Categories of Processes Enriched in Final Coalgebras

Sava Krstić[1], John Launchbury[1], and Duško Pavlović[2]

[1] Oregon Graduate Institute {krstic,jl}@cse.ogi.edu
[2] Kestrel Institute dusko@kestrel.edu

Abstract. Simulations between processes can be understood in terms of coalgebra homomorphisms, with homomorphisms to the final coalgebra exactly identifying bisimilar processes. The elements of the final coalgebra are thus natural representatives of bisimilarity classes, and a denotational semantics of processes can be developed in a final-coalgebra-enriched category where arrows are processes, canonically represented. In the present paper, we describe a general framework for building final-coalgebra-enriched categories. Every such category is constructed from a multivariant functor representing a notion of process, much like Moggi's categories of computations arising from monads as notions of computation. The "notion of process" functors are intended to capture different flavors of processes as dynamically extended computations. These functors may involve a computational (co)monad, so that a process category in many cases contains an associated computational category as a retract. We further discuss categories of resumptions and of hyperfunctions, which are the main examples of process categories. Very informally, the resumptions can be understood as computations extended in time, whereas hypercomputations are extended in space.

1 Introduction

A map $A \times X \longrightarrow B \times X$ can be construed as a function from A to B *extended in time*, represented by a set X of states. For each state in X and each input from A, such a function gives an output in B, and the next state in X. This is the usual representation of a *transducer*.

A map $A^X \longrightarrow B^X$, on the other hand, can be construed as a function from A to B *extended in space*, represented by a set X of storage cells. For each assignment of A-values to all X-cells, such a function gives an assignment of B-values to the same cells.

By transposition, a transducer $A \times X \longrightarrow B \times X$ can equivalently be viewed

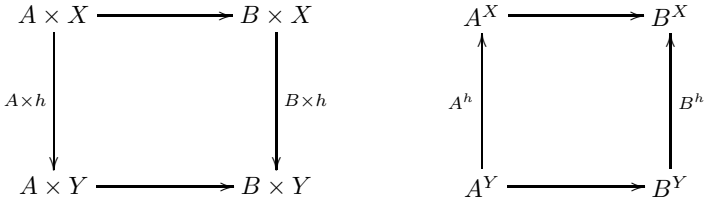
- as a Kleisli morphism $A \longrightarrow (B \times X)^X$ for the monad $B \mapsto (B \times X)^X$, or
- as a coalgebra $X \longrightarrow (B \times X)^A$ for the functor $X \mapsto (B \times X)^A$.

Similarly, a *function with storage* $A^X \longrightarrow B^X$ can be viewed

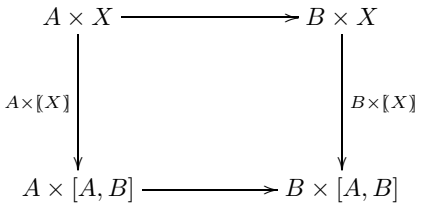
- as a Kleisli morphism $A^X \times X \longrightarrow B$ for the comonad $A \mapsto A^X \times X$, or

– as a coalgebra $X \longrightarrow B^{A^X}$ for the functor $X \mapsto B^{A^X}$.

In both cases, there is a double category [Gra74] displaying the structure and behaviour as the horizontal and the vertical arrows respectively. It has pairs $\langle A, X\rangle$ as objects, where the component A is to be thought of as a data type, and X as a state space. The horizontal arrows represent computations, captured as transducers (functions with storage), or by transposition as the Kleisli morphisms for the corresponding (co)monad. By the other transposition, they become coalgebras, and the vertical arrows arise as the coalgebra homomorphisms between them. They capture dynamics of the computations. The double morphisms are thus, respectively, the commutative squares of the form



This picture can be relativized, in favorable situations, over computational monads, or comonads. The problem with it, however, is a lot of redundancy, since arbitrary sets as state spaces provide a very loose representation of computational behavior. To pin down the computational semantics in the case of transducers, where many of them can be computationally equivalent, one usually seeks a way to construct *minimal representatives*. Recently, coalgebra has been proposed as a uniform framework for such constructions, where minimal representatives are obtained as elements of final coalgebras [Acz88,TR98,JR97]. Given a transducer $A \times X \longrightarrow B \times X$, there is a unique homomorphism $(\!|X|\!)$ to the final coalgebra $[A, B]_R$ of the functor $RX = (B \times X)^A$



It turns out that two states $x \in X$ and $y \in Y$ (Y being the state space of another transducer $A \times Y \longrightarrow B \times Y$) lead to equivalent computations if and only if $(\!|X|\!)(x) = (\!|Y|\!)(y)$.

These remarks can be repeated for functions with storage: the stores $x \in X$ and $y \in Y$ can be reasonably assumed to be computationally indistinguishable if and only if their images in the final coalgebra $[A, B]_H$ of the functor $HX = B^{A^X}$ are equal.

For historical perspective, we note that methods of representing processes in categories by coalgebra-like structures go back at least to the systems theory work of Arbib and Manes in the late sixties [Arb66,AM74]. The idea that

minimal representatives of processes can be viewed and obtained as elements of final coalgebras has probably originated in Aczel's work [Acz88] on semantics of concurrency in terms of hypersets, which form final coalgebras of the powerset functor.

Elements of $[A, B]_R$ are known as *resumptions*. Composition of transducers induces composition of resumptions, and resumption domains are the arrow sets of a category. The observation that process categories are final coalgebra-enriched was developed in an unpublished joint work of Abramsky and the third author; some instances are given in [Abr96b]. On the other hand, *hyperfunction domains* $[A, B]_H$ also form a category, as recently discovered in [LKS00]. While double categories present both structural (Kleisli) and behavioral (coalgebra) aspect of transducers and functions with storage, by passing to final coalgebras we extract the canonical behaviors while preserving the structural aspect. The (one-dimensional) categories of resumptions and hyperfunctions provide a more convenient ambient than the double categories from which they are distilled.

Following the idea that minimal representatives of processses are elements of final coalgebras, in the next section we describe an abstract situation when final coalgebras for a family of functors yield a category, *viz.* its hom-sets. This is our general framework of the *coalgebra enriched categories*. It applies to the endofunctors $RX = (B \times X)^A$ and $HX = B^{A^X}$, and yields, respectively, the categories of resumptions and hyperfunctions; their morphisms can be thought of as (abstract, computational) functions extended in time, or space. The same construction applies, however, to a wide class of more general endofunctors, involving, for example, various notions of computation captured by computational monads and comonads [Mog91,BG92]. Notably, the functors

$$R_M XAB = (M(B \times X))^A \qquad \text{and} \qquad H_G XAB = B^{G(A^X)}$$

considered in Section 3, for suitable monads M and comonads G lead to coalgebra enriched categories of computations extended in time, or space.

In Section 4, we specialize to hyperfunctions and discuss the question of equivalence of coinductive definitions of hyperfunction operations with the recursive definitions given in the *Haskell* package of [LKS00]. This turned out to be a surprisingly difficult problem, so we limit ourselves to a brief discussion, leaving out the proofs.

2 Final coalgebra enrichment

We represent a notion of process as a functor, or type constructor, T, which for types X, A and B yields the type $TXAB$ of computations with inputs of type A and outputs of type B, parametrized by a *process type* X representing the process state. The process type can be understood as a space of states or storage cells, as described in the previous section, or, more generally, it can be any kind of space in which some computational, or even physical processes can be analyzed. Processes can then be presented as coalgebras

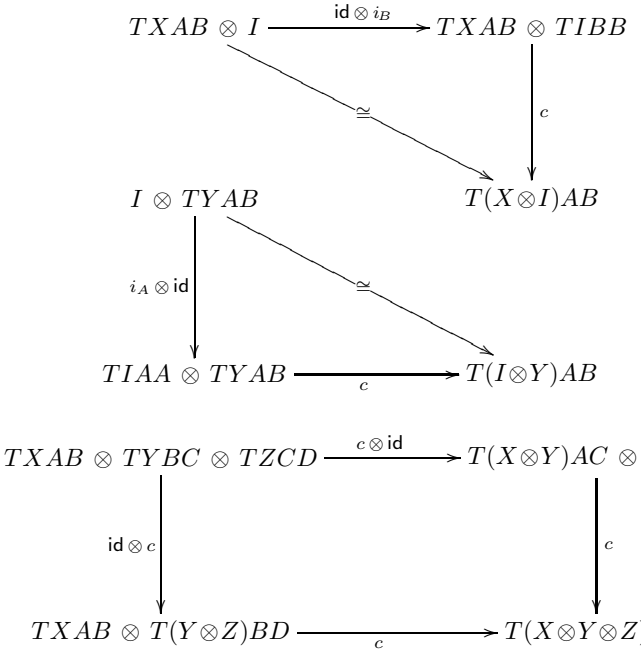
$$X \longrightarrow TXAB$$

assigning to each point of space a computation, involving the inputs from A, the outputs from B, as well as some further elements of X. For example, when X is the space of states, the computation yields the next states; when X is the set of storage cells, the computation may use and update their contents. Processes are thus viewed as computations extended through the process space X, which may be viewed as an abstraction of the temporal, or of the spatial dimension, in various ways.

Definition 1. *Let $\langle \mathcal{C}, \otimes, I \rangle$ be a monoidal category. A* notion of process *over $\mathcal{C}$ is a functor $T \colon \mathcal{C} \times \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$ equipped with extranatural families of maps*

$$i_A : I \longrightarrow TIAA$$
$$c_{XYABC} : TXAB \otimes TYBC \longrightarrow T(X \otimes Y)AC$$

satisfying the conditions



(For unexplained terminology, the reader is referred to [Kel82].)

The idea behind the structure attached to each notion of process T is that

- for each type A, i_A denotes the identity process, which simply outputs its inputs, over the trivial process space I;
- for all types A, B and C, c_{XYABC} sequentially composes the processes from A to B over the space X with the processes from B to C over Y; it hides the intermediary data from B, and produces processes from A to C, over the product space $X \otimes Y$.

The imposed commutativity conditions pin down these intended meanings.

Examples of process notions are given by the functors R_M and H_G defined in the Introduction; see the next section.

As we already mentioned, the representation of processes as coalgebras is loose: many different coalgebras may represent behaviorally equivalent processes, indistinguishable in terms of their computational interpretation. This motivates a whole branch of research on the various notions of simulation, bisimulation, observational equivalence etc. The role of coalgebra in this realm is well known [TR98,Rut96]. The minimal representatives of coalgebras with respect to the canonical notions of behavior are usually captured as the elements of the final coalgebras.

So one naturally tries to use the final process coalgebras as the hom-sets of (canonically represented) processes. And indeed, the structure of the notion of process gives rise to the category structure upon such hom-sets.

Theorem 1. *If T is a notion of process over a category $\mathcal{C}$ and if final $T(-)AB$-coalgebras, denoted $[A,B]$, exist for every A,B, then there exists a $\mathcal{C}$-enriched category $\mathcal{P}$ whose objects are the same as the objects of $\mathcal{C}$ and whose hom-objects are the coalgebras $[A,B]$.*

The category $\mathcal{P} = \mathcal{P}_T$ is called the process category *induced by the notion of process T.*

Proof. We will use the notation $\xi = \xi_{AB}$ for the final coalgebras $[A,B] \longrightarrow T[A,B]AB$.

The map i_A is a $T(-)AA$-coalgebra, so we define the $\mathcal{P}$-identity on A as the anamorphism $u_A \colon I \longrightarrow [A,A]$ induced by i_A.

To define the $\mathcal{P}$-composition

$$k_{ABC} : [A,B] \otimes [B,C] \longrightarrow [A,C]$$

we use the composite

$$\begin{aligned} \kappa_{ABC} \;:\; [A,B] \otimes [B,C] &\xrightarrow{\xi\otimes\xi} T[A,B]AB \otimes T[B,C]BC \\ &\xrightarrow{c} T([A,B]\otimes[B,C])AC \end{aligned}$$

It is a $T(-)AC$-coalgebra on $[A,B]\otimes[B,C]$, and k_{ABC} is defined as the anamorphism induced by it.

To prove the associativity of k, it suffices to show that the maps occurring in the diagram

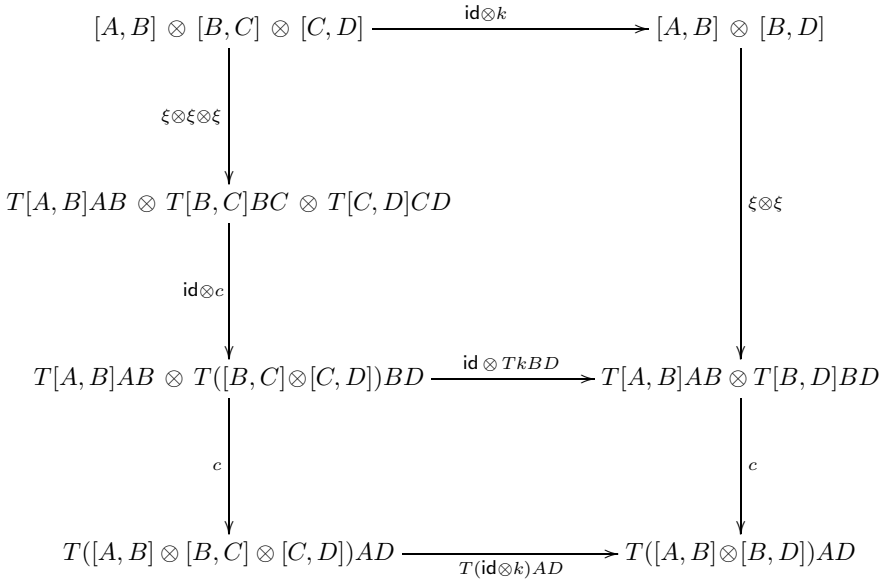
$$\begin{array}{ccc} [A,B]\otimes[B,C]\otimes[C,D] & \xrightarrow{\mathsf{id}\otimes k} & [A,B]\otimes[B,D] \\ \downarrow{\scriptstyle k\otimes\mathsf{id}} & & \downarrow{\scriptstyle k} \\ [A,C]\otimes[C,D] & \xrightarrow[k]{} & [A,D] \end{array}$$

are actually maps of $T(-)AD$-coalgebras. The diagram will then commute because $[A, D]$ is final. The two maps labelled k are k_{ABD} and k_{ACD}, and they are coalgebras by definition. Thus, we only need to define a $T(-)AD$-coalgebra κ_{ABCD} on $[A, B] \otimes [B, C] \otimes [C, D]$ and show that the maps $k_{ABC} \otimes \mathsf{id}$ and $\mathsf{id} \otimes k_{BCD}$ of the last diagram are coalgebra maps from κ_{ABCD} to κ_{ACD} and κ_{ABD} respectively.

We define κ_{ABCD} as the composite

$$\begin{aligned} \kappa_{ABCD} : [A, B] \otimes [B, C] \otimes [C, D] &\longrightarrow \\ \xrightarrow{\xi \otimes \xi \otimes \xi} T[A, B]AB \otimes T[B, C]BC \otimes T[C, D]CD &\\ \xrightarrow{k'} T([A, B] \otimes [B, C] \otimes [C, D])AD & \end{aligned}$$

where κ' is the diagonal of the rectangle expressing the associativity condition $c \circ (c \otimes \mathsf{id}) = c \circ (\mathsf{id} \otimes c)$ in Definition 1, with X, Y, Z instantiated with $[A, B]$, $[B, C]$, $[C, D]$ respectively. Thus, κ_{ABCD} is the left column map in the diagram



while κ_{ABD} is seen as the right column map. The diagram commutes because the two constituent diagrams commute: the top pentagon by definition of k_{ABD} and the bottom rectangle by the naturality of c.

An analogous diagram shows that $k \otimes \mathsf{id}$ is a coagebra map from κ_{ABCD} to κ_{ACD}, and this finishes the proof of associativity of the $\mathcal{P}$-composition k.

The unit axioms for u_A are checked in a similar fashion. Finally, the naturality of $\mathcal{P}$-composition and its units follows from extranaturality of families i_A and c_{XYABC}.

3 Examples: state and storage enrichment

Although the definition of the notion of process, and the general construction of Theorem 1 only require the monoidal structure on $\mathcal{C}$, our main motivating examples arise in the framework of *autonomous*, i.e. closed symmetric monoidal categories. In fact, the most interesting situations, some of which are referred to in the propositions below, are supported by the *cartesian* closed structure. The symbols $\multimap$ and $\Rightarrow$ will denote cotensor operations in autonomous and cartesian closed categories respectively.

3.1 Static process categories

For every monad M over an autonomous category $\mathcal{C}$, the Kleisli category $\mathcal{K}_M$ (the category of *M-computations* in the sense of Moggi [Mog91]) can be viewed as the process category $\mathcal{P}_T$ for the functor

$$T_M X A B = A \multimap M B$$

This is, of course, a degenerate process category: its hom-objects are the final coalgebras of functors constant in X. Indeed, if processes are construed as computations extended in time or space, then static processes boil down to mere computations, and the corresponding process categories $\mathcal{P}_{T_M}$ to the categories of computations $\mathcal{K}_M$.

More generally, there is a *static* process category corresponding to every functor $E\colon \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathcal{C}$ that enriches $\mathcal{C}$ over itself; the requisite notion of process is $TXAB = EAB$.

Every process notion T has an associated static one, T_{st}, defined by $T_{st}XAB = TIAB$. Its final coalgebras are $[A,B]_{st} = TIAB$, and by precomposing the map $T\,!\,\mathsf{id}\,\mathsf{id}\colon T[A,B]AB \longrightarrow [A,B]_{st}$ with $\xi\colon [A,B] \longrightarrow T[A,B]AB$ we define

$$\mathsf{proj}\colon [A,B] \longrightarrow [A,B]_{st}$$

which, as one can easily check, is an identity-on-objects functor

$$\mathsf{proj}\colon \mathcal{P}_T \longrightarrow \mathcal{P}_{T_{st}}.$$

In the most interesting examples, the static process category is some category of computations. Since computations can be treated as (static) processes, proj will usually be a retraction.

3.2 Resumptions

Every strong monad M over an autonomous category $\mathcal{C}$ with enough final coalgebras defines a category $\mathcal{R}_M$ of *M-resumptions*. By definition, $\mathcal{R}_M$ is the proces category $\mathcal{P}_{R_M}$ induced by the notion of process consisting of the functor $R_M : \mathcal{C} \times \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$, defined by

$$R_M X A B = A \multimap M(X \otimes B)$$

and the transformations

$$\begin{array}{lll} i: & I & \longrightarrow A \multimap M(I \otimes A) \\ c: & \left.\begin{array}{l} A \multimap M(X \otimes B) \otimes \\ B \multimap M(Y \otimes C) \end{array}\right\} & \longrightarrow A \multimap M(X \otimes Y \otimes C) \end{array}$$

obtained by transposing, respectively,

- the monad unit $I \otimes A \longrightarrow M(I \otimes A)$, and
- the composite

$$\begin{aligned} & A \otimes (A \multimap M(X \otimes B)) \otimes (B \multimap M(Y \otimes C)) \\ \xrightarrow{(1)} & M(X \otimes B) \otimes (B \multimap M(Y \otimes C)) \\ \xrightarrow{(2)} & M\,(X \otimes B \otimes (B \multimap M(Y \otimes C))) \\ \xrightarrow{(3)} & M\,(X \otimes M(Y \otimes C)) \\ \xrightarrow{(4)} & MM\,(X \otimes Y \otimes C) \\ \xrightarrow{(5)} & M\,(X \otimes Y \otimes C) \end{aligned}$$

  where maps (1) and (3) are derived from the evaluation $U \otimes (U \multimap V) \longrightarrow V$, (2) and (4) from the tensorial strength $U \otimes MV \longrightarrow M(U \otimes V)$, and (5) is a component of the monad cochain $MM \longrightarrow M$.

The fact that this structure satisfies the conditions of Definition 1 can be checked by a routine, though lengthy diagram chase.

When $\mathcal{C}$ is the category of sets and M is the power set functor, we obtain the classical example of resumptions: $[A, B]$ is the set of $A \times B$-labelled hypersets (synchronization trees) [Acz88], and $\mathcal{R}_M$ fully embeds in the category SProc of [Abr96a].

Proposition 1. *If $\mathcal{C}$ is* cartesian *closed and the category $\mathcal{R}_M$ of M-resumptions exists, then the category $\mathcal{K}_M$ of M-computations is a retract of $\mathcal{R}_M$.*

Proof. Since $R_M IAB = A \Rightarrow M(I \times B) \cong A \Rightarrow MB$, the static process category associated with $\mathcal{R}_M$ is indeed $\mathcal{K}_M$; we need maps

$$\mathsf{lift}\colon (A \Rightarrow MB) \longrightarrow [A, B]$$

that split $\mathsf{proj}\colon [A, B] \longrightarrow (A \Rightarrow MB)$.

Since $[A, B] = [A, B]_{R_M}$ is the final coalgebra for the functor $R_M(-)AB$, we can define lift as the anamorphism for the coalgebra on $A \Rightarrow MB$ obtained by transposing the composite

$$A \times (A \Rightarrow MB) \xrightarrow{\langle \epsilon, \pi' \rangle} MB \times (A \Rightarrow MB) \xrightarrow{\vartheta} M(B \times (A \Rightarrow MB))\ ,$$

where ε is evaluation and ϑ is tensorial strength.

It remains to prove $\mathsf{proj} \circ \mathsf{lift} = \mathsf{id}$, but this follows from the definitions.

Viewing functions as degenerate computations, i.e. the base category $\mathcal{C}$ as the Kleisli category $\mathcal{K}_{\mathsf{Id}}$, we derive that $\mathcal{C}$ is a retract of $\mathcal{R} = \mathcal{R}_{\mathsf{Id}}$.

3.3 Hyperfunctions and hypercomputations

The *G-hypercomputations* are the arrows of the process category $\mathcal{H}_G$ induced by the notion of process consisting of the functor $H_G : \mathcal{C} \times \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \longrightarrow \mathcal{C}$ given on objects by

$$H_G X A B = G(X \multimap A) \multimap B$$

and the transformations

$$\begin{array}{lll} i : & I & \longrightarrow G(I \multimap A) \multimap A \\ c : & \left.\begin{array}{l} G(X \multimap A) \multimap B \otimes \\ G(Y \multimap B) \multimap C \end{array}\right\} & \longrightarrow G\,(X \otimes Y \multimap A) \multimap C \end{array}$$

where G is a comonad possessing a tensorial strength $A \otimes GB \longrightarrow G(A \otimes B)$ and a *cotensorial strength*

$$(GA \multimap B) \longrightarrow G(A \multimap B).$$

Space constraints do not allow us to elaborate on the non-standard notion of cotensorial strength; the coherence conditions that need to be imposed on it are analogous to the well-known ones for the tensorial strength. Computationally, tensorial strength allows variables to enter and exit computations, whereas the cotensorial strength ensures the same for the abstraction.

Of course, just like functions are degenerate computations relative to the identity monad, *hyperfunctions* are degenerate hypercomputations, obtained by omitting G in the above definition.

The transformations i and c are obtained by transposing

- the counit $G(I \multimap A) \longrightarrow (I \multimap A)$, and
- the composite

$$\begin{aligned} & G(X \otimes Y \multimap A) \otimes (G(X \multimap A) \multimap B) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(1)}\ & G\,(Y \multimap (X \multimap A)) \otimes (G(X \multimap A) \multimap B) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(2)}\ & G((Y \multimap (X \multimap A)) \otimes ((X \multimap A) \multimap B)) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(3)}\ & G(Y \multimap B) \otimes (G(Y \multimap B) \multimap C) \\ \xrightarrow{(4)}\ & C \end{aligned}$$

  where cotensorial strength and currying are used for (1), tensorial strength for (2), and evaluation for (3) and (4).

Checking that the requirements of a process notion are satisfied is again routine, but this time fairly tedious.

Proposition 2. *If $\mathcal{C}$ is* cartesian *closed and the category $\mathcal{H}_G$ of G-hypercomputations exists, then the category $\mathcal{K}_G$ of G-computations is a retract of $\mathcal{H}_G$.*

Proof. The static process category associated with $\mathcal{H}_G$ is $\mathcal{K}_G$. The injection

$$\mathsf{lift}\colon (GA \Rightarrow B) \longrightarrow [A, B]$$

can be defined as the anamorphism for the coalgebra on $GA \Rightarrow B$, obtained by transposing the composite

$$\begin{aligned} & G((GA \Rightarrow B) \Rightarrow A) \times (GA \Rightarrow B) \\ & \xrightarrow{\langle \pi', \vartheta \rangle} (GA \Rightarrow B) \times G((GA \Rightarrow B) \times ((GA \Rightarrow B) \Rightarrow A)) \\ & \xrightarrow{\mathsf{id} \times G\epsilon} (GA \Rightarrow B) \times GA \\ & \xrightarrow{\epsilon'} B \end{aligned}$$

Again, $\mathsf{proj} \circ \mathsf{lift} = \mathsf{id}$ has a straightforward proof.

As a corollary (when $G = \mathsf{Id}$) we obtain that the base category $\mathcal{C}$ is a retract of the category $\mathcal{H}$ of hyperfunctions.

4 Hyperfunctions recursively

Hyperfunction types can be easily defined in a programming language which allows recursive definitions of data types (*Haskell*, for example). Any programming with them, however, would require using the basic operators like hyperfunction units, composition, and the lifting function. We have defined all of them as anamorphisms and it is not clear how such definitions can translate into a convenient programming language. The question is really not so much of translating, but of matching, for all basic hyperfunction operations were introduced in [LKS00] by means of recursive definitions, and we now have a problem of showing that each recursive definition is equivalent to the corresponding coalgebraic one.

Of course, the recursive definitions take place in a more specific setting provided by a category of domains. Without being too specific, let us assume that our base category is one of pointed domains, suitable for modeling recursive types by standard bilimit construction [AJ94]. The notation $[A, B]$ is now for the canonical solution of the equation $X \cong (X \Rightarrow A) \Rightarrow B$. By the Bekić Lemma, the pair of domains $[A, B]$ and $[B, A]$ must be the canonical solution to the system of equations

$$\begin{aligned} X &\cong Y \Rightarrow B \\ Y &\cong X \Rightarrow A. \end{aligned}$$

Consequently, there is a (canonical) isomorphism $[A, B] \longrightarrow ([B, A] \Rightarrow B)$ whose transpose

$$(\cdot) : [A, B] \times [B, A] \longrightarrow B$$

can be considered a *hyperfunction application operation*. Thus, any equation $u \cdot v = e$, where e is an expression of type B (assuming u and v are of types $[A, B]$ and $[B, A]$ respectively) defines a hyperfunction u. For example, for each b in B

there exists a "constant hyperfunction" k_b, defined by $k_b \cdot v = b$. More generally, for every function $f\colon A \longrightarrow B$ there exists a corresponding hyperfunction $\widetilde{f}$ recursively defined by

$$\widetilde{f} \cdot u = f(u \cdot \widetilde{f}). \tag{1}$$

Note that $\widetilde{f} \cdot k_a = f(a)$ so that $\widetilde{f}$ extends f in some sense, and the first question arises: *Does the equation (1) define the same function as* lift *of Proposition 2?*

Consider now the equation

$$\theta_A \cdot u = u \cdot \theta_A, \tag{2}$$

where both θ_A and u are in $[A, A]$. We can view the equation as a recursive definition of θ_A. It follows from (1) that $\widetilde{\mathsf{id}_A}$ satisfies it. But, are θ_A and $\widetilde{\mathsf{id}_A}$ equal? *Are they the same as the identity hyperfunction as defined in Section 3.3?*

Computing a little more with the use of (1),

$$\widetilde{f} \cdot \widetilde{g} = f(\widetilde{g} \cdot \widetilde{f}) = f(g(\widetilde{f} \cdot \widetilde{g}) = (f \circ g)(\widetilde{f} \cdot \widetilde{g}),$$

suggests that

$$\widetilde{f} \cdot \widetilde{g} = \mathsf{fix}(f \circ g),$$

but is it true? A special case is particularly interesting:

$$\mathsf{fix}(f) = \theta \cdot \widetilde{f}. \tag{3}$$

Does the least fixpoint operator coincide with the application of the unit hyperfunction?

Finally, let us look for a binary operation $\#\colon [B, C] \times [A, B] \longrightarrow [A, C]$ which behaves on the lifts of ordinary functions as the usual composition. In other words, we expect the equality $\widetilde{f} \,\#\, \widetilde{g} = \widetilde{f \circ g}$ to hold. The chain of equalities, some of them resting on the hypothetical equations above,

$$\begin{aligned}(\widetilde{f} \,\#\, \widetilde{g}) \cdot \widetilde{h} &= \widetilde{f \circ g} \cdot \widetilde{h} \\ &= \mathsf{fix}((f \circ g) \circ h) \\ &= \mathsf{fix}(f \circ (g \circ h)) \\ &= \widetilde{f} \cdot \widetilde{g \circ h} \\ &= \widetilde{f} \cdot (\widetilde{g} \,\#\, \widetilde{h})\end{aligned}$$

suggests the following recursive definition of $\#$:

$$(u \,\#\, v) \cdot w = u \cdot (v \,\#\, w). \tag{4}$$

Is this $\#$ associative? Do hyperfunctions θ behave like units for $\#$? Is $\#$ the same as the (reversed) hyperfunction composition, as defined in Section 3.3?

Theorems 2 and 3 below imply the answer yes to all our questions.

A hyperfunction package in *Haskell*

The above definitions can be coded in *Haskell*, with necessary modifications, as follows.

```
newtype H a b = Phi (H b a -> b)

(@) :: H a b -> H b a -> b
(Phi f) @ k = f k

konst :: a -> H b a
konst p = Phi (\k -> p)

(<<) :: (a -> b) -> H a b -> H a b
f << q = Phi (\k -> f (k @ q))

lift :: (a->b) -> H a b
lift f = f << lift f

proj :: H a b -> a -> b
proj f a = f @ (konst a)

self :: H a a
self = lift id

(#) :: H b c -> H a b -> H a c
f # g = Phi (\k -> f @ (g # k))

run :: H a a -> a
run f = f @ self
```

An explicit isomorphism, denoted `Phi`, is needed to identify the hyperfunction type `H a b` with the function type `H b a -> b`. Note that `@` is the application operator and `konst p` is a constant hyperfunction. The `(<<)` operator acts rather like a *cons* operator, taking a function element `f` and adding to the stack of functions `q` (if we can think of hyperfunctions as being stacks of functions, which is only partially true). `self` is the hyperfunction unit, and `run` is the application of it, so that `run (lift f)` is just `fix f`.

Hyperdomains

If the base category is cartesian closed with hyperfunction coalgebras, then there are natural maps

$$\phi_{AB} \colon ([B, A] \Rightarrow B) \longrightarrow [A, B]$$

defined as anamorphisms corresponding to the $T(-)AB$-coalgebras

$$(\xi_{BA}^{-1} \Rightarrow B) \colon ([B, A] \Rightarrow B) \;\longrightarrow\; ((([B, A] \Rightarrow B) \Rightarrow A) \Rightarrow B).$$

We will call our category a *category of hyperdomains* if ϕ_{AB} is an isomorphism for every A and B, and we will use the notation

$$p_{AB} \colon [B, A] \times [A, B] \longrightarrow B$$

for the transpose of the inverse of ϕ_{AB}. This is just the hyperfunction application operator, with a reversed order of arguments.

The application operator is all we need to give categorical formulation of the equations (1), (2) and (4) used in the domain setting to define the lifting function and the hyperfunction unit and composition. It turns out that no further assumptions are needed for proving that recursive definitions given by these equations are equivalent to the corresponding coalgebraic definitions.

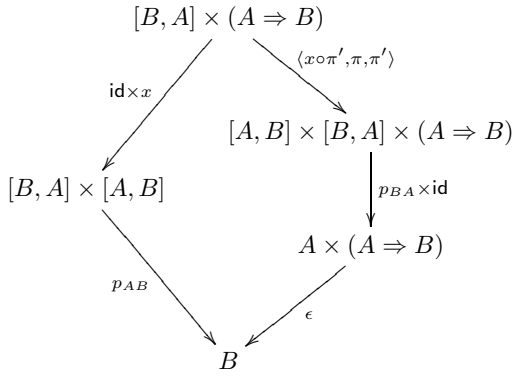


Diagram 1: Translation of
the equation $\widetilde{f} \cdot u = f(u \cdot \widetilde{f})$

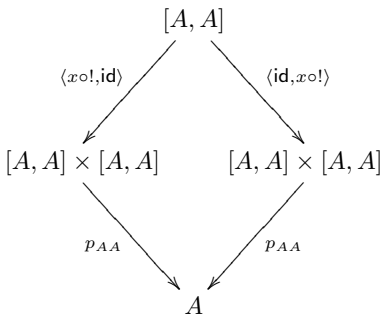


Diagram 2: Translation of
$\theta \cdot u = u \cdot \theta$.

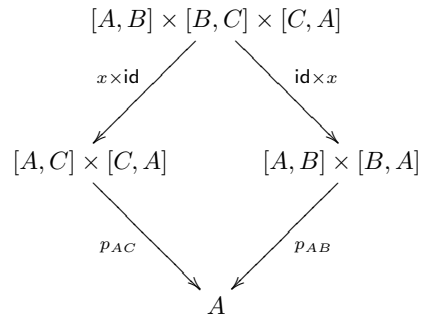


Diagram 3: Translation of
$(u \# v) \cdot w = u \cdot (v \# w)$.

The translations are given in Diagrams 1–3. Diagram 1 is an equation for $x \colon (A \Rightarrow B) \longrightarrow [A, B]$ and Diagram 2 is an equation for $x \colon I \longrightarrow [A, A]$. In

Diagram 3, however, the two occurrences of x refer to morphisms with different source and target. The unknown x here is a collection of functions $x_{ABC} : [A, B] \times [B, C] \longrightarrow [A, C]$, where A, B, C range over all objects. For each triple A, B, C, Diagram 3 expresses a relation between x_{ABC} and x_{BCA}. Due to the cyclic nature of the equation, this infinite system of equations partitions itself into systems of three equations in three unknowns x_{ABC}, x_{BCA} and x_{CAB}.

Theorem 2. *In any category of hyperdomains:*

(a) Diagram 1 commutes if and only if $x = \mathsf{lift}_{AB}$;
(b) Diagram 2 commutes if and only if $x = u_A$;
(c) Diagram 3 commutes (for every A, B, C) if and only if $x_{ABC} = k_{ABC}$ (for every A, B, C).

Turning to the remaining equation (3), we note first that the existence of fixpoints in a category of hyperdomains is granted by a result of Mulry [Mul99]: a fixpoint map $(A \Rightarrow A) \longrightarrow A$ exists whenever there is a retraction $X \longrightarrow (X \Rightarrow A)$, and we take $X = [A, A]$. This, of course, does not imply the equation (3), which we may rewrite as $\mathsf{fix} = Y$, where

$$Y_A : (A \Rightarrow A) \xrightarrow{\mathsf{lift}} [A, A] \xrightarrow{\langle \mathsf{id}, u_A \circ ! \rangle} [A, A] \times [A, A] \xrightarrow{p} A.$$

Theorem 3. *Over any category of hyperdomains, the operator Y is dinatural.*

For definition of dinaturality and the fact that all dinatual operators of type $(A \Rightarrow A) \longrightarrow A$ are necessarily fixpoint operators we refer to [BFSS90] and [SP00]. The equation (3) follows by combining Theorem 3 with a theorem of Simpson [Sim93] implying that the least fixpoint operator in standard categories of domains is the only dinatural fixpoint operator.

Another consequence of Theorem 3 is that the existence of hyperfunction coalgebras does not imply the isomorphisms $[A, B] \cong [B, A] \Rightarrow B$; a counterexample can be found using the PER model of polymorphism [Hyl88,LM91], as suggested by one of the referees.

5 Future work

We have initiated a study of the general structure of process categories. Under suitable conditions, there is premonoidal structure, others give fixpoints and recursion, and some induce the full framework for axiomatic domain theory. Finding such conditions precisely and determining their computational meaning seems to be a demanding but worthwhile task.

Special process categories deserve further study, starting with hyperfunctions. The recent example of hyperfunctions used as an instrumental datatype in a solution of a difficult programming problem [LKS00] suggests that the programming potential of process categories seems worth exploring as well.

References

[Abr96a] S. Abramsky. Interaction categories. In *Theory and Formal Methods '93*, Workshops in Computer Science, pages 57–70. Springer-Verlag, 1996.

[Abr96b] S. Abramsky. Retracing some paths in process algebra. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17, 1996.

[Acz88] P. Aczel. *Non-Well-Founded Sets*. CSLI Publications, 1988.

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 3*. Clarendon Press, 1994.

[AM74] M. A. Arbib and E. G. Manes. Machines in a category: An expository introduction. *SIAM Review*, 16:163–192, 1974.

[Arb66] M. A. Arbib. A common framework for automata theory - a rapprochement. *Automatica*, 3:161–189, 1966.

[BFSS90] E. Bainbridge, P.J. Freyd, A. Scedrov, and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[BG92] S. Brookes and S. Geva. Computational comonads and intensional semantics. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Categories in Computer Science*, pages 1–44. Cambridge University Press, 1992.

[Gra74] J. W. Gray. *Formal Category Theory: Adjointness for 2-categories*. Springer, 1974.

[Hyl88] M. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40:135–165, 1988.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[Kel82] G. M. Kelly. *Basic Concepts of Enriched Category Theory*. Cambridge University Press, 1982.

[LKS00] J. Launchbury, S. Krstić, and T. E. Sauerwein. Zip fusion with hyperfunctions. Technical report, Oregon Graduate Institute, 2000. Preprint available on `http://www.cse.ogi.edu/~krstic`.

[LM91] G. Longo and E. Moggi. onstructive natural deduction and its 'omega-set' interpretation. *Mathematical Structures in Computer Science*, 1:215–254, 1991.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mul99] P. S. Mulry. Categorical fixed-point semantics. *Theoretical Computer Science*, 118:301–314, 199.

[Rut96] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 1996.

[Sim93] A. Simpson. A characterization of the least-fixed-point operator by dinaturality. *Theoretical Computer Science*, 118:301–314, 1993.

[SP00] A. Simpson and G. Plotkin. Complete axioms for categorical fixed-point operators. In *15th Symposium on Logic in Computer Science (LICS 2000)*. IEEE Computer Society, 2000.

[TR98] D. Turi and J. J. M. M. Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Mathematical Structures in Computer Science*, 8:481–540, 1998.