Should I use a Monad or a Comonad?

Draft – Last updated June 16, 2012

Dominic Orchard
Computer Laboratory
University of Cambridge, UK
dominic.orchard@cl.cam.ac.uk

The category theoretic structures of *monads* and *comonads* can be used as an abstraction mechanism for simplifying both language semantics and programs. Monads have been used to structure impure computations, whilst comonads have been used to structure context-dependent computations. Interestingly, the class of computations structured by monads and the class of computations structured by comonads are not mutually exclusive. This paper formalises and explores the conditions under which a monad and a comonad can both structure the same notion of computation: when a comonad is *left adjoint* to a monad. Furthermore, we examine situations where a particular monad/comonad model of computation is deficient in capturing the essence of a computational pattern and provide a technique for calculating an alternative monad or comonad structure which *fully* captures the essence of the computation. Included is some discussion on how to choose between a monad or comonad structure in the case where either can be used to capture a particular notion of computation.

1 Introduction

Shall we be pure or impure? Today we will be pure- but how?

Following the seminal work of Moggi [11, 12], it is well known that the *monad* structure from category theory can be used to structure the semantics of impure notions of computation such as partiality, non-determinism, continuations, and state. Monads were further popularised in functional programming, first by Wadler [21, 22], and are now a ubiquitous programming technique. Some languages even provide specialised let-binding syntax with an overloadable monadic semantics (`do` in Haskell; `let!` in F#).

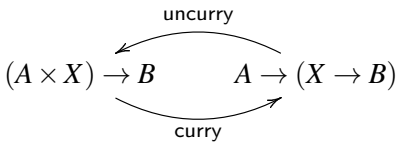
Monads are useful because they abstract over operations which define composition of computations, or functions, with structured output e.g. functions of type $X \to \mathsf{T}Y$ where T is some (parametric) structure. One of the joys of category theory is that every concept or theorem comes with an additional concept or theorem for free: the *dual*. The dual of monads are *comonads* which, instead of providing composition over structured output, provide composition over computations/functions with structured *input* e.g. for functions of type $\mathsf{D}X \to Y$ where D is some parametric structure. Comonads have been used to structure stream and dataflow computations [18], array computations [14], cellular automata [4], game semantics [6, 15], intensional semantics [3], environment passing, and more [8].

In 1995, Wadge proposed that the semantics of the dataflow language Lucid, which can be understood as an equational language for infinite streams, could be structured by a monad [19]. Ten years later, Uustalu and Vene gave a semantics for Lucid in terms of a comonad, and stated that "notions of dataflow cannot be structured with monads" [18]. There is an apparent conflict, which raises a number of questions: what does "cannot be structured" mean? Is the monadic semantics flawed in some way? Is one approach (comonadic or monadic) better than the other? What does "better" mean? And more generally: are there other notions of computations that can be structured by either a monad or a comonad, and if so how should one decide which structure to use? This paper provides answers to these questions.

Submitted to:
MSFP 2012

Parameterised computations are another notion of computation that can be structured by either a monad or a comonad. Traditionally the *exponent monad* (known as the *reader monad* in functional programming [22]) can be used to describe computations whose result depends on an additional parameter. The exponent monad has the structure $\mathsf{Exp}_X\ A = X \to A$ for some parameter type X, thus computations are of the type $A \to \mathsf{Exp}_X\ B \equiv A \to (X \to B)$. Interestingly, the *product comonad* can also be used to structure a computation with an additional parameter, where $\mathsf{Prod}_X\ A = A \times X$ for some parameter type X, thus computations are of type $\mathsf{Prod}_X\ A \to B \equiv (A \times X) \to B$, describing a computation with an extended input or extended free-variable context[1].

It is not a surprise that both the exponent monad and product comonad can be used for the same task as computations structured by either are *isomorphic* via the currying/uncurrying isomorphism:



This isomorphism between computations structured by a product comonad and those structured by an exponent monad is due to products and exponents being *adjoint functors*. In this case, the product comonad Prod_X is *left adjoint* to the exponent monad Exp_X (denoted $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$). Adjoint functors $\mathsf{L} \dashv \mathsf{R}$ have an isomorphism between morphisms $\mathsf{L}X \to Y \cong X \to \mathsf{R}Y$ of which curry/uncurry is an example.

Previously, Eilenberg and Moore studied adjoint functors where a monad T is *left adjoint* to a comonad D i.e. $\mathsf{T} \dashv \mathsf{D}$ [5]. In Section 3 we study the dual case, of comonads left adjoint to monads $\mathsf{D} \dashv \mathsf{T}$, which we show is the necessary and sufficient condition for monads and comonads to be equivalent in power (in terms of structuring computations) and in which cases there is then a choice between using a monad or a comonad. *Equivalence in power* is defined as an isomorphism between the category of computations structured by a monad (the *Kleisli* category) and the category of computations structured by a comonad (the *coKleisli category*). The equivalence is at the semantic-level; a syntactic source-to-source translation is not discussed.

For the semantics of Lucid, the monad used by Wadge and the comonad used by Uustalu and Vene are not adjoint but, as we will show in Section 4, the two approaches are equivalent, although there are some good reasons why the comonadic approach is preferable.

Whilst Section 3 studies equivalence between (co)monads where computations *exactly* fit the model of computation provided, Section 4 studies equivalence between (co)monads where computations *almost* fit the model of computation, but require some additional operations of type $\mathsf{D}A \to \mathsf{D}B$ or $\mathsf{T}A \to \mathsf{T}B$ i.e. the same (co)monad structure appears both on the input and output.

Section 5 provides a discussion on making a choice between a monad or a comonad when there is some equivalence. The laws and properties of a comonad, particularly their *shape preserving* property, and their implications to the utility of comonads, are discussed.

While monads are popular and widely used, comonads have remained somewhat underutilised despite interesting examples. It appears that in some cases monads are used when a comonad could be used equivalently or perhaps even more appropriately, and vice versa. This paper goes some way to understanding under which conditions a monad could be usurping a comonad, or vice versa.

We begin with a review of monads, comonads, and adjoint functors.

[1]The idea of using comonads to structure parameter passing is mentioned in the conclusion of the paper by Lewis et al. on implicit parameters in Haskell [10]. Others have since mentioned this use [17], and there is a product comonad library for GHC/Haskell named *Control.Comonad.Reader* [1] suggesting its use for parameter passing akin to the reader monad.

2 Review of monads, comonads, and adjoint functors

Notation $\mathscr{C}, \mathscr{D}$ are categories. $|\mathscr{C}|$ denotes the objects of a category $\mathscr{C}$. $\mathscr{C}(A,B)$ denotes the set of morphisms of $\mathscr{C}$ from A to B. Functors, or the functor of a monad/comonad, are always in sans font e.g. $\mathsf{F}, \mathsf{T}, \mathsf{D}$. A monad's functor is usually given the letter T and a comonad's functor D. To ease understanding of commutative diagrams, overline and underline are used as markers to show when a particular functor persists.

Background motivation Consider the following typing judgment for a term e of the simply-typed λ-calculus in a context Γ of free variables:

$$\Gamma : \tau \vdash e : \tau'$$

where $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$ and thus τ is the type of the whole environment (perhaps constructed with a record or product type). A *categorical semantics* of the language can be given where terms are interpreted as morphisms, mapping from a free variable context to a result value, in some category $\mathscr{C}$ [9] e.g.:

$$[\![\Gamma : \tau \vdash e : \tau']\!] : [\![\tau]\!] \to [\![\tau']\!] \in \mathscr{C}$$

where the types are interpreted as objects in the category/domain of the semantics [9].

Moggi proposed that various *impure* notions of computation can be described by a pure semantics with morphisms: $\tau \to \mathsf{T}\,\tau'$, were T is a structure which captures an impure computation of a value τ' [11]. For example, the functor $\mathsf{T}A = A + \bot$ provides a structure modelling partial computations. Moggi showed that T should be a *monad* structure (also known as a *triple*), which provides compositionality (with a unit) for such computations: i.e. if $f : A \to \mathsf{T}B$, $g : B \to \mathsf{T}C$ then $g \mathbin{\hat{\circ}} f : A \to \mathsf{T}C$.

Dually, a comonadic semantics describes programs as mappings $\mathsf{D}\,\tau \to \tau'$, where D is a comonad structure over the context/input. As with monads, the operations and laws of a comonad provide compositionality of such computations.

2.1 Monads

Monads will be familiar to Haskell programmers as defined by the type class:

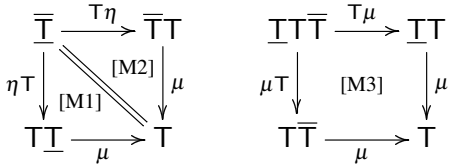
```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

where `m` is a parametrically polymorphic data type. This form of a monad is known as the *Kleisli triple* form in mathematics. The `>>=` operator (usually called *bind* in programming) is commonly called the *extension* operation in mathematics. In this paper we will predominantly use a different, but equivalent, formulation and will use a symbolic syntax for compactness.

Definition A *monad* (T, η, μ) comprises an endofunctor $\mathsf{T} : \mathscr{C} \to \mathscr{C}$ with two natural transformations:

$$\eta : 1_{\mathscr{C}} \to \mathsf{T} \qquad \mu : \mathsf{T}\mathsf{T} \to \mathsf{T}$$

$$\text{[M1]} \quad \mu \circ \eta\mathsf{T} = 1_{\mathsf{T}} \qquad \text{[M2]} \quad \mu \circ \mathsf{T}\eta = 1_{\mathsf{T}} \qquad \text{[M3]} \quad \mu \circ \mu\mathsf{T} = \mu \circ \mu\mathsf{T}$$



called *unit* and *multiplication* respectively, such that [M1-3] hold.

Example 2.1. The *exponent monad* on a category $\mathscr{C}$ comprises the endofunctor $\mathsf{Exp}_X : \mathscr{C} \to \mathscr{C}$ where:

- $\mathsf{Exp}_X A = X \to A$ for all objects $A \in |\mathscr{C}|$
- $\mathsf{Exp}_X f = \lambda e\,.\, f \circ e$ for all morphisms $f : A \to B \in \mathscr{C}$

and operations:

- $\eta a = \lambda x.\, a$ where $\eta : A \to (X \to A)$
- $\mu e = \lambda x.(e\,x)\,x$ where $\mu : (X \to (X \to A)) \to (X \to A)$

The exponent monad is often called the *reader monad* in functional programming and is commonly used for passing a parameter to a computation [7, 22].

Definition Given a monad T on a category $\mathscr{C}$, the *Kleisli category* $\mathscr{C}_\mathsf{T}$, has objects $|\mathscr{C}|$ and morphisms $\mathscr{C}_\mathsf{T}(A,B) = \mathscr{C}(A, \mathsf{T}B)$ (for all objects A, B) with:

- Composition: $\hat{\circ} : \mathscr{C}_\mathsf{T}(B,C) \to \mathscr{C}_\mathsf{T}(A,B) \to \mathscr{C}_\mathsf{T}(A,C)$ defined: $g \,\hat{\circ}\, f = \mu \circ \mathsf{T}g \circ f$
- Identities: $\hat{id}_A : \mathscr{C}_\mathsf{T}(A,A)$ defined $\hat{id} = \eta$

A Kleisli category thus captures the computations of an impure language (as structured by a monad). Functions/morphisms of type $A \to \mathsf{T}B$ will be referred to as *Kleisli morphisms* of the monad T.

The *extension operator* of a monad is an operation of type: $(-)^* : \mathscr{C}(A, \mathsf{T}B) \to \mathscr{C}(\mathsf{T}A, \mathsf{T}B)$ (which corresponds to the `(>>=)` operator in Haskell shown above) where (for all $f : A \to \mathsf{T}B, g : B \to \mathsf{T}C$):

$$\text{[K1]} \quad \eta^* = id_\mathsf{T} \qquad \text{[K2]} \quad f^* \circ \eta = f \qquad \text{[K3]} \quad (g^* \circ f)^* = g^* \circ f^*$$

Definition The *Kleisli triple form* of a monad comprises an object mapping $\mathsf{T} : |\mathscr{C}| \to |\mathscr{C}|$, extension operator $(-)^*$, and η, satisfying [K1-3].

Example 2.2. In Kleisli triple form, the *exponent monad* on a category $\mathscr{C}$ has object mapping $\mathsf{Exp}_X : |\mathscr{C}| \to |\mathscr{C}|$ where $\mathsf{Exp}_X A = X \to A$ for all objects $A \in |\mathscr{C}|$, and operations:

- $\eta a = \lambda x.\, a$
- $f^* e = \lambda x\,.\,(f\,(e\,x))\,x$ for all $f : A \to \mathsf{Exp}_X B$

The Kleisli triple and standard form of a monad (in terms of μ) are equivalent by the following equalities:

$$f^* = \mu \circ \mathsf{T}f \qquad \mu = id_\mathsf{T}{}^* \qquad \mathsf{T}f = (\eta \circ f)^*$$

By the left-most equation here, composition in a Kleisli category can be redefined: $g \,\hat{\circ}\, f = g^* \circ f$.
In programming, an operation `join` modelling μ is often defined in terms of bind as above:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id
```

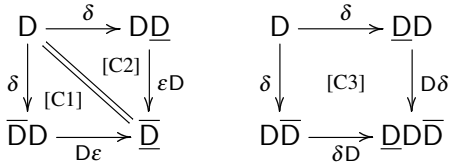
2.2 Comonads

The definitions in this section are dual to those of the previous.

Definition A *comonad* $(\mathsf{D}, \varepsilon, \delta)$ comprises an endofunctor $\mathsf{D} : \mathscr{C} \to \mathscr{C}$ with two natural transformations:

$$\varepsilon : \mathsf{D} \to 1_\mathscr{C} \qquad \delta : \mathsf{D} \to \mathsf{DD}$$

$$\text{[C1]} \quad \varepsilon\mathsf{D} \circ \delta = 1_\mathsf{D} \qquad \text{[C2]} \quad \mathsf{D}\varepsilon \circ \delta = 1_\mathsf{D} \qquad \text{[C3]} \quad \delta\mathsf{D} \circ \delta = \mathsf{D}\delta \circ \delta$$



called *counit* and *comultiplication* respectively, such that [C1-3] hold.

A potential intuition for $\mathsf{D}A$ is that it represents *context-dependent* computations i.e. the computation of a value of type A is dependent on, or parameterised by, some notion of the context of its evaluation. An intuition for ε is that it evaluates a computation in some empty or "current" context. For δ, context dependence is duplicated i.e. a context-dependent computation is turned into a context-dependent context-dependent computation where the inner "current" context is provided by the outer context.

Example 2.3. The *product comonad* on a category $\mathscr{C}$ comprises an endofunctor $\mathsf{Prod}_X : \mathscr{C} \to \mathscr{C}$ where:

- $\mathsf{Prod}_X A = A \times X$ for all objects $A \in |\mathscr{C}|$
- $\mathsf{Prod}_X f = \lambda(a,x).(f\ a,x)$ for all morphisms $f : A \to B \in \mathscr{C}$

and operations:

- $\varepsilon(a,x) = a$ where $\varepsilon : (A \times X) \to A$
- $\delta(a,x) = ((a,x),x)$ where $\delta : (A \times X) \to ((A \times X) \times X)$

As mentioned, the product comonad can be used to structure parameterised computations.

Definition Given a comonad D on a category $\mathscr{C}$, the *coKleisli category* ${}_{\mathsf{D}}\mathscr{C}$, has objects $|\mathscr{C}|$ and morphisms ${}_{\mathsf{D}}\mathscr{C}(A,B) = \mathscr{C}(\mathsf{D}A,B)$ (for all objects A,B) where:

- Composition: $\mathbin{\check{\circ}} : {}_{\mathsf{D}}\mathscr{C}(B,C) \to {}_{\mathsf{D}}\mathscr{C}(A,B) \to {}_{\mathsf{D}}\mathscr{C}(A,C)$ defined: $g \mathbin{\check{\circ}} f = g \circ \mathsf{D}f \circ \delta$
- Identities: $\check{id}_A : {}_{\mathsf{D}}\mathscr{C}(A,A)$ defined $\check{id} = \varepsilon$.

A coKleisli category thus captures the computations of a context-dependent language. Functions/morphisms of type $\mathsf{D}A \to B$ will be referred to as *coKleisli morphisms* of the comonad D.

In the same way as monads, comonads permit a *coextension operator*: $(-)^\dagger : \mathscr{C}(\mathsf{D}A,B) \to \mathscr{C}(\mathsf{D}A,\mathsf{D}B)$ where (for all $f : \mathsf{D}A \to B, g : \mathsf{D}B \to C$):

$$\text{[coK1]}\quad \varepsilon^\dagger = id_{\mathsf{D}} \qquad \text{[coK2]}\quad \varepsilon \circ f^\dagger = f \qquad \text{[coK3]}\quad (g \circ f^\dagger)^\dagger = g^\dagger \circ f^\dagger$$

An intuition for coextension is that a function from a context-dependent computation $\mathsf{D}A$ to a value B can have the context-dependence propagated to its result, returning a context-dependent computation $\mathsf{D}B$.

Definition Comonads may be presented in *coKleisli triple* form, in terms an object mapping $\mathsf{D} : |\mathscr{C}| \to |\mathscr{C}|$, the coextension operator $(-)^\dagger$, and ε, satisfying [coK1-3].

The coKleisli triple and standard form of a comonad (in terms of δ) are equivalent by the following:

$$f^\dagger = \mathsf{D}f \circ \delta \qquad \delta = id_{\mathsf{D}}{}^\dagger \qquad \mathsf{D}f = (f \circ \varepsilon)^\dagger$$

By the left-most equation, composition for a coKleisli category can be redefined: $g \mathbin{\check{\circ}} f = g \circ f^\dagger$.

Example 2.4. In coKleisli triple form, the *product comonad* on a category $\mathscr{C}$ has object mapping $\mathsf{Prod}_X : |\mathscr{C}| \to |\mathscr{C}|$ where $\mathsf{Prod}_X A = A \times X$ for all objects $A \in |\mathscr{C}|$, and operations:

- $\varepsilon(a,x) = a$
- $f^\dagger(a,x) = (f\,(a,x),x)$ for all $f : \mathsf{Prod}_X A \to B$

CoKleisli morphisms of the product comonad are reminiscent of extending a computation's context i.e. if we have morphisms $[\![\Gamma]\!] \to A$ then the product comonad extends the context: $[\![\Gamma]\!] \times X \to A$.

2.3 Adjoint Functors

There are several equivalent ways to define the adjointness relationship between two functors. Two such definitions will be of use to us: the *hom-set adjunction* and the *counit-unit adjunction*.

Definition (*Hom-set adjunction*) For categories $\mathscr{C}$ and $\mathscr{D}$, a pair of functors $\mathsf{L} : \mathscr{D} \to \mathscr{C}$ and $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ is adjoint if there exists a family of bijections:

$$\phi_{Y,X} : \hom_{\mathscr{C}}(\mathsf{L}Y, X) \cong \hom_{\mathscr{D}}(Y, \mathsf{R}X)$$

for all objects X in $\mathscr{C}$ and Y in $\mathscr{D}$, thus ϕ is a *natural isomorphism*.

L is called the *left adjoint* to R, and conversely R is the *right adjoint* to L, denoted $\mathsf{L} \dashv \mathsf{R}$.

Example 2.5. As mentioned in the introduction, $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$, where both Prod_X and Exp_X are endofunctors on some category $\mathscr{C}$, where $\mathsf{Prod}_X A = A \times X$ and $\mathsf{Exp}_X A = X \to A$, and the hom-set adjunction is provided by: $\mathsf{curry}/\mathsf{uncurry} : (\mathsf{Prod}_X A \to B) \cong (A \to \mathsf{Exp}_X B)$.

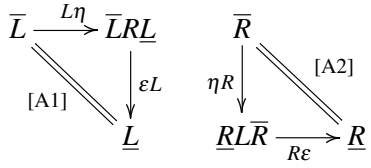
Definition (*Unit-counit adjunction*) Alternatively, a pair of functors $\mathsf{L} : \mathscr{D} \to \mathscr{C}$ and $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ is adjoint if there exists two natural transformations called the *counit* and *unit* of the adjunction respectively:

$$\varepsilon : \mathsf{LR} \to 1_{\mathscr{C}} \qquad \eta : 1_{\mathscr{D}} \to \mathsf{RL}$$

satisfying [A1-2].

[A1] $\varepsilon_{\mathsf{L}} \circ \mathsf{L}\eta = 1_{\mathsf{L}}$

[A2] $\mathsf{R}\varepsilon \circ \eta_{\mathsf{R}} = 1_{\mathsf{R}}$



Example 2.6. The unit and counit of $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$ are defined as such:

$$\eta : 1 \to \mathsf{Exp}_X \mathsf{Prod}_X \qquad \varepsilon : \mathsf{Prod}_x \mathsf{Exp}_X \to 1$$
$$\eta\ a = \lambda x.(a,x) \qquad \varepsilon\ (f,n) = fn$$

Lemma 2.1. *Given adjoint functors* $\mathsf{L} : \mathscr{D} \to \mathscr{C}$ *and* $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ *where* $\mathsf{L} \dashv \mathsf{R}$ *with counit* $\varepsilon : \mathsf{LR} \to 1_{\mathscr{C}}$ *and unit* $\eta : 1_{\mathscr{D}} \to \mathsf{RL}$ *there is an equivalent hom-set adjunction* $\phi : hom_{\mathscr{D}}(\mathsf{L}Y, X) \cong hom_{\mathscr{C}}(Y, \mathsf{R}X)$ *where:*

$$\begin{aligned} \phi\ f &= \mathsf{R}f \circ \eta \quad (\forall f : \mathsf{L}A \to B) \\ \phi^{-1}\ g &= \varepsilon \circ \mathsf{L}g \quad (\forall g : A \to \mathsf{R}B) \end{aligned} \tag{1}$$

We omit the proof of this construction for sake of brevity.

Lemma 2.2. *Every adjunction* $(\mathsf{L} \dashv \mathsf{R}, \varepsilon, \eta)$ *(where* $\mathsf{L} : \mathscr{D} \to \mathscr{C}$ *and* $\mathsf{R} : \mathscr{C} \to \mathscr{D}$*) gives rise to a monad* (T, η, μ) *in* $\mathscr{D}$ *where* $T = \mathsf{RL}$*, the unit of the monad is the unit of the adjunction* $\eta : 1 \to \mathsf{RL}$*, and* $\mu : \mathsf{RLRL} \to \mathsf{RL}$ *define* $\mu = \mathsf{R}\varepsilon_{\mathsf{L}}$ *(proof in [5]).*

Lemma 2.3. *Dually, every adjunction* $(\mathsf{L} \dashv \mathsf{R}, \varepsilon, \eta)$ *(where* $\mathsf{L} : \mathscr{D} \to \mathscr{C}$ *and* $\mathsf{R} : \mathscr{C} \to \mathscr{D}$*) gives rise to a comonad* (D, ε, δ) *in* $\mathscr{C}$ *where* $D = \mathsf{LR}$*, the counit of the comonad is counit of the adjunction* $\varepsilon : \mathsf{LR} \to 1$*, and* $\delta : \mathsf{LR} \to \mathsf{LRLR}$ *defined* $\delta = \mathsf{L}\eta_{\mathsf{R}}$ *(proof in [5]).*

Example 2.7. The adjunction $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$ induces a commonly used monad, the *state* monad, and a commonly used comonad, the *costate* comonad.

$$\mathsf{State}_X = \mathsf{Exp}_X \mathsf{Prod}_X = X \to (A \times X)$$
$$\mathsf{CoState}_X = \mathsf{Prod}_X \mathsf{Exp}_X = (X \to A) \times X$$

where the State_X monad and the $\mathsf{CoState}_X$ comonad have the unit and counit operation shown in **Example 2.6** respectively. By **Lemma 2.2** and **Lemma 2.3**, and the definitions of Kleisli/coKleisli triples, the respective extension and coextension operations are defined:

- $f^* = \mu \circ \mathsf{Exp}_X \mathsf{Prod}_X f = \mathsf{Exp}_X \varepsilon_{\mathsf{Prod}_X} \circ \mathsf{Exp}_X \mathsf{Prod}_X f = \lambda e.(\lambda x.(\lambda (e', x').(f\ e')\ x')\ (e\ x))$
- $f^\dagger = \mathsf{Prod}_X \mathsf{Exp}_X f \circ \delta = \mathsf{Prod}_x \mathsf{Exp}_\mathsf{X} f \circ \mathsf{Prod}_X \eta_{\mathsf{Exp}_X} = \lambda (e, x).(\lambda x'.(f(e, x'), x))$

The State_X monad can be used to thread mutable state, of type X, through a computation. $\mathsf{State}_X B = X \to (B \times X)$ is a map from the current state X to a result B and a new state X.

The $\mathsf{CoState}_X$ comonad can be used for modelling general context-dependence, where X represents some type of context. $\mathsf{CoState}_X A = (X \to A) \times X$ represents a map from contexts X to values A paired with a "current context" X.

2.4 Other notions

Definition The identity endofunctor $1_\mathscr{C} : \mathscr{C} \to \mathscr{C}$ maps objects and morphisms to themselves.

Definition Two categories $\mathscr{C}$ and $\mathscr{D}$ are isomorphic if there is a functor $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ and a functor $\mathsf{G} : \mathscr{D} \to \mathscr{C}$ such that $\mathsf{FG} = 1_\mathscr{D}$ and $\mathsf{GF} = 1_\mathscr{C}$. We denote isomorphic categories $\mathscr{C} \cong \mathscr{D}$.

3 Equivalent Monads and Comonads

As discussed in the introduction, parameterised computations can be structured by either the exponent monad $\mathsf{Exp}_X A = X \to A$ or the product comonad $\mathsf{Prod}_X A = A \times X$. Since $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$, morphisms $A \to \mathsf{Exp}_X B$ and $\mathsf{Prod}_X A \to B$ are isomorphic by the hom-set adjunction ϕ. Thus, the hom-set adjunction ϕ provides an isomorphism between the morphisms of the $\mathscr{C}_\mathsf{Exp}$ Kleisli category and the ${}_\mathsf{Prod}\mathscr{C}$ coKleisli category.

Here we show that, for *any* comonad D left-adjoint to a monad T ($\mathsf{D} \dashv \mathsf{T}$), the hom-set adjunction ϕ (and its inverse ϕ^{-1}) are *functorial* i.e. the isomorphism between the morphisms of ${}_\mathsf{D}\mathscr{C}$ and $\mathscr{C}_\mathsf{T}$ also preserves composition and identity in ${}_\mathsf{D}\mathscr{C}$ and $\mathscr{C}_\mathsf{T}$ i.e.

[F1]	$\phi(\check{id}_A) = \hat{id}_A$	[F1']	$\phi^{-1}(\hat{id}_A) = \check{id}_A$
[F2]	$\phi(g \mathbin{\check{\circ}} f) = \phi(g) \mathbin{\hat{\circ}} \phi(f)$	[F2']	$\phi^{-1}(g \mathbin{\hat{\circ}} f) = \phi^{-1}(g) \mathbin{\check{\circ}} \phi^{-1}(f)$

(recall $\check{id}$ and $\check{\circ}$ are the identity and composition of a coKleisli category; $\hat{id}$ and $\hat{\circ}$ the identity and composition of a Kleisli category). Since ϕ and ϕ^{-1} are also mutually inverse, they form mutual inverse functors, thus provide an isomorphism between the coKleisli and Kleisli categories ${}_\mathsf{D}\mathscr{C} \cong \mathscr{C}_\mathsf{T}$, thus the comonad D and monad T are equivalent in terms of capturing a particular notion of computation i.e. all computations and sub-computations are isomorphic.

Theorem 3.1. *Given a monad* (T, η, μ) *and a comonad* $(\mathsf{D}, \varepsilon, \delta)$*, if* $\mathsf{D} \dashv \mathsf{T}$ *then* ${}_\mathsf{D}\mathscr{C} \cong \mathscr{C}_\mathsf{T}$ *i.e. if a comonad is left adjoint to a monad then the Kleisli category of the monad and coKleisli category of the comonad are isomorphic categories.*

Proof Since $\mathsf{D} \dashv \mathsf{T}$, where $\mathsf{T} : \mathscr{C} \to \mathscr{C}$ and $\mathsf{D} : \mathscr{C} \to \mathscr{C}$, let the counit and unit adjunctions be $\alpha : \mathsf{DT} \to 1$ and $\beta : 1 \to \mathsf{TD}$ respectively, and thus via **Lemma 2.1** the hom-set adjunction ϕ for $\mathsf{D} \dashv \mathsf{T}$ is defined:

$$\phi f = \mathsf{T} f \circ \beta \qquad \phi^{-1} f = \alpha \circ \mathsf{D} f$$

Eilenberg and Moore showed that given a monad T, if T has a *right adjoint* R i.e. $\mathsf{T} \dashv \mathsf{R}$ then R is a comonad [5]. This result easily dualises, such that if D is a comonad and has a right adjoint R i.e. $\mathsf{D} \dashv \mathsf{R}$, then R is a monad (or if T is a monad and $\mathsf{L} \dashv \mathsf{T}$ then L is a comonad).

Without loss of generality[2], we construct the monad T from the comonad $(\mathsf{D}, \varepsilon, \delta)$ where $\varepsilon : \mathsf{D} \to 1$ and $\delta : \mathsf{D} \to \mathsf{DD}$. The operations of the monad for T can be constructed as such:

$$\mu : \mathsf{TT} \xrightarrow{\beta\mathsf{TT}} \mathsf{TDTT} \xrightarrow{\mathsf{T}\delta\mathsf{TT}} \mathsf{TDDTT} \xrightarrow{\mathsf{TD}\alpha\mathsf{T}} \mathsf{TDT} \xrightarrow{\mathsf{T}\alpha} \mathsf{T}$$
$$\mu = \mathsf{T}\alpha \circ \mathsf{TD}\alpha\mathsf{T} \circ \mathsf{T}\delta\mathsf{TT} \circ \beta\mathsf{TT} \tag{2}$$

$$\eta : 1 \xrightarrow{\beta} \mathsf{TD} \xrightarrow{\mathsf{T}\varepsilon} \mathsf{T}$$
$$\eta = \mathsf{T}\varepsilon \circ \beta \tag{3}$$

The monad laws [M1-3] can be proved for these constructions (omitted for brevity, see accompanying technical report [13]). Thus T is a *monad*.

From ϕ we construct the functor $\hat{\phi} : {}_{\mathsf{D}}\mathscr{C} \to \mathscr{C}_{\mathsf{T}}$ where $\hat{\phi}A = A$ on objects and $\hat{\phi}f = \phi f$ on morphisms and $\hat{\phi}^{-1} : \mathscr{C}_{\mathsf{T}} \to {}_{\mathsf{D}}\mathscr{C}$ where $\hat{\phi}^{-1}A = A$ on objects and $\hat{\phi}^{-1}f = \phi^{-1}f$ on morphisms. Trivially, $\hat{\phi}$ and $\hat{\phi}^{-1}$ are mutually inverse by the isomorphism of ϕ, i.e. $\hat{\phi}\hat{\phi}^{-1} = 1_{\mathscr{C}_{\mathsf{T}}}$ and $\hat{\phi}^{-1}\hat{\phi} = 1_{{}_{\mathsf{D}}\mathscr{C}}$.

The functor laws for $\hat{\phi}$ and $\hat{\phi}^{-1}$ ([F1-2] and [F1'-2'] above) must be proved. These laws related the operations of the monad and comonad, e.g.

$$\begin{array}{lll} \text{[F2]} & \phi(g \mathbin{\check{\circ}} f) & = \phi(g) \mathbin{\hat{\circ}} \phi(f) \\ & \mathsf{T}g \circ \mathsf{TD}f \circ \mathsf{T}\delta \circ \beta & = \mu \circ \mathsf{TT}g \circ \mathsf{T}\beta \circ \mathsf{T}f \circ \beta \end{array}$$

Since, the monad is defined in terms of the comonad via the adjunction ((2) and (3)) [F1-2] and [F1'-2'] can be proved via the laws of adjunctions, functors, natural transformations, and comonads, by unfolding the definitions of the monadic operations e.g. after unfolding ϕ, $\check{\circ}$, and $\hat{\circ}$:

$$\begin{array}{lll} \text{[F2]} & \phi(g \mathbin{\check{\circ}} f) & = \phi(g) \mathbin{\hat{\circ}} \phi(f) \\ & \mathsf{T}g \circ \mathsf{TD}f \circ \mathsf{T}\delta \circ \beta & = \mathsf{T}\alpha \circ \mathsf{TD}\alpha\mathsf{T} \circ \mathsf{T}\delta\mathsf{TT} \circ \beta\mathsf{TT} \circ \mathsf{TT}g \circ \mathsf{T}\beta \circ \mathsf{T}f \circ \beta \\ & \ldots & \end{array}$$

The proof for [F1'-2'] is dual to the proof for [F1-2] and can be proved independently, or can be proved via the isomorphism of ϕ/ϕ^{-1} and the proof for [F1-2]. The proof is omitted here, but can be found in the accompanying technical report [13].

Thus $\hat{\phi}$ and $\hat{\phi^{-1}}$ are mutually inverse functors, therefore ${}_{\mathsf{D}}\mathscr{C} \cong \mathscr{C}_{\mathsf{T}}$. $\square$.

Since $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$, by **Theorem 3.1** ${}_{\mathsf{Prod}}\mathscr{C} \cong \mathscr{C}_{\mathsf{Exp}}$, therefore the product comonad and exponent monad are equivalent in power. In Section 5 we will discuss whether there is a preference between using a monad or a comonad if they are equivalent in power.

A language with a monadic semantics and a language with a comonadic semantics could be interoperated in the case where the monad and comonad are equivalent, under the conditions we have shown here. Likewise, a program structured by a monad a program structured by an equivalent comonad could be composed in principle.

Thus we have an equivalence between (co)monads for computations which *exactly* fit within the model of computation provided i.e. for coKleisli and Kleisli morphisms.The next section studies comonads/monads where a computation *almost* fits within the model of computation provided, but the (co)monad deficient in that not all subcomputations fit the pattern. In some cases, this computation can be more succinctly and clearly structured by an alternate dual structure. The next section explores when this occurs and introduces a general technique for constructing a more appropriate structure from a less appropriate.

[2] i.e. by duality, we could start with the monad T and construct the comonad D and proceed with the proof this way.

4 Further Equivalences Between Monads and Comonads

Consider a programmer/semanticist using the product comonad for parameter passing. At some point, they realise that the task at hand requires a parameter's value to vary during a computation. Changing a parameter's value cannot be done with a coKleisli morphism $\mathsf{Prod}_X A \to B$, as the result does not have the product structure. However a morphism of type $\mathsf{Prod}_X A \to \mathsf{Prod}_X B$ may change the parameter in the returned product structure e.g. (reminder: $\mathsf{Prod}_X A = A \times X$):

$$\begin{aligned} &\mathit{change} : X \to \mathsf{Prod}_X A \to \mathsf{Prod}_X A \\ &\mathit{change}\ x' = \lambda(a,x).(a,x') \end{aligned}$$

The *change* operation can be composed with the coextension of coKleisli morphisms to change the environment during a computation, e.g. for an integer environment $X = \mathbb{Z}$ we might have:

$$f = \varepsilon \circ (\lambda(a,x).(a * x))^\dagger \circ (\mathit{change}\ 2) \circ (\lambda(a,x).a * x)^\dagger$$

where $f : \mathsf{Prod}_{\mathbb{Z}}\mathbb{Z} \to \mathbb{Z}$ first multiples the input with the parameter, then changes the environment parameter to 2, then multiplies the new parameter by the current value e.g. $f(3,4) = (3*4)*2 = 24$.

(Note that there is no analogous *change* function, that changes the parameter value for future computations, that can be defined for the exponent monad, even as a function of type $\mathsf{Exp}_X A \to \mathsf{Exp}_X B$.)

Use of the *change* function here is however unfortunate; there are no general laws governing its interaction with the operations of the comonad as it is not a coKleisli morphism, nor is it derived from one. We call such morphisms *sub-coKleisli* morphisms.

Definition A *sub-coKleisli* morphism for a comonad D has type $g : \mathsf{D}A \to \mathsf{D}B$, for some A, B, for which there does not exist a morphism $g' : \mathsf{D}A \to B$ such that $g = g'^\dagger$.

Definition A *sub-Kleisli* morphism for a monad T has type $f : \mathsf{T}A \to \mathsf{T}B$, for some A, B, for which there does not exist a morphism $f' : A \to \mathsf{T}B$ such that $f = f'^*$.

Sub-Kleisli and sub-coKleisli morphisms, with the same monad/comonad structure on "both ends", are awkward: there are no laws governing their interaction with the operations of a monad/comonad, and they usually do not fit well with syntactic extensions to languages (e.g. `do` notation in Haskell relies on subcomputations being of the form $A \to \mathsf{T}B$.).

The presence of sub-(co)Kleisli morphisms in a program or semantics may suggest that the (co)monad structure used does not provide a general enough abstraction for the computations being structured and that there may be a more appropriate monad or comonad structure that eliminates sub-(co)Kleisli morphisms, replacing them with (co)Kleisli morphisms of an alternative structure.

For the product comonad, the *change* function was added to allow *updateable* parameters, thus it is reasonable to assume that a more appropriate structure might be the *state monad* rather than the product comonad. In this case the *state monad* can be calculated from the structure of the product comonad and sub-coKleisli morphisms. Recall that $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$, thus there is the hom-set adjunction $\phi_{A,B} : \mathscr{C}(\mathsf{Prod}_X A, B) \cong \mathscr{C}(A, \mathsf{Exp}_X B)$. Applying ϕ to some sub-coKleisli morphism $f : \mathsf{Prod}_X A \to \mathsf{Prod}_X B$ gives a Kleisli morphism $\phi f : A \to \mathsf{Exp}_X \mathsf{Prod}_X A$ for the monad $\mathsf{Exp}_X \mathsf{Prod}_X$ induced by the adjunction, which is the State_X monad! (see **Example 2.7**). Thus, computations structured by the Prod_X comonad, with sub-coKleisli morphisms, can be translated to the computations structured by the State_X monad, without any sub-Kleisli morphisms.

In this section, the above result is generalised such that, given a semantics in terms of the comonad D with a number of sub-coKleisli morphisms $\mathsf{D}A \to \mathsf{D}B$, if there exists a functor F such that $\mathsf{D} \dashv \mathsf{F}$ then the induced monad FD can be used instead of the comonad D, where all the coKleisli morphisms $\mathsf{D}A \to B$ and the sub-coKleisli morphisms $\mathsf{D}A \to \mathsf{D}B$ are replaced by isomorphic Kleisli morphisms $A \to \mathsf{FD}B$. This result dualises in the obvious way.

Definition Given an endofunctor $\mathsf{F} : \mathscr{C} \to \mathscr{C}$, a *functor subcategory* for F is a *full subcategory* on $\mathscr{C}$, denoted ${}^{\mathsf{F}}\mathscr{C}$, with objects $|{}^{\mathsf{F}}\mathscr{C}| = |\mathscr{C}|$ and morphisms $hom_{{}^{\mathsf{F}}\mathscr{C}}(X,Y) = hom_{\mathscr{C}}(\mathsf{F}X, \mathsf{F}Y)$.

Theorem 4.1. *The functor subcategory ${}^{\mathsf{D}}\mathscr{C}$ for an endofunctor $\mathsf{D} : \mathscr{C} \to \mathscr{C}$ is isomorphic to a Kleisli category $\mathscr{C}_{\mathsf{T}}$, if there exists $\mathsf{R} : \mathscr{C} \to \mathscr{C}$ such that $\mathsf{D} \dashv \mathsf{R}$ where $\mathsf{T} = \mathsf{RD}$ is the monad induced by the adjunction.*

Proof Assuming an $R : \mathscr{C} \to \mathscr{C}$ such that $\mathsf{D} \dashv \mathsf{R}$, then let $\beta : 1 \to \mathsf{RD}$, $\alpha : \mathsf{DR} \to 1$ be the unit and counit of adjunction, and ϕ be the hom-set adjunction:

$$\phi_{A,B} : \mathscr{C}(\mathsf{D}A, B) \cong \mathscr{C}(A, \mathsf{R}B)$$

where $\phi f = \mathsf{R}f \circ \beta$ and $\phi^{-1} f = \alpha \circ \mathsf{D}f$.

Let $(\mathsf{T}, \eta, \mu) = (\mathsf{RD}, \beta, \mathsf{R}\alpha\mathsf{D})$ be the monad induced by the adjunction $\mathsf{D} \dashv \mathsf{R}$.

For ${}^{\mathsf{D}}\mathscr{C}$ to be isomorphic to $\mathscr{C}_T$ we require two functors $\mathsf{F} : {}^{\mathsf{D}}\mathscr{C} \to \mathscr{C}_T$ and $\mathsf{G} : \mathscr{C}_{\mathsf{T}} \to {}^{\mathsf{D}}\mathscr{C}$ where $\mathsf{FG} = 1_{\mathscr{C}_{\mathsf{T}}}$ and $\mathsf{GF} = 1_{{}^{\mathsf{D}}\mathscr{C}}$. Taking ϕ with the second component restricted to $\mathsf{D}B$ objects[3], $\theta_{A,B} = \phi_{A,\mathsf{D}B}$:

$$\theta_{A,B} : \mathscr{C}(\mathsf{D}A, \mathsf{D}B) \cong \mathscr{C}(A, \mathsf{RD}B)$$

provides an isomorphism between the morphisms of the ${}^{\mathsf{D}}\mathscr{C}$ functor subcategory, and the $\mathscr{C}_{\mathsf{RD}}$ Kleisli category. Thus F and G can be defined in terms of θ:

$$\begin{array}{ll} \mathsf{F} : {}^{\mathsf{D}}\mathscr{C} \to \mathscr{C}_{\mathsf{T}} & \mathsf{G} : \mathscr{C}_{\mathsf{T}} \to {}^{\mathsf{D}}\mathscr{C} \\ \mathsf{F}X = X & \mathsf{G}X = X \\ \mathsf{F}f = \theta f & \mathsf{G}f = \theta^{-1} f \end{array}$$

The functor laws for F and G must then be proved:

$$\begin{array}{llll} \text{[F1]} & \mathsf{F}\, id_{\mathsf{D}} = \hat{id} & \text{[F1']} & \mathsf{G}\, \hat{id} = id_{\mathsf{D}} \\ \text{[F2]} & \mathsf{F}\,(g \circ f) = (\mathsf{F}\, g) \mathbin{\hat{\circ}} (\mathsf{F}\, f) & \text{[F2']} & \mathsf{G}(g \mathbin{\hat{\circ}} f) = (\mathsf{G}g) \circ (\mathsf{G}f) \end{array}$$

$$\begin{array}{lll} \text{[F2]} & \mathsf{F}(g \circ f) & \\ = & \mathsf{R}(g \circ f) \circ \beta & \{\mathsf{F}/\theta \text{ def}\} \\ = & \mathsf{R}g \circ (\mathsf{R}f \circ \beta) & \{\text{functor}\} \\ = & \mathsf{R}g \circ \mathsf{R}(\alpha\mathsf{D} \circ \mathsf{D}\beta) \circ (\mathsf{R}f \circ \beta) & \text{[A1]} \\ = & \mathsf{R}(g \circ \overline{\alpha\mathsf{D}}) \circ \mathsf{RD}\beta \circ (\mathsf{R}f \circ \beta) & \{\text{functor}\} \\ = & \mathsf{R}(\overline{\alpha\mathsf{D}} \circ \mathsf{DR}g) \circ \mathsf{RD}\beta \circ (\mathsf{R}f \circ \beta) & (4) \\ = & \mathsf{R}\alpha\mathsf{D} \circ \mathsf{RD}(\mathsf{R}g \circ \beta) \circ (\mathsf{R}f \circ \beta) & \{\text{functor}\} \\ = & (\mathsf{R}g \circ \beta) \mathbin{\hat{\circ}} (\mathsf{R}f \circ \beta) & \{\hat{\circ} \text{ def}\} \\ = & \mathsf{F}g \mathbin{\hat{\circ}} \mathsf{F}f \quad \square & \{\mathsf{F}/\theta \text{ def}\} \end{array}$$

$$\begin{array}{lll} \text{[F1]} & \mathsf{F}id_{\mathsf{D}} & \\ = & \mathsf{R}\,id_{\mathsf{D}} \circ \beta & \{\mathsf{F}, \theta \text{ def}\} \\ = & \beta \quad \square & \{\text{functor } id\} \end{array}$$

Naturality of α:

$$\begin{array}{ccc} \mathsf{DR}(\mathsf{D}X) & \xrightarrow{\mathsf{DR}f} & \mathsf{DR}(\mathsf{D}Y) \\ {\scriptstyle \alpha(\mathsf{D}X)}\downarrow & & \downarrow{\scriptstyle \alpha(\mathsf{D}Y)} \\ \mathsf{D}X & \xrightarrow[f]{} & \mathsf{D}Y \end{array} \qquad (4)$$

[3] i.e. pre-compose the *dinatural* – natural in two components – isomorphism ϕ with D functor for the second component, which is still a dinatural isomorphism.

The proof of functoriality for G is dual, or follows from the proof for F and the isomorphism of θ, thus it is omitted for brevity. Therefore, F and G are functors, which are mutually inverse by the natural isomorphism of θ. Therefore, F and G provide the isomorphism ${}^{\mathsf{D}}\mathscr{C} \cong \mathscr{C}_{\mathsf{RD}}$. □

Lemma 4.2. *Given a comonad* D*, there exists a functor* $\mathsf{J} : {}_{\mathsf{D}}\mathscr{C} \to {}^{\mathsf{D}}\mathscr{C}$ *mapping the coKleisli category for* D *to the functor subcategory for* D*.*

Proof The construction of J is straightforward:

- $\mathsf{J}A = A$ for all objects $A \in \mathscr{C}$.
- $\mathsf{J}f = f^{\dagger}$ for all morphisms $f : \mathsf{D}A \to B \in \mathscr{C}$.

The functor laws are satisfied:

$$\begin{array}{lll} [\text{F1}] \quad \mathsf{J}\,\check{id} & = \mathsf{J}\varepsilon & \{\text{def } \check{id}\} \\ & = \varepsilon^{\dagger} & \{\text{def J}\} \\ & = id_{\mathsf{F}} & [\text{coK1}] \quad \square \end{array} \qquad \begin{array}{lll} [\text{F2}] \quad \mathsf{J}\,(g \check{\circ} f) & = (g^{\dagger} \circ f)^{\dagger} & \{\text{def J}, \check{\circ}\} \\ & = g^{\dagger} \circ f^{\dagger} & [\text{coK3}] \\ & = \mathsf{J}g \circ \mathsf{J}f & \{\text{def J}\} \quad \square \end{array}$$

Given a program/semantics comprising coKleisli morphisms $\mathsf{D}A \to B$ and sub-coKleisli morphisms $\mathsf{D}A \to \mathsf{D}B$, if we have a functor $\mathsf{R} : \mathscr{C} \to \mathscr{C}$ such that $\mathsf{D} \dashv \mathsf{R}$, then by **Theorem** (4.1) and **Lemma** (4.2) all of the coKleisli and sub-coKleisli morphisms can be mapped to Kleisli morphisms for the monad RD.

There are, of course, dual theorems and lemmas to the above:

Theorem 4.3. *The functor subcategory* ${}^{\mathsf{T}}\mathscr{C}$ *for an endofunctor* $\mathsf{T} : \mathscr{C} \to \mathscr{C}$ *is isomorphic to a coKleisli category* ${}_{\mathsf{D}}\mathscr{C}$*, if there exists* $\mathsf{L} : \mathscr{C} \to \mathscr{C}$ *such that* $\mathsf{L} \dashv \mathsf{T}$ *where* $\mathsf{D} = \mathsf{LT}$ *is the comonad induced by the adjunction.*

Lemma 4.4. *Given a monad* T*, there exists a functor* $\mathsf{J} : \mathscr{C}_{\mathsf{T}} \to {}^{\mathsf{T}}\mathscr{C}$ *mapping the Kleisli category for* T *to the functor subcategory for* T*.*

We are now ready to address the apparent conflict between Wadge's monadic semantics and Uustalu and Vene's comonadic semantics for Lucid.

Dataflow: a monad or a comonad? As discussed in the introduction, Uustalu and Vene gave a comonadic semantics for the dataflow language Lucid and stated that "notions of dataflow cannot be structured with monads" [18], however, a decade earlier Bill Wadge presented a monadic view of Lucid, arguing that monads can structure *aggregate* data such as streams hence the semantics of Lucid *can* be structured with a monad [19]. In fact, Wadge's monadic approach is isomorphic to Uustalu and Vene's comonadic approach, which we show here via **Theorem** (4.3).

The Lucid dataflow language is essentially a calculus of infinite streams with pointwise stream operations, constant streams, and a number of special stream functions, called *intensional* operators, which can compute an element of a stream from any other elements of the parameter streams (i.e. non-pointwise), the two most important of which are *next* and *fby* (followed by) [20]. Using $\langle a, b, c, \ldots \rangle$ as a notation for infinite streams, the main connectives of Lucid behave in the following way:

$$\begin{aligned} \mathit{1} &\rightsquigarrow \langle 1, 1, 1, \ldots \rangle \\ \langle x_0, x_1, x_2, \ldots \rangle \;+\; \langle y_0, y_1, y_2, \ldots \rangle &\rightsquigarrow \langle x_0 + y_0, x_1 + y_1, x_2 + y_2, \ldots \rangle \\ \mathit{next}\, \langle x_0, x_1, x_2, \ldots \rangle &\rightsquigarrow \langle x_1, x_2, \ldots \rangle \\ \langle x_0, x_1, x_2, \ldots \rangle \;\mathit{fby}\; \langle y_0, y_1, y_2, \ldots \rangle &\rightsquigarrow \langle x_0, y_0, y_1, \ldots \rangle \end{aligned}$$

In [19], Wadge's stream monad is essentially the exponent monad with natural numbers in the domain of the exponent: $\mathsf{Stream}\ A = \mathsf{Exp}_{\mathbb{N}} A = \mathbb{N} \to A$, with η and μ defined in the same way[4] as the exponent monad in **Example 2.2**. Multiplication operation takes the *diagonal* of a "doubly indexed" stream (i.e. a pair of contexts like a two-dimensional matrix), and unit defines a constant stream.

The pointwise operations in Lucid are given a semantics in terms of the non-stream function composed with η e.g. $[\![+]\!] = \eta \circ (+)$. Wadge however does not directly address how the intensional operators (*next* and *fby*) fit into the semantics. These functions return streams where the value at position n is computed from values in the parameter stream at positions other than n, thus the entire parameter stream is required in order to perform this computation. i.e. they are not Kleisli morphisms of type $A \to \mathsf{Stream}\ B$ or lifted pointwise operators, but can only be defined as sub-Kleisli morphisms $\mathsf{Stream}\, A \to \mathsf{Stream}\, B$:

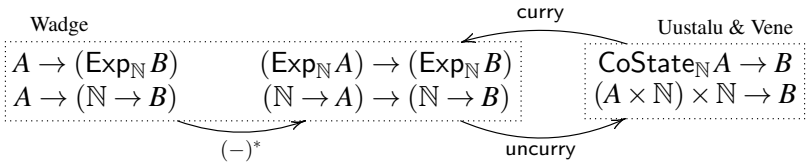
$$
\begin{aligned}
&\mathit{fby} : \mathsf{Stream}\ (A \times A) \to \mathsf{Stream}\ A \\
&\mathit{fby}\ s = \lambda n.\, \mathsf{if}\ (n = 0)\ \mathsf{then}\ (\pi_1 \circ s)\ 0\ \mathsf{else}\ (\pi_2 \circ s)\ (n-1) \\
&\mathit{next} : \mathsf{Stream}\ A \to \mathsf{Stream}\ A \\
&\mathit{next}\ s = \lambda n.\, s\ (n+1)
\end{aligned}
$$

Uustalu and Vene define the semantics of Lucid using a "streams with a position comonad" [17, 18, 16] which is the costate comonad, arising from $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$, shown in **Example 2.7**, with natural numbers for context, i.e. $\mathsf{CoState}_{\mathbb{N}} = \mathsf{Prod}_{\mathbb{N}} \mathsf{Exp}_{\mathbb{N}} = (\mathbb{N} \to A) \times \mathbb{N}$. Thus, $\mathsf{CoState}_{\mathbb{N}}$ defines a stream as a map from positions to values along with a "current" position. The ε and coextension operation $(-)^\dagger$ are defined in the same way as in **Example 2.7**. Coextension for $\mathsf{CoState}_{\mathbb{N}}$ can be illustrated (using the stream notation from above) as such:

$$
\begin{aligned}
f^\dagger\ (\langle x_0, x_1, \ldots \rangle, n) \quad = (\langle & f\ (\langle x_0, x_1, \ldots \rangle, 0), \\
& f\ (\langle x_0, x_1, \ldots \rangle, 1), \\
& \ldots \rangle,\ n)
\end{aligned}
$$

Wadge's semantics use the exponent monad $\mathsf{Exp}_{\mathbb{N}}$ which has the product functor $\mathsf{Prod}_{\mathbb{N}}$ as a left adjoint: $\mathsf{Prod}_{\mathbb{N}} \dashv \mathsf{Exp}_{\mathbb{N}}$, which we have been using throughout. Thus, by **Theorem 4.1** and **Lemma 4.2**, Wadge's monadic semantics, with sub-Kleisli morphisms, can be converted into a comonadic semantics with only coKleisli morphisms for the comonad $\mathsf{Prod}_{\mathbb{N}} \mathsf{Exp}_{\mathbb{N}}$ which is exactly the $\mathsf{CoState}_{\mathbb{N}}$ comonad used by Uustalu and Vene.

The hom-set adjunction for $\mathsf{Prod}_{\mathbb{N}} \dashv \mathsf{Exp}_{\mathbb{N}}$ is the curry/uncurry isomorphism (seen in **Example 2.5**) which provides the conversion from Wadge's semantics to Uustalu and Vene's:



Thus, the two approaches are equivalent. Wadge's approach however requires sub-Kleisli morphisms, thus Uustalu and Vene's assertion, that dataflow cannot be structured by a monad, is true if we understand *monadic structuring* as excluding sub-Kleisli morphisms. As discussed earlier, there are good reasons to avoid sub-(co)Kleisli morphisms. We discuss the trade-offs further now.

[4]Note: Wadge used the symbol $*$ for the η operation of the monad and the functor, and $\downarrow$ for the multiplication (join) operation μ. We use the standard notation here.

5 Choosing Between a Monad and a Comonad

Choice in the presence of sub-(co)Kleisli morphisms A computation solely captured by Kleisli or coKleisli morphisms benefits from equational laws for reasoning and optimisation by simplification, as well as better syntactic support in languages. Such benefits are not available for sub-(co)Kleisli morphisms. Furthermore, a computation with sub-(co)Kleisli morphisms may have code repetition, where code is repeated in order to emulate the (co)extension operation of the more appropriate structure.

By constructing an equivalent dual structure (e.g. a monad instead of a comonad) and eliminating sub-(co)Kleisli morphisms, any redundancy is captured within the new, more applicable, structure and general laws and syntactic extensions can be applied once again. Thus, if **Theorem 4.1** or **Theorem 4.3** is applicable, it is generally more useful to use the structure without sub-(co)Kleisli morphisms.

Choice given an equivalent monad and comonad In the case of a monad and a comonad that are exactly equivalent in power (see **Theorem 3.1**) the choice between structures is more subtle and subjective.

In the case of parameter passing, it is not clear whether the product comonad or exponent monad is objectively better. An argument in favour of the comonadic approach is that the read-only property of parameter passing is a consequence of the comonad laws, but in the monadic approach the read-only property must be independently proved and does not similarly come *for free*.

The read-only property of parameters in the product comonad is a consequence of the property that coextension (for any comonad) preserves the *shape* of the the parameter object.

Definition For some functor F the *shape* of a particular object $x : \mathsf{F}A$ is given by $(\mathsf{F}\,const_A)\ x$, where $const_A : A \to 1$ is unique morphism from A to the terminal object 1 of $\mathscr{C}$.

Thus in programming, if F is some data type, then $\mathsf{F}const_A$ replaces the elements of a data structure with the terminal object (in Haskell this is usually the empty tuple $()$).

Proposition 5.1. *Let $(D, \varepsilon, (-)^\dagger)$ be a coKleisli triple in $\mathscr{C}$. Coextension of any coKleisli morphism $f : \mathsf{D}A \to B$ yields a* shape preserving *morphism $f^\dagger : \mathsf{D}A \to \mathsf{D}B$ i.e. the shape of the parameter object is preserved in the result thus:*

$$\mathsf{D}\,const_B \circ f^\dagger = \mathsf{D}\,const_A \tag{5}$$

Proof By the terminal object property of $1 \in \mathscr{C}$, $const_A : A \to 1$ is a *unique* morphism mapping an object A to 1. Therefore, for all morphisms $f : A \to X$, $g : A \to Y$:

$$const_X \circ f = const_A = const_Y \circ g \tag{6}$$

by the uniqueness of morphisms to the terminal object. Thus, *shape preservation* (5.1) is proved:

$$\begin{array}{lll} & \mathsf{D}\,const \circ f^\dagger & \\ = & \mathsf{D}\,const \circ \mathsf{D}f \circ \delta & \{(-)^\dagger \text{ def}\} \\ = & \mathsf{D}(const \circ f) \circ \delta & \{\text{functor}\} \\ = & \mathsf{D}(const \circ \varepsilon) \circ \delta & (6) \\ = & \mathsf{D}\,const \circ \mathsf{D}\varepsilon \circ \delta & \{\text{functor}\} \\ = & \mathsf{D}\,const \quad \square & [\text{C2}] \end{array}$$

The crucial step in the proof is [C2] (equivalent to [coK2]), which enforces preservation of shape for the comonad. Monads on the other hand permit shape to be changed. This is a useful property to have in mind when deciding whether to use a monad or comonad. If changes in shape are required, then a monad

must be used (unless shape-changing sub-coKleisli morphisms are used). Dually, if shape preservation is required, a comonad might be the right choice as this property is intrinsic to the comonad.

In the case of the product comonad, the type of coKleisli morphisms $(A \times X) \to B$ shows that a coKleisli morphism itself cannot provide a new value for the parameter; there is no value X in the result. The shape preservation property confirms that coextension does not modify the parameter. For this reason, the product comonad might be preferred over the exponent monad for (read-only) parameter passing, although a prove can be given for the exponent monad that it too preserves the parameter.

6 Concluding Remarks and Further Work

Summary This paper showed and proved three main theorems:

1. (**Theorem 3.1**) Given a monad T and a comonad D, if $\mathsf{D} \dashv \mathsf{T}$, then ${}_{\mathsf{D}}\mathscr{C} \cong \mathscr{C}_{\mathsf{T}}$.
2. (**Theorem 4.1**) Given a comonad D, if there exists a functor R such that $\mathsf{D} \dashv \mathsf{R}$ then ${}^{\mathsf{D}}\mathscr{C} \cong \mathscr{C}_{\mathsf{RD}}$, where ${}^{\mathsf{D}}\mathscr{C}$ is the *functor subcategory* (with morphisms $\mathsf{D}A \to \mathsf{D}B$) and RD is the induced monad.
3. (**Theorem 4.3**) Dually, given a *monad* T, if there exists a functor L such that $\mathsf{L} \dashv \mathsf{T}$ then ${}^{\mathsf{T}}\mathscr{C} \cong {}_{\mathsf{Lt}}\mathscr{C}$, where ${}^{\mathsf{T}}\mathscr{C}$ is the *functor subcategory* (with morphisms $\mathsf{T}A \to \mathsf{T}B$) and LT is the induced *comonad*.

Product/exponent adjunction In a programming setting, the main (useful) example adjunction is that between the product functor $\mathsf{Prod}_X = A \times X$ and the exponent functor $\mathsf{Exp}_X = X \to A$ where $\mathsf{Prod}_X \dashv \mathsf{Exp}_X$. This adjunction was applied for each of the three theorems above, which had the following implications:

1. ${}_{\mathsf{Prod}}\mathscr{C} \cong \mathscr{C}_{\mathsf{Exp}}$, thus *parameter passing* can be equivalently structured by either the product comonad or exponent monad.
2. ${}^{\mathsf{Prod}}\mathscr{C} \cong \mathscr{C}_{\mathsf{State}}$ where $\mathsf{State}_X = \mathsf{Exp}_X\mathsf{Prod}_X$. Thus, parameter passing structured by the product comonad with the *change* $: X \to \mathsf{Prod}_X A \to \mathsf{Prod}_X A$ sub-coKleisli morphism is equivalent to using the State_X monad.
3. ${}^{\mathsf{Exp}}\mathscr{C} \cong {}_{\mathsf{CoState}}\mathscr{C}$ where $\mathsf{CoState}_X = \mathsf{Prod}_X\mathsf{Exp}_X$. Thus, Wadge's monadic semantics for Lucid in terms of the $\mathsf{Exp}_{\mathbb{N}}$ monad, with intensional operators $\mathsf{Exp}_{\mathbb{N}}A \to \mathsf{Exp}_{\mathbb{N}}B$, is equivalent to using the $\mathsf{CoState}_{\mathbb{N}}$ comonad used by Uustalu and Vene to give a comonadic semantics for Lucid.

Containers The results on structuring streams can be generalised to *containers* [2]. *Containers* have a set of shapes S, and for a particular shape $s \in S$ there is a set of positions Ps. The container functor is defined $\mathsf{F}A = \sum_{s:S}(Ps \to A)$ i.e. a coproduct of maps from a set of positions to values, for each possible shape. Containers can be both a monad and a comonad.

Mono-shape containers have just one shape s. For example, lists of length n have $S = \{n\}$ where $Pn = \mathbb{N}_{<n}$ i.e. positions are natural numbers less than n. Thus, $\mathsf{List}_n A = \mathbb{N}_{<n} \to A$.

For a mono-shape container $\mathsf{M}_s A = (Ps \to A)$, all morphisms $\mathsf{M}_s A \to \mathsf{M}_s B$ are shape-preserving, and by **Theorem 4.1** there is an equivalent comonad $(Ps \to A) \times Ps$ i.e. the costate comonad $\mathsf{CoState}_{Ps}$. Uustalu and Vene mention the costate comonad for containers in [17].

Popularity of Comonads Whilst monads are popular and widely accepted, comonads have remained somewhat underutilised and little understood. We suggest four possible reasons for the relative underuse of comonads in programming compared to monads:

1. Redundancy: some programs/computations can be structured equally well with a monad instead of a comonad.
2. Utility: the laws and operations of comonads means that there are less useful comonads for programming i.e. the structure is dual, but its utility is not.
3. Support: unlike monads, there is a lack of syntactic support in languages.
4. Sociological: monads were popularised first and are still an advanced topic for many; comonads are viewed as even more esoteric/complicated/confusing.

The first of these reasons has been the main focus of this paper in Sections 3 and 4. By the equivalences in this paper, we have shown that there are cases where a monad could be usurping a comonad, which may be a more appropriate structure than a monad in some cases.

Some attention was given to the second reason in Section 5. The *shape preservation* property of comonads is quite strong and may be one of the limiting factors in the utility of comonads; there may be fewer situations in which a computation is shape preserving, and more situations in which shape is changed throughout a computation, in which case a monad is more appropriate.

The third reason can be remedied by providing syntax for comonadic programming in languages, dualising Haskell's `do` (perhaps the `od` or `codo` notation) and F#'s `let!`. This is addressed in the author's upcoming PhD thesis in which such notation is introduced.

Acknowledgements

Thanks are due to Jeremy Gibbons, Daniel James, Alan Mycroft, and Tomas Petricek for discussions and comments on an earlier version of this paper, and to Pierre Clairambault and Andrew Pitts for category theory help and discussions. Thank you to Guillaume Munch for pointing out relevant material. Finally, my thanks to Bill Wadge for a discussion several years ago that planted the seed of this paper.

This research was kindly supported by an EPSRC Doctoral Training Award.

References

[1] `http://hackage.haskell.org/packages/archive/category-extras/0.53.5/doc/html/Control-Comonad-Reader.html`.

[2] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[3] Stephen Brookes and Kathryn V Stone. Monads and Comonads in Intensional Semantics. Technical report, Pittsburgh, PA, USA, 1993.

[4] S. Capobianco and T. Uustalu. A Categorical Outlook on Cellular Automata. *Arxiv preprint arXiv:1012.1220*, 2010.

[5] S. Eilenberg and J.C. Moore. Adjoint functors and triples. *Illinois Journal of Mathematics*, 9(3):381–398, 1965.

[6] Russ Harmer, Martin Hyland, and Paul-Andre Mellies. Categorical Combinatorics for Innocent Strategies. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 379–388, Washington, DC, USA, 2007. IEEE Computer Society.

[7] J. Hughes. Global variables in Haskell. *Journal of Functional Programming*, 14(05):489–502, 2004.

[8] Richard B. Kieburtz. Codata and Comonads in Haskell, 1999.

[9] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*. Cambridge Univ Pr, 1988.

[10] J.R. Lewis, J. Launchbury, E. Meijer, and M.B. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 118. ACM, 2000.

[11] E. Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23. IEEE, 1989.

[12] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[13] Dominic Orchard. Proofs for "Should I use a Monad or a Comonad?", 2011. Technical report. `http://www.cl.cam.ac.uk/~dao29/drafts/monad-or-comonad-techreport-orchard11.pdf`.

[14] Dominic Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: Declarative, Parallel Structured Grid Programming. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 15–24, New York, NY, USA, 2010. ACM.

[15] Nikos Tzevelekos. Nominal Game Semantics, PhD Thesis. Technical Report CS-RR-09-18, 2008.

[16] Tarmo Uustalu and Varmo Vene. Signals and comonads. *Journal of universal computer science*, 11(7):1310–1326, 2005.

[17] Tarmo Uustalu and Varmo Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008.

[18] Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, November 2006.

[19] William Wadge. Monads and Intensionality. In *International Symposium on Lucid and Intensional Programming '95*, 1995.

[20] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[21] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[22] P. Wadler. Monads for functional programming. *Advanced Functional Programming*, pages 24–52, 1995.