delta definition, the other defined inductively) coincide. This principle is closely related to Brouwer's 'Bar Theorem', for which he presented a fascinating but fallacious[1] argument in three articles. There is an extensive discussion of Brouwer's argument by Dummett in [4, pp 68–75], and a more formal analysis of bar induction by Howard and Kreisel in [8]. A very penetrating discussion of bar induction that is closely related to our representation of continuous functions by well-founded trees is given by Tait in [18].

The inspiration for our representation of continuous functions with discrete codomain was in fact Brouwer's argument for bar-induction, that conjures an inductive structure from a proof of a Π^1_1 statement $\forall \alpha\, \exists n \ldots$. This inductive content was made explicit by Kreisel and Troelstra [20, 8.4, p225] in the form of the class 'K' of neighbourhood functions central to their so-called elimination of choice sequences, discussed in [4, pp 75–81] and [18]. In this paper, we put this inductive structure into a datatype. In fact the paper of Tait's just cited contains (at the bottom of p.195) a definition of what amounts to the function *eat* in section 2.1 below, differing only in notation. We claim no originality for this insight.

Now what about stream-valued continuous functions on A^ω with values in B^ω? 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. Whenever you arrive at a node labelled with an element of B, as you will inevitably do infinitely often, 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.

Is this representation complete? It turns out that every stream-valued function on streams is representable by a non-wellfounded tree of the kind we have described, though the argument is perhaps a little intricate.

Our main contribution, non-trivially extending the state of the art in the 1960's, is to formulate a represention of stream processing components (continuous functions between

[1]Brouwer did not place any restriction such as monotonicity on B. As explained by Dummett [4, pp 68–75] this is definitely an error.

streams, including their composition), different from that customary in the logical literature[2], that fits better with practical implementation of stream computation.

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 not entirely satisfying. The chief advantage of using our data type to program stream processing components is that it ensures liveness, through the use of mixed inductive-coinductive types. The foundations of Haskell are located in a theory of *partial* functions, and not functions in the standard mathematical sense. Totality is something extrinsic, beyond the scope of the type system. Our approach guarantees that the stream processors are total. So it might be better expressed in a language for *total* functional programming, as advocated by Turner [21], and approximated in systems such as Epigram and Agda. This means that evaluation of the constructor form of the value of a function at an argument in its domain must terminate, in our opinion something to be striven for in a practical programming language.

It seems that there are lessons to be learnt from this work for the design of formalisms and systems for developing dependently typed programs. It is not yet entirely clear what facilities for coinductive definition and reasoning such systems need to provide, and in what form. It seems firstly that facilities for inductive-recursive definition may be needed in connection with coinductive structures: the neighbourhoods in coinductive types have an inductive-recursive construction. (Admittedly, this structure does not become fully evident until we consider more general coinductive datatypes than streams.) Secondly, inductive and coinductive definitions are sometimes nested within each other (as in $(\nu X)\,(\mu Y)\,B \times X + Y^A$). Coding our constructions in current systems for dependently typed programming has revealed a number of deficiencies and errors in these systems. Dealing with recursive definitions in which induction and coinduction are combined needs careful analysis, that in our opinion should be based on the universal properties of initial algebras and final coalgebras.

The paper is organised as follows.

- Section 1: preliminaries.
- Section 2: we define the representation of the continuous function space $A^\omega \Rightarrow B$ 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 $A^\omega \Rightarrow B^\omega$ 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.
- Section 5: in conclusion, we summarise what has been done, point out related work, and indicate some directions for further work.

[2]According to this, if k_f represents a function $f : \omega^\omega \to B$, then $\phi : \omega^\omega \to B^\omega$ is represented by k_f where $f(n \mathbin{⨾} \alpha) = \phi(\alpha, n)$. This manoeuver works only when α is a stream of natural numbers, or encodable as such.

The main definitions of the paper can be transcribed quite simply into Haskell. A Haskell encoding can be found at `http://personal.cis.strath.ac.uk/~ng/eating.hs`.

1. Preliminaries

We assume the reader is familiar with the categorical notions of product, coproduct, and exponential, and standard notations associated with these. We use '$\cdot$' as infix notation for composition, with (as usual) postponent at left and preponent at right.

1.1. Streams.

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

We use Greek letters α, β, ... as variables over stream types. We overload the infix operator ($\fatsemi$) (with section notation) our basic means of constructing both streams and non-empty lists. Thus if $a : A$, then the following functions prefix a to streams and to lists.

$$(a\,\fatsemi) : A^\omega \Rightarrow A^\omega \qquad (a\,\fatsemi) : A^* \Rightarrow A^*$$

We also have the empty list $\diamond : A^*$.

As destructors of streams we use hd and tl.

$$hd : A^\omega \rightarrow A$$
$$tl : A^\omega \rightarrow A^\omega$$

For all $a : A$ and $\alpha : A^\omega$ we have

$$hd(a\,\fatsemi\,\_) = a : A$$
$$tl(\_\,\fatsemi\,\alpha) = \alpha : A^\omega$$
$$\alpha = (hd\,\alpha)\,\fatsemi\,(tl\,\alpha) : A^\omega$$

Here we have for clarity written $\_$ for parts of expressions that need not be named. The destructors hd and tl are used implicitly in pattern-matching definitions.

We sometimes write α_0 for $hd\,\alpha$, and α' for $tl\,\alpha$.

We use the function $\overline{()} : A^\omega \rightarrow (A^*)^\omega$ which returns the stream of finite prefixes of its argument. It is defined by $\overline{\alpha}(0) = \diamond$ and $\overline{\alpha}(n+1) = \alpha(0)\,\fatsemi\,(\overline{\alpha'})(n)$.

Streams are endowed with a topology in which the neighbourhoods are given by finite sequences $c : A^*$. Each such represents the predicate $N\,c = \{\,\alpha \mid c = \overline{\alpha}(len\,c)\,\}$ 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^ω to X, where X is either a discrete space D, or a space D^ω 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 α 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^ω.

- A stream-valued continuous function $f : A^\omega \to B^\omega$ is continuous at $\alpha : A^\omega$ if $(\forall n \in \omega)(\exists m \in \omega) f(\overline{\alpha}(m)) \subseteq \overline{f\,\alpha}\,n$, 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^ω. If m does not depend on α, 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 *in* for the structure map into the carrier of an initial algebra. Thus $in : F(\mu\, F) \to \mu(F)$. Given an algebra $C, \gamma : FC \to C$, we let $fold(C;\gamma)$, or simply $fold\,\gamma$ to denote the unique morphism $\delta : \mu\, F \to C$ such that

$$\delta \cdot in = \gamma \cdot F\,\delta.$$

We use in^{-1} for the inverse of the structure map, namely $fold(F\,in)$.

Example: finite sequences $A^* \triangleq (\mu\, X)\, 1 + A \times X$. We use $\diamond$ and $(⨟)$ as constructors associated with $\_^*$, so

$$\begin{tikzcd} 1 \arrow[r, "\diamond"] & A^* \\ A \times A^* \arrow[r, "(⨟)"] & A^* \end{tikzcd}$$
$$in = [\diamond|(⨟)] : 1 + A \times A^* \to A^*$$

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, $T_A : \mathsf{Set} \to \mathsf{Set}$ is actually the free monad over the functor $(\_)^A$ (alias $(A \to)$, 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 **Ret** and **Get** for the constructors associated with T_A. Thus

$$\begin{tikzcd} B \arrow[r, "\mathbf{Ret}"] & T_A B \\ (T_A B)^A \arrow[r, "\mathbf{Get}"] & T_A B \end{tikzcd}$$
$$in = [\mathbf{Ret}|\mathbf{Get}] : B + (T_A B)^A \to T_A B$$

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

$$out \cdot \delta = F\,\delta \cdot \gamma$$

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

Example: streams A^ω. We use hd and tl to access components of a stream. $out = \langle hd, tl\rangle : A^\omega \to A \times A^\omega$, while $out^{-1}\langle a, \alpha\rangle = a\,⨟\,\alpha$.

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

2. Discrete codomain

Recall (from section 1.2) that $T_A B \triangleq (\mu X)\, B + X^A$. In this section we define a function *eat* of type $T_A B \to A^\omega \to B \times A^\omega$ that allows us to represent continuous functions in $A^\omega \Rightarrow B$ using elements of $T_A B$. Then we give a non-constructive argument that this representation is complete.

2.1. Definition of *eat*.

Let $M_A B \triangleq A^\omega \to (B \times A^\omega)$ be the state monad, with state set A^ω. (The state is the suffix of the input stream that remains unread.) The unit and bind (infix $\gg\!=$) operators of the state monad are as follows.

$$\begin{array}{l} \eta : B \to M_A B \\ (\gg\!=) : M_A B \to (B \to M_A C) \to M_A C \\ \eta(b) \triangleq (\lambda\, \alpha)\, \langle b, \alpha \rangle \\ (m \gg\!= f) \triangleq (\lambda\, \alpha)\, \textbf{let}\ \langle i, \alpha' \rangle = m\alpha\ \textbf{in}\ \ f(i, \alpha') \end{array}$$

Note that M_A supports the operation of reading one input:

$$\begin{array}{l} get : M_A A \\ get\, \alpha = \langle \alpha_0, \alpha' \rangle \end{array}$$

This function plays an important rôle below in the guise of $\langle hd, tl \rangle : A^\omega \cong A \times A^\omega$.

The most straightfoward definition of *eat* is by structural recursion.

$$\begin{array}{ll} eat : T_A B & \to M_A B \\ eat(\textbf{Ret}\, b) & = \eta\, b \\ eat(\textbf{Get}\, \phi) & = get \gg\!= (eat \cdot \phi) \end{array}$$

2.2. Completeness.

The following result is in essence well known.

Theorem 2.1. (Completeness of representation of $A^\omega \Rightarrow B$ by $T_A B$.) *There is a function $rep : (A^\omega \Rightarrow B) \to T_A B$ such that if $f : A^\omega \Rightarrow B$ then $\pi_0 \cdot (eat(rep\ f)) = f$*

Note that *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.

Proof. (Classical) Suppose that some function $f : A^\omega \Rightarrow B$ has no representative. We 'construct' an argument $\alpha : A^\omega$ at which f is not continuous. Thence, if f is continuous at all arguments, there exists some $r : T_A B$ such that $eat(r)$ equals f.

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

$$f_0 = f, f_1 = f_0 \cdot (\alpha_0\, ⨾), f_2 = f_1 \cdot (\alpha_1\, ⨾) = f \cdot (\alpha_0\, ⨾) \cdot (\alpha_1\, ⨾), \ldots$$

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

The structure of this proof is discussed in Dummett [4, pp 49–55], and Troelstra and van Dalen [20, 8.7, p227]. Of course, there are other proofs that do not make use of constructively illicit forms of contraposition. For example, there seem to be proofs that use instead monotone bar-induction, and are arguably intuitionistically valid. However as indicated by Tait in [18, pp 194–196], the best we can hope to achieve from a constructive point of view is to find models of suitable systems of constructive reasoning to which we have adjoined an axiom asserting that a function of type $A^\omega \to B$ continuous in the weak ϵ–δ sense is always continuous in the strong inductive sense. It is to be expected that such a model would refute Church's thesis.

3. Stream codomain

Recall (from section 1.2) that $P_A B \triangleq (\nu\, X)\, T_A(B \times X)$. The previous section gave a complete representation of discrete valued continuous functions $A^\omega \Rightarrow B$, where A and B are discrete. We turn now to stream-valued functions. First we define a function eat_∞ with type $P_A B \to A^\omega \Rightarrow B^\omega$. We have not been able to find a similar representation in the literature. Then we provide it with a right-inverse rep_∞.

Of course, $A^\omega \Rightarrow B^\omega$ is isomorphic to $(A^\omega \Rightarrow B)^\omega$, and so its elements can be represented by streams of representations of $A^\omega \Rightarrow B$. However such a representation would be unusable in practice, as the same input stream would have to be scanned again and again to produce successive items in the output stream.

3.1. **Definition of eat_∞.** We define eat_∞ 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^ω is a final coalgebra, to define a function into it, it is enough to define a coalgebra for $(B\times)$ with carrier $P_A B \times A^\omega$.

$$\begin{tikzcd}
P_A B \times A^\omega \arrow[rr, "e"] \arrow[d, "out \times 1"] & & B^\omega \arrow[dddd, "\langle hd, tl \rangle"] \\
T_A(B \times P_A B) \times A^\omega \arrow[d, "eat \times 1"] & & \\
M_A(B \times P_A B) \times A^\omega \arrow[d, "app"] & & \\
(B \times P_A B) \times A^\omega \arrow[d, "assoc"] & & \\
B \times (P_A B \times A^\omega) \arrow[rr, "1 \times e"] & & B \times B^\omega
\end{tikzcd}$$

So

$$e : P_A B \times A^\omega \to B^\omega$$
$$e = unfold(assoc \cdot app \cdot ((eat \cdot out) \times 1)\ .$$

Then

$$eat_\infty : P_A B \to A^\omega \Rightarrow B^\omega$$
$$eat_\infty = curry\, e\ .$$

A more down-to-earth or humane presentation of the definition of eat_∞ follows, as it might be written in a functional programming language.

```
data T a b  =  Ret b | Get(a → T a b)
eat :: T a b → Str a → (b, Str a)
eat (Ret b) as          =  (b, as)
eat (Get f) (a : as)    =  eat (f a) as

data P a b  =  P(T a (b, P a b))
eat∞ :: P a b → Str a → Str b
eat∞ (P t) as  =  let ((b, p), as′) = eat t as in
                    b : eat∞ p as′
```

Remark: this definition generalises effortlessly to the case when the codomain is an arbitrary final coalgebra for a strong functor F (that is, one equipped with a suitable natural transformation $strength : F\,X \times Y \to F(X \times Y))$. Let $R \triangleq \nu(T_A \cdot F)$.

$$\begin{tikzcd}
R \times A^\omega \arrow[r, "e'"] \arrow[d, "out \times 1"] & \nu F \arrow[ddd, "out"] \\
T_A(FR) \times A^\omega \arrow[d, "app \cdot (eat \times 1)"] & \\
(FR) \times A^\omega \arrow[d, "strength"] & \\
F(R \times A^\omega) \arrow[r, "F(e')"] & F(\nu F)
\end{tikzcd}$$

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 the codomain νF. In fact, this is possible for functors that represent a single sorted signature of finite arity operators. We hope to substantiate this remark in a subsequent publication.

3.2. **Definition of rep_∞.**

The function eat_∞ 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_∞ that picks a representative for any such continuous function. In the following subsection, we'll show that rep_∞ is right-inverse to eat_∞.

As the codomain of rep_∞ is to be the carrier of a final coalgebra for the functor $T_A(B \times \_)$, we define rep_∞ 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.

$$\begin{tikzcd}
(A^\omega \Rightarrow B^\omega) \arrow[rr, "rep_\infty"] \arrow[d, "\tau"'] & & P_A B \arrow[dd, "out"] \\
T_A B \times (A^\omega \Rightarrow B^\omega) \arrow[d, "\rho"'] & & \\
T_A(B \times (A^\omega \Rightarrow B^\omega)) \arrow[rr, "T_A(1\times rep_\infty)"] & & T_A(B \times P_A B)
\end{tikzcd}$$

So $rep_\infty = unfold(\rho \cdot \tau)$. Here

$$\begin{array}{rcl}
\tau : (A^\omega \Rightarrow B^\omega) & \to & T_A B \times (A^\omega \Rightarrow B^\omega) \\
\tau\, f & = & \langle rep(hd \cdot f), tl \cdot f\rangle
\end{array}$$

The other component ρ of the structure map of our $T_A \cdot (B\times)$-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.

$$\begin{array}{rcl}
\rho : T_A B \times (A^\omega \Rightarrow C) & \to & T_A(B \times (A^\omega \Rightarrow C)) \\
\rho\langle \mathbf{Ret}\, b, f\rangle & = & \mathbf{Ret}\langle b, f\rangle \\
\rho\langle \mathbf{Get}\, \phi, f\rangle & = & \mathbf{Get}\, (\lambda\, a)\, \rho\langle \phi\, a, f \cdot (a\, ⨾)\rangle
\end{array}$$

Remarks: ρ 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, $(\pi_0 \cdot eat)(t, \alpha) = (\pi_0 \cdot assoc \cdot eat)(\rho(t, f), \alpha)$ for any $t : T_A B$ and $f : A^\omega \Rightarrow C$.

Although *rep* cannot be defined constructively, at least without postulating some form of bar-induction, the construction of rep_∞ from *rep* is a simple matter of programming.

3.3. Completeness of eat_∞.

Now we want to show that the function eat_∞ is surjective. It is enough to show that rep_∞ is a right inverse for eat_∞.

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

$$(eat_\infty \cdot rep_\infty) = 1_{A^\omega \Rightarrow B^\omega} \; .$$

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

$$R = \{(f\, \alpha, eat_\infty(rep_\infty\, f, \alpha)) \mid \alpha : A^\omega, f : A^\omega \Rightarrow B^\omega\}$$

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

(1) $hd(f\, \alpha) = hd(eat_\infty(rep_\infty\, f, \alpha))$, and
(2) $tl(f\, \alpha)\; R\; tl(eat_\infty(rep_\infty\, f, \alpha))$.

As for (1),

$$\begin{array}{cl} & hd(eat_\infty(rep_\infty\, f, \alpha)) \\ = & (\pi_0 \cdot assoc \cdot eat)(out(rep_\infty\, f), \alpha) \\ = & (\pi_0 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho \cdot (rep \times 1) \cdot \langle hd\cdot, tl\cdot\rangle)f, \alpha) \\ = & (\pi_0 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(rep(hd \cdot f), tl \cdot f), \alpha) \\ = & \{(T_A \cdot (B\times))rep_\infty \cdot \rho \textit{ doesn't affect shape, or first coordinate of data } \} \\ & (\pi_0 \cdot eat)(rep(hd \cdot f), \alpha) \\ = & \{\textit{Completeness in the discrete-valued case } \} \\ & hd(f\, \alpha) \end{array}$$

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

$$\begin{array}{cl} & tl(eat_\infty(rep_\infty\, f, \alpha)) \\ = & (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(rep(hd \cdot f), tl \cdot f), \alpha) \end{array}$$

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

$$tl(f\, \alpha)\ R\ (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(rep(hd \cdot f), tl \cdot f), \alpha)\ .$$

By completeness in the discrete-valued case, it is enough to show that for all $t \in T_A B$, $f' : A^\omega \Rightarrow B^\omega$ and $\alpha : A^\omega$,

$$f'(\alpha)\ R\ (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(t, f'), \alpha)\ .$$

We argue by induction on the wellfounded structure t.

- In the base case that t has the form $\mathbf{Ret}\, b$, calculation shows that

  $$\begin{array}{cl} & (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(t, f'), \alpha) \\ = & eat_\infty(rep_\infty(f'), \alpha)\ . \end{array}$$

  But $(f'\, \alpha)\ R\ eat_\infty(rep_\infty(f'), \alpha))$, so we are done with this case.
- In the step case that t has the form $\mathbf{Get}\, \phi$, calculation shows that

  $$\begin{array}{cl} & (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(t, f'), \alpha) \\ = & (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(\phi(\alpha_0), f' \cdot (\alpha_0\, ⨾), \alpha')\ . \end{array}$$

  But by induction hypothesis,

  $$\begin{array}{cl} & (f' \cdot (\alpha_0\, ⨾))(\alpha') \\ R & (eat_\infty \cdot \pi_1 \cdot assoc \cdot eat)((T_A \cdot (B\times))rep_\infty \cdot \rho)(\phi(\alpha_0), f' \cdot (\alpha_0\, ⨾), \alpha')\ , \end{array}$$

  and moreover $(f' \cdot (\alpha_0\, ⨾))(\alpha') = f'(\alpha)$. So we are done with this case too. □

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 $\phi \cdot \psi$. However, the argument for completeness is less than entirely constructive. Can we directly program such an r from p and q? Yes! In fact, in at least two different ways, one 'lazy', or demand driven, and the other 'greedy', or data driven. The computation is reminiscent of cut-elimination in proof theory, though in this case the objects that interact with each other 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 (‘$\otimes$’) on representations of stream functions that represents the composition of those functions, in the sense

$$eat_\infty(p \otimes q) = eat_\infty p \cdot eat_\infty q$$

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

First, we define a coalgebra χ for the functor $T_A \cdot (C\times)$. The carrier will be the product $S \triangleq T_B(C \times P_B C) \times T_A(B \times P_A B)$. First, we present the defining equations for χ in pattern-matching format, as they might be written in a functional program. (This means that in the third equation, t_{bc} must have the form $\mathbf{Get}\,\phi$.) Then we show how to analyse this code into nested structural recursions, and so demonstrate that χ is not just a piece of code, but actually a function defined by universal properties of the functors T_B and T_A.

$$\begin{array}{llll}
\chi : & \overbrace{T_B(C \times P_B C)}^{\textit{postponent}} \times & \overbrace{T_A(B \times P_A B)}^{\textit{preponent}} \to & T_A(C \times S) \\
\chi\langle & \mathbf{Ret}\,\langle c, p_{bc}\rangle & , t_{ab}\rangle & = \mathbf{Ret}\,\langle c, \langle out\,p_{bc}, t_{ab}\rangle\rangle \\
\chi\langle & \mathbf{Get}\,\phi & , \mathbf{Ret}\,\langle b, p_{ab}\rangle\rangle & = \chi\langle \phi\, b, out\,p_{ab}\rangle \\
\chi\langle & t_{bc} & , \mathbf{Get}\,\psi\rangle & = \mathbf{Get}\,(\lambda\, a)\,\chi\langle t_{bc}, \psi\, a\rangle
\end{array}$$

Note that in the second equation, χ at $\mathbf{Get}\,\phi$ is defined in terms of χ at $\phi\, b$, and hence the postponent ‘goes down’ one step in the outer structural recursion (though the preponent may ‘go up’, arbitrarily far).

It is routine to tease the recursion into the form of nested structural recursions. The outer recursion is on the structure of the postponent $T_B(C \times P_B C)$, with an inner or subordinate recursion on the structure of the preponent $T_A(B \times P_A B)$. To write it down, we use a polymorphic function

$$\begin{array}{ll}
\multicolumn{2}{l}{\textit{fold} : (B \to C) \to ((A \to C) \to C) \to T_A B \to C} \\
\textit{fold}\, p\, g\, (\mathbf{Ret}\, b) & = p\, b \\
\textit{fold}\, p\, g\, (\mathbf{Get}\, \phi) & = g\,((\lambda\, a : A)\, \textit{fold}\, p\, g\, (\phi a))
\end{array}$$

to express structural recursion over wellfounded trees, or in categorical terms the initiality of $[\mathbf{Ret}|\mathbf{Get}]$ among algebras $[p|g] : (B + C^A) \to C$. The definition of χ can then be given in the form

$$\begin{array}{ll}
\multicolumn{2}{l}{\chi\langle t_{bc}, t_{ab}\rangle = \textit{fold}\, p\, g\, t_{bc}\, t_{ab}} \\
\mathbf{where} & p : (C \times P_B C) \to T_A(B \times P_A B) \to T_A(C \times S) \\
& p\,\langle c, p_{bc}\rangle\, t_{ab} = \mathbf{Ret}\langle c, \langle out\, p_{bc}, t_{ab}\rangle\rangle \\
& g : (B \to T_A(B \times P_A B) \to T_A(C \times S)) \to T_A(B \times P_A B) \to T_A(C \times S) \\
& g\, f = \textit{fold}((\lambda\,\langle b, p_{ab}\rangle)\, f\, b\, (out\, p_{ab}))\, \mathbf{Get}\ .
\end{array}$$

Note that the carrier for the algebra of the outer recursion is the function space $T_A(B \times P_A B) \to T_A(C \times S)$, while that for the inner recursion is $T_A(C \times S)$.

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.

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

$$\begin{array}{l}
\otimes : P_B C \times P_A B \to P_A C \\
p \otimes q \triangleq (\textit{unfold}\,\chi)\langle out\, p, out\, q\rangle
\end{array}$$

We call $\otimes$ *lazy* composition, since the internal actions of the composite are the minimum necessary to respond to demand for data.

Altenkirch and Swierstra noticed that there is another such coalgebra. We present its definition first in functional programming style, using pattern matching; below we show how the equations can be teased into nested recursions.

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

Because of the top-to-bottom reading of the equations, it is implicit in the last equation that t_{ab} has the form $\mathbf{Ret}(b, p_{ab})$. Anthropomorphically, this form of composition gives priority to the preponent's 'greedy' desire to read input. Whereas with the 'lazy' form, output is produced as soon as the postponent is ready, regardless of the form of the preponent, in this greedier form of composition no output is produced until *both* the postponent and preponent are writing. We call the composition combinator $\otimes'$ to which χ' gives rise *greedy* composition, since the internal actions of the composite are driven by the arrival of data at the input.

What is the mathematical structure of the code for this form of composition? Again, it is definition by nested recursion. One might think[3] that the outer recursion is this time on the structure of the preponent, and the inner recursion on the postponent. In fact, this would not work. In the crucial middle clause (in which the two components communicate), the postponent 'goes down' in the structural order, while the preponent may 'go up', arbitrarily far. A more careful analysis shows that, again, the outer recursion is on the structure of the postponent, with subordinate recursions on the structure of the preponent. In fact there is little formal difference from our definition of χ above, except that the base case of the outer recursion uses another inner recursion rather than a simple explicit definition. The local function p then becomes

$$\begin{array}{l}
p : (C \times P_B C) \to T_A(B \times P_A B) \to T_A(C \times S) \\
p\,\langle c, p_{bc}\rangle = \mathit{fold}\,((\lambda\,\langle b, p_{ab}\rangle)\, \mathbf{Ret}\langle c, \langle \mathit{out}\, p_{bc}, \mathbf{Ret}\langle b, p_{ab}\rangle\rangle\rangle)\, \mathbf{Get}
\end{array}$$

Unfortunately we currently have little of substance to say about how these forms of composition are related. One might well expect that a pipeline implemented with greedy composition would be less responsive (*i.e.* deliver results later) than one expressed with the lazy form.

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* χ. 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 $\mathit{out} : P_A B \cong T_A(B \times P_A B)$ is left implicit.

Lemma 4.1. *Both composition operators* $\copyright \in \{\otimes, \otimes'\}$ *satisfy the following laws:*
(1) $\mathbf{Ret}(c, p_{bc}) \copyright t_{ab} = \mathbf{Ret}(c, p_{bc} \copyright t_{ab})$ *(where* $t_{ab} = \mathbf{Ret}(b, p_{ab})$ *in case* $\copyright = \otimes'$*)*
(2) $(\mathbf{Get}\, \phi) \copyright \mathbf{Ret}(b, p_{ab}) = \phi\, b \copyright p_{ab}$
(3) $p_{bc} \copyright (\mathbf{Get}\, \psi) = \mathbf{Get}\,(\lambda a.\ p_{bc} \copyright \psi\, a)$ *(where* $p_{bc} = \mathbf{Get}\, \phi$ *in case* $\copyright = \otimes$*)*

[3]As did we, at first.

Proof. By unfolding the definitions. Actually, it is the desired effect of the definitions that we have these properties. □

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

Lemma 4.2.
$$R = \{(eat_\infty(p_{bc} \copyright t_{ab}, \alpha), eat_\infty(p_{bc}, eat_\infty(t_{ab}, \alpha))) \mid \alpha \in A^\omega\}$$
is a bisimulation on C^ω *if* $\copyright \in \{\otimes, \otimes'\}$.

Proof. It is enough to prove that
(1) $hd(eat_\infty(p_{bc} \copyright t_{ab}, \alpha)) = hd(eat_\infty(p_{bc}, eat_\infty(t_{ab}, \alpha)))$
(2) $tl(eat_\infty(p_{bc} \copyright t_{ab}, \alpha)), tl(eat_\infty(p_{bc}, eat_\infty(t_{ab}, \alpha))) \in R$
for all $p_{bc} \in P_B C$ and 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 $p_{bc} = \mathbf{Ret}(c, q_{bc})$.
$$eat_\infty(p_{bc} \copyright t_{ab}, \alpha) = c \mathbin{;} eat_\infty(q_{bc} \copyright t_{ab}, \alpha)$$
$$eat_\infty(p_{bc}, eat_\infty(t_{ab}, \alpha)) = c \mathbin{;} eat_\infty(q_{bc}, eat_\infty(t_{ab}, \alpha))$$
Case $p_{bc} = \mathbf{Get}\,\phi$ *and* $t_{ab} = \mathbf{Ret}(b, t_{ab})$.
$$eat_\infty(p_{bc} \copyright t_{ab}, \alpha) = eat_\infty((\phi\, b) \copyright t_{ab}, \alpha)$$
$$eat_\infty(p_{bc}, eat_\infty t_{ab} \alpha) = eat_\infty(\phi\, b, eat_\infty(t_{ab}, \alpha)).$$
Case $t_{ab} = \mathbf{Get}\,\psi$.
$$eat_\infty(p_{bc} \copyright t_{ab}, \alpha) = eat_\infty(p_{bc} \copyright \psi\, a, \alpha)$$
$$eat_\infty(p_{bc}, eat_\infty(t_{ab}, a \mathbin{;} \alpha)) = eat_\infty(p_{bc}, eat_\infty(\psi\, a, \alpha)).$$
The claim for $\copyright = \otimes$ now follows by nested structural recursion, the outer induction on the postponent, the inner induction on the preponent; for $\otimes'$ the nesting is reversed. □

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*
$$eat_\infty(p_{bc} \copyright t_{ab}) = eat_\infty t_{bc} \cdot eat_\infty p_{ab}.$$
for $\copyright \in \{\otimes, \otimes'\}$.

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. □

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.

We consider the main point of this paper to be i) a representation of stream processors as trees - this ensures that our stream processors are total as opposed to the partial functions which exist in the Haskell function space $A^\omega \to B^\omega$; ii) a guarantee that all stream processors can be represented by such trees; and iii) a demonstration that these trees are well suited to computation — this takes the form of an implementation of the composition of stream processing functions directly on the representatives themselves.

There may also be advantages of a more technical nature. Very often when a function is represented by a data structure, such as a wellfounded or infinite tree, the function is automatically 'memoised' – its values for particular arguments are recorded in the data, and need not be recomputed if they are needed again. For example, the representation of functions on finitary inductive types by coinductive trees (in general, final coalgebras for certain rank 2 functors) discovered by Hinze [7] and Altenkirch [1] have this property. The same phenomenon may occur with our representation of stream functions. However their work is concerned with functions on inductive types, as is natural with initial algebras, whereas ours is primarily concerned with functions on coinductive types, which is in the opposite direction from the universal maps associated with final coalgebras.

Our representations are not unique, though different representations of the same function correspond to computationally different behaviour. 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 hyperarithmetic or worse (when the data items consumed and produced are natural numbers).

Another question that may deserve further study is to understand and compare the relation between the lazy and greedy forms of composition introduced in section 4. More generally, it may be worth investigating whether there is a real connection between these forms of composition and superficially similar forms of composition in cut-elimination, and algorithmic game theory.

The set A^ω 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 [13], analysed by Buchholz in [3]. Another connection that might be made is with Brotherston and Simpson's non-wellfounded proof systems in [2]. Yet another is with Niwiński and Walukiewicz's infinitary proof trees in [15].

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 [16], 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 [12]: 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 [11] also used streams (sometimes in a ring) for communication among system processes.

The programming system Fudgets [14] 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 [9], 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 may be natural to focus on *contractive* functions, because of Banach's fixed point theorem (see the references in the paper [3]). This states that contractive functions have unique fixed points. In their paper "Ensuring streams flow" [19] 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, Buchholz has designed a calculus for writing (recursive) stream processing functions, (and even functions processing certain not-well-founded trees) which ensures that functions are contractive where required [3]. We have not specifically examined the representation of contractive functions, though they are prominent in the form of the functions $(\alpha_0 \mathbin{;})$ in our constructions. Nor have we yet considered representations of uniformly continuous functions.

The notion of arrow, introduced to functional programming by Hughes [10] was developed to express compositional infrastructure in programming generalising that of Kleisli morphisms for a monad, and crucially interacting with a tensor combinator according to some reasonable laws. The reference [6] provides a useful perspective. Abstractly, an arrow is a monoid in a certain category of bifunctors. 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 $+$, $\times$ and other multi-input combinators. It may be that one has to get to grips with notions of fairness, such as fair merging, in connection with such combinators. Another direction for further development is to investigate combinations of stream processors in which, as in many applications of stream processing, there are forms of feedback, or looping.

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 for pointing out (and debugging) the 'greedy' form of composition mentioned in section 4. Their model of teletype IO in [17], while close to that expounded in this paper, does not address productivity and continuity. Finally we thank the referees for their close scrutiny of the paper, and many valuable suggestions.

References

[1] T. Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 8 – 21, 2001.

[2] J. Brotherston and A. Simpson. Complete sequent calculi for induction and infinite descent. In *Proceedings of LICS-22*, pages 51–60. IEEE Computer Society, July 2007.

[3] W. Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Ann. Pure Appl. Logic*, 136(1-2):75–90, 2005.

[4] M. Dummett. *Elements of intuitionism.* Clarendon Press, Oxford, 2000. 2nd edition.

[5] A. D. Gordon. *Functional programming and input/output.* Cambridge University Press, New York, NY, USA, 1994.

[6] C. Heunen and B. Jacobs. Arrows, like monads, are monoids. *Electronic Notes in Theoretical Computer Science*, 158:219–236, May 2006.

[7] R. Hinze. Memo functions, polytypically! In J. Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 17–32, jul 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[8] W. A. Howard and G. Kreisel. Transfinite induction and bar induction of types zero and one, and the role of continuity in intuitionistic analysis. *J. Symb. Log.*, 31(3):325–358, 1966.

[9] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[10] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000.

[11] S. B. Jones and A. F. Sinclair. Functional programming and operating systems. *Comput. J.*, 32(2):162–174, 1989.

[12] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, 1965.

[13] G. Mints. Finite investigations of transfinite derivations. *J. Sov. Math.*, 10, 1978.

[14] A. K. Moran, D. Sands, and M. Carlsson. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Computer Science*. Springer-Verlags, Apr. 1999.

[15] D. Niwiński and I. Walukiewicz. Games for the μ-calculus. *Theor. Comput. Sci.*, 163(1-2):99–116, 1996.

[16] J. E. Stoy and C. Strachey. Os6 - an experimental operating system for a small computer. part 1: general principles and structure. *Comput. J.*, 15(2):117–124, 1972.

[17] W. Swierstra and T. Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 25–36, 2007.

[18] W. W. Tait. Constructive reasoning. In B. V. Rootselaar and J. Staal, editors, *Logic, Methodology and Philosophy of Science III*, Studies in Logic and the Foundations of Mathematics, pages 185–200, Amsterdam, 1968. North-Holland.

[19] A. Telford and D. Turner. Ensuring streams flow. In *Algebraic Methodology and Software Technology*, pages 509–523, 1997.

[20] A. Troelstra and D. van Dalen. *Constructivism in Mathematics.* North-Holland, 1988. 2 volumes.

[21] D. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.