CMCS 2008

Comonadic Notions of Computation

Tarmo Uustalu[1]

Institute of Cybernetics at Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia

Varmo Vene[2]

Dept. of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia

Abstract

We argue that symmetric (semi)monoidal comonads provide a means to structure context-dependent notions of computation such as notions of dataflow computation (computation on streams) and of tree relabelling as in attribute evaluation. We propose a generic semantics for extensions of simply typed lambda calculus with context-dependent operations analogous to the Moggi-style semantics for effectful languages based on strong monads. This continues the work in the early 90s by Brookes, Geva and Van Stone on the use of computational comonads in intensional semantics.

Keywords: context-dependent computation, dataflow computation, tree transformations, symmetric monoidal comonads, coKleisli semantics

1 Introduction

Since the seminal work by Moggi in the late 80s [25], *monads*, more precisely, *strong monads*, have become a generally accepted tool for structuring *effectful* notions of computation, such as computation with exceptions, output, computation using an environment, state-transforming, nondeterministic and probabilistic computation etc. The idea is to use a Kleisli category as the category of impure, effectful functions, with the Kleisli inclusion giving an embedding of the pure functions from the base category. Although finer and coarser accounts of effects based on Lawvere theories [27] (this reference is only the first in a series of papers; for more recent presentations, see [28,18]) and arrows/Freyd categories [17,30] also exist, the monadic approach remains central and best known. In particular, monads are part of the standard libraries of Haskell.

[1] tarmo@cs.ioc.ee

[2] varmo@cs.ut.ee

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

But monads do not capture all meaningful kinds of impure computation. In particular, they exclude some natural notions where, instead of producing effects, computations consume something beyond just values ("contexts" of values). Hence it is natural to ask whether *comonads*, likely with some additional structure (as strength for monads), can deliver a solution for such cases. (Combinations of monads and comonads via *distributive laws* can then hopefully account for notions of computation that are both effectful and context-dependent.) That this may well be so was hinted very early on by Brookes et al. [8,9] who demonstrated that what they called *computational comonads* can be used to make denotational semantics intensional. While that work was not directly involved with general context-dependent computation, intensional semantics is a form of impure instrumentation of denotational semantics that fits well into this idiom.

In functional programming, Kieburtz [20] was first to advocate comonads as tools for structuring context-dependent computations and gave some interesting examples. The specific application of comonads to environment-passing computation or implicit parameters has been discussed by Lewis et al. [22].

In this paper, we proceed directly from the motivation to treat some important notions of context-dependent computation, namely notions of dataflow computation (stream-based computation) and notions of computation on trees such as tree relabellings in attribute evaluation. We demonstrate that a rather elegant framework for working with these notions of computation is given by *symmetric (semi)monoidal comonads*. Reassuringly, strong monads appear associated with symmetric monoidal comonads also in works on the categorical semantics of intuitionistic linear and modal logic [4,7]. We describe some aspects of the structure of coKleisli categories corresponding to symmetric (semi)monoidal comonads and describe then a general interpretation of languages for context-dependent computation into such categories.

We have previously described our proposal at work on language processors for dataflow computation [34] and attribute evaluation [35] implemented in Haskell. In this paper, written with a different slant, we look into the underlying theory, concentrating on the issue of the most appropriate additional structure for comonads.

The organization of the paper is the following. First, we present a compressed recap of strong monads, their Kleisli categories and the semantics of effectful languages à la Moggi. Then we develop our analogous account of context-dependent computation based on coKleisli categories of symmetric (semi)monoidal comonads. We emphasize the important differences resulting from the fact that, despite dualizing from monads to comonads, we are still interested in transferring as much of a given Cartesian closed structure (possibly with coproducts and a uniform parameterized fixpoint operation) as possible. Finally, we briefly comment on the relation to computational comonads and some important advanced issues that we intend to treat in due detail elsewhere.

We assume that the reader knows symmetric monoidal closed and Cartesian closed categories and the categorical semantics of simply typed lambda calculus. We reproduce some basics about monads, comonads, strong functors/monads and symmetric monoidal functors/comonads.

2 Monads and effectful computation

We begin by a schematic review of the monad-based approach to effectful computation. The purpose is to recall the central ideas, the technical machinery and the big "scheme of things" of this case, so we can establish a standard of what we want to achieve in the case of comonads and context-dependent computation.

2.1 Monads

The starting-point in the monadic approach to (call-by-value) effectful computation is the idea that impure, effectful functions from A to B must be nothing else than pure functions from A to TB. Here pure functions live in a base category $\mathcal{C}$ and T is an endofunctor on $\mathcal{C}$ that describes the notion of effect of interest; it is useful to think of TA as the type of effectful computations of values of a given type A.

For this to work, impure functions must have identities and compose. Therefore T cannot merely be a functor, but must be a monad.

A *monad* on a category $\mathcal{C}$ is given by a functor $T : \mathcal{C} \to \mathcal{C}$ (the *underlying functor*), two natural transformations $\eta : \mathsf{Id}_{\mathcal{C}} \dot{\to} T$ (the *unit*) and $\mu : TT \dot{\to} T$ (the *multiplication*) satisfying the conditions

$$
\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[d, "T\eta_A"'] \arrow[dr, equal] & TTA \arrow[d, "\mu_A"] \\
TTA \arrow[r, "\mu_A"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TTTA \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & TTA \arrow[d, "\mu_A"] \\
TTA \arrow[r, "\mu_A"'] & TA
\end{tikzcd}
$$

This definition says that (T, η, μ) is a monoid in the endofunctor category $[\mathcal{C}, \mathcal{C}]$ wrt. its $(\mathsf{Id}_{\mathcal{C}}, *)$ monoidal structure.

A monad T on a category $\mathcal{C}$ induces a category $\mathbf{Kl}(T)$ called the *Kleisli category* of T defined by

- an object is an object of $\mathcal{C}$,
- a map of from A to B is a map of $\mathcal{C}$ from A to TB,
- $\mathsf{id}^T_A =_{\mathrm{df}} A \xrightarrow{\eta_A} TA$,
- if $k : A \to^T B$, $\ell : B \to^T C$, then $\ell \circ^T k =_{\mathrm{df}} A \xrightarrow{k} TB \xrightarrow{\ell^\star} TC$ where $\ell^\star =_{\mathrm{df}} TB \xrightarrow{T\ell} TTC \xrightarrow{\mu_C} TC$.

Note that it is the unit η and multiplication μ that make the Kleisli identity id^T and composition $\circ^T$ possible; the laws of the identity and composition follow from those of η and μ.

From $\mathcal{C}$ there is an identity-on-objects *inclusion functor* J to $\mathbf{Kl}(T)$, defined on maps by: if $f : A \to B$, then $Jf =_{\mathrm{df}} A \xrightarrow{f} B \xrightarrow{\eta_B} TB = A \xrightarrow{\eta_A} TA \xrightarrow{Tf} TB$. (It is truly an inclusion, if η is mono, but we ignore this.) It has a right adjoint $U : \mathbf{Kl}(T) \to \mathcal{C}$ given by: $UA =_{\mathrm{df}} TA$ and, if $k : A \to^T B$, then $Uk =_{\mathrm{df}} TA \xrightarrow{k^\star} TB$.

In our application, we use the Kleisli category $\mathbf{Kl}(T)$ as the category of effectful functions and J as an embedding from the category of pure functions $\mathcal{C}$. Accordingly, for us, the function $\eta_A = J\mathsf{id}_A : A \to TA$ is the pure identity function on A turned into a trivially effectful function. $Jf : A \to TB$ is a general pure function $f : A \to B$ viewed as trivially effectful. $\mu_A = \mathsf{id}^\star_{TA} : TTA \to TA$ "flattens" an effectful computation of an effectful computation. $k^\star : TA \to TB$ is an effectful function $k : A \to TB$ extended into one that can input an effectful computation.

Well-known examples of monads on Cartesian categories[3] (optionally with coproducts, optionally closed), e.g., **Set**, include the following.

The *exceptions monad* is given by

- $TA =_{\mathrm{df}} A + E$ where E is some object (of exceptions),
- $\eta_A =_{\mathrm{df}} A \xrightarrow{\mathsf{inl}} A + E$,
- $\mu_A =_{\mathrm{df}} (A + E) + E \xrightarrow{[\mathsf{id},\mathsf{inr}]} A + E$

and is used for computation with exceptions. Properly impure functions are possible thanks to the error-raise operation $\mathsf{raise}_A =_{\mathrm{df}} E \xrightarrow{\mathsf{inr}} A + E$.

The *output monad* is given by

- $TA =_{\mathrm{df}} A \times E$ where (E, e, m) is some monoid (of output traces), e.g., the type of lists of a fixed element type with nil and append,
- $\eta_A =_{\mathrm{df}} A \xrightarrow{\mathsf{ur}} A \times 1 \xrightarrow{\mathsf{id} \times e} A \times E$,
- $\mu_A =_{\mathrm{df}} (A \times E) \times E \xrightarrow{\mathsf{a}} A \times (E \times E) \xrightarrow{\mathsf{id} \times m} A \times E$

and is used to handle observable output or time. The important operation for generating impure functions is $\mathsf{print} : E \xrightarrow{\langle !, \mathsf{id} \rangle} 1 \times E$.

The *environment monad* is given by:

- $TA =_{\mathrm{df}} E \Rightarrow A$ where E is some object (of environments),
- $\eta_A =_{\mathrm{df}} \Lambda(A \times E \xrightarrow{\mathsf{fst}} A) : A \to E \Rightarrow A$,
- $\mu_A =_{\mathrm{df}} \Lambda((E \Rightarrow (E \Rightarrow A)) \times E \xrightarrow{\langle \mathsf{ev}, \mathsf{snd} \rangle} (E \Rightarrow A) \times E \xrightarrow{\mathsf{ev}} A)$
  $: E \Rightarrow (E \Rightarrow A) \to E \Rightarrow A$.

It is used to deal with a readable environment. The native operation is $\mathsf{ask} =_{\mathrm{df}} \Lambda(1 \times E \xrightarrow{\mathsf{snd}} E) : 1 \to E \Rightarrow E$.

Further important well-known examples include the state monad, the continuations monad, free monads and free completely iterative monads [2].

2.2 *Strong monads*

To be able to use a Kleisli category as a category of computation, we also need it to have datatypes. At the very least, it must support something product-like and in particular something approximating local composition to interpret *let*.

In order for these to exist, the monad must be strong.

A *strong functor* on a monoidal category $(\mathcal{C}, I, \otimes)$ is given by an endofunctor F on $\mathcal{C}$ together with a natural transformation $\mathsf{sl}_{A,B} : A \otimes FB \to F(A \otimes B)$ (the

[3] By a Cartesian category we mean one with finite products (the word is also used to refer to categories with finite limits).

(tensorial) strength) satisfying

$$
\begin{tikzcd}
I \otimes FA \arrow[r, "\mathsf{sl}_{I,A}"] \arrow[d, "\mathsf{ul}_{FA}"'] & F(I \otimes A) \arrow[d, "F\mathsf{ul}_{A}"] \\
FA \arrow[r, equal] & FA
\end{tikzcd}
\quad
\begin{tikzcd}
(A \otimes B) \otimes FC \arrow[rr, "\mathsf{sl}_{A \otimes B,C}"] \arrow[d, "\mathsf{a}_{A,B,FC}"'] & & F((A \otimes B) \otimes C) \arrow[d, "F\mathsf{a}_{A,B,C}"] \\
A \otimes (B \otimes FC) \arrow[r, "\mathsf{id}_{A} \otimes \mathsf{sl}_{B,C}"'] & A \otimes F(B \otimes C) \arrow[r, "\mathsf{sl}_{A,B \otimes C}"'] & F(A \otimes (B \otimes C))
\end{tikzcd}
$$

(Note that a monad can generally have more than one strength.)

A *strong natural transformation* between two strong functors (F, sl^F), (G, sl^G) is a natural transformation $\tau : F \dot{\to} G$ satisfying

$$
\begin{tikzcd}
A \otimes FB \arrow[r, "\mathsf{sl}^F_{A,B}"] \arrow[d, "\mathsf{id}_{A} \otimes \tau_{B}"'] & F(A \otimes B) \arrow[d, "\tau_{A \otimes B}"] \\
{} & {}
\end{tikzcd}
$$

$$A \otimes GB \xrightarrow[\mathsf{sl}^G_{A,B}]{} G(A \otimes B)$$

A *strong monad* on a monoidal category $(\mathcal{C}, I, \otimes)$ is a monad (T, η, μ) where T is a strong functor and η, μ are strong natural transformations. The latter part means that η, μ satisfy

$$\begin{tikzcd}
A \otimes B \arrow[r, equal] \arrow[d, "\mathsf{id}_A \otimes \eta_B"'] & A \otimes B \arrow[d, "\eta_{A \otimes B}"] \\
A \otimes TB \arrow[r, "\mathsf{sl}_{A,B}"'] & T(A \otimes B)
\end{tikzcd}
\qquad
\begin{tikzcd}
A \otimes TTB \arrow[r, "\mathsf{sl}_{A,TB}"] \arrow[d, "\mathsf{id}_A \otimes \mu_B"'] & T(A \otimes TB) \arrow[r, "T\mathsf{sl}_{A,B}"] & TT(A \otimes B) \arrow[d, "\mu_{A \otimes B}"] \\
A \otimes TB \arrow[rr, "\mathsf{sl}_{A,B}"'] & & T(A \otimes B)
\end{tikzcd}$$

(Note that Id is canonically strong and strong F, G make GF canonically strong.)

A strong functor (F, sl) on a symmetric monoidal category $(\mathcal{C}, I, \otimes)$ is automatically bistrong: it is also endowed with a *costrength* $\mathsf{sr}_{A,B} : FA \otimes B \to F(A \otimes B)$ whose properties are symmetric to those of a strength. It is defined by

$$\mathsf{sr}_{A,B} =_{\mathrm{df}} FA \otimes B \stackrel{\mathsf{c}_{FA,B}}{\longrightarrow} B \otimes FA \stackrel{\mathsf{sl}_{B,A}}{\longrightarrow} F(B \otimes A) \stackrel{F\mathsf{c}_{B,A}}{\longrightarrow} F(A \otimes B)$$

A bistrong monad $(T, \mathsf{sl}, \mathsf{sr})$ is called *commutative*, if it satisfies

$$TA \otimes TB \xrightarrow{\mathsf{sl}_{TA,B}} T(TA \otimes B) \xrightarrow{T\mathsf{sr}_{A,B}} TT(A \otimes B)$$

$$
\begin{tikzcd}
{} \arrow[d, "\mathsf{sr}_{A,TB}"'] & & {} \arrow[dd, "\mu_{A\otimes B}"] \\
T(A \otimes TB) \arrow[d, "T\mathsf{sl}_{A,B}"'] & & \\
TT(A \otimes B) \arrow[rr, "\mu_{A\otimes B}"'] & & T(A \otimes B)
\end{tikzcd}
$$

Strength is not a very restrictive condition. In particular, on **Set**, every monad is strong, with a unique strength. The reason is that any endofunctor F on **Set** has a unique functorial strength $fs_{X,Y} : X \Rightarrow Y \to FX \Rightarrow FY$ (internalizing its functoriality) and any natural transformation between endofunctors on **Set** is functorially strong. Functorial strength implies tensorial strength. Commutativity is rarer. An example of a commutative monad is the exponent monad.

Given a Cartesian category $\mathcal{C}$ and a $(1, \times)$ strong monad T on it, we can manufacture "preproducts" in $\mathbf{Kl}(T)$ using the products of $\mathcal{C}$ and the strength sl like this:

$$
\begin{array}{rclcrcl}
1^T & =_{\mathrm{df}} & 1 & \qquad & A_0 \times^T A_1 & =_{\mathrm{df}} & A_0 \times A_1 \\
 & & & & \mathsf{fst}^T & =_{\mathrm{df}} & \eta \circ \mathsf{fst} \\
 & & & & \mathsf{snd}^T & =_{\mathrm{df}} & \eta \circ \mathsf{snd} \\
!^T & =_{\mathrm{df}} & \eta \circ ! & & \langle k_0, k_1 \rangle^T & =_{\mathrm{df}} & \mathsf{sl}^\star \circ \mathsf{sr} \circ \langle k_0, k_1 \rangle
\end{array}
$$

With this definition, the typing rules for products hold, but not all laws. In particular, the beta-laws $\mathsf{fst}^T \circ^T \langle k_0, k_1 \rangle^T = k_0$ and $\mathsf{snd}^T \circ^T \langle k_0, k_1 \rangle^T = k_1$ do not hold. And the binary operation $\times^T$ on objects does not extend to a bifunctor (although it is a functor in each argument separately). But some other important laws, such as, e.g., the eta-law $\langle \mathsf{fst}^T, \mathsf{snd}^T \rangle^T = \mathsf{id}^T$, survive. In particular, $(\mathbf{Kl}(T), J)$ is a Freyd category on $\mathcal{C}$, i.e., a symmetric premonoidal category with an identity-on-objects functor from $\mathcal{C}$ strictly preserving the $(1, \times)$ symmetric premonoidal structure of $\mathcal{C}$ and also centrality. (Since all maps of $\mathcal{C}$ are central, the latter part really means sending all maps of $\mathcal{C}$ to central maps of $\mathbf{Kl}(T)$).

If $\mathcal{C}$ is Cartesian closed, then "pre-exponents" in $\mathbf{Kl}(T)$ can be defined from the exponents of $\mathcal{C}$ by

$$\begin{aligned} A \Rightarrow^T B &=_{\mathrm{df}} A \Rightarrow TB \\ \mathsf{ev}^T &=_{\mathrm{df}} \mathsf{ev} \\ \Lambda^T(k) &=_{\mathrm{df}} \eta \circ \Lambda(k) \end{aligned}$$

It is not true that $A \Rightarrow^T - : \mathbf{Kl}(T) \to \mathbf{Kl}(T)$ is right adjoint to $- \times^T A : \mathbf{Kl}(T) \to \mathbf{Kl}(T)$. So $\Rightarrow^T$ is not a true exponent functor wrt. the preproduct functor $\times^T$. However $A \Rightarrow^T - : \mathbf{Kl}(T) \to \mathcal{C}$ is right adjoint to $J(- \times A) : \mathcal{C} \to \mathbf{Kl}(T)$:

$$\cfrac{\cfrac{\cfrac{J(C \times A) \to^T B}{C \times A \to TB}}{C \to A \Rightarrow TB}}{C \to A \Rightarrow^T B}$$

2.3 Semantics of effectful languages

Given a strong monad T on a Cartesian closed category $\mathcal{C}$, the pure part of an effectful language can be interpreted into $\mathbf{Kl}(T)$ in the standard way, relying on the generic pre-(Cartesian closed) structure of Kleisli categories of strong monads.

$$
\begin{array}{rcll}
[\![K]\!]^T & =_{\mathrm{df}} & \text{an object of } \mathbf{Kl}(T) & = \text{that object of } \mathcal{C} \\
[\![A \times B]\!]^T & =_{\mathrm{df}} & [\![A]\!]^T \times^T [\![B]\!]^T & = [\![A]\!]^T \times [\![B]\!]^T \\
[\![A \Rightarrow B]\!]^T & =_{\mathrm{df}} & [\![A]\!]^T \Rightarrow^T [\![B]\!]^T & = [\![A]\!]^T \Rightarrow T[\![B]\!]^T \\
[\![\underline{C}]\!]^T & =_{\mathrm{df}} & [\![C_0]\!]^T \times^T \ldots \times^T [\![C_{n-1}]\!]^T & = [\![C_0]\!]^T \times \ldots \times [\![C_{n-1}]\!]^T \\
[\![(\underline{x})\, x_i]\!]^T & =_{\mathrm{df}} & \pi_i^T & = \eta \circ \pi_i \\
[\![(\underline{x})\, \mathit{let}\ x \leftarrow t\ \mathit{in}\ u]\!]^T & =_{\mathrm{df}} & [\![(\underline{x}, x)\, u]\!]^T \circ^T \langle \mathsf{id}^T, [\![(x)\, t]\!]^T \rangle^T & = ([\![(\underline{x}, x)\, u]\!]^T)^\star \circ \mathsf{sl} \circ \langle \mathsf{id}, [\![(x)\, t]\!]^T \rangle \\
[\![(\underline{x})\, \mathit{fst}\ t]\!]^T & =_{\mathrm{df}} & \mathsf{fst}^T \circ^T [\![(\underline{x})t]\!]^T & = T\mathsf{fst} \circ [\![(\underline{x})t]\!]^T \\
[\![(\underline{x})\, \mathit{snd}\ t]\!]^T & =_{\mathrm{df}} & \mathsf{snd}^T \circ^T [\![(\underline{x})t]\!]^T & = T\mathsf{snd} \circ [\![(\underline{x})t]\!]^T \\
[\![(\underline{x})\, (t_0, t_1)]\!]^T & =_{\mathrm{df}} & \langle [\![(\underline{x})t_0]\!]^T, [\![(\underline{x})t_1]\!]^T \rangle^T & = \mathsf{sl}^\star \circ \mathsf{sr} \circ \langle [\![(\underline{x})t_0]\!]^T, [\![(\underline{x})t_1]\!]^T \rangle \\
[\![(\underline{x})\, \lambda x t]\!]^T & =_{\mathrm{df}} & \Lambda^T([\![(\underline{x}, x)t]\!]^T) & = \eta \circ \Lambda([\![(\underline{x}, x)t]\!]^T) \\
[\![(\underline{x})\, t\, u]\!]^T & =_{\mathrm{df}} & \mathsf{ev}^T \circ^T \langle [\![(\underline{x})t]\!]^T, [\![(\underline{x})u]\!]^T \rangle^T & = \mathsf{ev}^\star \circ \mathsf{sl}^\star \circ \mathsf{sr} \circ \langle [\![(\underline{x})t]\!]^T, [\![(\underline{x})u]\!]^T \rangle
\end{array}
$$

(The notation $(\underline{x})\, t$ denotes a term t with free variables $\underline{x}$; we have left out types; for well-typed terms, the interpretation is well-defined.) In the second column, there is the "standard" semantics in terms of the pre-(Cartesian closed) structure of $\mathbf{Kl}(T)$. In the third column the same appears spelled out in more primitive terms, after simplifications. Note, e.g., that in the semantics of *let*, sr is not needed, although the definition of $\langle -, - \rangle^T$ involves it; it gets cancelled out in the simplification.

Constructs specific to particular notions of effect must be interpreted specifically. E.g., for exceptions we can use the coproduct monad T and set

$$[\![(\underline{x})\ \mathit{raise}\ t]\!]^T =_{\mathrm{df}} \mathsf{raise}^\star \circ [\![(\underline{x})t]\!]^T$$

As $\mathbf{Kl}(T)$ is only pre-(Cartesian closed), we have soundness of typing: $\underline{x} : \underline{C} \vdash t : A$ implies $[\![(\underline{x})\, t]\!]^T : [\![\underline{C}]\!]^T \to^T [\![A]\!]^T$. But of course not all equations of lambda-calculus are validated.

In particular, we have that $\vdash t : A$ implies $[\![t]\!]^T : 1 \to^T [\![A]\!]^T$. So a closed term t of a type A denotes an element of $T[\![A]\!]^T$.

3 Comonads and context-dependent computation

We proceed to an analysis of context-dependent computation.

3.1 Comonads

Basics and first examples

We go straight to the definition and first properties of comonads. Comonads are the dual of monads. A *comonad* is a functor $D : \mathcal{C} \to \mathcal{C}$ (the *underlying functor*)

together with natural transformations $\varepsilon : D \dot{\to} \mathsf{Id}_{\mathcal{C}}$ (the *counit*), $\delta : D \dot{\to} DD$ (the *comultiplication*) satisfying

$$\begin{tikzcd} DA \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] \arrow[dr, equal] & DDA \arrow[d, "D\varepsilon_A"] \\ DDA \arrow[r, "\varepsilon_{DA}"'] & DA \end{tikzcd} \qquad \begin{tikzcd} DA \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] & DDA \arrow[d, "D\delta_A"] \\ DDA \arrow[r, "\delta_{DA}"'] & DDDA \end{tikzcd}$$

In other words, a comonad is a comonoid in $([\mathcal{C}, \mathcal{C}], \mathsf{Id}_{\mathcal{C}}, *)$.

Dually to Kleisli categories, a comonad D on a category $\mathcal{C}$ induces a category $\mathbf{CoKl}(D)$ called the *coKleisli category* of D. This is defined by:

- an object is an object of $\mathcal{C}$,
- a map of from A to B is a map of $\mathcal{C}$ from DA to B,
- $\mathsf{id}^D_A =_{\mathrm{df}} DA \xrightarrow{\varepsilon_A} A$,
- if $k : A \to^D B$, $\ell : B \to^D C$, then $\ell \circ^D k =_{\mathrm{df}} DA \xrightarrow{k^\dagger} DB \xrightarrow{\ell} C$ where $k^\dagger = DA \xrightarrow{\delta_A} DDA \xrightarrow{Dk} DB$.

From $\mathcal{C}$ there is an identity-on-objects *inclusion functor* J to $\mathbf{CoKl}(D)$, defined on maps by: if $f : A \to B$, then $Jf =_{\mathrm{df}} DA \xrightarrow{\varepsilon_A} A \xrightarrow{f} B = DA \xrightarrow{Df} DB \xrightarrow{\varepsilon_B} B$.

The functor J has a left adjoint $U : \mathbf{CoKl}(D) \to \mathcal{C}$ given by: $UA =_{\mathrm{df}} DA$ and, if $k : A \to^D B$, then $Uk =_{\mathrm{df}} DA \xrightarrow{k^\dagger} DB$.

The intuitive basis for the use of coKleisli categories as categories of impure computation should be the following. As before, we think of $\mathcal{C}$ as the category of pure functions, but D describes a notion of context. DA is the type of values of a given type A placed into a context. The category $\mathbf{CoKl}(D)$, whose maps are maps $DA \to B$ of the base category, is the category of context-dependent functions.

The function $\varepsilon_A : DA \to A$ is then the identity on A made trivially context-dependent, i.e., turned into a function discarding any given context. The function $Jf : DA \to B$ is a general pure function $f : A \to B$ regarded as trivially context-dependent in a similar fashion. The function $\delta_A : DA \to DDA$ duplicates the context of a value while $k^\dagger : DA \to DB$ is a context-dependent function $k : DA \to B$ extended into one that outputs a value of in a context (so it can be postcomposed with a context-dependent function).

Some computationally meaningful examples with $\mathcal{C}$ a Cartesian (closed) category, e.g., $\mathbf{Set}$, are the following.

The *product comonad* is defined by:

- $DA =_{\mathrm{df}} A \times E$, where E is a fixed object of $\mathcal{C}$,
- $\varepsilon_A =_{\mathrm{df}} A \times E \xrightarrow{\mathsf{fst}} A$,

- $\delta_A =_{\mathrm{df}} A \times E \xrightarrow{\langle \mathsf{id}, \mathsf{snd} \rangle} (A \times E) \times E$.

This is the dual of the exceptions monad. But its use is the same as that of the environment monad: for $TA =_{\mathrm{df}} E \Rightarrow A$ we have $\mathbf{CoKl}(D) \cong \mathbf{Kl}(T)$. Hence the product comonad can be used for dependence on an environment. The important native operation of this comonad leading to impure computations is $\mathsf{ask} =_{\mathrm{df}} 1 \times$

$E \xrightarrow{\mathsf{snd}} E$.

The *exponent comonad* is given by:

- $DA =_{\mathrm{df}} S \Rightarrow A$ where (S, e, m) is a monoid in $\mathcal{C}$,
- $\varepsilon_A =_{\mathrm{df}} (S \Rightarrow A) \xrightarrow{\mathsf{ur}^{-1}} (S \Rightarrow A) \times 1 \xrightarrow{\mathsf{id} \times e} (S \Rightarrow A) \times S \xrightarrow{\mathsf{ev}} A$,
- $\delta_A =_{\mathrm{df}} \Lambda(\Lambda(\delta'_A)) : S \Rightarrow A \to S \Rightarrow (S \Rightarrow A)$ where
  $\delta'_A =_{\mathrm{df}} ((S \Rightarrow A) \times S) \times S \xrightarrow{\mathsf{a}} (S \Rightarrow A) \times (S \times S) \xrightarrow{\mathsf{id} \times m} (S \Rightarrow A) \times S \xrightarrow{\mathsf{ev}} A$.

We come to computational uses soon, but some interesting cases are, e.g., $(S, e, m) =_{\mathrm{df}} (\mathsf{Nat}, 0, +)$ and $(S, e, m) =_{\mathrm{df}} (\mathsf{Nat}, 0, \max)$.

The *costate comonad* is given by:

- $DA =_{\mathrm{df}} (S \Rightarrow A) \times S$ where S is an object of $\mathcal{C}$,

- $\varepsilon_A =_{\mathrm{df}} (S \Rightarrow A) \times S \xrightarrow{\mathsf{ev}} A$,
- $\delta_A =_{\mathrm{df}} (S \Rightarrow A) \times S \xrightarrow{\mathsf{coev} \times \mathsf{id}} (S \Rightarrow ((S \Rightarrow A) \times S)) \times S$.

This comonad arises from the composition in the appropriate order of the adjoint functors $S \times - \dashv S \Rightarrow -$. Composition the other way around gives rise to the state monad T defined by $TA = S \Rightarrow (A \times S)$. Again we defer the discussion of the computational utility.

The *cofree comonad* on an endofunctor H on $\mathcal{C}$ is given by $DA =_{\mathrm{df}} \nu X. A \times HX$ (i.e., the carrier of the final $A \times H(-)$-coalgebra, which is the cofree H-coalgebra on A). The functor $DA =_{\mathrm{df}} \mu X. A \times HX$ (i.e., the carrier of the initial $A \times H(-)$-algebra) is also a comonad, the *cofree recursive comonad* [33] (dual to the free completely iterative monad [2]). The set DA is the set of nonwellfounded resp. wellfounded A-labelled H-branching trees. Think, e.g., of the case $HX =_{\mathrm{df}} 1 + X \times X$, leading to binary branching with a termination option. The counit $\varepsilon_A : DA \to A$ extracts the root label of a given tree (so the root label is the focus value in a tree and the rest of the tree is its "context"). The comultiplication $\delta_A : DA \to DDA$ replaces the label of every node with the subtree rooted by that node (thus equipping every node label with a local copy of its context).

Comonads for dataflow computation

Next we look at dataflow computation (stream-based computation). We are interested in notions of computation where impure functions from A to B are general, causal or anticausal functions from $\mathsf{Str}A$ to $\mathsf{Str}B$ where $\mathsf{Str}A =_{\mathrm{df}} \nu X.A \times X$ is the set of streams with elements from A. The physical intuition here is that of discrete-time signal transformers; streams represent histories of signals. Causality (corresponding to what is physically feasible with signals) means that the present value of the output signal can only depend on the present and past values of the input signal. Anticausality means dependence on the present and future alone.

Streams are in natural bijection with functions from natural numbers: $\mathsf{Str}A \cong \mathsf{Nat} \Rightarrow A$. Hence, general stream functions $\mathsf{Str}A \to \mathsf{Str}B$ (as used in dataflow languages like Lucid [3]) are in natural bijection with maps $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \to B$, i.e., with coKleisli maps of the costate comonad $DA =_{\mathrm{df}} (S \Rightarrow A) \times S$ where $S =_{\mathrm{df}} \mathsf{Nat}$. Moreover, the identities and composition of general stream functions

and the coKleisli identities and composition agree. So this particular instantiation of the costate comonad describes the notion of context used in general dataflow computation. We call it the *streams with a position comonad*, since it pairs an input stream of a stream function with a chosen position of interest in the output stream. The important operations supported this comonad are fby and next. The fby ('followed by') operation corresponds to initialized unit delay of the input signal, while next operation corresponds to unit anticipation. In both cases, the stream is shifted but the designated position stays the same.

For clarity, here is the concrete description for the case $\mathcal{C} = \mathbf{Set}$:

$$DA =_{\mathrm{df}} (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat}$$

$$\begin{array}{rcrcl}
\varepsilon_A & : & (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} & \longrightarrow & A \\
 & & (f, n) & \longmapsto & f\, n \\
\delta_A & : & (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} & \longrightarrow & (\mathsf{Nat} \Rightarrow ((\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat})) \times \mathsf{Nat} \\
 & & (f, n) & \longmapsto & (\lambda m.(f, m), n) \\
\mathsf{fby}_A & : & A \times ((\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat}) & \longrightarrow & A \\
 & & (a_{00}, (f, 0)) & \longmapsto & a_{00} \\
 & & (a_{00}, (f, n+1)) & \longmapsto & f\, n \\
\mathsf{next}_A & : & (\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} & \longrightarrow & A \\
 & & (f, n) & \longmapsto & f(n+1)
\end{array}$$

Further, a position in a stream splits it into two parts, a list of elements before the position (the past of the signal) and a stream of all remaining elements (the present and future): $(\mathsf{Nat} \Rightarrow A) \times \mathsf{Nat} \cong \mathsf{List} A \times \mathsf{Str} A$. From here it is not a long way to see that the causal stream functions (where the present value of the output signal can depend on the present and past values of the input signal, but not on the future; programs in the Lustre [13] and Lucid Synchrone [29] dataflow languages denote causal stream functions) correspond precisely to the coKleisli maps of the comonad $DA =_{\mathrm{df}} \mathsf{List} A \times A \cong \mathsf{NEList} A \cong \mu X. A \times (1 + X)$ (this is the cofree recursive monad on H defined by $HX =_{\mathrm{df}} 1 + X$, we call it the *nonempty list comonad*). And the anticausal ones (where the present of the output signal may only depend on the present and future values of the input signal) correspond to the coKleisli maps of the comonad $DA =_{\mathrm{df}} \mathsf{Str} A = \nu X. A \times X$ (the cofree comonad on $\mathsf{Id}_{\mathcal{C}}$, also equivalent to the exponent comonad $DA =_{\mathrm{df}} S \Rightarrow A$ with $(S, e, m) =_{\mathrm{df}} (\mathsf{Nat}, 0, +)$, we call it the *stream comonad*). Again, in each case the identities and composition of stream functions agree with those of the coKleisli category. Of the operations discussed above, the first comonad supports only unit delay fby, while the second one only supports unit anticipation next.

The concrete description of the nonempty list comonad is this:

$$DA =_{\mathrm{df}} \mathsf{NEList}\, A$$

$$
\begin{array}{rccc}
\varepsilon_A : & \mathsf{NEList}\,A & \longrightarrow & A \\
& (a_0, \ldots, a_{n-1}, a_n) & \longmapsto & a_n \\
\delta_A : & \mathsf{NEList}\,A & \longrightarrow & \mathsf{NEList}\,(\mathsf{NEList}\,A) \\
& (a_0, \ldots, a_{n-1}, a_n) & \longmapsto & ((a_0), \ldots, (a_0, \ldots, a_{n-1}), (a_0, \ldots, a_{n-1}, a_n)) \\
\mathsf{fby}_A : & A \times \mathsf{NEList}\,A & \longrightarrow & A \\
& (a_{00}, (a_0)) & \longmapsto & a_{00} \\
& (a_{00}, (a_0, \ldots, a_n, a_{n+1})) & \longmapsto & a_n
\end{array}
$$

The comonad for anticausal dataflow computation is concretely defined by:

$$
DA =_{\mathrm{df}} \mathsf{Str} A
$$

$$
\begin{array}{rccc}
\varepsilon_A : & \mathsf{Str} A & \longrightarrow & A \\
& (a_n, a_{n+1}, \ldots) & \longmapsto & a_n \\
\delta_A : & \mathsf{Str} A & \longrightarrow & \mathsf{Str}(\mathsf{Str} A) \\
& (a_n, a_{n+1}, \ldots) & \longmapsto & ((a_n, a_{n+1}, \ldots), (a_{n+1}, \ldots), \ldots) \\
\mathsf{next}_A : & \mathsf{Str} A & \longrightarrow & A \\
& (a_n, a_{n+1}, \ldots) & \longmapsto & a_{n+1}
\end{array}
$$

Comonads for tree transformations

A similar example is given by relabelling tree transformations, often specified with attribute grammars [21]. Let $H : \mathcal{C} \to \mathcal{C}$. We are interested in relabelling tree functions $\mathsf{Tree}A \to \mathsf{Tree}B$ where $\mathsf{Tree}A =_{\mathrm{df}} \mu X.A \times H X$ is the type of wellfounded A-labelled H-branching trees. (Relabellings are equally plausible for nonwellfounded trees, but we concentrate on the wellfounded case.) At any node of interest in the output tree the label is determined by the label at the same node in the input tree plus maybe some more nodes. In the case of general relabellings, dependence on the labels in all of the remaining tree is allowed. In bottom-up relabellings, only labels at and below the position of interest may influence the result. The idea is again to mark the node of interest.

The comonad for general relabellings (corresponding to general attribute grammars with both synthesized and inherited attributes) is the *trees with a position comonad*, the comonad structure on the zipper datatype of Huet [16]. This is defined by $DA =_{\mathrm{df}} \mathsf{Tree}'A \times A \cong \mathsf{Path}A \times \mathsf{Tree}A$ where $\mathsf{Path}A =_{\mathrm{df}} \mathsf{List}(A \times H'(\mathsf{Tree}A)) \cong \mu X.1 + A \times H'(\mathsf{Tree}A) \times X$. Here F' denotes the derivative of a functor (container) F; intuitively it is the type of one-hole versions of the container F [23,1]. An element of the type $\mathsf{Tree}'A \times A$ consists of an A-labelled tree with the label of one node omitted, just to mark this node, along with the omitted label attached separately. An element of the type $\mathsf{Path}A \times \mathsf{Tree}A$ is an A-labelled tree, split into the path from a node of interest up to the root, together with all side subtrees, and the subtree rooted by this node of interest.

As an example of how the path type is computed for a branching factor H, for $HX =_{\mathrm{df}} 1 + X \times X$ we have $H'X = 2 \times X$, so $\mathsf{Path}A \cong \mathsf{List}(A \times 2 \times \mathsf{Tree}A))$. An element in this type is a list of triples (the label of my parent, I am the left or right child, the side subtree rooted by my sibling) for every node from the focus node up until the root node (excluded) in a tree to relabel.

The comonad for bottom-up tree relabellings (corresponding to purely synthesized attribute grammars) is the *tree comonad*, defined by $DA =_{\mathrm{df}} \mathsf{Tree} A$ (the cofree recursive comonad on H). An element here represents a subtree of a global tree to be relabelled rooted by a node of interest.

(Notice that bottom-up tree relabellings generalize anticausal stream functions.)

The important operations of the comonad are for navigation in the tree: up to the parent of a given node (this is possible in the case of general tree relabellings) and down to the children (possible in both cases).

From streams and trees to containers

It is worth noticing that the tree with a position comonad is in fact a coproduct of costate comonads and the tree comonad is a coproduct of exponent comonads, just as the stream with a position comonad is a costate comonad and the stream comonad an exponent comonad.

The observation is that trees (just as streams) are a special case of *containers* [1], i.e., set functors $FA =_{\mathrm{df}} \coprod_{s\in S}(P_s \Rightarrow A)$ where S is a set (of *shapes*) and P an assignment of sets (of *positions*) to shapes.

Shape-preserving functions $FA \to FB$ are thus families of maps $(P_s \Rightarrow A \to P_s \Rightarrow B)_{s\in S}$, in other words, maps $\coprod_{s\in S}((P_s \Rightarrow A) \times P_s) \to B$. The functor $DA =_{\mathrm{df}} \coprod_{s\in S}((P_s \Rightarrow A) \times P_s) \cong F'A \times A$ carries the comonad for the general relabellings.

To speak of abstract bottom-upness, we must confine ourselves to containers with some structure, namely: (1) for any shape $s \in S$ and position $p \in P_s$, a shape $s \downarrow p \in S$ (for the shape of the subcontainer below position p in containers of shape s), (2) for any shape $s \in S$, a position $0_s \in P_s$ (the root position), (3) for any shape $s \in S$ and positions $p \in P_s$ and $p' \in P_{s\downarrow p}$, a position $p \cdot p' \in P_s$ (the position p' in the subcontainer as one in the global container), such that $s \downarrow 0_s = s$, $s \downarrow (p \cdot p') = (s \downarrow p) \downarrow p'$, $p \cdot 0_{s\downarrow p} = p$, $0_s \cdot p = p$, and $(p \cdot p') \cdot p'' = p \cdot (p' \cdot p'')$ — a form of dependent monoid on the family P.

The comonad for bottom-up relabellings is then $DA =_{\mathrm{df}} \coprod_{s\in S}(P_s \Rightarrow A) \cong FA$.

3.2 Symmetric (semi)monoidal comonads

If $\mathcal{C}$ is Cartesian, then the coKleisli category $\mathbf{CoKl}(D)$ of a comonad D on $\mathcal{C}$ is straightforwardly Cartesian, as $J : \mathcal{C} \to \mathbf{CoKl}(D)$ is a right adjoint and preserves limits.

Explicitly, this structure is given by:

$$
\begin{array}{rlrl}
1^D & =_{\mathrm{df}} 1 & A_0 \times^D A_1 & =_{\mathrm{df}} A_0 \times A_1 \\
 & & \mathsf{fst}^D & =_{\mathrm{df}} \mathsf{fst} \circ \varepsilon = \varepsilon \circ D\mathsf{fst} \\
 & & \mathsf{snd}^D & =_{\mathrm{df}} \mathsf{snd} \circ \varepsilon = \varepsilon \circ D\mathsf{snd} \\
!^D & =_{\mathrm{df}} \,! & \langle k_0, k_1 \rangle^D & =_{\mathrm{df}} \langle k_0, k_1 \rangle
\end{array}
$$

But finding a structure to mimic the exponents of $\mathcal{C}$, if $\mathcal{C}$ is Cartesian closed, is not so easy. Given that exponents should be internal homsets, it is natural to choose

$$A \Rightarrow^D B =_{\mathrm{df}} DA \Rightarrow B$$

It is also unproblematic to match this up with the definition

$$\mathsf{ev}^D_{A,B} =_{\mathrm{df}} D((DA \Rightarrow B) \times A) \xrightarrow{\mathsf{s}} D(DA \Rightarrow B) \times DA \xrightarrow{\varepsilon \times \mathsf{id}} (DA \Rightarrow B) \times DA \xrightarrow{\mathsf{ev}} B$$

where $\mathsf{s}_{A,B} =_{\mathrm{df}} \langle D\mathsf{fst}, D\mathsf{snd} \rangle : D(A \times B) \to DA \times DB$.

But given $k : C \times^D A \to^D B$, i.e., $D(C \times A) \to B$, how should we define $\Lambda^D(k) : DC \to DA \Rightarrow B$, i.e., $C \to^D A \Rightarrow^D B$? It only makes sense to set $\Lambda^D(k) =_{\mathrm{df}} \Lambda(k')$ where $k' = DC \times DA \xrightarrow{?} D(C \times A) \xrightarrow{k} B$.

Using a strength of D (if available), we could use one of the maps

$$DC \times DA \xrightarrow{\varepsilon \times \mathsf{id}} C \times DA \xrightarrow{\mathsf{sl}} D(C \times A)$$
$$DC \times DA \xrightarrow{\mathsf{id} \times \varepsilon} DC \times A \xrightarrow{\mathsf{sr}} D(C \times A)$$

but this gives a solution where the order of two arguments of a binary function is important and the context of the value of one of the arguments is discarded.

The answer lies in symmetric monoidal and semimonoidal comonads. We review the definitions.

A *strong [lax] symmetric monoidal functor* between symmetric monoidal categories $(\mathcal{C}, I, \otimes)$ and $(\mathcal{C}', I', \otimes')$ is a functor on $F : \mathcal{C} \to \mathcal{C}'$ together with an isomorphism [map] $\mathsf{e} : I' \to FI$ and a natural isomorphism [transformation] with components $\mathsf{m}_{A,B} : FA \otimes' FB \to F(A \otimes B)$ satisfying

$$
\begin{tikzcd}
FA \otimes' I' \arrow[r, "\mathsf{id} \otimes' \mathsf{e}'"] \arrow[d, "\mathsf{ur}'_{FA}"'] & FA \otimes' FI \arrow[r, "\mathsf{m}_{A,I}"] & F(A \otimes I) \arrow[d, "F\mathsf{ur}_A"] \\
FA \arrow[rr, equal] & & FA
\end{tikzcd}
\qquad
\begin{tikzcd}
FA \otimes' FB \arrow[r, "\mathsf{m}_{A,B}"] \arrow[d, "\mathsf{c}'_{FA,FB}"'] & F(A \otimes B) \arrow[d, "F\mathsf{c}_{A,B}"] \\
FB \otimes' FA \arrow[r, "\mathsf{m}_{B,A}"'] & F(B \otimes A)
\end{tikzcd}
$$

$$
\begin{tikzcd}
(FA \otimes' FB) \otimes' FC \arrow[r, "\mathsf{m}_{A,B} \otimes \mathsf{id}"] \arrow[d, "\mathsf{a}'_{FA,FB,FC}"'] & F(A \otimes B) \otimes' FC \arrow[r, "\mathsf{m}_{A \otimes B,C}"] & F((A \otimes B) \otimes C) \arrow[d, "F\mathsf{a}_{A,B,C}"] \\
FA \otimes' (FB \otimes' FC) \arrow[r, "\mathsf{id} \otimes \mathsf{m}_{B,C}"'] & FA \otimes' F(B \otimes C) \arrow[r, "\mathsf{m}_{A,B \otimes C}"'] & F(A \otimes (B \otimes C))
\end{tikzcd}
$$

A *symmetric monoidal natural transformation* between two (strong or lax) symmetric monoidal functors $(F, \mathsf{e}^F, \mathsf{m}^F)$, $(G, \mathsf{e}^G, \mathsf{m}^G)$ is a natural transformation $\tau :$

$F \dot{\to} G$ satisfying

$$
I' \xrightarrow{\mathsf{e}^F} FI \qquad FA \otimes' FB \xrightarrow{\mathsf{m}^F_{A,B}} F(A \otimes B)
$$

$$\begin{tikzcd} {} \arrow[d, equal] & {} \arrow[d, "\tau_I"] \\ I' \arrow[r, "\mathsf{e}^G"'] & GI \end{tikzcd} \qquad \begin{tikzcd} {} \arrow[d, "\tau_A \otimes' \tau_B"'] & {} \arrow[d, "\tau_{A\otimes B}"] \\ GA \otimes' GB \arrow[r, "\mathsf{m}^G_{A,B}"'] & G(A \otimes B) \end{tikzcd}$$

A *strong [lax] symmetric monoidal comonad* on a symmetric monoidal category $(\mathcal{C}, I, \otimes)$ is a comonad (D, ε, δ) where the underlying functor D is a strong [lax] symmetric monoidal functor (with preservation of I, $\otimes$ witnessed by e, m) and the counit and comultiplication ε, δ are symmetric monoidal natural transformations. The latter means that we have

$$\begin{tikzcd} I \arrow[r, "\mathsf{e}"] \arrow[d, equal] & DI \arrow[d, "\varepsilon_I"] \\ I \arrow[r, equal] & I \end{tikzcd} \qquad \begin{tikzcd} DA \otimes DB \arrow[r, "\mathsf{m}_{A,B}"] \arrow[d, "\varepsilon_A \otimes \varepsilon_B"'] & D(A \otimes B) \arrow[d, "\varepsilon_{A\otimes B}"] \\ A \otimes B \arrow[r, equal] & A \otimes B \end{tikzcd}$$

$$\begin{tikzcd} I \arrow[rr, "\mathsf{e}"] \arrow[d, equal] & & DI \arrow[d, "\delta_I"] \\ I \arrow[r, "\mathsf{e}"'] & DI \arrow[r, "D\mathsf{e}"'] & DDI \end{tikzcd} \qquad \begin{tikzcd} DA \otimes DB \arrow[rr, "\mathsf{m}_{A,B}"] \arrow[d, "\delta_A \otimes \delta_B"'] & & D(A \otimes B) \arrow[d, "\delta_{A\otimes B}"] \\ DDA \otimes DDB \arrow[r, "\mathsf{m}_{DA,DB}"'] & D(DA \otimes DB) \arrow[r, "D\mathsf{m}_{A,B}"'] & DD(A \otimes B) \end{tikzcd}$$

(Note that Id is canonically symmetric monoidal and that F, G being symmetric monoidal make GF canonically symmetric monoidal, so the definition is meaningful. Note also that the conditions on e are identical to those of an Eilenberg-Moore coalgebra structure on 1.)

Fairly often, as we will see shortly, the full structure of a symmetric monoidal comonad is not achievable, but not necessary either. We speak of a symmetric *semimonoidal* category, if the unit I is not present (or exists but is not important for us), and of a symmetric *semimonoidal* functor/comonad, if the unit preservation witness e is not present. We will later (in Sec. 3.4) show that that this imperfection can be avoided by switching to "typed" comonads on presheaf categories.

Let us revisit our examples. The product comonad, given by $DA =_{\mathrm{df}} A \times E$ with E an object, is lax symmetric monoidal (semimonoidal) as soon as E carries some commutative monoid (semigroup) structure $e : 1 \to E$, $m : E \times E \to E$. Indeed, we can then choose $\mathsf{e} =_{\mathrm{df}} 1 \xrightarrow{\langle \mathsf{id}, e \rangle} 1 \times E$, $\mathsf{m}_{A,B} =_{\mathrm{df}} (A \times E) \times (B \times E) \longrightarrow (A \times B) \times (E \times E) \xrightarrow{\langle \mathsf{id}, m \rangle} (A \times B) \times E$.

The exponent comonad, given by $D =_{\mathrm{df}} S \Rightarrow A$ with S carrying a monoid structure, is strong symmetric monoidal as witnessed by the isomorphism $\mathsf{e} =_{\mathrm{df}} \Lambda(1 \times S \xrightarrow{!} 1) : 1 \cong S \Rightarrow 1$ and natural isomorphism $\mathsf{m}_{A,B} =_{\mathrm{df}} \Lambda(((S \Rightarrow A) \times (S \Rightarrow B)) \times S \xrightarrow{\langle \mathsf{ev} \circ (\mathsf{fst} \times \mathsf{id}), \mathsf{ev} \circ (\mathsf{snd} \times \mathsf{id}) \rangle} A \times B) : (S \Rightarrow A) \times (S \Rightarrow B) \cong S \Rightarrow (A \times B)$.

The cofree comonad and cofree recursive comonad on any polynomial functor $HX \cong 1 + H'X$ are lax symmetric semimonoidal: we can choose m to "zip" two trees together, truncating wherever the branchings at a pair of corresponding nodes disagree. In the case of nonempty lists ($H'X =_{\mathrm{df}} X$), this is exactly the customary truncating zipping operation of nonempty lists. Alternatively, a single-node tree can be returned for trees of different shapes, pairing just the values at the roots.

Given a symmetric (semi)monoidal comonad D on a Cartesian closed category $\mathcal{C}$, we can define: if $k : C \times^D A \to^D B$, then $\Lambda^D(k) =_{\mathrm{df}} \Lambda(k')$ where $k' = DC \times DA \xrightarrow{\mathsf{m}_{C,A}} D(C \times A) \xrightarrow{k} B$.

How good are the pre-exponents obtained as imitations of exponents?

If D is strong symmetric (semi)monoidal, then $A \Rightarrow^D -$ is right adjoint to $- \times^D A$:

$$\frac{\dfrac{\dfrac{\dfrac{C \times^D A \to^D B}{D(C \times A) \to B}}{DC \times DA \to B}}{DC \to DA \Rightarrow B}}{C \to^D A \Rightarrow^D B}$$

Hence $\Rightarrow^D$ is a true exponent functor and $\mathbf{CoKl}(D)$ is Cartesian closed just as $\mathcal{C}$.

If D is only lax symmetric (semi)monoidal however, then the binary operation $\Rightarrow^D$ extends to a functor, but it has few properties of the exponent functor.

An intermediate case arises when (e and) m satisfy

$$
\begin{tikzcd}
DA \arrow[r, equal] \arrow[d, "!_{DA}"'] & DA \arrow[d, "D!_A"] \\
1 \arrow[r, "\mathsf{e}"'] & D1
\end{tikzcd}
\qquad
\begin{tikzcd}
DA \arrow[r, equal] \arrow[d, "\Delta_{DA}"'] & DA \arrow[d, "D\Delta_A"] \\
DA \times DA \arrow[r, "\mathsf{m}_{A,A}"'] & D(A \times A)
\end{tikzcd}
$$

where $\Delta_A =_{\mathrm{df}} \langle \mathsf{id}, \mathsf{id} \rangle : A \to A \times A$. These conditions are automatic, if D is strong symmetric (semi)monoidal, but not in the lax case. When met, they yield $\mathsf{e} \circ !_{D1} = \mathsf{id}_{D1}$ and $\mathsf{m}_{A,B} \circ \mathsf{s}_{A,B} = \mathsf{id}_{D(A \times B)}$. As a consequence, $\Rightarrow^D$ becomes a weak exponent operation on objects, i.e., we get $\mathsf{ev}^D \circ^D (\Lambda(k) \times^D \mathsf{id}^D) = k$.

We note that $(!_A, \Delta_A)$ (corresponding to the structural rules of weakening and contraction) give a uniform comonoid structure on all objects A of $\mathcal{C}$. Also, the map and natural transformation $(!_{D1}, \mathsf{s})$ witness that D is an *oplax* symmetric monoidal comonad that also respects the uniform comonoid structure $(!, \Delta)$.

3.3 Semantics of context-dependent languages

We are ready to define a general coKleisli semantics of context-dependent languages. We interpret lambda-calculus into the coKleisli category $\mathbf{CoKl}(D)$ of a symmetric (semi)monoidal comonad D on a given Cartesian closed category $\mathcal{C}$ of pure computations. We do this, as if $\mathbf{CoKl}(D)$ was Cartesian closed, even if it may be merely

Cartesian "preclosed" in one of the senses discussed above. We get:

$$
\begin{array}{rcll}
[\![K]\!]^D & =_{\mathrm{df}} & \text{an object of } \mathbf{CoKl}(D) & = \text{that object of } \mathcal{C} \\
[\![1]\!]^D & =_{\mathrm{df}} & 1^D & = 1 \\
[\![A \times B]\!]^D & =_{\mathrm{df}} & [\![A]\!]^D \times^D [\![B]\!]^D & = [\![A]\!]^D \times [\![B]\!]^D \\
[\![A \Rightarrow B]\!]^D & =_{\mathrm{df}} & [\![A]\!]^D \Rightarrow^D [\![B]\!]^D & = D[\![A]\!]^D \Rightarrow [\![B]\!]^D \\
[\![\underline{C}]\!]^D & =_{\mathrm{df}} & [\![C_0]\!]^D \times^D \ldots \times^D [\![C_{n-1}]\!]^D & = [\![C_0]\!]^D \times \ldots \times [\![C_{n-1}]\!]^D \\
[\![(\underline{x})\, x_i]\!]^D & =_{\mathrm{df}} & \pi_i^D & = \pi_i \circ \varepsilon \\
[\![(\underline{x})\, let\ x \leftarrow t\ in\ u]\!]^D & =_{\mathrm{df}} & [\![(\underline{x}, x)\, u]\!]^D \circ^D \langle \mathsf{id}^D, [\![(\underline{x})\, t]\!]^D \rangle^D & = [\![(\underline{x}, x)\, u]\!]^D \circ \langle \varepsilon, [\![(\underline{x})\, t]\!]^D \rangle^\dagger \\
[\![(\underline{x})\, ()]\!]^D & =_{\mathrm{df}} & !^D & = \,!
\end{array}
$$

$$
\begin{aligned}
[\![(\underline{x})\,\mathit{fst}\ t]\!]^D &=_{\mathrm{df}} \mathsf{fst}^D \circ^D [\![(\underline{x})\,t]\!]^D && = \mathsf{fst} \circ [\![(\underline{x})\,t]\!]^D \\
[\![(\underline{x})\,\mathit{snd}\ t]\!]^D &=_{\mathrm{df}} \mathsf{snd}^D \circ^D [\![(\underline{x})\,t]\!]^D && = \mathsf{snd} \circ [\![(\underline{x})\,t]\!]^D \\
[\![(\underline{x})\,(t_0, t_1)]\!]^D &=_{\mathrm{df}} \langle [\![(\underline{x})\,t_0]\!]^D, [\![(\underline{x})\,t_1]\!]^D \rangle^D && = \langle [\![(\underline{x})\,t_0]\!]^D, [\![(\underline{x})\,t_1]\!]^D \rangle \\
[\![(\underline{x})\,\lambda x t]\!]^D &=_{\mathrm{df}} \Lambda^D([\![(\underline{x}, x)\,t]\!]^D) && = \Lambda([\![(\underline{x}, x)\,t]\!]^D \circ \mathsf{m}) \\
[\![(\underline{x})\,t\,u]\!]^D &=_{\mathrm{df}} \mathsf{ev}^D \circ^D \langle [\![(\underline{x})\,t]\!]^D, [\![(\underline{x})\,u]\!]^D \rangle^D && = \mathsf{ev} \circ \langle [\![(\underline{x})\,t]\!]^D, ([\![(\underline{x})\,u]\!]^D)^\dagger \rangle
\end{aligned}
$$

Any construct specific to a particular notion of context receives a specific interpretation. E.g., for the *ask* construct of a language for computing with an environment we can use the product comonad and define:

$$[\![(\underline{x})\,\mathit{ask}]\!]^D =_{\mathrm{df}} \mathsf{ask} \circ D!$$

And for the constructs of a general/causal/anticausal dataflow language we can use the appropriate comonad and define:

$$
\begin{aligned}
[\![(\underline{x})\,t_0\ \mathit{fby}\ t_1]\!]^D &=_{\mathrm{df}} \mathsf{fby} \circ \langle [\![(\underline{x})\,t_0]\!]^D, ([\![(\underline{x})\,t_1)]\!]^D)^\dagger \rangle \\
[\![(\underline{x})\,\mathit{next}\ t]\!]^D &=_{\mathrm{df}} \mathsf{next} \circ ([\![(\underline{x})\,t]\!]^D)^\dagger
\end{aligned}
$$

Again, we have soundness of typing, in the form $\underline{x} : \underline{C} \vdash t : A$ implies $[\![(\underline{x})t]\!]^D : [\![\underline{C}]\!]^D \rightarrow^D [\![A]\!]^D$, but not all equations of the lambda-calculus are validated.

For a closed term $\vdash t : A$, soundness of typing says that $[\![t]\!]^D : 1 \to^D [\![A]\!]^D$, i.e., $D1 \to [\![A]\!]^D$, so closed terms are evaluated relative to a contextuated value of the unit type.

In case of general or causal stream functions, an element of $D1$ is a list over 1, i.e., a natural number, for the time elapsed from the beginning of the history at a moment of interest. Of course it identifies a stream position.

If D is strong or lax symmetric monoidal (not just semimonoidal), we have a canonical choice $\mathsf{e} : 1 \to D1$. This happens, for example, in the case of the stream comonad for anticausal dataflow computation. This is adequate as all closed terms in an anticausal language must denote constant streams, with the same value at every position. Indeed, there is no way to identify a position with an anticausal computation on no input.

In what sense is this semantics correct? We could compare the generic coKleisli semantics to some other generic semantics, e.g., an operational semantics, if we had one available. Unfortunately this is not the case: generic operational semantics for context-dependent languages is future work for us.

But we can compare the coKleisli semantics of specific languages to their standard denotational semantics. Here we can observe the following. Standard dataflow languages (Lucid, Lustre/Lucid Synchrone) are first-order and here the coKleisli and standard (stream-function) semantics agree fully. How to combine dataflow constructs and higher-orderness has been unclear; various designs have been proposed, e.g., Colaço et al.'s design with two flavors of function spaces [10]. The coKleisli semantics offers a neat design motivated by mathematical considerations, namely imitation of closed structure.

3.4 Precise comonads for dataflow computation and tree transformations

Several of notions of context that we looked at do not correspond to strong symmetric monoidal comonads. Rather, they correspond to lax symmetric semimonoidal comonads, for the reason that m should morally be partial and the total version fails to be an isomorphism and rules out the existence of a cohering e. Here indexing in the form of use of comonads on presheaf categories can help.

Consider the case of causal dataflow. Instead of the lax symmetric monoidal comonad on **Set** we could work with a more refined strong symmetric monoidal comonad on $[\mathbb{N}, \mathbf{Set}]$, where $\mathbb{N}$ is the set of natural numbers (seen as a discrete category).

We define:

$$(DA)_n =_{\mathrm{df}} \prod_{j=0}^{n} A_j$$

$$\begin{array}{rrcl}
(\varepsilon_A)_n : & (DA)_n & \to & A_n \\
& (a_0, \ldots, a_n) & \mapsto & a_n \\
(\delta_A)_n : & (DA)_n & \to & \prod_{j=0}^{n} (DA)_j \\
& (a_0, \ldots, a_n) & \mapsto & ((a_0), \ldots, (a_0, \ldots, a_n))
\end{array}$$

The fby operation can be defined for those A which extend to a functor $\omega \to \mathbf{Set}$ where ω is the poset of natural numbers: we need to be able to delay stream elements. Constant sets are typical examples.

$$\begin{array}{rrcl}
(\mathsf{fby}_A)_n : & A_0 \times (DA)_n & \to & A_n \\
& (a_{00}, (a_0)) & \mapsto & a_{00}
\end{array}$$

$$(a_{00}, (a_0, \ldots, a_n, a_{n+1})) \mapsto A_{n \to n+1}\, a_n$$

(By $m \to n$ we denote the map that is there, if $m \leq n$.) All of the above are same definitions as for the causal dataflow comonad defined before, except that we are "typed" by the positions in the single shape that streams can have.

This comonad is unproblematically strong symmetric monoidal in the right way, with e and m defined by

$$
\begin{array}{rccc}
\mathsf{e}_n : & 1 & \longrightarrow & (D1)_n \\
& () & \longmapsto & \underbrace{((), \ldots, ())}_{n+1 \text{ times}} \\
(\mathsf{m}_{A,B})_n : & (DA)_n \times (DA)_n & \longrightarrow & (DA)_n \\
& ((a_0, \ldots, a_n), (a'_0, \ldots, a'_n)) & \longmapsto & ((a_0, a'_0), \ldots, (a_n, a'_n))
\end{array}
$$

A closed term in a causal dataflow language now denotes a natural transformation $D1 \to A$ where $1 \cong D1$. The nth component is thus an element of A_n, i.e., the term denotes an element of A_n for any position n. Of course typical base types would be constant: $K_n =_{\mathrm{df}} K$.

A similar treatment is possible for general dataflow computation and for bottom-up and general tree relabelling. In the case of trees, the indexing is by a pair of a shape and position.

4 Discussion

Brookes, Geva and Van Stone's computational comonads

Brookes and Geva's [8] original example of comonadic computation was intensional semantics. In its simplest form it is this: As the base category we use the category $\omega\mathbf{Cpo}$ of ω-cpos and ω-continuous functions. The ω-cpo DA is given by ω-chains of elements of an ω-cpo A, partially ordered pointwise. The counit computes the limit of an ω-chain. The multiplication sends an ω-chain to the ω-chain of its prefixes (seen as ω-chains by repeating the last element).

Notably, this is very similar in spirit to the comonad D on $\mathbf{Set}$ given by nonempty lists, for causal dataflow, except that nonempty lists are finite sequences and they do not have to be chains wrt. some partial order.

The data and laws of computational comonads were slightly different from those of symmetric (semi)monoidal comonads. Most notably, instead of $\mathsf{e} : 1 \to D1$, they had a natural transformation η with components $\eta_A : A \to DA$, required to form a uniform Eilenberg-Moore coalgebra structure on all objects A. The natural transformations m and s ("merge" and "split") were governed by laws similar, but not fully identical or equivalent to those of a lax and oplax symmetric semimonoidal comonad.

Coproducts and recursion

As left adjoints preserve colimits, the Kleisli category of any monad on a coCartesian category $\mathcal{C}$ inherits the coproducts of $\mathcal{C}$. The coKleisli category of a comonad on a coCartesian category is generally not coCartesian (dually to the case of Kleisli categories and Cartesian structure).

Approximating general recursion (a uniform parameterized fixpoint operation, equivalently a uniform trace operation [14]) of a Cartesian base category in a Kleisli category is a subtle issue that has received considerable attention [12,26,6]. This

is an interesting (and non-dual!) problem also in the case of coKleisli categories. Some initial work has been done by N. Frisby in the FP community.

The coKleisli category of the cofree recursive comonad on a functor H, defined by $DA =_{\mathrm{df}} \mu X. A \times H X$, always has a partial uniform parameterized fixpoint operation, alternatively a partial uniform trace operation, implementing guarded recursion. This can also be reformulated in terms of recursive coalgebras, dualizing Milius's completely iterative algebras [24]. We recall that cofree recursive comonads describe the context-dependence manifested in bottom-up tree relabellings.

Combining effects and context-dependence

It is feasible that a notion of computation combines both effectfulness and context-dependence. Such combinations can correspond to distributive laws of a comonad over a monad in which case the category of impure functions is the bi-

Kleisli category of the distributive law. This design appeared already in the work of Brookes and Van Stone [9]. We have applied it to clocked causal dataflow computation, combining causal dataflow and exceptions [34]. Power and Watanabe [32] have given a definitive account of the mathematics of distributive laws between monads and comonads.

Lawvere theories and arrows/Freyd categories

Lawvere theories [27] and arrows/Freyd categories [17,30] [4] are finer and coarser approaches to effectful computation. Lawvere theories make the effectful operations of a notion of effect explicit. Arrows/Freyd categories, generalizing strong monads/their Kleisli categories, were proposed as an axiomatization of notions of impure computation reaching beyond Moggi-style effects.

Similar treatments of context-dependent computation should also be possible; this is an avenue for investigation.

In the direction of Lawvere theories, we would like to proceed from Power and Shkaravska's [31] analysis of the array comonad (here called the costate comonad) as one generated by a Lawvere cotheory.

Concerning the other direction, we note that arrows/Freyd categories as such are not a very interesting analysis of context-dependent computation, since any comonad/coKleisli category gives trivially an arrow/Freyd category. While Freyd categories are there to axiomatize the aspects of the Cartesian structure of the base category that must survive in the category of impure computations, for a coKleisli category we know that it is properly Cartesian. The interesting issue is to give a useful axiomatization of the additionally desirable symmetric "preclosed" structure. Freyd categories were not devised for this. Rather, we are after a suitable weakening of the notions of a symmetric closed category à la Eilenberg and Kelly (closed structure without monoidal structure) and of the adjunction in symmetric monoidal closed structure [11]. We have begun studying this matter with J. Adàmek and J. Velebil.

[4] Jacobs et al. [15,19] have developed a thorough account of the interrelationship of arrows and Freyd categories.

5 Conclusions

We have demonstrated that a number of notions of context-dependent computation admit an analysis using symmetric (semi)monoidal comonads, leading to a systematic semantics of corresponding languages, analogous in spirit to Moggi's account of effects in terms of strong monads. Our analysis is not distant from that of Brookes et al., but we have added important new types of examples (notions of dataflow computation, tree labelling) and streamlined the specific data and laws for obtaining an approximation of Cartesian closed structure guided by category-theoretic criteria of canonicity. The resulting picture is quite elegant, especially with the view that the important examples where the basic comonad is only lax symmetric semimonoidal can be reworked into examples of strong symmetric monoidal comonads using appropriate indexing.

Recursion (both general recursion and guarded recursion) and finer and coarser alternatives to comonads analogous to Lawvere theories and arrows are important special topics that we plan to address elsewhere. Likewise we defer to future research the study of computational uses of comonad resolutions other than the coKleisli resolution and generic operational semantics of context-dependent languages.

Acknowledgements

We are grateful to Thorsten Altenkirch, Bart Jacobs, Paul Levy, Conor McBride, Marino Miculan for useful discussions.

This work was partially supported by the Estonian Science Foundation grant no. 6940 and by the EU FP6 IST coordination action TYPES.

References

[1] Abbott, M., T. Altenkirch, and N. Ghani, *Containers: constructing strictly positive types*, Theor. Comput. Sci. **342**(1) (2005), pp. 3–27.

[2] Aczel, P., J. Adàmek, S. Milius, and J. Velebil, *Infinite trees and completely iterative theories: a coalgebraic view*, Theor. Comput. Sci. **300**(1–3) (2003), pp. 1–45.

[3] Ashcroft, E. A. and W. W. Wadge, "LUCID, the Dataflow Programming Language," Academic Press, 1985.

[4] Benton, N., G. Bierman, V. de Paiva, and M. Hyland, *Linear lambda-calculus and categorical models revisited*, in: E. Börger et al., eds, "Proc. of 6th Wksh. on Computer Science Logic, CSL '92," Lect. Notes in Comput. Sci. **702**, Springer, 1993, pp. 61–84.

[5] Benton, N., J. Hughes, and E. Moggi, *Monads and effects*, in: G. Barthe, P. Dybjer, L. Pinto, J. Saraiva, eds., "Advanced Lectures from Int. Summer School on Applied Semantics, APPSEM 2000," Lect. Notes in Comput. Sci. **2395**, Springer, 2002, pp. 42–122.

[6] Benton, N. and M. Hyland, *Traced premonoidal categories*, Theor. Inform. and Appl. **37**(4) (2003), pp. 273–299.

[7] Bierman, G. and V. de Paiva, *On an intuitionistic modal logic*, Studia Logica **65**(3) (2000), pp. 383–416.

[8] Brookes, S. and S. Geva, *Computational comonads and intensional semantics*, in: M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., "Applications of Categories in Computer Science," London Math. Society Lecture Note Series **177**, Cambridge Univ. Press, 1992, pp. 1–44.

[9] Brookes, S. and K. Van Stone, "Monads and comonads in intensional semantics," Techn. Report CMU-CS-93-140, School of Comput. Sci., Carnegie Mellon Univ., 1993, 41 pp.

[10] Colaço, J.-L., A. Girault, G. Hamon, and M. Pouzet, *Towards a higher-order synchronous data-flow language*, in: "Proc. of 4th ACM Int. Conf. on Embedded Software, EMSOFT'04," ACM Press, 2004, pp. 230–239.

[11] Eilenberg, S. and G. M. Kelly, *Closed categories*, in: "Proc. of Conf. on Categorical Algebra (La Jolla, 1965)," Springer, 1966, pp. 421–562.

[12] Erkök, L. and J. Launchbury, *Recursive monadic bindings*, in: "Proc. of 5th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '00," ACM Press, 2000, pp. 174–185.

[13] Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud, *The synchronous data flow programming language LUSTRE*, Proc. of IEEE **79**(9) (1991), pp. 1305–1320.

[14] Hasegawa, M, *The uniformity principle on traced monoidal categories*, in: R. Blute, P. Selinger, eds., "Proc. of 9th Conf. on Category Theory and Computer Science, CTCS '02," Electron. Notes in Comput. Sci. **69**, Elsevier, 2003, pp. 137–155.

[15] Heunen, C. and B. Jacobs, *Arrows, like monads, are monoids*, in: S. Brookes, M. Mislove, eds., "Proc. of 22nd Ann. Conf. on Mathematical Foundations of Programming Semantics, MFPS XXII," Electron. Notes in Theor. Comput. Sci. **158**, Elsevier, 2006, pp. 219–236.

[16] Huet, G., *The zipper*, J. of Funct. Program. **7**(5) (1997), pp. 549–554.

[17] Hughes, J., *Generalising monads to arrows*, Sci. of Comput. Program. **37**(1–3) (2000), pp. 67–111.

[18] Hyland, M., G. Plotkin, and J. Power, *Combining effects: sum and tensor*, Theor. Comput. Sci. **357**(1–3) (2006), pp. 70–99.

[19] Jacobs, B. and I. Hasuo, *Freyd is Kleisli, for arrows*, in: C. McBride, T. Uustalu, eds., "Proc. of Wksh. on Mathematically Structured Functional Programming, MSFP 2006," Electron. Wkshs. in Computing, BCS, 2006, 13 pp.

[20] Kieburtz, R. B., *Codata and comonads in Haskell*, unpublished manuscript, 1999.

[21] Knuth, D., *Semantics of context-free languages*, Math. Syst. Theory **2**(2) (1968), pp. 127–145. Corrigendum, ibid., **51**(1) (1971), pp. 95–96.

[22] Lewis, J. R., M. B. Shields, E. Meijer, and J. Launchbury, *Implicit parameters: dynamic scoping with static types*, in: "Proc. of 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '00," ACM Press, 2000, pp. 108–118.

[23] McBride, C., *The derivative of a regular type is the type of its one-hole contexts*, manuscript, 2000.

[24] Milius, S., *Completely iterative algebras and completely iterative monads*, Inform. and Comput. **196**(1) (2005), pp. 1–41.

[25] Moggi, E., *Notions of computation and monads*, Inform. and Comput. **93**(1) (1991), pp. 55–92.

[26] Moggi, E. and A. Sabry, *An abstract monadic semantics for value recursion*, Theor. Inform. and Appl. **38**(4) (2004), pp. 375–400.

[27] Plotkin, G. and J. Power, *Semantics for algebraic operations*, in: S. Brookes, M. Mislove, eds., "Proc. of 17th Conf. on Mathematical Foundations of Programming Semantics, MFPS-XVII," Electron. Notes in Theor. Comput. Sci. **45**, Elsevier, 2001, pp. 332–345.

[28] Plotkin, G. and J. Power, *Computational effects and operations: an overview*, in: M. H. Escardó, A. Jung, eds., "Proc. of Wksh. on Domains VI," Electron. Notes in Theor. Comput. Sci. **73**, Elsevier, 2004, pp. 149–163.

[29] Pouzet, M., *Lucid Synchrone, version 3: tutorial and reference manual*, unpublished manuscript, 2006.

[30] Power, J. and E. Robinson, *Premonoidal categories and notions of computation*, Math. Structures in Comput. Sci. **7**(5) (1997), pp. 453–468.

[31] Power, J. and O. Shkaravska, *From comodels to coalgebras: state and arrays*, in: J. Adámek, S. Milius, eds., "Proc. of Wksh. on Coalgebraic Methods in Comput. Sci., CMCS 2004," Electron. Notes in Theor. Comput. Sci. **106**, Elsevier, 2004, pp. 297–314.

[32] Power, J. and H. Watanabe, *Combining a monad and a comonad*, Theor. Comput. Sci. **280**(1–2) (2002), pp. 137–162.

[33] Uustalu, T. and V. Vene, *The dual of substitution is redecoration*, in: K. Hammond, S. Curtis, eds., "Trends in Functional Programming 3," Intellect, 2002, pp. 99–110.

[34] Uustalu, T. and V. Vene, *The essence of dataflow programming (full version)*, in: Z. Horváth, ed., "Revised Selected Lectures from 1st Central-European Functional Programming School, CEFP 2005," Lect. Notes in Comput. Sci. **4164**, Springer, 2006, pp. 135–167.

[35] Uustalu, T. and V. Vene. *Comonadic functional attribute evaluation*, in: M. van Eekelen, ed., "Trends in Functional Programming 6," Intellect, 2007, pp. 145–162.