Computational Comonads and Intensional Semantics

Stephen Brookes Shai Geva

October 1991
CMU-CS-91-190

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported in part by National Science Foundation grant CCR-9006064 and in part by DARPA/NSF grant CCR-8906483.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, DARPA or the U.S. government.

Keywords: theory, applicative (functional) programming, semantics, parallelism, category theory, lazy evaluation

Abstract

We explore some foundational issues in the development of a theory of intensional semantics. A programming language may be given a variety of semantics, differing in the level of abstraction; one generally chooses the semantics at an abstraction level appropriate for reasoning about a particular kind of program property. Extensional semantics are typically appropriate for proving properties such as partial correctness, but an intensional semantics at a lower abstraction level is required in order to reason about computation strategy and thereby support reasoning about intensional aspects of behavior such as order of evaluation and efficiency. It is obviously desirable to be able to establish sensible relationships between two semantics for the same language, and we seek a general category-theoretic framework that permits this.

Beginning with an "extensional" category, whose morphisms we can think of as functions of some kind, we model a notion of computation as a comonad with certain extra structure and we regard the Kleisli category of the comonad as an intensional category. An intensional morphism, or algorithm, can be thought of as a function from computations to values. This view accords with a lazy operational interpretation of programs. Under certain rather general assumptions the underlying category can be recovered from the Kleisli category by taking a quotient, derived from a congruence relation that we call extensional equivalence. We then focus on the case where the underlying category is cartesian closed. Under further assumptions the Kleisli category satisfies a weak form of cartesian closure: application morphisms exist, currying and uncurrying of morphisms make sense, and the diagram for exponentiation commutes up to extensional equivalence. When the underlying category is an ordered category we identify conditions under which the exponentiation diagram commutes up to an inequality. We illustrate these ideas and results by introducing some notions of computation on domains and by discussing the properties of the corresponding categories of algorithms on domains.

1 Introduction

Most existing denotational semantic treatments of programming languages are extensional, in that they abstract away from computational details and ascribe essentially extensional meanings to programs. For instance, in the standard denotational treatment of imperative while-programs the meaning of a program is taken to be a partial function from states to states; and in the standard denotational model of the simply typed λ-calculus, the meaning of a term of function type is taken to be a continuous function. Extensional models are appropriate if one wants to reason only about extensional properties of programs, such as partial correctness of while-programs. However, such models give no insight into questions concerning essentially intensional aspects of program behavior, such as efficiency or complexity. For instance, in a typical extensional model all sorting programs denote the same function, regardless of their computation strategy, and therefore regardless of their worst- or average-case behavior. It is desirable to have a semantic model in which sensible comparisons can be made between programs with the same extensional behavior, on the basis of their computation strategy.

We emphasize that we regard intensionality as a relative term; given a programming language we might wish to provide an extensional semantics and also an intensional semantics that contains more computational information and is thus at a lower level of abstraction. We would like to be able to extract extensional meanings from intensional meanings, and to show that the intensional semantics "fits properly" on top of the extensional semantics. Suppose that we have an extensional semantics provided in a category whose objects represent sets of data values and whose morphisms are functions of some kind; and that we have an intensional semantics in a category with the same objects but with morphisms that we regard as algorithms, which correspond to functions equipped with a computation strategy. It seems reasonable that we should be able to define an equivalence relation on algorithms (in the same hom-set) that identifies all pairs of algorithms with the same "extensional part"; that composition of algorithms should respect this equivalence; and that quotienting the algorithms from A to B by this equivalence relation should yield precisely the extensional morphisms.

In this paper we set out a basis for a category-theoretic approach to intensional semantics, motivated by the following intuition. If the extensional meaning of a program may be modelled as some kind of function from data values to data values, then we can obtain an intensional semantics by introducing a notion of computation and defining an intensional meaning to be a function from computations to values. This accords with an intuitive operational semantics for programs in which the execution of a program proceeds lazily in a coroutine-like manner [10]: the program responds to requests for output (say, from a user) by performing input computation until it has sufficient information to supply an output value. We formalize what we mean by a notion of computation in abstract terms as follows. Suppose that extensional meanings are given in some category $\mathcal{C}$. Then, for each object A, we specify an object TA of computations over A and we specify how to lift a morphism f from A to B into a morphism Tf from TA to TB; we require that T be a functor on $\mathcal{C}$. We specify, for each object A, a morphism $\epsilon_A : TA \to A$ from computations to values and a morphism $\delta_A : TA \to T^2A$ that maps a computation over A to a computation over TA. Intuitively, ϵ maps a computation to the value it computes, and δ shows how a computation may itself be computed. We require that (T, ϵ, δ) be a comonad over $\mathcal{C}$. Then we regard the Kleisli category of this comonad as an intensional category; it has the same objects as $\mathcal{C}$, and an intensional morphism from A to B is just a morphism in $\mathcal{C}$ from TA to B.

We say that a comonad is computational if there is a natural way to convert a data value into a degenerate computation returning that value. This enables us to extract from an algorithm a function from values to values, and we obtain an extensional equivalence relation on algorithms by identifying all pairs of algorithms that determine the same function. We show that if the comonad is computational then the Kleisli category collapses onto $\mathcal{C}$ under extensional equivalence.

We then show that, assuming certain further conditions concerning products, the Kleisli category satisfies an intensional analogue of the cartesian closedness property. This generalizes from the known result that if the underlying category is cartesian closed and the (functor part of the) comonad preserves products then the Kleisli category is also cartesian closed. When the underlying category $\mathcal{C}$ is an ordered category, we identify conditions under which the Kleisli construction preserves certain lax forms of cartesian closedness.

Throughout the paper we motivate our definitions and results by means of notions of computation on domains. We focus primarily on three forms of computation at differing levels of abstraction. At the end of the paper we discuss briefly some further examples that indicate the broader applicability of our ideas.

We assume familiarity with elementary category theory and domain theory. We refer the reader to [11] and [1] for categorical background and to [8] for the relevant domain theory.

2 Computations, Comonads and Algorithms

Let $\mathcal{C}$ be a category that we regard as providing an extensional framework. We wish to encapsulate in abstract terms what a notion of computation over $\mathcal{C}$ is, and to build an "intensional" category whose morphisms can be thought of as extensional morphisms equipped with additional computational information. We model a notion of computation over $\mathcal{C}$ as a comonad over $\mathcal{C}$, the functor part of which maps an object A to an object TA representing computations over A. The two other components of the comonad describe how to extract a value from a computation, and how a computation is built up from its sub-computations. We then take an intensional morphism from A to B to be an extensional morphism from TA to B, essentially a morphism from *input computations* over A to *output values* in B. This leads us to use for our intensional category the Kleisli category $\mathcal{C}_T$ [11], which has the same objects as $\mathcal{C}$ and in which the morphisms from A to B are exactly the $\mathcal{C}$-morphisms from TA to B. Typically $\mathcal{C}$ is a category in which morphisms are functions of some kind, and we will refer to intensional morphisms in $\mathcal{C}_T$ as *algorithms* to emphasize their computational content. In case we need to compare Kleisli categories for different comonads over the same underlying category we will use the term T-algorithm, indicating the comonad explicitly.

2.1 Comonads and the Kleisli category

Definition 2.1 A *comonad* over a category $\mathcal{C}$ is a triple (T, ϵ, δ) where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\epsilon : T \dot{\to} I_{\mathcal{C}}$ is a natural transformation from T to the identity functor, and $\delta : T \dot{\to} T^2$ is a natural transformation from T to T^2, such that the following associativity and identity conditions hold, for every object A:

$$\begin{array}{rcl} T(\delta_A) \circ \delta_A & = & \delta_{TA} \circ \delta_A \\ \epsilon_{TA} \circ \delta_A & = & T(\epsilon_A) \circ \delta_A \ = \ \mathsf{id}_{TA} . \end{array}$$

Figures 1 and 2 express these requirements in diagrammatic form. •

Definition 2.2 Given a comonad (T, ϵ, δ) over $\mathcal{C}$, the *Kleisli category* $\mathcal{C}_T$ is defined by:

$$\begin{tikzcd}
TA \arrow[r, "Tf"] \arrow[d, "\epsilon_A"'] & TB \arrow[d, "\epsilon_B"] & TA \arrow[r, "Tf"] \arrow[d, "\delta_A"'] & TB \arrow[d, "\delta_B"] \\
A \arrow[r, "f"'] & B & T^2A \arrow[r, "T^2f"'] & T^2B
\end{tikzcd}$$

Figure 1: Naturality of ϵ and δ in a comonad: these diagrams commute, for all A, B, $f : A \rightarrow^{\mathcal{C}} B$.

$$\begin{tikzcd}
TA & T^2A \arrow[l, "\epsilon_{TA}"'] \arrow[r, "T\epsilon_A"] & TA & TA \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] & T^2A \arrow[d, "\delta_{TA}"] \\
& TA \arrow[ul, "\mathsf{id}_{TA}"] \arrow[u, "\delta_A"'] \arrow[ur, dashed, "\mathsf{id}_{TA}"'] & & T^2A \arrow[r, "T\delta_A"'] & T^3A
\end{tikzcd}$$

Figure 2: Identity and associativity laws of a comonad: these diagrams commute, for all A.

- The objects of $\mathcal{C}_T$ are the objects of $\mathcal{C}$.
- The morphisms from A to B in $\mathcal{C}_T$ are the morphisms from TA to B in $\mathcal{C}$.
- The identity morphism $\widehat{\mathsf{id}}_A$ on A in $\mathcal{C}_T$ is $\epsilon_A : TA \rightarrow^{\mathcal{C}} A$.
- The composition in $\mathcal{C}_T$ of $a : A \rightarrow^{\mathcal{C}_T} B$ and $a' : B \rightarrow^{\mathcal{C}_T} C$, denoted $a' \mathbin{\bar{\circ}} a$, is the composition in $\mathcal{C}$ of $\delta_A : TA \rightarrow^{\mathcal{C}} T^2A$, $Ta : T^2A \rightarrow^{\mathcal{C}} TB$ and $a' : TB \rightarrow^{\mathcal{C}} C$, *i.e.*,

$$a' \mathbin{\bar{\circ}} a = a' \circ Ta \circ \delta_A.$$

The associativity and identity laws of the comonad ensure that $\mathcal{C}_T$ is a category [11]. $\bullet$

This use of morphisms from TA to B to model algorithms from A to B fits well with an intuitive operational semantics based on the coroutine mechanism [10]. A program responds to requests for output by performing some computation on its input (typically, to evaluate some portion of the input) until it has enough information to determine what output value to produce. Execution is lazy, in that computation is demand-driven. The operational behavior of algorithm composition can be described as follows. Let $a : A \rightarrow^{\mathcal{C}_T} B$ and $a' : B \rightarrow^{\mathcal{C}_T} C$. Then $a' \mathbin{\bar{\circ}} a$ responds to a request for output (in C) by performing an input computation t over A, transforming this into a computation t' over B by applying a to the prefixes of t, and supplying t' as argument to a'. For further details concerning operational semantics we refer to [5].

3 Notions of computation on domains

Our main examples will be based on a category of domains and continuous functions. To avoid repetition and to be precise, let us remark that by a domain we mean a directed-complete, bounded-complete, algebraic partial order with a least element. That is, a domain $(D, \sqsubseteq)$ is a set D equipped with a partial order $\sqsubseteq$ satisfying the following conditions:

- D has a least element, denoted $\perp_D$.
- Every non-empty directed subset $X \subseteq D$ has a least upper bound $\bigsqcup X$.
- Every non-empty bounded (or consistent) subset $X \subseteq D$ has a least upper bound $\bigsqcup X$.
- Every element of D is the least upper bound of its (directed set of) finite approximations.

A set X is directed iff for all $x, y \in X$ there is a $z \in X$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$. A set X is bounded iff there is a $z \in D$ such that $x \sqsubseteq z$ for all $x \in X$. An element $e \in D$ is finite iff, for all directed subsets $X \subseteq D$, if $e \sqsubseteq \bigsqcup X$ then $e \sqsubseteq x$ for some $x \in X$.

We remark that none of our example comonads really requires that the underlying domain be algebraic, and nor does the presence of a least element play any prominent rôle (except, of course, in justifying the existence of least fixed points). We could just as well work in the category of directed-complete, bounded complete partial orders and continuous functions. Nevertheless, the property of algebraicity is very natural in the computational setting and all of our example functors on domains preserve algebraicity. At the end of the paper we will discuss further examples based on different categories and different types of domain.

3.1 Increasing paths

The first notion of computation that we introduce models in abstract terms a sequence of time steps in which some incremental evaluation is being performed. For example, a program with two inputs may need to evaluate one or more of its input arguments and it may attempt to perform evaluations in parallel or in sequence; moreover, it may only require partial information about its arguments, as is typically the case, say, when an argument is a function and the program needs to apply that argument to an already known parameter. One natural way to formalize this form of computation is as an increasing sequence of values drawn from a domain, whose partial order reflects the amount of information inherent in a value.

We define the comonad T_1 of "increasing paths" as follows[1].

- For a given domain $(D, \sqsubseteq)$, let $T_1 D$ be the set of finite or infinite increasing sequences over D, ordered componentwise. For convenience we represent a finite sequence as an eventually constant infinite sequence. Thus, the elements of $T_1 D$ have form $\langle d_n \rangle_{n=0}^{\infty}$, where for each $n \geq 0$, $d_n \sqsubseteq d_{n+1}$; and we define $\langle d_n \rangle_{n=0}^{\infty} \sqsubseteq_{T_1 D} \langle d'_n \rangle_{n=0}^{\infty}$ iff for all $n \geq 0$, $d_n \sqsubseteq_D d'_n$.
- For a continuous function $f : D \to D'$, let $T_1 f : T_1 D \to T_1 D'$ be the function that applies f componentwise. That is, $(T_1 f)\langle d_n \rangle_{n=0}^{\infty} = \langle f d_n \rangle_{n=0}^{\infty}$.
- For $t \in T_1 D$ let $\epsilon_D t$ be the least upper bound of t. That is, $\epsilon \langle d_n \rangle_{n=0}^{\infty} = \bigsqcup_{n=0}^{\infty} d_n$.

[1] This comonad, adapted for Scott domains, was first introduced in [4].

$$\begin{tikzcd}
 & \langle\top,\top\rangle \arrow[dl] \arrow[dr, dash] & \\
\langle\top,\bot\rangle \arrow[dr, dash] & & \langle\bot,\top\rangle \arrow[dl] \\
 & \langle\bot,\bot\rangle &
\end{tikzcd}$$

Figure 3: The domain $\texttt{Two} \times \texttt{Two}$.

- For $t \in T_1 D$ let $\delta_D t$ be the sequence of (finite) prefixes of t. That is, if $t = \langle d_n \rangle_{n=0}^{\infty}$, then for each $n \geq 0$, $(\delta t)_n = d_0 \ldots d_n d_n^{\omega}$.

Intuitively, a computation may be viewed as a (time-indexed) sequence of increments in the amount of information known about a data value, and the value "computed" by such a computation is its least upper bound; each computation is itself built up progressively from its prefixes. Equivalently, we can regard a computation over D as a continuous function from the domain $\texttt{VNat}$ of "vertical" natural numbers (together with limit point ω) to D. Our ordering on computations then corresponds exactly to the pointwise ordering on such functions.

The least element of $T_1 D$ is the sequence $\bot^{\omega}$. The finite elements of $T_1 D$ are just the eventually constant sequences all of whose elements are finite in D. It is easy to verify that T_1 maps domains to domains and is indeed a functor. The comonad laws hold: naturality of ϵ corresponds to continuity of f; naturality of δ states that the operation of applying a function componentwise to a sequence "commutes" with taking prefixes; every computation is the least upper bound of its prefixes; and every prefix of a prefix of t is also a prefix of t.

For illustration, let $\texttt{Two}$ be the domain $\{\bot, \top\}$. The domain $\texttt{Two} \times \texttt{Two}$ is shown in Figure 3, and Figure 4 shows the six continuous functions from $\texttt{Two} \times \texttt{Two}$ to $\texttt{Two}$, ordered pointwise. We give these functions mnemonic names: $\bot$ and $\top$ are constant functions; l is strict in its left argument; r is strict in its right argument; b is strict in both arguments; poll returns $\top$ if either of its two arguments is $\top$, so that poll is not strict in either argument. Each function is depicted by a Hasse diagram corresponding to Figure 3, in which the nodes are shaded to indicate their image under the function being described: $\circ$ corresponds to $\bot$, $\bullet$ to $\top$.

Figure 5 shows part of $T_1(\texttt{Two} \times \texttt{Two})$. Figure 6 shows some of the T_1-algorithms from $\texttt{Two} \times \texttt{Two}$ to $\texttt{Two}$, ordered pointwise. The notation for describing algorithms is based on Figure 5, again with $\circ$ representing $\bot$ and $\bullet$ representing $\top$. In each case the intended algorithm is the least continuous function on paths consistent with this description. The nomenclature is intended to indicate (as yet only informally) the function computed by each algorithm and what computation strategy the algorithm uses. For instance, the algorithms pb, lb, rb and db all compute the function b; pb computes both arguments in parallel immediately, lb computes left-first and then right, rb computes right-first and then left, and db computes both arguments in either order. Since the diagram includes only one algorithm for poll, for $\bot$, and for $\top$, in these cases we use the same name for the algorithm as for the function.

Since $T_1(\texttt{Two} \times \texttt{Two})$ includes paths with repeated steps, we can also make distinctions between algorithms which differ not in the order in which they evaluate their arguments, but in the amount of time they are prepared to wait for each successive increment to be achieved. For instance, for the function b there are algorithms pb_n, lb_n, rb_n and db_n for each $n \geq 0$, characterized as the least

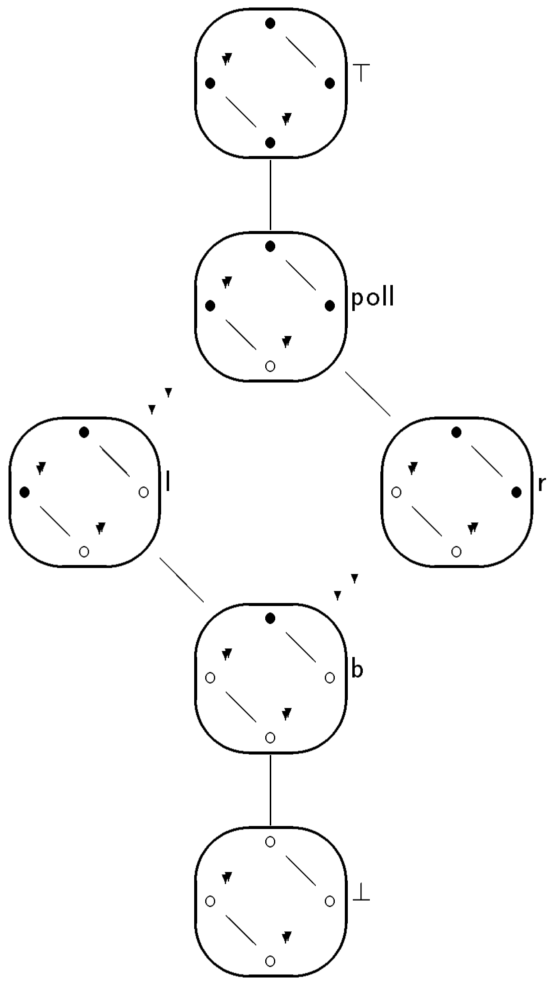

Figure 4: Continuous functions from `Two` $\times$ `Two` to `Two`, ordered pointwise.

$$\begin{tikzcd}
 & \langle\top,\top\rangle^\omega \arrow[dl, dash] \arrow[dr, dash] & \\
\langle\top,\bot\rangle\langle\top,\top\rangle^\omega & & \langle\bot,\top\rangle\langle\top,\top\rangle^\omega \\
\langle\top,\bot\rangle^\omega \arrow[u, dash] & & \langle\bot,\top\rangle^\omega \arrow[u, dash] \\
 & \langle\bot,\bot\rangle^\omega \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

Figure 5: Some paths in T_1(`Two` $\times$ `Two`).

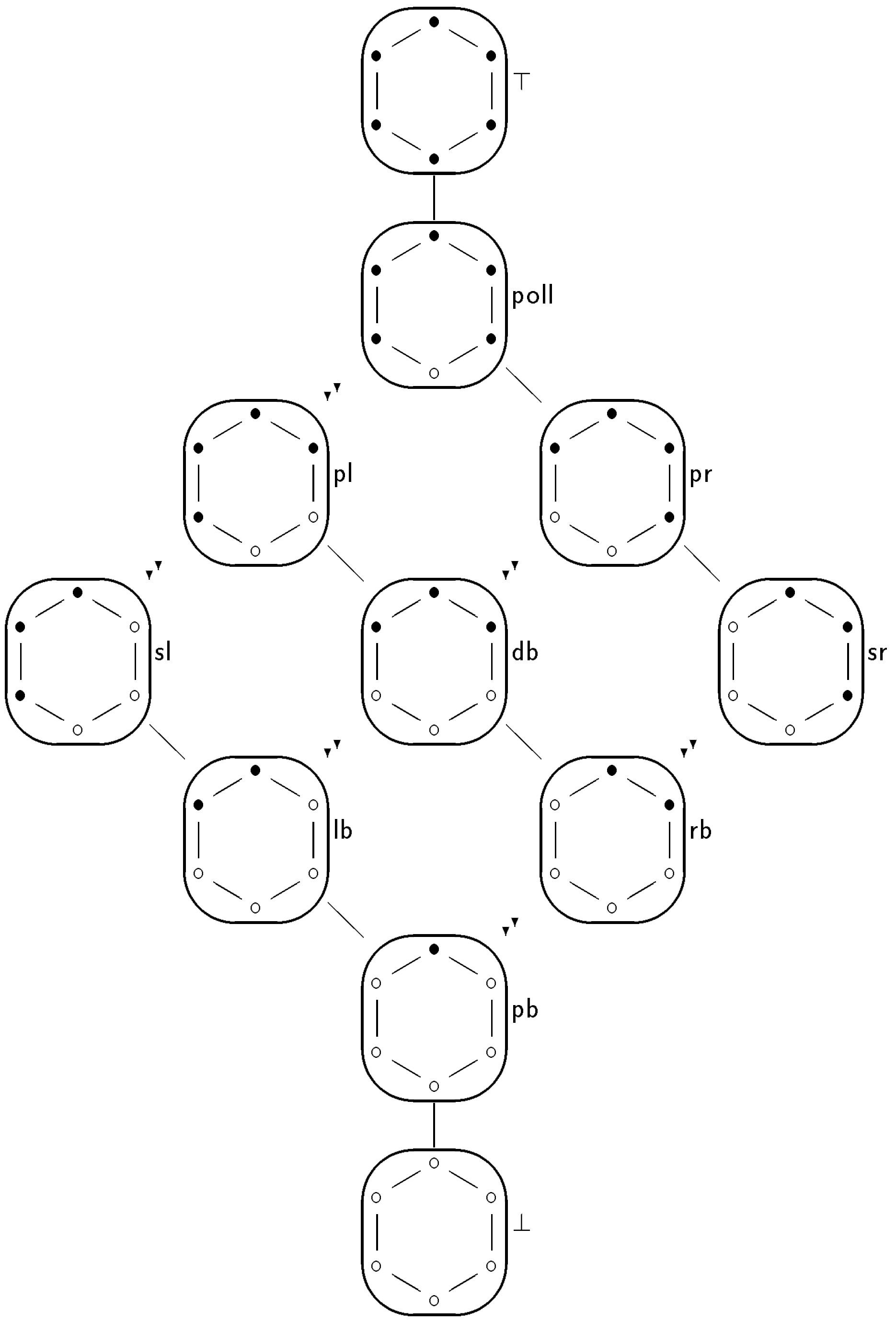

Figure 6: Some T_1-algorithms from `Two` $\times$ `Two` to `Two`, ordered pointwise.

functions on paths such that

$$\begin{array}{l}
\mathsf{pb}_n(\langle\bot,\bot\rangle^n\,\langle\top,\top\rangle^\omega) = \top \\
\mathsf{lb}_n(\langle\bot,\bot\rangle^n\,\langle\top,\bot\rangle\,\langle\top,\bot\rangle^n\,\langle\top,\top\rangle^\omega) = \top \\
\mathsf{rb}_n(\langle\bot,\bot\rangle^n\,\langle\bot,\top\rangle\,\langle\bot,\top\rangle^n\,\langle\top.\top\rangle^\omega) = \top \\
\mathsf{db}_n(\langle\bot,\bot\rangle^n\,\langle\top,\bot\rangle\,\langle\top,\bot\rangle^n\,\langle\top,\top\rangle^\omega) = \top \\
\mathsf{db}_n(\langle\bot,\bot\rangle^n\,\langle\bot,\top\rangle\,\langle\bot,\top\rangle^n\,\langle\top,\top\rangle^\omega) = \top.
\end{array}$$

Informally, pb_n is the algorithm which needs to evaluate both arguments and returns $\top$ provided each evaluation succeeds (with result $\top$) in at most n time steps. Similarly, lb_n evaluates both arguments and returns $\top$ provided evaluation of the left argument succeeds in at most n time steps and evaluation of the right argument succeeds in at most $2n+1$ time steps.

The following relationships hold, for all $n \geq 0$:

$$\begin{array}{l}
\mathsf{pb}_n \sqsubseteq \mathsf{lb}_n \sqsubseteq \mathsf{pb}_{2n+1} \\
\mathsf{pb}_n \sqsubseteq \mathsf{rb}_n \sqsubseteq \mathsf{pb}_{2n+1} \\
\mathsf{db}_n = \mathsf{lb}_n \sqcup \mathsf{rb}_n.
\end{array}$$

Moreover, $\mathsf{pb}_n \sqsubseteq \mathsf{pb}_{n+1}$, $\mathsf{lb}_n \sqsubseteq \mathsf{lb}_{n+1}$, $\mathsf{r}_n \sqsubseteq \mathsf{r}_{n+1}$, and $\mathsf{db}_n \sqsubseteq \mathsf{db}_{n+1}$ for all $n \geq 0$. Each of these sequences of algorithms has the same least upper bound, characterized as the algorithm b_* that maps every path with lub $\langle\top,\top\rangle$ to $\top$. Of course, in Figure 6, pb is just pb_0, and so on.

3.2 Strictly increasing paths

In the comonad T_1 a computation has a built-in measure of the number of time steps it takes between successive proper increments. We obtain a more abstract notion of computation by retaining only the increments themselves, so that we may still make distinctions on the basis of the order of evaluation of arguments. To do this we model a computation as a "strictly increasing path". We define the strictly increasing path comonad T_2 as follows.

- Let T_2D be the set of finite or infinite strictly increasing sequences over D. Again, for convenience, we represent a finite sequence as an eventually constant infinite sequence. That is, the elements of T_2D are either of form $\langle d_n\rangle_{n=0}^{\infty}$, with $d_n \sqsubset_D d_{n+1}$ for all $n \geq 0$; or of form $d_0 \ldots d_{N-1}d_N^\omega$, where $N \geq 0$ and $d_n \sqsubset_D d_{n+1}$ for $0 \leq n < N$. Let $\sqsubseteq_{T_2D}$ be the least partial order on T_2D such that

$$d_0 \ldots d_{N-1}d_N^\omega \sqsubseteq_{T_2D} d_0 \ldots d_{N-1}t \quad \text{if } t \in T_2D \ \&\ d_N \sqsubseteq_D t_0.$$

  This ordering is based on the prefix ordering on sequences, but adjusted to take appropriate account of the underlying order on data values. The order $\sqsubseteq_{T_2D}$ is actually the stable ordering [2] on T_2D, when we regard the elements of T_2D as (strictly increasing, possibly eventually constant) stable functions from `VNat` to D. Note that every continuous function from `VNat` to D is also stable.

- For a continuous function $f : D \to D'$ we define T_2f to be the function which applies f componentwise and suppresses any repetitions (except for constant suffixes). That is, T_2f is the least continuous function such that for all $d \in D$, for all $d_0, d_1 \in D$ such that $d_0 \sqsubset d_1$, and for all $t \in T_2D$,

$$\begin{array}{rcll}
T_2f(d^\omega) & = & (fd)^\omega & \\
T_2f(d_0d_1t) & = & (fd_0)(T_2f(d_1t)) & \text{if } fd_0 \neq fd_1 \\
 & = & T_2f(d_1t) & \text{if } fd_0 = fd_1.
\end{array}$$

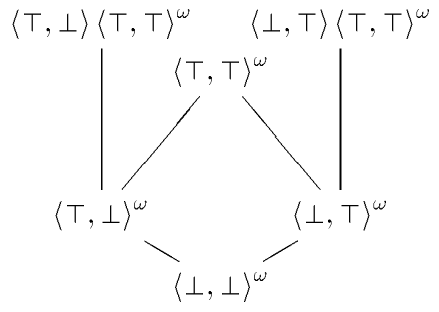

Figure 7: Part of $T_2(\texttt{Two} \times \texttt{Two})$.

- For all $t \in T_2 D$, let $\epsilon_D t$ be the lub of t. That is, $\epsilon \langle d_n \rangle_{n=0}^{\infty} = \bigsqcup_{n=0}^{\infty} d_n$.
- For all $t \in T_2 D$, let $\delta_D t$ be the sequence of prefixes of t. Again, if $t = \langle d_n \rangle_{n=0}^{\infty}$ then for each $n \geq 0$, $(\delta t)_n = d_0 \ldots d_n d_n^{\omega}$. Note that if t is strictly increasing, so is δt.

The least element of $T_2 D$ is the sequence $\bot^{\omega}$. The lub of a directed (or consistent) set of strictly increasing paths is again a strictly increasing path[2]. The finite elements of $T_2 D$ are the eventually constant sequences of form $\langle d_0 \ldots d_{N-1} \rangle d_N^{\omega}$, where $N \geq 0$ and d_N is a finite element of D. Every element of $T_2 D$ is the lub of its finite approximations. Thus, T_2 maps domains to domains. Functoriality is easily checked.

Although the order is not pointwise, it is still true that every $t \in T_2 D$ is the lub of its prefixes. The comonad laws hold for (T_2, ϵ, δ).

Figure 7 shows some of the paths in $T_2(\texttt{Two} \times \texttt{Two})$. Figure 8 shows some of the T_2-algorithms from $\texttt{Two} \times \texttt{Two}$ to $\texttt{Two}$, ordered pointwise, using a notation based on Figure 7. Again the nomenclature is chosen to indicate the function computed and the computation strategy. Each of the T_1-algorithms of Figure 6 has a corresponding T_2-algorithm, for which we use the same name; but because of the different ordering on paths, there are three additional T_2-algorithms. Note also that since $T_2 D$ does not include paths with repeated elements, only pb_0 and pb_1 of the family of pb_n algorithms defined above have corresponding T_2-algorithms.

3.3 Timed data

A simple notion of computation over domains is obtained by regarding a computation as a pair consisting of a data value and a natural number, intuitively representing the amount of time or the cost associated with the calculation of the value. With this intuitition it seems reasonable to regard one computation $\langle d, n \rangle$ as approximating another $\langle d', n' \rangle$ iff $d \sqsubseteq d'$ and $n' \leq n$; that is, a better computation produces a more precise data value in less time. This suggests the use of the following comonad:

- $T_3 D = D \times \texttt{VNat}^{\mathrm{op}}$, ordered componentwise.
- For $f : D \to D'$, $(T_3 f) \langle d, n \rangle = \langle f d, n \rangle$.
- $\epsilon \langle d, n \rangle = d$.

[2] However, this would not be the case if we ordered $T_2 D$ componentwise, since $T_2 D$ is not directed-complete under the componentwise ordering.

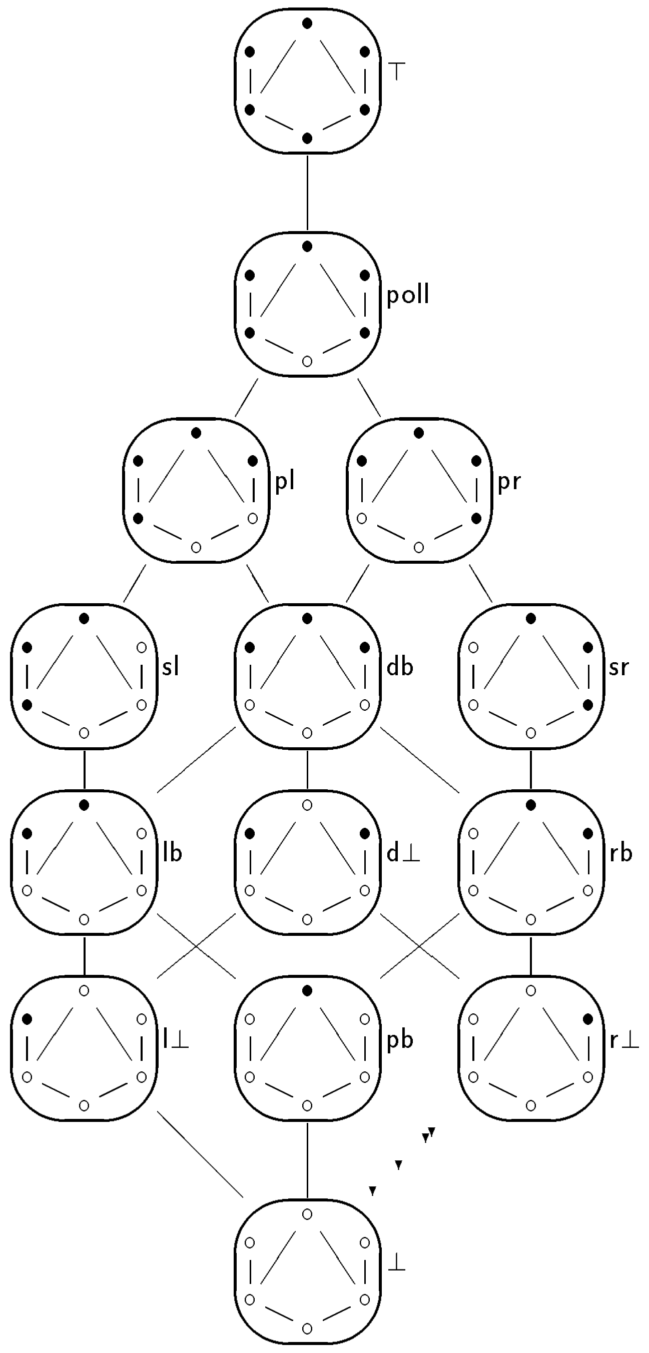

Figure 8: Some T_2-algorithms from `Two` × `Two` to `Two`, ordered pointwise.

- $\delta \langle d, n \rangle = \langle \langle d, n \rangle, n \rangle$.

Here $\mathtt{VNat}^{\mathrm{op}}$ is the domain consisting of the natural numbers together with ω, ordered by the reverse of the usual ordering, so that ω is the least element. The least element of $T_3 D$ is $\langle \bot_D, \omega \rangle$.

We may define for each continuous function $f : D \to D'$ and each $n \in \mathtt{VNat}$ an algorithm f_n from D to D':

$$\begin{aligned} f_n \langle d, k \rangle &= fd && \text{if } k \leq n \\ &= \bot && \text{otherwise.} \end{aligned}$$

Clearly, whenever $f \sqsubseteq g$ we also get $f_n \sqsubseteq g_n$. Moreover, because of our ordering on $T_3 D$, we get $f_n \sqsubseteq f_{n+1}$ for each $n \geq 0$; and f_ω is simply $\lambda \langle d, n \rangle . fd$. The lub of the f_n is the function $f_* = \lambda \langle d, n \rangle . (n = \omega \to \bot, fd)$, which is below f_ω. Using this nomenclature, we show some of the T_3-algorithms from $\mathtt{Two} \times \mathtt{Two}$ to $\mathtt{Two}$ in Figure 9.

It is also possible to define a comonad based on the functor $TD = D \times \mathtt{VNat}$, using the usual ordering on the integer component.

4 Relating algorithms and functions

4.1 Computational comonads

We will say that a comonad is *computational* if for each object A, every data value in A can be regarded as a "degenerate" computation in TA, and degenerate computations possess certain simple properties. This permits us to extract from an algorithm a function from values to values, by looking at the algorithm's effect when applied to degenerate computations. Two algorithms are called extensionally equivalent iff they determine the same function.

More precisely, we require the existence of a natural transformation $\gamma : I_{\mathcal{C}} \dot{\to} T$ satisfying some axioms which capture formally what we mean by degeneracy. We then show that this permits us to define an "extensional equivalence" relation on each hom-set in $\mathcal{C}_T$. Extensional equivalence is preserved by composition, so that we actually have a congruence on $\mathcal{C}_T$. The underlying category $\mathcal{C}$ may then be recovered from $\mathcal{C}_T$ by taking a quotient.

Definition 4.1 A *computational comonad* over a category $\mathcal{C}$ is a quadruple $(T, \epsilon, \delta, \gamma)$ where (T, ϵ, δ) is a comonad over $\mathcal{C}$ and $\gamma : I_{\mathcal{C}} \dot{\to} T$ is a natural transformation such that, for every object A,

- $\epsilon_A \circ \gamma_A = \mathsf{id}_A$
- $\delta_A \circ \gamma_A = \gamma_{TA} \circ \gamma_A$.

Naturality guarantees that, for every morphism $f : A \to^{\mathcal{C}} B$,

- $Tf \circ \gamma_A = \gamma_B \circ f$.

Figure 10 shows these properties in diagrammatic form. •

As an immediate corollary of these properties, ϵ_A is epi and γ_A is mono, for every object A.

Proposition 4.2 *If $(T, \epsilon, \delta, \gamma)$ is a computational comonad, then there is a pair of functors* $(\mathsf{alg}, \mathsf{fun})$ *between $\mathcal{C}$ and $\mathcal{C}_T$ with the following definitions and properties:*

- $\mathsf{alg} : \mathcal{C} \to \mathcal{C}_T$ *is the identity on objects, and* $\mathsf{alg}(f) = f \circ \epsilon_A$, *for every $f : A \to^{\mathcal{C}} B$.*

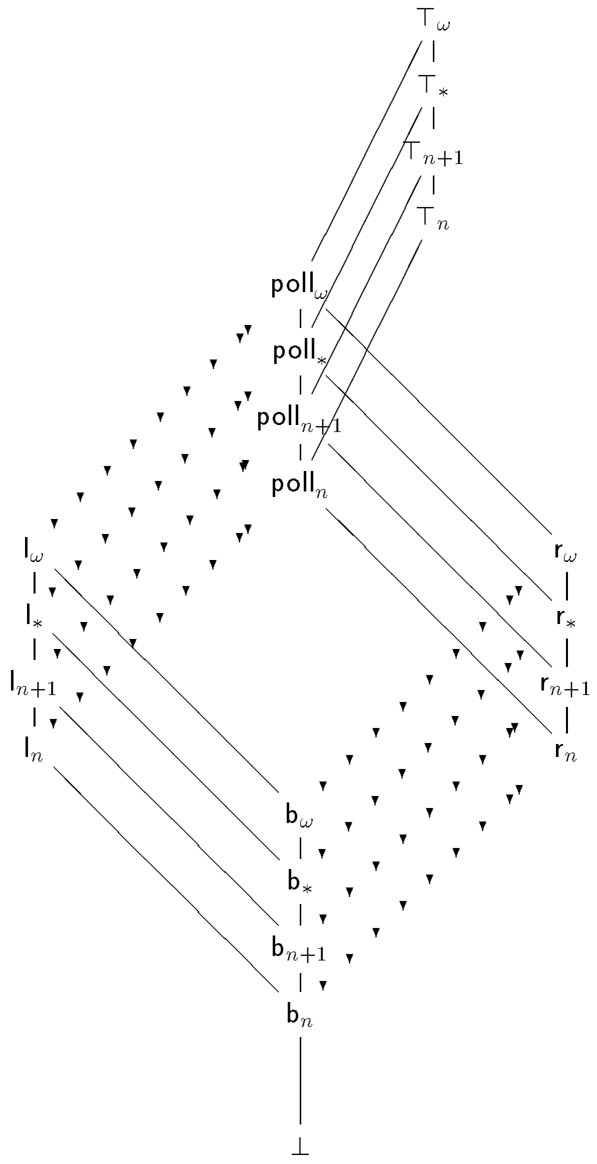

Figure 9: Some T_3-algorithms from `Two` $\times$ `Two` to `Two`, ordered pointwise.

$$\begin{tikzcd}
A \arrow[rr, "\mathrm{id}_A"] \arrow[dr, "\gamma_A"'] & & A \\
& TA \arrow[ur, "\epsilon_A"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\gamma_A"] \arrow[d, "\gamma_A"'] & TA \arrow[d, "\delta_A"] \\
TA \arrow[r, "\gamma_{TA}"'] & T^2A
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "\gamma_A"'] & B \arrow[d, "\gamma_B"] \\
TA \arrow[r, "Tf"'] & TB
\end{tikzcd}$$

Figure 10: Properties of a computational comonad: these diagrams commute, for all A, B, and all morphisms $f : A \to^{\mathcal{C}} B$.

- $\mathsf{fun} : \mathcal{C}_T \to \mathcal{C}$ *is the identity on objects, and* $\mathsf{fun}(a) = a \circ \gamma_A$, *for all* $a : A \to^{\mathcal{C}_T} B$.
- fun *induces an equivalence relation* $=^e$ *on* $\mathcal{C}_T$, *given by* $a_1 =^e a_2 \iff \mathsf{fun}(a_1) = \mathsf{fun}(a_2)$. *This relation is a congruence; that is, for all* $a_1, a_2 : A \to^{\mathcal{C}_T} B$ *and* $a_1', a_2' : B \to^{\mathcal{C}_T} C$,

$$a_1 =^e a_2 \ \&\ a_1' =^e a_2' \quad \Rightarrow \quad a_1' \mathbin{\bar{\circ}} a_1 =^e a_2' \mathbin{\bar{\circ}} a_2.$$

- *The quotient category of* $\mathcal{C}_T$ *by* $=^e$ *is isomorphic to* $\mathcal{C}$.
- $\mathsf{fun} \circ \mathsf{alg} = I_{\mathcal{C}}$. *That is, for all* $f : A \to^{\mathcal{C}} B$, $\mathsf{fun}(\mathsf{alg}\ f) = f$.
- $\mathsf{alg} \circ \mathsf{fun} =^e I_{\mathcal{C}_T}$, *in that for all* $a : A \to^{\mathcal{C}_T} B$, $\mathsf{alg}(\mathsf{fun}\ a) =^e a$.

Proof: Functoriality of alg and fun are straightforward. For instance:

$$\begin{array}{rcll}
\mathsf{fun}(a' \mathbin{\bar{\circ}} a) & = & (a' \mathbin{\bar{\circ}} a) \circ \gamma & \\
& = & (a' \circ Ta \circ \delta) \circ \gamma & \\
& = & a' \circ Ta \circ \gamma \circ \gamma & \text{since } \delta \circ \gamma = \gamma \circ \gamma \\
& = & a' \circ \gamma \circ a \circ \gamma & \text{by naturality of } \gamma \\
& = & \mathsf{fun}(a') \circ \mathsf{fun}(a). &
\end{array}$$

A similar calculation shows that $=^e$ is a congruence.

The quotient of $\mathcal{C}_T$ by $=^e$ has the same objects as $\mathcal{C}_T$ (and therefore the same objects as $\mathcal{C}$), and the morphisms in the quotient category from A to B are the $=^e$-equivalence classes of morphisms from A to B in $\mathcal{C}_T$. Let us write $[a]$ for the equivalence class of a. Clearly, the map $f \mapsto [\mathsf{alg}(f)]$ is an isomorphism of hom-sets, showing that $\mathcal{C}_T/_{=^e}$ is isomorphic to $\mathcal{C}$.

The facts that $\mathsf{fun} \circ \mathsf{alg} = I_{\mathcal{C}}$ and $\mathsf{alg} \circ \mathsf{fun} =^e I_{\mathcal{C}_T}$ are elementary consequences of the definitions.

∎

We say that $\mathsf{fun}(a)$ is the extensional morphism computed by a. Since $\mathsf{fun}(\mathsf{alg} f) = f$, every extensional morphism f is computed by some (not necessarily unique) intensional morphism.

These results show that every computational comonad can be used to produce an intensional category that yields back the underlying extensional category when we take the extensional quotient. Next we show that fun and alg are natural transformations. Let *Set* be the category of sets and functions.

$$\begin{tikzcd}
\mathcal{C}(TA,B) \arrow[r, "\mathcal{C}(Tf,g)"] \arrow[d, "\mathsf{fun}"'] & \mathcal{C}(TA',B') \arrow[d, "\mathsf{fun}"] \\
\mathcal{C}(A,B) \arrow[r, "\mathcal{C}(f,g)"'] & \mathcal{C}(A',B')
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{C}(TA,B) \arrow[r, "\mathcal{C}(Tf,g)"] & \mathcal{C}(TA',B') \\
\mathcal{C}(A,B) \arrow[r, "\mathcal{C}(f,g)"'] \arrow[u, "\mathsf{alg}"] & \mathcal{C}(A',B') \arrow[u, "\mathsf{alg}"']
\end{tikzcd}$$

Figure 11: Naturality of fun and alg for a computational comonad: these diagrams commute, for all $f : A' \rightarrow^{\mathcal{C}} A$ and $g : B \rightarrow^{\mathcal{C}} B'$.

Definition 4.3 The two-variable hom-functor $\mathcal{C}(T(\perp),\perp)$ from $\mathcal{C}^{\mathrm{op}} \times \mathcal{C}$ to *Set* takes a pair (A,B) of objects to $\mathcal{C}(TA,B)$ and takes a pair of morphisms (f,g) with $f : A' \rightarrow^{\mathcal{C}} A$ and $g : B \rightarrow^{\mathcal{C}} B'$ to $\mathcal{C}(Tf,g) : \mathcal{C}(TA,B) \rightarrow \mathcal{C}(TA',B')$, where for all $a : TA \rightarrow^{\mathcal{C}} B$,

$$\mathcal{C}(Tf,g)(a) = g \circ a \circ Tf.$$

Similarly, the two-variable hom-functor $\mathcal{C}(\perp,\perp)$ takes (A,B) to $\mathcal{C}(A,B)$ and (f,g) to $\mathcal{C}(f,g)$, where for all $h : A \rightarrow^{\mathcal{C}} B$,

$$\mathcal{C}(f,g)(h) = g \circ h \circ f.$$

•

Proposition 4.4 *Let $(T,\epsilon,\delta,\gamma)$ be a computational comonad over a category $\mathcal{C}$. Then* fun *and* alg*, as defined in Proposition 4.2, are natural transformations:*

$$\begin{aligned}
&\mathsf{fun} : \mathcal{C}(T(\perp),\perp) \dot{\rightarrow} \mathcal{C}(\perp,\perp)\\
&\mathsf{alg} : \mathcal{C}(\perp,\perp) \dot{\rightarrow} \mathcal{C}(T(\perp),\perp).
\end{aligned}$$

That is, for all $f : A' \rightarrow^{\mathcal{C}} A$ and $g : B \rightarrow^{\mathcal{C}} B'$, the following identities hold:

$$\begin{aligned}
&\mathsf{fun} \circ \mathcal{C}(Tf,g) = \mathcal{C}(f,g) \circ \mathsf{fun}\\
&\mathcal{C}(Tf,g) \circ \mathsf{alg} = \mathsf{alg} \circ \mathcal{C}(f,g).
\end{aligned}$$

Figure 11 expresses these properties in diagram form.

Proof: Straightforward, using naturality of γ and ϵ. ∎

We now show that the three example comonads introduced earlier can be extended to become examples of computational comonads.

4.2 Examples.

1. For the increasing paths comonad T_1, let $\gamma_D : D \to T_1 D$ be defined by $\gamma_D d = d^\omega$, for all $d \in D$. Clearly γ_D is continuous, and γ is a natural transformation. Moreover, the computational comonad laws hold, since the lub of d^ω is d, and all prefixes of d^ω are equal to d^ω.

   The functor fun maps each algorithm from $\mathtt{Two} \times \mathtt{Two}$ to $\mathtt{Two}$ into a function from $\mathtt{Two} \times \mathtt{Two}$ to $\mathtt{Two}$. In particular, $\mathsf{fun}(\mathsf{pb}) = \mathsf{fun}(\mathsf{lb}) = \mathsf{fun}(\mathsf{rb}) = \mathsf{fun}(\mathsf{db}) = \mathsf{b}$. Similarly, $\mathsf{fun}(\mathsf{pb}_n) = \mathsf{b}$ for all $n \geq 0$, and $\mathsf{fun}(\mathsf{b}_*) = \mathsf{b}$. In fact, $\mathsf{b}_* = \mathsf{alg}(\mathsf{b})$.

   Figure 12 illustrates the result of taking the extensional quotient of Figure 6. Boxes enclose equivalence classes of algorithms, arcs between boxes represent the quotient ordering, and within each box we retain the pointwise order to ease comparison with Figure 6. As expected, the quotient figure is isomorphic to Figure 4 when we identify each equivalence class with the function computed.

2. For the strictly increasing paths comonad, we may again let $\gamma_D : D \to T_2 D$ be $\gamma_D d = d^\omega$. Again this is a continuous function, and γ is a natural transformation. Again the computational comonad laws hold. Figure 13 shows the quotient of Figure 8 by extensional equivalence. Note that $\mathsf{fun}(\mathsf{d}\bot) = \mathsf{fun}(\mathsf{l}\bot) = \mathsf{fun}(\mathsf{r}\bot) = \mathsf{fun}(\bot) = \bot$.

   Again the quotient diagram is isomorphic to Figure 4.

3. For the timed data comonad, we obtain a suitable γ by deciding what cost to associate with a degenerate computation. For each $k \in \mathtt{VNat}$ we may take $\gamma_k d = \langle d, k \rangle$ and obtain a natural transformation satisfying the requirements of a computational comonad. Define fun_k to be the functor whose action on algorithms is given by $\mathsf{fun}_k(a) = a \circ \gamma_k$, and let $=^e{}_k$ be the equivalence relation induced by γ_k. For example, we have, for each $k \geq 0$:

   $$\begin{aligned} \mathsf{fun}_k(\mathsf{b}_n) &= \mathsf{b} && \text{if } k \leq n \\ &= \bot && \text{if } k > n. \end{aligned}$$

   Clearly, $\mathsf{b}_n =^e{}_k \mathsf{b}_{n+1}$ iff $k \neq n+1$.

   Again the Kleisli category quotients onto the underlying category under the congruence induced by γ_k. Figure 14 shows the quotient of Figure 9 under the equivalence induced by γ_{n+1}.

5 Products and Exponentiation

5.1 Products

Now suppose that the underlying category $\mathcal{C}$ has products, and for each pair of objects A_1 and A_2 there is a distinguished product, which we denote $A_1 \times A_2$, with π_i ($i = 1, 2$) being the projections. It is easy to show that distinguished product objects in $\mathcal{C}$ are also product objects in $\mathcal{C}_T$, with projections in $\mathcal{C}_T$ given by:

$$\begin{aligned} \widehat{\pi}_i &: A_1 \times A_2 \to^{\mathcal{C}_T} A_i \\ \widehat{\pi}_i &= \epsilon_{A_i} \circ T\,\pi_i \\ &= \pi_i \circ \epsilon_{A_1 \times A_2}. \end{aligned}$$

Pairing of morphisms in $\mathcal{C}_T$ is the pairing of morphisms in $\mathcal{C}$, and the combination $\langle T\,\pi_1, T\,\pi_2 \rangle$ plays a special rôle in light of the following properties.

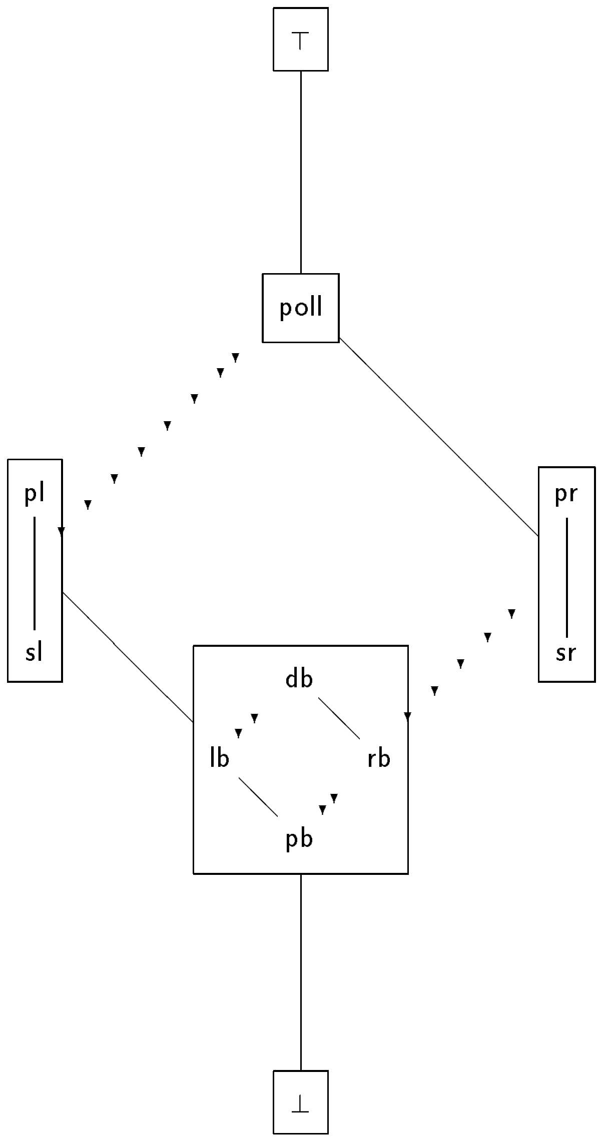

Figure 12: Quotient of Figure 6 by extensional equivalence.

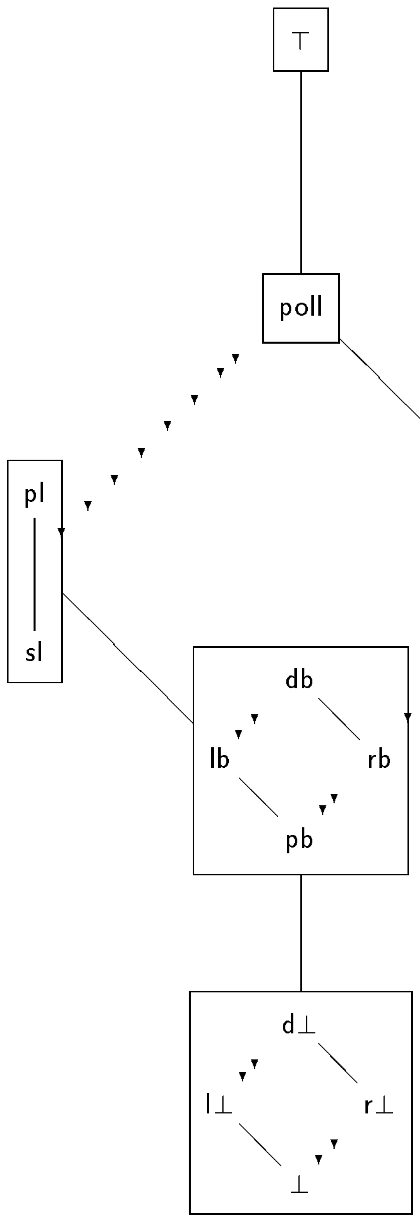

Figure 13: Quotient of Figure 8 by extensional equivalence.

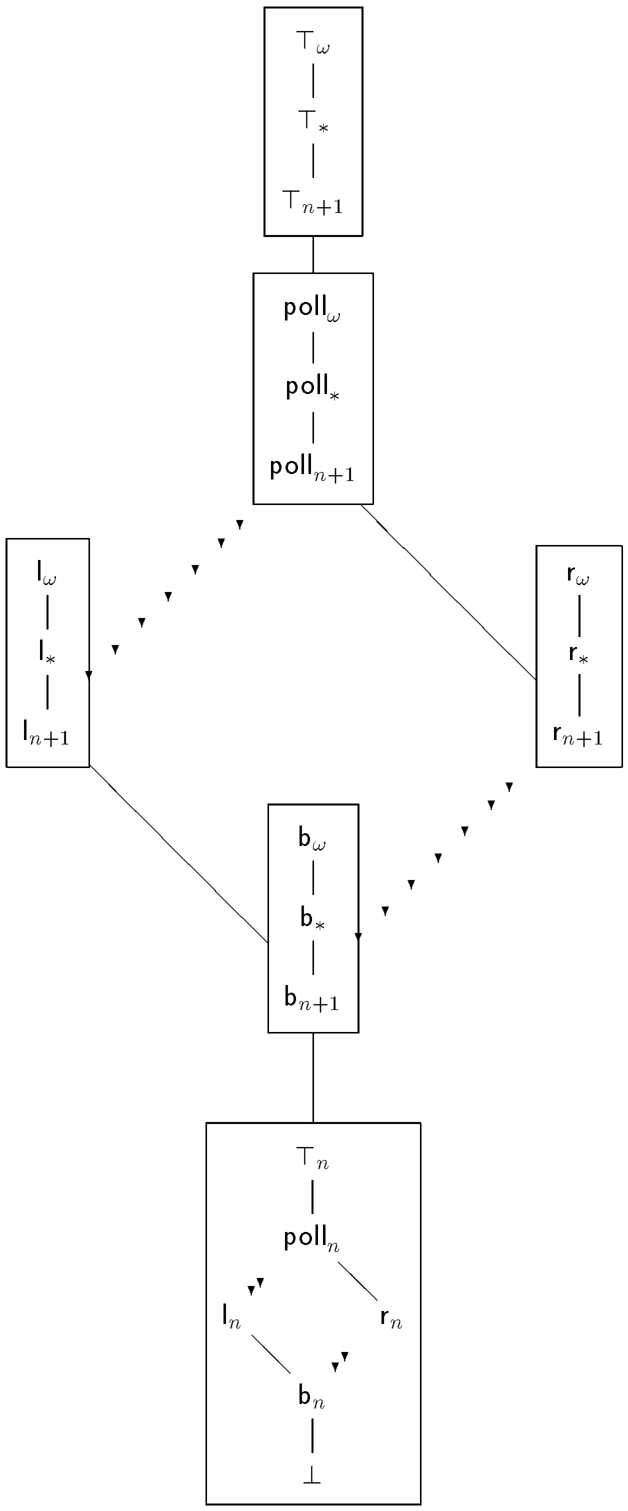

Figure 14: Quotient of Figure 9 by the equivalence induced by γ_{n+1}.

Proposition 5.1 *If $\mathcal{C}$ has products then $\sigma : T(\bot \times \bot) \dot{\to} T(\bot) \times T(\bot)$ defined by:*

$$\sigma_{A,B} : T(A \times B) \to TA \times TB$$
$$\sigma_{A,B} = \langle T\,\pi_1, T\,\pi_2 \rangle$$

is a natural transformation such that, for all objects A and B,

$$\begin{array}{rcl} (\epsilon_A \times \epsilon_B) \circ \sigma_{A,B} & = & \epsilon_{A\times B} \\ (\delta_A \times \delta_B) \circ \sigma_{A,B} & = & \sigma_{TA,TB} \circ T\sigma_{A,B} \circ \delta_{A\times B} \\ \sigma_{A,B} \circ \gamma_{A\times B} & = & \gamma_A \times \gamma_B. \end{array}$$

Proof: Naturality of σ follows from the universal property of products and the functoriality of T. The remaining properties are easy consequences of naturality of ϵ, δ, and γ. Note the identity $\sigma \circ T\langle f, g\rangle = \langle Tf, Tg\rangle$. In particular, $\sigma \circ T\sigma = \langle T^2\,\pi_1, T^2\,\pi_2\rangle$. ∎

5.2 Exponentiation

Now suppose that the underlying category $\mathcal{C}$ is cartesian closed. That is, we assume that $\mathcal{C}$ has a distinguished terminal object and distinguished binary products, and that for every pair of objects B and C there is a distinguished exponentiation object $B \to C$ satisfying the usual requirements: for all B and C there is a morphism $\mathsf{app}_{B,C} : (B \to C) \times B \to^{\mathcal{C}} C$ such that, for all A and all morphisms $f : A \times B \to^{\mathcal{C}} C$ there is a unique morphism $\mathsf{curry}(f) : A \to^{\mathcal{C}} (B \to C)$ such that $\mathsf{app}_{B,C} \circ (\mathsf{curry}(f) \times \mathsf{id}_B) = f$.

Equivalently, a category is cartesian closed if it has finite products and there is a pair of natural isomorphisms

$$\mathsf{curry} : \mathcal{C}(\bot \times B, C) \dot{\to} \mathcal{C}(\bot, B \to C)$$
$$\mathsf{uncurry} : \mathcal{C}(\bot, B \to C) \dot{\to} \mathcal{C}(\bot \times B, C).$$

Here $\mathcal{C}(\bot \times B, C)$ and $\mathcal{C}(\bot, B \to C)$ are contravariant hom-functors from $\mathcal{C}^{\mathrm{op}}$ to the category *Set*, with the standard definitions [1]. This is the same as requiring that $\mathsf{curry}(\mathsf{uncurry}\,h) = h$ and $\mathsf{uncurry}(\mathsf{curry}\,g) = g$, together with the following naturality conditions: for all $f : A \to^{\mathcal{C}} A'$, $g : A' \times B \to^{\mathcal{C}} C$, and $h : A' \to^{\mathcal{C}} (B \to C)$,

$$\mathsf{curry}(g \circ (f \times \mathsf{id})) = (\mathsf{curry}\,g) \circ f$$
$$\mathsf{uncurry}(h) \circ (f \times \mathsf{id}) = \mathsf{uncurry}(h \circ f).$$

It follows easily from these conditions that one can choose $\mathsf{app} = \mathsf{uncurry}(\mathsf{id})$ as a suitable application morphism.

We want to give some general conditions under which analogous properties can be obtained for the Kleisli category $\mathcal{C}_T$. Assuming that $\mathcal{C}$ is cartesian closed, the obvious candidate in $\mathcal{C}_T$ for the exponential object of B and C is $TB \to C$. Moreover, we know that there is a natural isomorphism between $\mathcal{C}_T(A, TB \to C)$ and $\mathcal{C}(TA \times TB, C)$. Since $\mathcal{C}_T(A \times B, C)$ is just $\mathcal{C}(T(A \times B), C)$, it is clear that we must make some assumptions about the relationship between $T(A \times B)$ and $TA \times TB$.

If $T(A \times B)$ and $TA \times TB$ are naturally isomorphic it is easy to show that $\mathcal{C}_T$ is cartesian closed whenever $\mathcal{C}$ is. This is apparently a "Folk Theorem". The comonad T_1 has this property, and we gave in [4] a proof using this property that the Kleisli category of T_1 is cartesian closed.

However, there are reasonable examples in which T does not preserve products, including T_2 and T_3 as described earlier. Instead, we will make a weaker assumption: that the comonad can be equipped with natural ways to move back and forth between $T(A \times B)$ and $TA \times TB$ that interact sensibly with the comonad operations ϵ, δ, and γ. This can be conveniently summarized by means of two natural transformations split and merge satisfying certain combinational laws.

Definition 5.2 Let $(T, \epsilon, \delta, \gamma)$ be a computational comonad. A *computational pairing* is a pair of natural transformations

$$\begin{array}{l} \mathsf{split} : T(\bot \times \bot) \dot{\rightarrow} T(\bot) \times T(\bot) \\ \mathsf{merge} : T(\bot) \times T(\bot) \dot{\rightarrow} T(\bot \times \bot) \end{array}$$

such that, for all objects A and B, the following properties hold:

$$\begin{array}{l} (\epsilon_A \times \epsilon_B) \circ \mathsf{split}_{A,B} = \epsilon_{A\times B} \\ \epsilon_{A\times B} \circ \mathsf{merge}_{A,B} = \epsilon_A \times \epsilon_B \\ \mathsf{split}_{A,B} \circ \gamma_{A\times B} = \gamma_A \times \gamma_B \\ \mathsf{merge}_{A,B} \circ (\gamma_A \times \gamma_B) = \gamma_{A\times B} \\ (\delta_A \times \delta_B) \circ \mathsf{split}_{A,B} = \mathsf{split}_{TA,TB} \circ T\,\mathsf{split}_{A,B} \circ \delta_{A\times B} \\ \mathsf{merge}_{TA,TB} \circ (\delta_A \times \delta_B) = T\,\mathsf{split}_{A,B} \circ \delta_{A\times B} \circ \mathsf{merge}_{A,B}\,. \end{array}$$

Naturality of split and merge requires that for all $f : A \rightarrow^{\mathcal{C}} A'$ and $g : B \rightarrow^{\mathcal{C}} B'$,

$$\begin{array}{l} \mathsf{split}_{A',B'} \circ T(f \times g) = (Tf \times Tg) \circ \mathsf{split}_{A,B} \\ \mathsf{merge}_{A',B'} \circ (Tf \times Tg) = T(f \times g) \circ \mathsf{merge}_{A,B}\,. \end{array}$$

We summarize these properties in diagram form in Figure 15.

•

The properties listed above formalize the sense in which we require the splitting and merging operations to interact sensibly with ϵ, δ, and γ. In particular, the following properties follow immediately.

Corollary 5.3 *Let $(T, \epsilon, \delta, \gamma)$ be a computational comonad and let* split *and* merge *form a computational pairing. Then for all A and B,*

$$\begin{array}{rcl} (\epsilon_A \times \epsilon_B) \circ \mathsf{split}_{A,B} \circ \mathsf{merge}_{A,B} & = & (\epsilon_A \times \epsilon_B) \\ \mathsf{split}_{A,B} \circ \mathsf{merge}_{A,B} \circ (\gamma_A \times \gamma_B) & = & (\gamma_A \times \gamma_B) \\ \epsilon_{A\times B} \circ \mathsf{merge}_{A,B} \circ \mathsf{split}_{A,B} & = & \epsilon_{A\times B} \\ \mathsf{merge}_{A,B} \circ \mathsf{split}_{A,B} \circ \gamma_{A\times B} & = & \gamma_{A\times B}. \end{array}$$

We have already seen that $\sigma = \langle T\,\pi_1, T\,\pi_2\rangle$ qualifies as a suitable split operation (Proposition 5.1). Despite this fact, split (and merge) are not generally uniquely determined by the computational pairing laws and we wish to permit the use of comonads with "non-standard" choices of split. Moreover, naturality of split and merge does not by itself imply the computational pairing laws.

5.3 Examples

1. Return again to the increasing path comonad T_1. The natural transformation $\sigma = \langle T_1\,\pi_1, T_1\,\pi_2\rangle$ is given by: $\sigma(u) = \langle \lambda n.\, \pi_1(u_n), \lambda n.\, \pi_2(u_n)\rangle$. This is actually an isomorphism, with inverse given by $\mathsf{merge}(\langle s, t\rangle) = \lambda n.\, \langle s_n, t_n\rangle$. Intuitively, each of these two operations works "in parallel" on the two components.

   Both σ and merge are natural transformations, and they satisfy the computational pairing laws, which state that:

   - Merging and splitting commute with componentwise application of functions.

$$\begin{tikzcd}
T(A \times B) \arrow[r, "T(f \times g)"] \arrow[d, "\mathsf{split}_{A,B}"'] & T(A' \times B') \arrow[d, "\mathsf{split}_{A',B'}"] \\
TA \times TB \arrow[r, "Tf \times Tg"'] & TA' \times TB'
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \times TB \arrow[r, "T(f \times g)"] \arrow[d, "\mathsf{merge}_{A,B}"'] & TA' \times TB' \arrow[d, "\mathsf{merge}_{A',B'}"] \\
T(A \times B) \arrow[r, "Tf \times Tg"'] & T(A' \times B')
\end{tikzcd}$$

$$\begin{tikzcd}
TA \times TB \arrow[d, "\mathsf{merge}_{A,B}"'] \arrow[dr, "\epsilon_A \times \epsilon_B"] & \\
T(A \times B) \arrow[r, "\epsilon_{A \times B}"] \arrow[d, "\mathsf{split}_{A,B}"'] & A \times B \\
TA \times TB \arrow[ur, dotted, "\epsilon_A \times \epsilon_B"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & TA \times TB \arrow[d, "\mathsf{merge}_{A,B}"] \\
A \times B \arrow[ur, dotted, "\gamma_A \times \gamma_B"] \arrow[r, "\gamma_{A \times B}"] \arrow[dr, "\gamma_A \times \gamma_B"'] & T(A \times B) \arrow[d, "\mathsf{split}_{A,B}"] \\
 & TA \times TB
\end{tikzcd}$$

$$\begin{tikzcd}
TA \times TB \arrow[rr, "\delta_A \times \delta_B"] \arrow[d, "\mathsf{merge}_{A,B}"'] & & T^2A \times T^2B \arrow[d, "\mathsf{merge}_{TA,TB}"] \\
T(A \times B) \arrow[r, "\delta_{A \times B}"] \arrow[d, "\mathsf{split}_{A,B}"'] & T^2(A \times B) \arrow[r, "T\,\mathsf{split}_{A,B}"] & T(TA \times TB) \arrow[d, "\mathsf{split}_{TA,TB}"] \\
TA \times TB \arrow[rr, "\delta_A \times \delta_B"'] & & T^2A \times T^2B
\end{tikzcd}$$

Figure 15: Properties of a computational pairing: these diagrams commute for all A, A', B, B' and all $f : A \to^{\mathcal{C}} A'$, $g : B \to^{\mathcal{C}} B'$.

- Merging and splitting respect lubs of sequences.
- Merging and splitting respect prefixes.
- Merging two degenerate computations produces a degenerate computation, and splitting a degenerate computation produces a pair of degenerate computations.

2. There are other intuitively sensible ways to split and merge in the increasing paths comonad T_1. We can define a form of (left-first) interleaving by:

$$\mathsf{lmerge}(\langle s,t\rangle) = \lambda n.\left\langle s_{\lceil n/2\rceil}, t_{\lfloor n/2\rfloor}\right\rangle .$$

For example, this gives:

$$\mathsf{lmerge}(s_0 s_1 s_2^\omega, t_0 t_1 t_2^\omega) = \langle s_0,t_0\rangle\langle s_1,t_0\rangle\langle s_1,t_1\rangle\langle s_2,t_1\rangle\langle s_2,t_2\rangle^\omega .$$

To go with lmerge, we define a split that operates only on alternate steps of a computation:

$$\mathsf{split}_2(u) = \langle \lambda n.\, \pi_1(u_{2n}), \lambda n.\, \pi_2(u_{2n})\rangle .$$

We then obtain the identity $\mathsf{split}_2 \circ \mathsf{lmerge} = \mathsf{id}$.

It is easy to verify that lmerge and split_2 are natural transformations, and that the computational pairing properties hold, making use of the equalities $\lfloor \min(i,2j)/2\rfloor = \min(\lfloor i/2\rfloor, j)$ and $\lceil \min(i,2j)/2\rceil = \min(\lceil i/2\rceil, j)$.

There is clearly also a right-first version of interleaving rmerge and this also interacts sensibly with split_2 as given above.

3. For the strictly increasing paths comonad, each of the split-merge combinations above adapts in the obvious way, modified so as to ensure that the result of splitting a strictly increasing sequence of pairs is a pair of strictly increasing sequences. Thus, for example,

$$\begin{array}{rcl}\sigma(\langle\bot,\bot\rangle\langle\top,\bot\rangle\langle\top,\top\rangle^\omega) & = & \langle\bot\top^\omega, \bot\top^\omega\rangle \\ \mathsf{merge}(\bot\top^\omega,\bot\top^\omega) & = & \langle\bot,\bot\rangle\langle\top,\top\rangle^\omega .\end{array}$$

In fact $T_2(A\times B)$ and $T_2A\times T_2B$ are not generally isomorphic, because a strictly increasing sequence of pairs does not necessarily increase strictly in *both* components at each stage. Nevertheless, σ and merge are still natural transformations satisfying the requirements listed above for a computational pairing, and we have the identity $\sigma\circ\mathsf{merge} = \mathsf{id}$.

The (appropriately adjusted) lmerge and split_2 operations also form a computational pairing, and $\mathsf{split}_2\circ\mathsf{lmerge} = \mathsf{id}$; similar properties hold for rmerge and split_2.

5.4 Pairing, currying and uncurrying on algorithms

Using the split operation of a computational pairing provides a way to combine a pair of algorithms into an algorithm on pairs. If split is taken to be σ, this is the standard way to form the product of two morphisms in the Kleisli category. We can also use split to define intensional analogues to the contravariant hom-functors $\mathcal{C}(\bot\times B, C)$ and $\mathcal{C}(\bot, B\to C)$.

Definition 5.4 Let $\mathcal{C}$ be a category with finite products, let $(T, \epsilon, \delta, \gamma)$ be a computational comonad, and let split be a natural transformation from $T(\bot \times \bot)$ to $T \bot \times T\bot$. For $f : A \rightarrow^{\mathcal{C}_T} A'$ and $g : B \rightarrow^{\mathcal{C}_T} B'$ we define $(f \hat{\times} g) : (A \times A') \rightarrow^{\mathcal{C}_T} (B \times B')$ by $f \hat{\times} g = (f \times g) \circ \mathsf{split}$. •

Proposition 5.5 *Let $\mathcal{C}$ be a category with finite products, let $(T, \epsilon, \delta, \gamma)$ be a computational comonad and let split and merge form a computational pairing. Then there is a functor $\hat{\times}$ from $\mathcal{C}_T \times \mathcal{C}_T$ to $\mathcal{C}_T$ that maps a pair of objects (A, B) to $A \times B$ and maps a pair of morphisms (a, b) to $(a \hat{\times} b) = (a \times b) \circ \mathsf{split}$.*

Proof: To show that $\hat{\times}$ maps identity morphisms to identity morphisms:

$$(\widehat{\mathsf{id}}_A \hat{\times} \widehat{\mathsf{id}}_B) = (\epsilon_A \times \epsilon_B) \circ \mathsf{split} = \epsilon_{A \times B} = \widehat{\mathsf{id}}_{A \times B}.$$

To show that $\hat{\times}$ preserves composition, let $a : A \rightarrow^{\mathcal{C}_T} A'$, $b : B \rightarrow^{\mathcal{C}_T} B'$, $a' : A' \rightarrow^{\mathcal{C}_T} A''$, and $b' : B' \rightarrow^{\mathcal{C}_T} B''$. Then:

$$\begin{aligned}
(a' \hat{\times} b') \mathbin{\bar{\circ}} (a \hat{\times} b) &= ((a' \times b') \circ \mathsf{split}) \mathbin{\bar{\circ}} ((a \times b) \circ \mathsf{split}) \\
&= (a' \times b') \circ \mathsf{split} \circ T(a \times b) \circ T\,\mathsf{split} \circ \delta \\
&= (a' \times b') \circ (Ta \times Tb) \circ \mathsf{split} \circ T\,\mathsf{split} \circ \delta \\
&= (a' \times b') \circ (Ta \times Tb) \circ (\delta \times \delta) \circ \mathsf{split} \\
&= ((a' \circ Ta \circ \delta) \times (b' \circ Tb \circ \delta)) \circ \mathsf{split} \\
&= ((a' \mathbin{\bar{\circ}} a) \times (b' \mathbin{\bar{\circ}} b)) \circ \mathsf{split} \\
&= (a' \mathbin{\bar{\circ}} a) \hat{\times} (b' \mathbin{\bar{\circ}} b).
\end{aligned}$$

■

Definition 5.6 Let $\mathcal{C}$ be a cartesian closed category, let $(T, \epsilon, \delta, \gamma)$ be a computational comonad, and let split and merge form a computational pairing. The contravariant functor $\mathcal{C}_T(\bot \hat{\times} B, C)$ from ${\mathcal{C}_T}^{\mathrm{op}}$ to *Set* is defined as follows.

- On objects the functor maps A to $\mathcal{C}_T(A \times B, C)$.
- On morphisms the functor maps $f : A \rightarrow^{\mathcal{C}_T} A'$ to the function $\lambda g.g \mathbin{\bar{\circ}} (f \hat{\times} \widehat{\mathsf{id}}_B)$ from $\mathcal{C}_T(A' \times B, C)$ to $\mathcal{C}_T(A \times B, C)$.

Similarly, the contravariant functor $\mathcal{C}_T(\bot, TB \rightarrow C)$ from ${\mathcal{C}_T}^{\mathrm{op}}$ to *Set*, is defined by:

- On objects the functor maps A to $\mathcal{C}_T(A, TB \rightarrow C)$.
- On morphisms the functor maps $f : A \rightarrow^{\mathcal{C}_T} A'$ to the function $\lambda h.h \mathbin{\bar{\circ}} f$ from $\mathcal{C}_T(A', TB \rightarrow C)$ to $\mathcal{C}_T(A, TB \rightarrow C)$.

•

Proposition 5.7 *Let $(T, \epsilon, \delta, \gamma)$ be a computational comonad and let split and merge be a computational pairing. Given $a : T(A \times B) \rightarrow^{\mathcal{C}} C$ and $b : TA \rightarrow^{\mathcal{C}} (TB \rightarrow C)$, define*

$$\begin{array}{ll}
\widehat{\mathsf{curry}}(a) : TA \rightarrow^{\mathcal{C}} (TB \rightarrow C) & \widehat{\mathsf{uncurry}}(b) : T(A \times B) \rightarrow^{\mathcal{C}} C \\
\widehat{\mathsf{curry}}(a) = \mathsf{curry}(a \circ \mathsf{merge}) & \widehat{\mathsf{uncurry}}(b) = \mathsf{uncurry}(b) \circ \mathsf{split}\,.
\end{array}$$

Then:

- $\widehat{\mathsf{curry}}$ *and* $\widehat{\mathsf{uncurry}}$ *are natural transformations:*

$$\widehat{\mathsf{curry}} : \mathcal{C}_T(\bot \hat{\times} B, C) \dot{\to} \mathcal{C}_T(\bot, TB \to C)$$
$$\widehat{\mathsf{uncurry}} : \mathcal{C}_T(\bot, TB \to C) \dot{\to} \mathcal{C}_T(\bot \hat{\times} B, C).$$

- *For all* $a : A \times B \to^{\mathcal{C}_T} C$, $\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}(a)) =^e a$.

- *For all* $f : A \times B \to^{\mathcal{C}} C$, $\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}(\mathsf{alg}\ f)) = \mathsf{alg}\ f$.

Proof:

- The naturality of $\widehat{\mathsf{curry}}$ amounts to the requirement that $\widehat{\mathsf{curry}}(g \mathbin{\bar{\circ}} (f \hat{\times} \widehat{\mathsf{id}}_B)) = (\widehat{\mathsf{curry}}\, g) \mathbin{\bar{\circ}} f$, for all $f : A \to^{\mathcal{C}_T} A'$ and $g : A' \times B \to^{\mathcal{C}_T} C$. This follows from naturality of currying in $\mathcal{C}$ and the properties of computational pairing:

$$\begin{aligned}
\widehat{\mathsf{curry}}(g \mathbin{\bar{\circ}} (f \hat{\times} \widehat{\mathsf{id}})) &= \widehat{\mathsf{curry}}(g \circ T(f \times \epsilon) \circ T\,\mathsf{split} \circ \delta) \\
&= \mathsf{curry}(g \circ T(f \times \epsilon) \circ T\,\mathsf{split} \circ \delta \circ \mathsf{merge}) \\
&= \mathsf{curry}(g \circ T(f \times \epsilon) \circ \mathsf{merge} \circ (\delta \times \delta)) \\
&= \mathsf{curry}(g \circ \mathsf{merge} \circ (Tf \times T\epsilon) \circ (\delta \times \delta)) \\
&= \mathsf{curry}(g \circ \mathsf{merge} \circ ((Tf \circ \delta) \times (T\epsilon \circ \delta))) \\
&= \mathsf{curry}(g \circ \mathsf{merge} \circ ((Tf \circ \delta) \times \mathsf{id})) \\
&= \mathsf{curry}(g \circ \mathsf{merge}) \circ (Tf \circ \delta) \\
&= \widehat{\mathsf{curry}}(g) \circ Tf \circ \delta \\
&= \widehat{\mathsf{curry}}(g) \mathbin{\bar{\circ}} f.
\end{aligned}$$

- Similarly, to show naturality of $\widehat{\mathsf{uncurry}}$ we need $\widehat{\mathsf{uncurry}}(h) \mathbin{\bar{\circ}} (f \hat{\times} \widehat{\mathsf{id}}) = \widehat{\mathsf{uncurry}}(h \mathbin{\bar{\circ}} f)$, for all $f : A \to^{\mathcal{C}_T} A'$ and $h : A' \to^{\mathcal{C}_T} (TB \to C)$. Again the proof is straightforward:

$$\begin{aligned}
\widehat{\mathsf{uncurry}}(h) \mathbin{\bar{\circ}} (f \hat{\times} \widehat{\mathsf{id}}) &= \mathsf{uncurry}(h) \circ \mathsf{split} \circ T(f \times \epsilon) \circ T\,\mathsf{split} \circ \delta \\
&= \mathsf{uncurry}(h) \circ (Tf \times T\epsilon) \circ \mathsf{split} \circ T\,\mathsf{split} \circ \delta \\
&= \mathsf{uncurry}(h) \circ (Tf \times T\epsilon) \circ (\delta \times \delta) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h) \circ ((Tf \circ \delta) \times (T\epsilon \circ \delta)) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h) \circ ((Tf \circ \delta) \times \mathsf{id}) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h \circ Tf \circ \delta) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h \mathbin{\bar{\circ}} f) \circ \mathsf{split} \\
&= \widehat{\mathsf{uncurry}}(h \mathbin{\bar{\circ}} f).
\end{aligned}$$

- Let $a : A \times B \to^{\mathcal{C}_T} C$. Then

$$\begin{aligned}
\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}(a)) \circ \gamma &= \mathsf{uncurry}(\mathsf{curry}(a \circ \mathsf{merge})) \circ \mathsf{split} \circ \gamma \\
&= (a \circ \mathsf{merge}) \circ \mathsf{split} \circ \gamma \\
&= a \circ (\mathsf{merge} \circ \mathsf{split} \circ \gamma) \\
&= a \circ \gamma,
\end{aligned}$$

  showing that $\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}\, a) =^e a$.

- Let $f : A \times B \to^{\mathcal{C}} C$. Then

$$\begin{array}{rcll}
\mathsf{un}\widehat{\mathsf{curry}}(\widehat{\mathsf{curry}}(\mathsf{alg}\ f)) & = & \mathsf{uncurry}(\mathsf{curry}(\mathsf{alg}(f) \circ \mathsf{merge})) \circ \mathsf{split} & \\
& = & \mathsf{alg}(f) \circ \mathsf{merge} \circ \mathsf{split} & \\
& = & f \circ \epsilon \circ \mathsf{merge} \circ \mathsf{split} & \text{by Corollary 5.3} \\
& = & f \circ \epsilon & \\
& = & \mathsf{alg}\ f. &
\end{array}$$

■

We have shown that an intensional pairing produces a weak form of exponentiation structure: we obtain notions of currying and uncurrying on algorithms that are natural transformations but satisfy a weaker condition than isomorphism. We may rephrase these properties in terms of the existence of a notion of "application" in the intensional category as follows.

Proposition 5.8 *Let $\mathcal{C}$ be a cartesian closed category, $(T, \epsilon, \delta, \gamma)$ be a computational comonad and let* split *and* merge *be a computational pairing. For all B and C there is an "application morphism"*

$$\widehat{\mathsf{app}}_{B,C} \ :\ [TB \to C] \times B \to^{\mathcal{C}_T} C$$

such that, for all $a : A \times B \to^{\mathcal{C}_T} C$

$$\widehat{\mathsf{app}}_{B,C} \,\bar{\circ} (\widehat{\mathsf{curry}}(a) \mathbin{\hat{\times}} \widehat{\mathsf{id}}_B) =^e a.$$

Proof: Define $\widehat{\mathsf{app}}_{B,C} = \mathsf{un}\widehat{\mathsf{curry}}(\widehat{\mathsf{id}}_{TB \to C}) = \mathsf{un}\widehat{\mathsf{curry}}(\epsilon_{TB \to C})$. As a corollary of the naturality of $\mathsf{un}\widehat{\mathsf{curry}}$ (Proposition 5.7), we get:

$$\begin{array}{rcl}
\widehat{\mathsf{app}} \,\bar{\circ} (b \mathbin{\hat{\times}} \widehat{\mathsf{id}}) & = & \mathsf{un}\widehat{\mathsf{curry}}(\widehat{\mathsf{id}}) \,\bar{\circ}\, (b \mathbin{\hat{\times}} \widehat{\mathsf{id}}) \\
& = & \mathsf{un}\widehat{\mathsf{curry}}(\widehat{\mathsf{id}} \,\bar{\circ}\, b) \\
& = & \mathsf{un}\widehat{\mathsf{curry}}(b).
\end{array}$$

Thus, in particular, $\widehat{\mathsf{app}} \,\bar{\circ} (\widehat{\mathsf{curry}}(a) \mathbin{\hat{\times}} \widehat{\mathsf{id}}) = \mathsf{un}\widehat{\mathsf{curry}}(\widehat{\mathsf{curry}}(a)) =^e a$.

Note that although $\widehat{\mathsf{curry}}(a)$ is not the unique morphism h such that $\widehat{\mathsf{app}} \,\bar{\circ} (h \mathbin{\hat{\times}} \mathsf{id}) =^e a$, all such morphisms satisfy the condition that $\mathsf{un}\widehat{\mathsf{curry}}(h) =^e a$. ■

Thus, we have a weak form of cartesian closedness: instead of the usual diagram for exponentiation we replace $=$ by $=^e$ and we relax the uniqueness condition. This is summarized in Figure 16.

Next we consider what happens if we make further assumptions on the relationship between split and merge.

Proposition 5.9 *Let $\mathcal{C}$ be a cartesian closed category and $(T, \epsilon, \delta, \gamma)$ be a computational comonad with a computational pairing* split *and* merge*.*

- *If* $\mathsf{merge} \circ \mathsf{split} = \mathsf{id}$ *then* $\mathsf{un}\widehat{\mathsf{curry}} \circ \widehat{\mathsf{curry}} = \mathsf{id}$.
- *If* $\mathsf{split} \circ \mathsf{merge} = \mathsf{id}$ *then* $\widehat{\mathsf{curry}} \circ \mathsf{un}\widehat{\mathsf{curry}} = \mathsf{id}$.

As a corollary we get the following version of the "Folk Theorem":

$$\begin{tikzcd}
A \times B \arrow[r, "a"] \arrow[d, "\widehat{\mathsf{curry}}(a) \mathbin{\hat{\times}} \widehat{\mathsf{id}}"'] \arrow[dr, phantom, "=^e"] & C \\
(TB \to C) \times B \arrow[ur, dotted, "\widehat{\mathsf{app}}"'] &
\end{tikzcd}$$

Figure 16: When T has computational pairing this diagram in $\mathcal{C}_T$ commutes up to extensional equivalence, for all $a : A \times B \to^{\mathcal{C}_T} C$.

Corollary 5.10 *If $\mathcal{C}$ is cartesian closed and $(T, \epsilon, \delta, \gamma)$ is a computational comonad with a computational pairing such that* $\mathsf{merge} \circ \mathsf{split} = \mathsf{id}$ *and* $\mathsf{split} \circ \mathsf{merge} = \mathsf{id}$ *then the category $\mathcal{C}_T$ is cartesian closed.*

Note the important fact that our definitions are parameterized by the choice of split and merge. Once these are chosen, $\widehat{\mathsf{app}}$, $\widehat{\mathsf{curry}}$ and $\mathsf{un}\widehat{\mathsf{curry}}$ are determined uniquely. The Kleisli category itself is independent of split and merge; what happens, however, is that each choice of these two natural transformations induces a (weak form of) exponentiation structure on this category. The Kleisli category may possess many different notions of merging and splitting, and therefore many different ways to curry, uncurry and apply algorithms. This means that one may use the Kleisli category to give an interpretation to a functional programming language containing several syntactically and semantically distinct forms of application. This would be desirable, for instance, if the language included both a strict and a non-strict form of application.

5.5 Examples

1. The Kleisli category based on the increasing path comonad T_1 is cartesian closed, with exponentiation structure built from the standard split-merge combination, which form an isomorphism.

   Using the computational pairing lmerge and split_2, we obtain intensional forms of currying, uncurrying, and application which we will call $\widehat{\mathsf{curry}}_l$, $\mathsf{un}\widehat{\mathsf{curry}}_l$ and $\widehat{\mathsf{app}}_l$. This provides a weak form of exponentiation: $\widehat{\mathsf{curry}}_l$ and $\mathsf{un}\widehat{\mathsf{curry}}_l$ are natural transformations, and for all $a : A \times B \to^{\mathcal{C}_{T_1}} C$ we get

   $$\widehat{\mathsf{app}}_l \,\bar{\circ}(\widehat{\mathsf{curry}}_l(a) \mathbin{\hat{\times}} \widehat{\mathsf{id}}) =^e a.$$

   Since $\mathsf{split}_2 \circ \mathsf{lmerge} = \mathsf{id}$, we have $\widehat{\mathsf{curry}}_l(\mathsf{un}\widehat{\mathsf{curry}}_l\, h) = h$ but $\mathsf{un}\widehat{\mathsf{curry}}_l(\widehat{\mathsf{curry}}_l\, g) =^e g$. For example,

   $$\mathsf{un}\widehat{\mathsf{curry}}_l(\widehat{\mathsf{curry}}_l \mathsf{pb}) = \mathsf{un}\widehat{\mathsf{curry}}_l(\widehat{\mathsf{curry}}_l \mathsf{lb}) = \mathsf{pb},$$

   and $\mathsf{un}\widehat{\mathsf{curry}}_l(\widehat{\mathsf{curry}}_l\, \mathsf{rb})$ is the least algorithm mapping the path $\langle\bot,\top\rangle\,\langle\bot,\top\rangle\,\langle\top,\top\rangle^\omega$ to $\top$. This algorithm of course computes the function b.

   Similar properties hold for the computation pairing rmerge and split_2, with the derived operations $\widehat{\mathsf{curry}}_r$, $\mathsf{un}\widehat{\mathsf{curry}}_r$ and $\widehat{\mathsf{app}}_r$.

2. The strictly increasing paths comonad T_2, with computational pairing σ and merge, again has operations $\widehat{\mathsf{curry}}$, $\widehat{\mathsf{uncurry}}$ and $\widehat{\mathsf{app}}$ that provide weak forms of exponentiation. As we remarked earlier, σ and merge are not isomorphisms. Instead, $\sigma \circ \mathsf{merge} = \mathsf{id}$ and for all $u \in T_2(A \times B)$ the computation $\mathsf{merge}(\sigma(u))$ is pointwise above u. Hence, $\widehat{\mathsf{curry}}(\widehat{\mathsf{uncurry}}\, h) = h$ and $\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}\, g) =^e g$. As an example, we have:

$$\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}\,\mathsf{lb}) = \widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}\,\mathsf{rb}) = \widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}\,\mathsf{pb}) = \mathsf{pb}.$$

The lmerge and split_2 computational pairing also gives rise to a weak form of exponentiation, as does the rmerge and split_2 computational pairing.

6 Ordered categories

So far, although our principal examples were based on a cartesian closed category of domains, we have not fully exploited the order structure. This permitted us to state and prove results that hold in a more general category-theoretic setting. Next we suppose that the underlying category is an ordered category: each hom-set is equipped with a complete partial order, and composition is continuous. A functor T of ordered categories is required to respect the ordering, in that for all $f, g : A \to^{\mathcal{C}} B$ if $f \leq g$ then $Tf \leq Tg$. Moreover, T must also be continuous (in its action on morphisms). All of our examples so far satisfy these conditions.

Suppose $(T, \epsilon, \delta, \gamma)$ is a computational comonad over an ordered category $\mathcal{C}$. Then clearly $\mathcal{C}_T$ is again an ordered category. All of the results of the previous sections go through in the ordered setting. In particular, the functors fun and alg introduced earlier respect the ordering; and the proof of Proposition 4.2 can be adapted to show that the extensional quotient of the ordering on $\mathcal{C}_T(A, B)$ is just the order on $\mathcal{C}(A, B)$.

We can also obtain some slightly stronger results by taking advantage of the ordering. We omit most of the proofs, which may be easily obtained from the results above, using monotonicity and continuity of composition.

Proposition 6.1 *Let $\mathcal{C}$ be an ordered ccc and let $(T, \epsilon, \delta, \gamma)$ be a computational comonad over $\mathcal{C}$ with a computational pairing.*

- *If* $\mathsf{split} \circ \mathsf{merge} \geq \mathsf{id}$ *then* $\widehat{\mathsf{curry}} \circ \widehat{\mathsf{uncurry}} \geq \mathsf{id}$.
- *If* $\mathsf{split} \circ \mathsf{merge} \leq \mathsf{id}$ *then* $\widehat{\mathsf{curry}} \circ \widehat{\mathsf{uncurry}} \leq \mathsf{id}$.
- *If* $\mathsf{merge} \circ \mathsf{split} \geq \mathsf{id}$ *then* $\widehat{\mathsf{uncurry}} \circ \widehat{\mathsf{curry}} \geq \mathsf{id}$.
- *If* $\mathsf{merge} \circ \mathsf{split} \leq \mathsf{id}$ *then* $\widehat{\mathsf{uncurry}} \circ \widehat{\mathsf{curry}} \leq \mathsf{id}$.

Next we introduce a simple generalization of the notion of cartesian closed ordered category, obtained by relaxing the requirement that currying and uncurrying form an isomorphism. Instead we allow currying and uncurrying to form an adjunction in each homset, so that we have an example of a *local adjunction* (see for example [9]) with additional properties. The relevance of "lax" notions of adjunction such as these in computational settings (albeit with different motivations) has been pointed out in different contexts by other authors, for instance in [14].

Definition 6.2 An ordered category $\mathcal{C}$ is *cartesian up-closed* if and only if it has finite products and for all pairs of objects B and C there is an object $B \to C$ and a pair of lax natural transformations curry, $\mathsf{uncurry}$ between $\mathcal{C}(\perp \times B, C)$ and $\mathcal{C}(\perp, B \to C)$ satisfying:

$$\begin{aligned}
\mathsf{curry}(\mathsf{uncurry}\, h) &\leq h \\
\mathsf{uncurry}(\mathsf{curry}\, g) &\geq g \\
\mathsf{curry}(g \circ (f \times \mathsf{id})) &\leq \mathsf{curry}(g) \circ f \\
\mathsf{uncurry}(h) \circ (f \times \mathsf{id}) &\geq \mathsf{uncurry}(h \circ f).
\end{aligned}$$

Similarly, we say that $\mathcal{C}$ is *cartesian down-closed* iff it has finite products and there is a pair of lax natural transformations curry, $\mathsf{uncurry}$ satisfying:

$$\begin{aligned}
\mathsf{curry}(\mathsf{uncurry}\, h) &\geq h \\
\mathsf{uncurry}(\mathsf{curry}\, g) &\leq g \\
\mathsf{curry}(g \circ (f \times \mathsf{id})) &\geq \mathsf{curry}(g) \circ f \\
\mathsf{uncurry}(h) \circ (f \times \mathsf{id}) &\leq \mathsf{uncurry}(h \circ f).
\end{aligned}$$

•

Definition 6.3 Let $\mathcal{C}$ be an ordered category with finite products.

- An *up-exponential* for objects B and C is an object $B \to C$ of $\mathcal{C}$ together with a morphism $\mathsf{app}_{B,C} : (B \to C) \times B \to^{\mathcal{C}} C$ such that for every $f : A \times B \to^{\mathcal{C}} C$ there is a least morphism $\mathsf{curry}(f) : A \to^{\mathcal{C}} (B \to C)$ such that

$$\mathsf{app} \circ (\mathsf{curry}(f) \times \mathsf{id}) \geq f.$$

- A *down-exponential* for objects B and C is an object $B \to C$ of $\mathcal{C}$ together with a morphism $\mathsf{app}_{B,C} : (B \to C) \times B \to^{\mathcal{C}} C$ such that for every $f : A \times B \to^{\mathcal{C}} C$ there is a greatest morphism $\mathsf{curry}(f) : A \to^{\mathcal{C}} (B \to C)$ such that

$$\mathsf{app} \circ (\mathsf{curry}(f) \times \mathsf{id}) \leq f.$$

•

The following result may be shown by adapting the usual proof that the two alternative definitions of cartesian closed categories are equivalent.

Proposition 6.4 *An ordered category $\mathcal{C}$ is cartesian up-closed iff it has finite products and up-exponentials.*

An ordered category is cartesian down-closed iff it has finite products and down-exponentials.

Note that if the same object $B \to C$ and morphism $\mathsf{app}_{B,C}$ qualifies simultaneously as an up- and a down-exponential then it forms the usual notion of exponentiation and the category is cartesian closed in the usual sense.

Proposition 6.5 *Let $\mathcal{C}$ be a cartesian up-closed category, let $(T, \epsilon, \delta, \gamma)$ be a computational comonad, and let split and merge be a computational pairing such that*

$$\begin{aligned}
&\mathsf{split} \circ \mathsf{merge} \leq \mathsf{id} \\
&\mathsf{merge} \circ \mathsf{split} \geq \mathsf{id} \\
&(\delta \times \delta) \circ \mathsf{split} \leq \mathsf{split} \circ T\, \mathsf{split} \circ \delta \\
&\mathsf{merge} \circ (\delta \times \delta) \geq T\, \mathsf{split} \circ \delta \circ \mathsf{merge}.
\end{aligned}$$

Then $\mathcal{C}_T$ is cartesian up-closed.

Proof:

- Let $a : A \times B \to^{\mathcal{C}_T} C$ and $b : A \to^{\mathcal{C}_T} (TB \to C)$. Then:

$$\widehat{\mathsf{uncurry}}(\widehat{\mathsf{curry}}\, a) = \mathsf{uncurry}(\mathsf{curry}(a \circ \mathsf{merge})) \circ \mathsf{split} \geq a \circ \mathsf{merge} \circ \mathsf{split} \geq a$$
$$\widehat{\mathsf{curry}}(\widehat{\mathsf{uncurry}}\, b) = \mathsf{curry}(\mathsf{uncurry}(b) \circ \mathsf{split} \circ \mathsf{merge}) \leq \mathsf{curry}(\mathsf{uncurry}\, b) \leq b.$$

- To show that $\widehat{\mathsf{curry}}$ is a lax natural transformation, let $f : A \to^{\mathcal{C}_T} A'$ and $g : A' \times B \to^{\mathcal{C}_T} C$. Then:

$$\begin{aligned}
\widehat{\mathsf{curry}}(g \mathbin{\bar\circ} (f \mathbin{\hat\times} \widehat{\mathsf{id}})) &= \widehat{\mathsf{curry}}(g \circ T(f \times \epsilon) \circ T\,\mathsf{split} \circ \delta) \\
&= \mathsf{curry}(g \circ T(f \times \epsilon) \circ T\,\mathsf{split} \circ \delta \circ \mathsf{merge}) \\
&\leq \mathsf{curry}(g \circ T(f \times \epsilon) \circ \mathsf{merge} \circ (\delta \times \delta)) \\
&= \mathsf{curry}(g \circ \mathsf{merge} \circ (Tf \times T\epsilon) \circ (\delta \times \delta)) \\
&= \mathsf{curry}(g \circ \mathsf{merge} \circ ((Tf \circ \delta) \times (T\epsilon \circ \delta))) \\
&= \mathsf{curry}(g \circ \mathsf{merge} \circ ((Tf \circ \delta) \times \mathsf{id})) \\
&\leq \mathsf{curry}(g \circ \mathsf{merge}) \circ (Tf \circ \delta) \\
&= \widehat{\mathsf{curry}}(g) \circ Tf \circ \delta \\
&= \widehat{\mathsf{curry}}(g) \mathbin{\bar\circ} f.
\end{aligned}$$

- To show that $\widehat{\mathsf{uncurry}}$ is a lax natural transformation, suppose that $f : A \to^{\mathcal{C}_T} A'$ and $h : A' \to^{\mathcal{C}_T} (TB \to C)$. Then

$$\begin{aligned}
\widehat{\mathsf{uncurry}}(h) \mathbin{\bar\circ} (f \mathbin{\hat\times} \widehat{\mathsf{id}}) &= \mathsf{uncurry}(h) \circ \mathsf{split} \circ T(f \times \epsilon) \circ T\,\mathsf{split} \circ \delta \\
&\geq \mathsf{uncurry}(h) \circ (Tf \times T\epsilon) \circ \mathsf{split} \circ T\,\mathsf{split} \circ \delta \\
&= \mathsf{uncurry}(h) \circ (Tf \times T\epsilon) \circ (\delta \times \delta) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h) \circ ((Tf \circ \delta) \times (T\epsilon \circ \delta)) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h) \circ ((Tf \circ \delta) \times \mathsf{id}) \circ \mathsf{split} \\
&\geq \mathsf{uncurry}(h \circ Tf \circ \delta) \circ \mathsf{split} \\
&= \mathsf{uncurry}(h \mathbin{\bar\circ} f) \circ \mathsf{split} \\
&= \widehat{\mathsf{uncurry}}(h \mathbin{\bar\circ} f).
\end{aligned}$$

∎

A similar result holds for a cartesian down-closed category with a computational pairing satisfying reversed inequalities.

7 Examples

We now return to the third comonad introduced earlier, after which we will introduce briefly some related of notions of computation on different categories of domains and functions.

7.1 Timed data

In the timed data comonad T_3, the standard split operation is:

$$\mathsf{split}\,\langle\langle a, b\rangle, n\rangle = \langle\langle a, n\rangle, \langle b, n\rangle\rangle.$$

Given our interpretation of $\langle d, n\rangle$ as a computation yielding d at cost n, an obvious choice for a merge operation is:

$$\mathsf{merge}(\langle a, m\rangle, \langle b, n\rangle) = \langle\langle a, b\rangle, \max(m, n)\rangle .$$

Both of these operations are natural transformations, and we obtain the following properties:

$$\begin{aligned}&\mathsf{split}\circ\mathsf{merge} \sqsubseteq \mathsf{id}\\ &\mathsf{merge}\circ\mathsf{split} = \mathsf{id}\\ &(\delta\times\delta)\circ\mathsf{split} = \mathsf{split}\circ T_3\,\mathsf{split}\circ\delta\\ &\mathsf{merge}\circ(\delta\times\delta) \sqsupseteq T_3\,\mathsf{split}\circ\delta\circ\mathsf{merge}.\end{aligned}$$

The underlying category is cartesian closed, hence also cartesian up-closed. It follows from Proposition 6.5 that the Kleisli category of T_3 is cartesian up-closed.

7.2 Strict algorithms

The category of domains and strict continuous functions is not cartesian closed, although the category does have products. For each pair of domains D and D', the set of strict continuous functions $D \to_s D'$, ordered pointwise, is again a domain. The usual uncurrying operation on functions preserves strictness, but the usual currying does not. Instead, we may define a variant form of currying by:

$$\begin{aligned}&\mathsf{curry}_s : (A\times B \to_s C) \to (A \to_s (B \to_s C))\\ &\mathsf{curry}_s(f) = \lambda x.\lambda y.(x = \bot \vee y = \bot \to \bot, f(x, y)).\end{aligned}$$

When f is strict, $\mathsf{curry}_s(f)$ is the best strict function approximating $\mathsf{curry}(f)$ pointwise. For instance, let lor, ror and sor be the left-strict, right-strict, and doubly-strict or-functions. Then

$$\mathsf{uncurry}(\mathsf{curry}_s\,\mathsf{lor}) = \mathsf{uncurry}(\mathsf{curry}_s\,\mathsf{ror}) = \mathsf{uncurry}(\mathsf{curry}_s\,\mathsf{sor}) = \mathsf{sor}.$$

It is easy to check that curry_s is a natural transformation (and so is $\mathsf{uncurry}$).

The following relationships hold, for all $f : A \times B \to_s C$ and all $g : A \to_s (B \to_s C)$:

$$\begin{aligned}&\mathsf{curry}_s(\mathsf{uncurry}\,g) = g\\ &\mathsf{uncurry}(\mathsf{curry}_s\,f) \sqsubseteq f.\end{aligned}$$

Hence, the category of domains and strict continuous functions is cartesian down-closed.

Let $T_1 D$ be the set of increasing paths over D (not just the strict continuous maps from `VNat` to D), ordered pointwise. The maps ϵ, δ and γ are all strict, as are all of the split and merge operations above. We may therefore use the Kleisli construction to build a model of strict parallel algorithms. To illustrate this model, note that all of the algorithms of Figure 6 also belong in this category, with the exception of $\top$, which is non-strict.

Since the underlying category is cartesian down-closed, each of the computational pairings discussed earlier for T_1 gives rise to a down-exponentiation structure, so that the category of domains and strict algorithms is again cartesian down-closed.

We may also adapt the T_2 and T_3 comonads to this category.

7.3 Computation on effectively given domains

The category of effectively given domains and computable functions is cartesian closed. A domain is effectively given iff its finite elements are recursively enumerable (hence, countable), it is decidable whether two finite elements are consistent, and (an index for) the lub of two consistent finite elements is decidable (as a function of their indices). An element of D is computable iff the set of (indices of) its finite approximations is recursively enumerable.

The functor T_1 can be adapted to this category, by defining $T_1 D$ for an effectively given domain D to be the computable increasing paths over D (equivalently, the computable continuous functions from `VNat` to D, ordered pointwise). All of the auxiliary operations (ϵ, δ, γ, and so on) are computable. Hence we obtain a category of effectively given domains and computable algorithms, and this category quotients onto the underlying category of effectively given domains and computable functions. This algorithms category is again cartesian closed.

The functor T_3 maps effectively given domains to effectively given domains, and again the auxiliary operations are computable. We therefore obtain a category of effectively given domains and T_3-algorithms that quotients onto the underlying category and is cartesian up-closed.

The functor T_2 preserves algebraicity but not ω-algebraicity, since $T_2 D$ may have uncountably many finite elements. The T_2 comonad therefore does not adapt to the category of effectively given domains and computable functions.

7.4 Computation on pre-domains

We use the term *pre-domain* for a "bottomless" domain: a directed-complete, bounded complete, algebraic partial order with no requirement that there be a least element. The category of pre-domains and continuous functions is cartesian closed.

Let $T_4 D$ be the set of non-empty finite or infinite increasing sequences over D, ordered by the prefix ordering. Clearly this forms a pre-domain, and the finite elements are just the finite sequences. $T_4 D$ is generally a pre-domain rather than a domain, even if D has a least element, because the prefix ordering does not relate sequences with different first elements. We make T into a functor by specifying that (as usual) $T_4 f$ applies f componentwise.

Again we let ϵ be the lub operation and let δt be the sequence of (non-empty) prefixes of t. Then (T_4, ϵ, δ) forms a comonad.

We may regard a computation of length 1 as degenerate, and this corresponds to defining the function γ from D to $T_4 D$ by $\gamma d = \langle d \rangle$. Although this function γ is not continuous, so that we cannot claim that T_4 is a computational comonad, we still obtain a congruence relation on algorithms by defining

$$a =^e a' \iff \forall d \in D . a\langle d \rangle = a'\langle d \rangle .$$

Note that for all $f, g : A \to B$ we have

$$(f \circ \epsilon) =^e (g \circ \epsilon) \Rightarrow f = g.$$

It is then easy to modify the proof of Proposition 4.2 to show that the Kleisli category of this comonad quotients onto the underlying category under $=^e$.

We may define splitting and merging operations as follows. The standard way to split is:

$$\begin{aligned}&\mathsf{split}(\langle x_0, y_0\rangle \ldots \langle x_k, y_k\rangle) = \langle x_0 \ldots x_k, y_0 \ldots y_k\rangle \\ &\mathsf{split}(\langle x_n, y_n\rangle_{n=0}^{\infty}) = \langle \langle x_n\rangle_{n=0}^{\infty}, \langle y_n\rangle_{n=0}^{\infty}\rangle .\end{aligned}$$

Let merge be the least continuous function satisfying:

$$\begin{aligned}
&\mathsf{merge}(x_0 \dots x_k, y_0 \dots y_m) = \langle x_0, y_0\rangle \dots \langle x_n, y_n\rangle \qquad (n = \min(m,k))\\
&\mathsf{merge}(x_0 \dots x_k, \langle y_i\rangle_{i=0}^{\infty}) = \langle x_0, y_0\rangle \dots \langle x_k, y_k\rangle\\
&\mathsf{merge}(\langle x_i\rangle_{i=0}^{\infty}, y_0 \dots y_m) = \langle x_0, y_0\rangle \dots \langle x_m, y_m\rangle\\
&\mathsf{merge}(\langle x_n\rangle_{n=0}^{\infty}, \langle y_n\rangle_{n=0}^{\infty}) = \langle x_n, y_n\rangle_{n=0}^{\infty}\,.
\end{aligned}$$

Clearly, $\mathsf{merge} \circ \mathsf{split} = \mathsf{id}$ and $\mathsf{split} \circ \mathsf{merge} \sqsubseteq \mathsf{id}$. These two operations are obviously natural and satisfy the computational pairing properties, except that $\epsilon \circ \mathsf{merge} \sqsubseteq \epsilon \times \epsilon$. The Kleisli category is cartesian down-closed.

8 Conclusions

We have described a category-theoretic approach to intensional semantics, based on the idea that a notion of computation or intensional behavior may be modelled by means of a computational comonad, and that the Kleisli category thus obtained can be viewed as an intensional model. The morphisms in this category map computations to values, and from such a morphism one may recover a map from values to values. One may define an equivalence relation that identifies all algorithms that compute the same function, and this equivalence relation can be used to collapse the Kleisli category onto the underlying category.

We have identified a set of conditions under which the Kleisli category possesses exponentiations or weaker types of exponentiation, based on the existence of natural ways to pair computations. We described a series of examples to illustrate the applicability of our definitions and results. In doing so, we have placed our recent work [4] in a wider context.

Our work arose out of an attempt, begun in [3], to generalize an earlier intensional model of Berry and Curien [6]. They defined a category of deterministic concrete data structures and sequential algorithms, showed that this category is cartesian closed, and that it collapses onto the category of deterministic concrete data structures and sequential functions under an obvious notion of extensional equivalence. The sequential functions category is not cartesian closed, and their construction of sequential algorithms was not based on a comonad. The operational semantics implicit in their work was again coroutine-like and lazy, but with the restriction that computation should proceed *sequentially*, with at most one argument being evaluated at a time. In our generalization of their model we relax the sequentiality restriction so as to permit parallel computation.

The *query model* of parallel algorithms between deterministic concrete data structures, described in [3], contains algorithms for non-sequential functions such as parallel-or. However, the model's construction was rather complex and we were unable to formulate a suitable notion of composition for algorithms. Instead, in [4] we presented a much more streamlined form of algorithm between Scott domains and for the first time we cast our construction in terms of a comonad. Of the comonads introduced in this paper, T_1 corresponds to the comonad used in [4]; T_2 is closer in spirit to the query model of [3], but we are able here to go considerably further.

Moggi has developed an abstract view of programming languages in which a notion of computation is modelled as a monad [12, 13]. Examples of notions of computation as monads include: computation with side-effects, computation with exceptions, partial computations, and non-deterministic computations. In this view, the meaning of a program is taken as a function from values to computations, and an intuitive operational semantics is that a program from A to B takes an input of type A and returns an output computation. This point of view is consistent with an input-driven lazy operational semantics. In contrast, our "opposite" point of view based on comonads (which are, after all, monads on the opposite category) is consistent with a demand-driven lazy operational semantics. Moggi states in [12] that his view of programs corresponds to call-by-value parameter passing, and he says that there is an alternative view of "programs as functions from computations to computations" corresponding to call-by-name. Our work shows that there is also a third alternative: programs as functions from computations to values. Common to these approaches is the realization that values should be distinguished from computations (and the use of an endofunctor T). Apart from that, the motivations and the operational intuitions behind the monad approach and the comonad approach are different, and we feel that the two approaches should be regarded as orthogonal or complementary. The extra structure and algebraic laws embodied in a monad seem appropriate in Moggi's context. Equally, the extra structure and algebraic laws embodied in a comonad seem appropriate in our context. We plan to explore to what extent (and to what effect) the two approaches can be combined. For instance, given a comonad T and a monad P over the same category $\mathcal{C}$ one might obtain (assuming that T and P satisfy certain properties) a category of (T, P)-algorithms, in which a morphism from A to B is a morphism in $\mathcal{C}$ from TA to PB.

We plan to investigate notions of computation in further domain-theoretic settings. We are already working on categories of algorithms on (generalized) concrete data structures [5]. It would be interesting to see if the Berry-Curien sequential algorithms category could be embedded in the Kleisli category of a suitable comonad over a sequential functions category. We intend to investigate notions of computation on the category of dI-domains and stable functions [2], and on the category of qualitative domains and linear functions [7].

Acknowledgements

The diagrams in this paper were drawn using macros designed by John Reynolds.

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall International, 1990.

[2] G. Berry. Stable models of typed λ-calculi. In *Proc. 5^{th} Coll. on Automata, Languages and Programming*, number 62 in Lecture Notes in Computer Science, pages 72–89. Springer-Verlag, July 1978.

[3] S. Brookes and S. Geva. Towards a theory of parallel algorithms on concrete data structures. In *Semantics for Concurrency, Leicester 1990*, pages 116–136. Springer Verlag, 1990.

[4] S. Brookes and S. Geva. A cartesian closed category of parallel algorithms between Scott domains. Technical Report CMU-CS-91-159, Carnegie Mellon University, School of Computer Science, 1991. Submitted for publication.

[5] S. Brookes and S. Geva. Continuous functions and parallel algorithms on concrete data structures. Technical Report CMU-CS-91-160, Carnegie Mellon University, School of Computer Science, 1991. To appear in the Proceedings of MFPS'91.

[6] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, 1986.

[7] J.-Y. Girard. *Proofs and Types*. Cambridge University Press, 1989.

[8] C. Gunter and D. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. MIT Press/Elsevier, 1990.

[9] C. B. Jay. Local adjunctions. *Journal of Pure and Applied Logic*, 53:227–238, 1988.

[10] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 1977*, pages 993–998. North Holland, 1977.

[11] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[12] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1989.

[13] E. Moggi. Notions of computation and monads. *Information and Computation*, 1991.

[14] R. A. G. Seely. Modeling computations: A 2-categorical framework. In *Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1987.