Coeffects: typing context-dependent computations using comonads

Tomas Petricek, Dominic Orchard, and Alan Mycroft

University of Cambridge, UK
{firstname.lastname}@cl.cam.ac.uk

Abstract. The literature describes various type systems which track how computations depend on some notion of *context*. A context-dependent computation might only execute in environments that provide certain capabilities or privileges. We refer to such properties of computations as *coeffects*, dualising the concept of effects.

We show how to track different forms of coeffect in a uniform way using a type system based on *comonads* and describe how the system follows from a comonadic semantics of the lambda calculus. Applications include tracking dynamically-scoped implicit parameters and checking execution environments that specify where a computation may execute in a distributed programming setting.

While there is only one form of monadic effect systems, there are by contrast multiple variants of comonadic coeffect systems, one of which is equivalent to monadic effect systems. As we explain, this is due to a different notion of context capture for functions.

1 Introduction

Types in functional languages have been traditionally used to classify expressions based on *what* values they can compute. However, types can be also used to classify expressions based on *how* they compute values.

The second approach has been used in *effect systems* introduced by Gifford and Lucassen [11]. Effect systems have been designed, for example, to track operations on memory locations, communication in message-passing systems [13] and atomicity in concurrent applications [10]. They are usually described as typing judgements of the form $\Gamma \vdash e : \tau!\sigma$, associating effects σ with the result.

Conversely, typing judgements of type systems that track the dependence on a context are usually of the form $\Gamma@\mathcal{C} \vdash e : \tau$, associating the properties $\mathcal{C}$ with the typing assumptions. We term such properties *coeffects*. For example, $\mathcal{C}$ may specify execution environments in which the expression e can be evaluated.

It is well known that *monads* can be used to structure impure computations [19]. Wadler and Thiemann showed that effect systems and monadic semantics can be unified, where monadic types are annotated with effect information. This paper shows the dual: unifying comonads, used for structuring context-dependent computations [32], with coeffects to give a comonadic coeffect system.

Section 2 introduces the notion of coeffects in more detail and shows a coeffect system for distributed programming. We then present our main contributions:

Type System for Coeffects. We define a type system for tracking properties of context-dependent computations (Section 3) and explain how the system follows from a categorical semantics of context-dependent computations based on *tagged comonads* (Section 4).

Variants and Applications. Differently to monadic effect systems, there are multiple variants of comonadic coeffect systems (Section 5). We show one variant for tracking dynamically scoped implicit parameters (Section 3.2) and another variant to track modality in distributed programming language (Section 5).

Comonadic and Monadic Types. Although effects and coeffects are intuitively different, we show that, under certain conditions, one variant of the comonadic coeffect system can be translated to a monadic effect system (Section 6).

2 Coeffects

Context is defined as *the circumstances that form the setting for an event, statement, or idea* [1]. The context of a computation may provide resources or capabilities to perform some action or it may restrict what the computation can do. We call the properties of such context-dependence *coeffects*. We start by showing an example of a coeffect system for tracking where an expression can be evaluated in a distributed language such as Links [7] or ML5 [20].

2.1 Example: Distributed Programming

Consider a language for distributed programming, which accesses external resources with the **access** construct. The following example function obtains data from an externally provided store and hashes it using a key entered by the user:

$$\begin{aligned}\textbf{let}\ \mathsf{hashInput} = \lambda().\ &\textbf{let}\ n = \textbf{access}\ \mathsf{store}\ \textbf{in}\\ &\textbf{let}\ k = \textbf{access}\ \mathsf{input}\ \textbf{in}\ \mathsf{hash}\ n\ k\end{aligned}$$

When executed on a phone, the function reads data from the phone memory, reads user input and calculates the hash. In a web browser, it fails at runtime because the browser cannot access local data and so the store is undefined.

Figure 1 shows typing rules that statically eliminate such errors. Typing judgements have form $\Gamma@r \vdash e : \tau$ which denote that the expression e can be evaluated in any execution environment in r yielding a value of type τ. Tags r are subsets of the set of all possible environments $\Omega = \{\mathsf{server}, \mathsf{phone}, \mathsf{browser}\}$. Resources are annotated with the environments in which they are available, thus $\mathsf{store} : \textbf{res}^{\{\mathsf{server},\mathsf{phone}\}}\mathsf{string}$ and $\mathsf{input} : \textbf{res}^{\{\mathsf{browser},\mathsf{phone}\}}\mathsf{string}$.

The *(access)* rule specifies that a resource can only be accessed in the execution environments where it is available. The *(let)* rule combines two subexpressions that can evaluate in different environments. The composed expression can only evaluate in the intersection of the environments, so the body of hashInput can evaluate only in the phone environment. The environments

$$(\text{var})\frac{x : \tau \in \Gamma}{\Gamma@\Omega \vdash x : \tau}$$

$$(\text{fun})\frac{(\Gamma, x : \tau_1)@r \vdash e : \tau_2}{\Gamma@\Omega \vdash \lambda x.e : \tau_1 \xrightarrow{r} \tau_2}$$

$$(\text{access})\frac{\Gamma@r_1 \vdash e : \mathbf{res}^{r_2}\tau}{\Gamma@r_1 \cap r_2 \vdash \mathbf{access}\, e : \tau}$$

$$(\text{app})\frac{\Gamma@r_1 \vdash e_1 : \tau_1 \xrightarrow{r_2} \tau_2 \quad \Gamma@r_3 \vdash e_2 : \tau_1}{\Gamma@r_1 \cap r_2 \cap r_3 \vdash e_1\, e_2 : \tau_2}$$

$$(\text{let})\frac{\Gamma@r_1 \vdash e_1 : \tau_1 \quad (\Gamma, x : \tau_1)@r_2 \vdash e_2 : \tau_2}{\Gamma@r_1 \cap r_2 \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau}$$

Fig. 1: Typing rules of a distributed programming language

in which a function body can evaluate are captured by a function type, thus $\mathsf{hashInput} : \mathsf{unit} \xrightarrow{r} \mathsf{string}$ where $r = \{\mathsf{phone}\}$.

The system can be developed further, including a proof of type safety, but the approach is ad-hoc, specialised to distributed programming. The next section introduces a general type system for tracking *coeffects*, based on *comonads*.

2.2 Effects and Monads, Coeffects and Comonads

For a typing judgement $\Gamma \vdash e : \tau$, the *context* or *type assumptions* Γ specifies types of free variables and τ specifies the result type. Effect systems associate the effects of expressions with the result of a typing judgement: $\Gamma \vdash e : \tau!\sigma$. For example, if e writes a reference cell r, then σ might be a set $\{\mathsf{write}(r)\}$.

Monads have been used for capturing impure computations in pure languages [19]. A semantics or a program structured by a monad has a monadic type M over the result type i.e. judgements $\Gamma \vdash e : M\tau$ and functions $\tau_1 \to M\tau_2$. An effect systems can be transposed to equivalent monadic type systems by annotating a monad with effect information: $\Gamma \vdash e : M^{\sigma}\tau$ [35].

We note that effect systems have a dual, *coeffect systems*, which associate information about context-dependence with the context of the typing judgement: $\Gamma@C \vdash e : \tau$. *Comonads*, the dual of monads, have been used for structuring context-dependent computations [32]. Dualising Wadler and Thiemann's work, we show that coeffect systems can be captured using comonads annotated with coeffects, with judgements $C^{\sigma}\Gamma \vdash e : \tau$, which represent a computation that can be evaluated only in a context σ. The function $\mathsf{hashInput}$, which can be applied only in the phone environment, could be given the type $C^{\{\mathsf{phone}\}}\mathsf{unit} \to \mathsf{string}$.

The type system for distributed programming has coeffect tags as sets of allowed execution environments, composing them using *intersection*. This is dual to effect systems that compose sets of performed effects using *union* [35]. However, this duality in the composition operation is not an inherent aspect of coeffects. The system could equally track the resources that must be available for an expression to evaluate, using *union*. To easily enable both options, we generalise tags to *monoids* $(T, 1, \otimes)$, providing a middle-ground between a concrete model (such as a sets) and a fully general model (such as a dependently-typed specification). The distributed programming example uses the monoid $(\mathcal{P}(\Omega), \Omega, \cap)$.

$$
\begin{array}{l}
v ::= x \mid !a \mid \lambda x.e \mid \lambda^{\circ}\, a.e \\
e ::= v \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{let}\ a \Leftarrow e_1\ \textbf{in}\ e_2 \mid e_1\ e_2 \mid e_1 \diamond e_2 \\
\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid C^r \tau \\
x \in Id \quad a \in CoId \qquad\qquad \Gamma = Id \to \tau \quad \Delta\ =\ CoId \to \tau
\end{array}
$$

Fig. 2: Syntax and types of the comonadic language.

3 Comonadic Coeffect Types

Monadic types give a useful template for building effect systems, but the same can be done for coeffects. Our type system based on *comonads* gives a template for building coeffect systems that track how computations depend on context.

3.1 Comonadic Language

The comonadic language in Figure 2 combines pure and comonadic versions of the lambda calculus. Although the type system could be demonstrated using just a comonadic version, languages that support monadic types like Haskell [26] and F# [30] also mix pure and monadic features. We believe that a combination is needed for practical support for comonads. This allows the user to gradually introduce comonadic typing without changing code of existing pure functions.

The pure language consists of variables x, abstraction $\lambda x.e$, application $e_1\ e_2$, let-binding and a function type $\tau_1 \to \tau_2$. The comonadic subset includes a separate syntactic category of context-carrying variables a, variable access $!a$, context-dependent lambda abstraction $\lambda^{\circ} x.e$, application $e_1 \diamond e_2$, and comonadic let-binding. The type $C^r \tau$ represents a value carrying a comonadic context.

Comonadic Sub-language. The system is based on comonads parameterised by tags r that statically specify additional information about the context. We first discuss typing rules for the comonadic sub-language shown in Figure 3 (a). A judgement $C^r \Delta; \Gamma \vdash e : \tau$ denotes that, given a free variable context Δ in a context annotated with a tag r, an expression e has a type τ. The context Γ is used for pure variables and is discussed later. The notation $C^r \Delta$ stands for a comonadic context containing a tuple with variable assignments. For variables $a_1 : \tau_1, \ldots, a_n : \tau_n$, the type of the context is $C^r(\tau_1 \times \ldots \times \tau_n)$.

Variable access *(counit)* extracts a pure value from a pure (empty) context tagged with the unit 1. Lambda abstraction *(cofun)* type-checks the body in a context $r \otimes s$, which is a combination of the context associated with the current outer scope r and the context s provided by the comonadic parameter. The context of a context-dependent application *(coapp)* combines the contexts of the two sub-expressions e_1 and e_2 with the context required by the comonadic function type $C^t \tau_1 \to \tau_2$. Finally, *(cobind)* types comonadic let-binding, combining the contexts of the two sub-expressions (which has the same type as $(\lambda^{\circ} a.e_2) \diamond e_1$).

Note that the type constructor C^r only appears as part of the context $C^r \Delta$ or immediately on the left of a function arrow $C^r \tau_1 \to \tau_2$, but never as a value.

$$(\text{cobind})\frac{C^r\Delta;\Gamma\vdash e_1:\tau_1 \quad C^s(\Delta,a:\tau_1);\Gamma\vdash e_2:\tau_2}{C^{r\otimes s}\Delta;\Gamma\vdash \textbf{let}\ a\Leftarrow e_1\ \textbf{in}\ e_2:\tau_2}$$

$$(\text{counit})\frac{a:\tau\in\Delta}{C^1\Delta;\Gamma\vdash !a:\tau}$$

$$(\text{cofun})\frac{C^{r\otimes s}(\Delta,a:\tau_1);\Gamma\vdash e:\tau_2}{C^r\Delta;\Gamma\vdash \lambda^\circ a.e:C^s\tau_1\to\tau_2}$$

$$(\text{coapp})\frac{C^r\Delta;\Gamma\vdash e_1:C^t\tau_1\to\tau_2 \quad C^s\Delta;\Gamma\vdash e_2:\tau_1}{C^{r\otimes s\otimes t}\Delta;\Gamma\vdash e_1\diamond e_2:\tau_2}$$

Fig 3. (a) Comonadic sub-langauge

$$(\text{let})\frac{C^r\Delta;\Gamma\vdash e_1:\tau_1 \quad C^r\Delta;(\Gamma,x:\tau_1)\vdash e_2:\tau_2}{C^r\Delta;\Gamma\vdash \textbf{let}\ x=e_1\ \textbf{in}\ e_2:\tau_2}$$

$$(\text{var})\frac{x:\tau\in\Gamma}{C^r\Delta;\Gamma\vdash x:\tau}$$

$$(\text{fun})\frac{C^{\hat{1}}\Delta;(\Gamma,x:\tau_1)\vdash e:\tau_2}{C^r\Delta;\Gamma\vdash \lambda x.e:\tau_1\to\tau_2}$$

$$(\text{app})\frac{C^r\Delta;\Gamma\vdash e_1:\tau_1\to\tau_2 \quad C^r\Delta;\Gamma\vdash e_2:\tau_1}{C^r\Delta;\Gamma\vdash e_1\ e_2:\tau_2}$$

Fig 3. (b) Pure sub-langauge

Full Language. Figure 3 (b) shows the typing rules for the pure sub-language. The additional context Γ maintains types of pure variables. The rules for the comonadic sub-language always propagate the pure context unmodified.

The rules *(app)* and *(let)* propagate the comonadic context, which makes it possible to use comonadic constructs inside an argument of a pure function application (i.e. it is possible to write $!a + !b$ in a scope with a unit context).

A pure function *(fun)* does not capture the comonadic context of the current scope and it cannot contain context-dependent function applications. To specify this, we introduce a special tag $\hat{1}$. Given a monoid $(T, 1, \otimes)$, a value $\hat{1} \notin T$ represents *empty context*. We extend the definition of $\otimes$ such that $\hat{1}\otimes r = r = r\otimes\hat{1}$ for any r. There is no way to obtain $\hat{1}$ as a combination of $r, s \in T$ thus pure functions cannot contain comonadic application, comonadic let-binding or counit, but they can contain comonadic function declarations, because $\hat{1}\otimes r = r$.

3.2 Example: Comonadic Implicit Parameters

Implicit parameters [16] are *dynamically scoped* and can be used to parameterise functions without explicit argument passing. As suggested by the authors, they can be modelled using comonads. To give a concrete example of our framework using implicit parameters, we extend our language as follows:

$$?p \in \mathit{DynId} \qquad e ::= \ldots \mid ?p \mid \textbf{dyn}\ ?p = e_1\ \textbf{in}\ e_2$$

$$\frac{C^r\Delta;\Gamma\vdash e_1:\tau_1 \quad C^{\{?p:\tau_1\}\cup r}\Delta;\Gamma\vdash e_2:\tau_2}{C^r\Delta;\Gamma\vdash \textbf{dyn}\ ?p=e_1\ \textbf{in}\ e_2:\tau_2} \qquad \frac{}{C^{r\cup\{?p:\tau\}\cup s}\Delta;\Gamma\vdash ?p:\tau}$$

The expression $?p$ reads a parameter and ($\textbf{dyn}\ ?p = e_1\ \textbf{in}\ e_2$) sets $?p$ to the value of e_1 and evaluates e_2 in a context containing $?p$. To track which parameters are in scope, we use a monoid $(\mathcal{P}(\mathit{DynId}\times\mathit{Type}), \emptyset, \cup)$. An interesting case is declaration of a nested comonadic function. The following function format does some pre-processing and then returns another comonadic function:

```
let format = λ°str . let lines ⇐ formatLines !str ?width in
                        (λ°rest . append !lines !rest ?width ?size)
let fmt = (dyn ?width = 5 in format ◇ "Hello")  in
dyn ?size = 10 in fmt ◇ "world"
```

The outer function requires a context $\{?\mathsf{width} : \mathsf{int}\}$, because it accesses ?width. The ?width parameter is captured by the inner, nested function, so calling the nested function on the last line only requires ?size, but not ?width. This semantics is captured by the type $C^{\{?\mathsf{width}:\mathsf{int}\}}\mathsf{string} \to C^{\{?\mathsf{size}:\mathsf{int}\}}\mathsf{string} \to \mathsf{string}$.

Implicit parameters are related to the *reader* monad [12]. Our system is more general because comonadic lambda abstraction captures the implicit parameters of a declaration scope, whilst monadic lambda abstraction does not. Interestingly, our system can be restricted to model the reader monad (Section 5).

4 Semantic Motivation

Comonads, the dual structure to a monads, have numerous applications in functional programming [33, 23, 14]. Uustalu and Vene model context-dependent computations with *symmetric semi-monoidal* comonads [32].

In categorical semantics, monads and comonads provide composition for computations with extra structure on their result or on their parameters, respectively. Given the judgement $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$, the comonadic semantics of e are modelled as a function $C(\tau_1 \times \ldots \times \tau_n) \to \tau$, where C is a comonad structure over the free-variable typing assumptions. We define the notion of a *strong monoidal tagged comonad*, which gives rise to the comonadic coeffect system.

4.1 Tagged Monads and Comonads

We use computations modelled as functions $C^r\tau_1 \to \tau_2$ where the comonad C is annotated with tags $r, s, \ldots \in T$ of a monoid $(T, 1, \otimes)$. Monadic functions can be tagged similarly, generalising sets [35]. The type signatures for the composition and *counit* (*unit*) operations of *tagged comonads* (*monads*) are:

$$\begin{array}{lll} \mathit{compose} & : & (C^r\tau_1 \to \tau_2) \ \to (C^s\tau_2 \to \tau_3) \ \to (C^{r\otimes s}\tau_1 \to \tau_3) \\ \mathit{counit} & : & C^1\tau \to \tau \end{array}$$

$$\begin{array}{lll} \mathit{compose} & : & (\tau_1 \to M^r\tau_2) \to (\tau_2 \to M^s\tau_3) \to (\tau_1 \to M^{r\otimes s}\tau_3) \\ \mathit{unit} & : & \tau \to M^1\tau \end{array}$$

The usual monad and comonad laws [19, 32] continue to hold for a tagged version, because $\otimes$ is associative and *counit* (*unit*) is tagged with the unit of the monoid 1. In the categorical semantics, the input of $C^r\tau_1 \to \tau_2$ represents the free-variable context. This means that *counit* is a computation that obtains values of variables without requiring any special property of the context.

Composition can be defined using the following *bind* and *cobind* operations:

$$\begin{array}{lll} \mathit{bind} & : & M^r\tau_1 \quad \to (\tau_1 \to M^s\tau_2) \to M^{r\otimes s}\tau_2 \\ \mathit{cobind} & : & C^{r\otimes s}\tau_1 \to (C^r\tau_1 \to \tau_2) \ \to C^s\tau_2 \end{array}$$

The result of *bind* combines effects of the first parameter with the effects resulting from the computation returned by the second parameter. The *cobind* operations evaluates a context-dependent computation (the second parameter) and wraps the result in a new context. The coeffects of the input context (first parameter) are a combination of coeffects of the context-dependent computation (second parameter) and the newly created context (result). The type signatures highlight that coeffects propagate *backwards*, whereas effects propagate *forwards*.

All the tagged operations here are actually *families* of operations, indexed by the tags, since a monoid does not necessarily have a normal form for its objects.

Additional Structure. Tagged comonads provide composition of functions $C^r\tau_1 \to \tau_2$, but a full semantics for the simply-typed lambda calculus requires extra structure to interpret merging of contexts in lambda abstraction. *Monoidal tagged comonads* provide this extra structure with the following additional operation:

$$\mathit{combine} \ : \ C^r\tau_1 \times C^s\tau_2 \to C^{r\otimes s}(\tau_1 \times \tau_2)$$

In the semantics of lambda abstraction, the *combine* operation combines the coeffects of a context in which the function is constructed and the coeffects of a context given to the function by the caller. The laws of *combine* [32] specify consistent behaviour with respect to other operations. Alternate tagging of the operation can be used to restrict the comonadic type system. In Section 5, we consider two variants obtained by tagging one of the arguments of *combine* with the unit 1. One such variant gives a system equivalent to monadic effect systems.

Let-binding and application require a typing context to be duplicated. This is modelled using a *split* operation, which splits coeffects between the two contexts. The operation is provided by a *strong monoidal tagged comonad*, which requires that *combine* is an isomorphism (with an inverse that we call *split*):

$$\mathit{split} \ : \ C^{r\otimes s}(\tau_1 \times \tau_2) \to C^r\tau_1 \times C^s\tau_2$$

Monoidal comonads also provide a unit operation for *combine*. This operation is not required thus we need only a strong *semi*-monoidal tagged comonad.

$$\begin{array}{rll} [\![(\overline{a})\ \lambda^\circ a.e]\!] & \Omega = \lambda a \to [\![(\overline{a}, a)\ e]\!]\ (\mathit{combine}\ (\Omega, a)) & \textit{(fun)} \\ [\![(\overline{a})\ \textbf{let}\ a \Leftarrow e_1\ \textbf{in}\ e_2]\!] & \Omega = [\![(\overline{a}, a)\ e_2]\!]\ (\mathit{cobind}\ \Omega\ (\lambda\Omega' \to & \textit{(let)} \\ & \qquad \textbf{let}\ \Omega_1, \Omega_2 = \mathit{split}\ \Omega'\ \textbf{in} & \\ & \qquad (\mathit{counit}\ \Omega_1,\ [\![(\overline{a})\ e_1]\!]\ \Omega_2))) & \\ [\![(\overline{a})\ e_1 \diamond e_2]\!] & \Omega = \textbf{let}\ \Omega_1, \Omega_2 = \mathit{split}\ \Omega\ \textbf{in} & \textit{(app)} \\ & \quad ([\![(\overline{a})\ e_1]\!]\ \Omega_1)\ (\mathit{cobind}\ \Omega_2\ (\lambda\Omega_3 \to [\![(\overline{e_2})\ a]\!]\ \Omega_3)) & \\ [\![(\overline{a})\ !a_i]\!] & \Omega = \pi_i\ (\mathit{counit}\ \Omega) & \textit{(var)} \end{array}$$

Fig. 4: Comonadic semantics using (strong semi-monoidal) tagged comonads.

4.2 Categorical Semantics

The semantics of the comonadic part of the language is shown in Figure 4. It differs from the model of Uustalu and Vene [32] in two ways. Firstly, our operations *cobind*, *counit* and *combine* are *tagged*. Secondly, we use *split* to duplicate the context when passing it to multiple sub-expressions. Due to space restrictions, the model omits pure contexts and constructs of the pure sub-language. These can be found in an extended semantics in Appendix A [25].

This simplified model maintains a single variable context representing the variables in $C^r\Delta$. Assume a term e has a type τ, comonadic free variables $a_1 : \tau_1, \ldots, a_n : \tau_n$ and expects a comonadic context with coeffect r. The semantic interpretation of the term is a function of type $C^r(\tau_1 \times \ldots \times \tau_n) \to \tau$. We write Ω for values of the first parameter. An extended semantics (Appendix A) includes an additional parameter Γ for values of the pure variables.

Lambda abstraction *(app)* uses *combine* to merge the outer context with the context carried by the function argument. The *(let)* rule uses *split* to duplicate the context in order to extract the original context using *counit* and evaluate the expression e_1 (the semantics is equivalent to the semantics of $(\lambda^\circ a\,.\,e_2) \diamond e_1$). Similarly, *(app)* uses *split* to obtain contexts for two sub-expressions.

Our type system is a direct result of the categorical semantics. To obtain the typing rules, we determine the types of individual occurrences of $[\![(\overline{a})\ e]\!]$ in each equation and then use the types (and tags) to define typing rules.

5 Variants of the Comonadic Coeffect System

The key difference between comonadic coeffect and monadic effect systems is the treatment of free-variable contexts in lambda abstraction, because the comonadic context carries extra structure. A function $C^r\tau_1 \to \tau_2$ defined in context $C^s\Delta$ is type-checked in a context that has a coeffect tag $r \otimes s$. In contrast, lambda abstraction in monadic effect systems is a pure computation with no effects. The following listing compares the two rules side-by-side:

$$\frac{C^{r\otimes s}(\Delta, a : \tau_1) \vdash e : \tau_2}{C^r\Delta \vdash \lambda^\circ a.e : C^s\tau_1 \to \tau_2} \qquad \frac{\Delta, a : \tau_1 \vdash e : M^r\tau_2}{\Delta \vdash \lambda^\circ a.e : M^1(\tau_1 \to M^r\tau_2)}$$

The rule for comonads is determined by the *combine* operation, which is used to merge the outer and inner context of a lambda. The most general type expects contexts with arbitrary tags and combines them: $C^r\tau_1 \times C^s\tau_2 \to C^{r\otimes s}(\tau_1 \times \tau_2)$. The type can be restricted by requiring one of the two arguments (contexts) to be tagged with the unit of the monoid instead of an arbitrary tag. This gives two interesting variants of the lambda abstraction, which are shown in Figure 5.

Comonads with Pure Functions. The variant in Figure 5(a) has the unit tag for the second parameter of *combine*, which corresponds to the context carried by the function argument. Thus, all context-dependent functions will be pure (no coeffects). The input type remains comonadic, but the tag is unit: $C^1\tau_1 \to \tau_2$.

$$\frac{C^r(\Delta, a : \tau_1); \Gamma \vdash e : \tau_2}{C^r\Delta; \Gamma \vdash \lambda^\circ a.e : C^1\tau_1 \to \tau_2}$$

$$\mathit{combine} \ : \ C^r\tau_1 \times C^1\tau_2 \to C^r(\tau_1 \times \tau_2)$$

Fig 5. (a) Pure functions

$$\frac{C^r(\Delta, a : \tau_1); \Gamma \vdash e : \tau_2}{C^1\Delta; \Gamma \vdash \lambda^\circ a.e : C^r\tau_1 \to \tau_2}$$

$$\mathit{combine} \ : \ C^1\tau_1 \times C^r\tau_2 \to C^r(\tau_1 \times \tau_2)$$

Fig 5. (b) Pure contexts

The rule specifies that any context-dependence has to be satisfied by the scope where the function is defined. The resulting function does not capture the context at the call site. If the rule was used with implicit parameters (Section 3.2), the type system would be reduced to ordinary lexical scoping — all parameters have to be available from the declaring scope, they are captured by a closure and the caller cannot override them.[1]

This variant can be used to type-check partially evaluated expressions in distributed languages. For example, when defining a function on the *server*, all sub-expressions in the body that require the *server* environment can be substituted with a value. The resulting pure function can be transferred to the *client*.

Comonadic Types Equivalent to Monads. The variant in Figure 5(b) requires unit tag for the first parameter of *combine*. This gives a variant of lambda abstraction that is similar to the rule in monadic effect systems. The rule specifies that all the context is provided by the caller of the function. For example, typing the format function from Section 3.2 using this rule gives a type $C^{\{\mathsf{?width:int}\}}\mathsf{string} \to C^{\{\mathsf{?width:int,?size:int}\}}\mathsf{string} \to \mathsf{string}$.

The outer function accesses the ?width parameter; the nested function accesses both parameters. Unlike the original interpretation, the ?width parameter is not captured by the inner function thus must be provided again by the caller. This is equivalent to the *reader* monad. In fact, a comonadic type system that uses this variant of lambda abstraction can be translated to a monadic type system that tracks the same property and vice versa (Section 6). As a result, this variant can be used to track properties that are traditionally viewed as effects.

Comonadic Distributed Language. The distributed language introduced in Section 2.1 uses a type system written in a coeffect style to track the environment in which a function can be executed. Lambda abstraction is constructed in a unit context, so we can encode it using the variant in Figure 5(b). We use a monoid $(\mathcal{P}(\Omega), \Omega, \cap)$ and add the following primitives:

$$\begin{array}{l} e ::= \ldots \mid \textbf{access}\ e \\ \tau ::= \ldots \mid \textbf{res}^r\tau \end{array} \qquad (\text{access})\frac{C^r\Delta; \Gamma \vdash e : \textbf{res}^s\tau}{C^{r\otimes s}\Delta; \Gamma \vdash \textbf{access}\ e : \tau}$$

[1] This variant also reverses the order of combined effects in comonadic **let** binding, meaning that the monoid of tags should be commutative.

Similarly to the earlier example, resources have a type $\mathsf{res}^r\tau$ and are accessed using **access**. As demonstrated in online Appendix B [25], the language can be used to implement and type-check the hashInput function from Section 2.1

6 When Coeffects and Effects are Equivalent

As mentioned, there are some program properties that can be tracked equivalently by an effect system or a coeffect system.

Section 6.1 describes the conditions for equivalence between information tracking in coeffect and effect systems, and shows that the general comonadic coeffect system discussed in Section 5 tracks the same information as the standard monadic effect system shown above in Figure 6.

Section 6.2 describes the conditions under which a monadic and comonadic semantics are equivalent. Two pairs of equivalent semantic models for (co)effect systems are shown: 1). using the identity comonad and identity monad, corresponding to adding *unwitnessed* (co)effects onto an existing language, without changing the semantics 2). using a product comonad and exponent monad for *witnessed* constraints/capabilities.

6.1 Equivalence of Types/Information

For a language, a coeffect system (with judgments $\vdash_C$) and effect system (with judgments $\vdash_M$) are equivalent in the information tracked if they use the same monoid of tags and if, for all expressions e, given $C^r\Delta \vdash_C e : \tau$ and $\Gamma \vdash_M e : M^r\tau'$ then $C^r\Delta \to \tau \leftrightharpoons \Gamma \to M^r\tau'$, where $\leftrightharpoons$ is a *pointwise*, recursive relation on types comparing equality of constructors and ground types, with one additional rule for monadic/comonadic function types:

$$\frac{\tau_1 \leftrightharpoons \tau_1' \qquad \tau_2 \leftrightharpoons \tau_2'}{(C^r\tau_1 \to \tau_2) \leftrightharpoons (\tau_1' \to M^r\tau_2')} \; [eq_{\to r}]$$

where the comonad and monad are annotated with the same tag r. The rest of the relation is straightforward, defining recursive equality (see Appendix C [25]).

Proposition 1. *The general comonadic coeffect system shown in Figure 3, with the variant of the (cofun) rule shown in Figure 5(b), is equivalent in the information tracked to the monadic effect system of Figure 6.*

$$(\text{bind})\frac{\Gamma \vdash e_1 : M^r\tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : M^s\tau_2}{\Gamma \vdash \mathbf{let}\, x \Leftarrow e_1\, \mathbf{in}\, e_2 : M^{r\otimes s}\tau_2} \qquad (\text{app})\frac{\Gamma \vdash e_1 : M^r(\tau_1 \to M^t\tau_2) \quad \Gamma \vdash e_2 : M^s\tau_1}{\Gamma \vdash e_1\; e_2 : M^{(r\otimes s\otimes t)}\tau_2}$$

$$(\text{fun})\frac{\Gamma, x : \tau_1 \vdash e : M^r\tau_2}{\Gamma \vdash \lambda x.e : M^1(\tau_1 \to M^r\tau_2)} \qquad (\text{unit})\frac{x : \tau_1 \in \Gamma}{\Gamma \vdash x : M^1\tau_1}$$

Fig. 6: Monadic effect system, for a tagged monad M^t

The proof proceeds by structural induction on the terms of the language and is straightforward; Appendix D [25] provides one case for demonstration.

For any additional rules of a coeffect system, further to the general framework of Figure 3, there must be a corresponding rule in the effect system. For the distributed programming example, an equivalent effect system requires a typing rule for the **access** construct corresponding to the coeffect rule e.g.:

$$\frac{\Gamma \vdash e : \mathsf{res}^r \tau}{\Gamma \vdash \mathsf{access}\ e : M^r\ \tau}(access)$$

6.2 Equivalence of Semantics

For a language L, semantics for a comonadic coeffect system $[\![-]\!]_C$ and a monadic effect system $[\![-]\!]_M$ are equivalent (or isomorphic), if:

1. the monadic effect system and comonadic effect system are equivalent in the information tracked (see Section 6.1)
2. there exists an isomorphism between the denotations:

$$\phi_{A,B} : (C\tau_1 \to \tau_2) \cong (\tau_1 \to M\tau_2) \tag{1}$$

   Such an isomorphism is always provided if C and M are *adjoint functors* (where C is *left adjoint*), where ϕ is known as the *hom-set adjunction*.
3. and, for every expression $e \in L$ that $[\![e]\!]_M = \phi[\![e]\!]_C$, which can be proved via structural induction over the terms of the language.

The isomorphism ϕ from the adjunction, if it exists, gives a way to convert one semantics into the other. There are two typical cases of adjoint monads and comonads which we discuss here: the identity comonad and identity monad, and the product comonad and exponent monad.

Identity comonad & identity monad The identity comonad ($CA = A$) and the identity monad ($MA = A$) are trivially adjoint where the isomorphism between $CA \to B$ into $A \to MB$ is the identity function.

A tagged identity comonad $C^tA = A$ and tagged identity monad $M^tA = A$ provide no value witnessing the effect/coeffect tag t. Since the identity (co)monad does not add extra structure, but is simply a wrapper, any language can be given an identity (co)monad semantics without changing the operational semantics of the language. Note that the static semantics of the language might change, as run-time bugs may now be prevented at compile-time by the stronger typing.

Proposition 2. *The semantics of the comonadic coeffect system for the tagged identity comonad and semantics of the monadic effect system for the tagged identity monad are equivalent.*

The proof is given by the induction over the structure of terms and is straightforward. All of the monad/comonad definitions for the identity comonad and monad are essentially identities. For demonstration purposes we show just one clause of the proof in Appendix E [25].

The identity comonad can be used to compose different coeffects, as a composite comonad can always be formed with the identity comonad.

Product comonad & exponent monad The product comonad pairs a value A with another value of a fixed type X i.e. $CA = A \times X$, thus functions $CA \to B$ describe computations with an extra, implicit parameter of type X. The exponent monad, often called the *reader* monad, parameterises a value A by some parameter X i.e. $MA = X \to A$, thus functions $A \to MB$ describe computations where the result depends on some extra, implicit parameter, of type X.

The functor of the product comonad is *left adjoint* to the exponent monad, where the *hom-set adjunction* is provided by currying and uncurrying:

$$(A \times X) \to B \quad \underset{\mathsf{curry}}{\overset{\mathsf{uncurry}}{\leftrightarrows}} \quad A \to (X \to B) \tag{2}$$

The tag of a tagged product comonad may be the type of the implicit parameter i.e. $C^X A = A \times X$, and likewise the tagged exponent monad $M^X A = X \to A$ (assuming their is a monoidal operation on such types). With the tagged product comonad and tagged exponent monad we can give a semantic meaning to capabilities. For example, with the implicit parameters of Section 3.2, the comonadic and monadic semantics of lookup $?p$ might be defined:

$$\begin{aligned} [\![(\bar{x})\ ?p]\!]_C &= \lambda(\Omega, x)\ .\ \textbf{if}\ x = \{?p\}\ \textbf{then}\ \mathit{lookup}\ p\ \textbf{else}\ \bot \\ [\![(\bar{x})\ ?p]\!]_M &= \lambda\Gamma\ .\ \lambda x.\ \textbf{if}\ x = \{?p\}\ \textbf{then}\ \mathit{lookup}\ p\ \textbf{else}\ \bot \end{aligned}$$

Note that: $[\![(\bar{x})\ ?p]\!]_M = \mathsf{curry}\ [\![(\bar{x})\ ?p]\!]_C$ holds, thus this constitutes one clause of the proof of the following proposition:

Proposition 3. *The semantics of the comonadic coeffect system for the product comonad and the monadic effect system for the exponent monad are equivalent.*

We do not provide the rest of the proof here for space reasons, but the remaining cases are straightforward given the definitions.

7 Related and Future Work

Modal and Linear Logic Systems. Bierman and De Paiva [3] developed a natural deduction system for intuitionistic S4 modal logic and model it in category theory using a comonad for the $\Box$ connective. The corresponding term language (also similar to the work of Pfenning and Wong [27]) bears some resemblance to our comonadically-typed lambda calculus. Their system is simpler, separating introduction and elimination of $\Box$, while our system is more complex in order to match standard lambda-calculus typing rules. Because of the correspondence between S4 modal logic and comonads, many of the type systems based on modal logic [20, 21, 24, 27] are also similar to our comonadic coeffect system.

Intuitionistic S5 modal logic (with both $\Box$ and $\Diamond$ modalities) has been used by to describe mobility and locality respectively in a distributed programming setting [20, 21]. Murphy et al. use explicit *possible worlds*, with judgements of the form $\Gamma \vdash e : \tau@\omega$ denoting a typing assignment at world ω. The ω annotation

resembles our coeffect tags, although ω is restricted to a single world. We believe that such systems could be captured by our general comonadic coeffect system.

Linear logic can be used to control resource-usage (e.g. [4]), usually omitting the standard contraction rule [17]. Our system allows contraction in the non-comonadic part of the language. In the comonadic part, contraction may be controlled using *split* and an appropriate monoid. Finding a variant of our system that can capture linearity is an important future work. Blute et al. [5] describe a linear logic for storage, modelling the ! connective as a monoidal comonad that provides contraction via an operation $!A \rightarrow !A \times !A$. This structure closely resembles our tagged comonads and it may fit within our coeffect framework.

Type Systems for Coeffects. Several existing type systems can be described as coeffect systems. The implicit parameters example in this paper was inspired by Lewis et al. [16] who mention comonadic semantics as future work. Our other example was motivated by distributed programming languages [20, 29]. Links [7] supports different execution environments explicitly and database dependency as an effect. Our systems provides a general framework for checking such properties.

Type systems that guarantee secure information flow [28, 34] can also be viewed as coeffect systems. The context specifies the secrecy of the information. Certain operations are only allowed when the context represents specific secrecy. In the calculus of capabilities [8], the context carries *capabilities* that allow access to memory regions. The context is modified by memory allocation and deallocation. We believe that such systems could use the general framework provided by our type system.

Effect and *Coeffect Systems* Some computations can be viewed as coeffect systems, but they also provide operations that modify the context. In a system for safe locking [9], the context carries a set of acquired locks (enabling access to a reference), but the synchronize construct modifies the context.

Such systems could be structured by both a monad and a comonad *at the same time*, with functions of type: $Ca \rightarrow Mb$, for some C comonad and M monad [6]. For example, partial dataflow computations can be structured by a stream comonad and partiality (exception) monad [33]. A *distributive law* between the two structures $dist : C(Ma) \rightarrow M(Ca)$ is required in order to define a composition operation: $compose : (Ca \rightarrow Mb) \rightarrow (Cb \rightarrow Mc) \rightarrow (Ca \rightarrow Mc)$.

We expect that comonadic coeffect and monadic effect systems can be composed, if they provide a *tagged* distributive law of type: $dist : C^r(M^s a) \rightarrow M^s(C^r a)$. This operation can be easily defined for the simple categorical model based on the product comonad and exponent monad.

Parameterised Monads and Comonads In a monadic setting, *parameterised monads* of Atkey [2] use tags that comprise pre- and post-conditions of a computation. McBride shows [18], that these tags are related to specifications in Hoare type theory [22]. Many practical systems [2, 15] use sets of labels.

Traditional effect systems [31] include a sub-typing rule that cannot be directly expressed using our *monoid* tags:

$$\frac{\Gamma \vdash e : \tau, F \qquad F \subseteq F'}{\Gamma \vdash e : \tau, F'}$$

An analogous rule for coeffect subtyping can be easily added (syntactically) to the coeffect system if the monoid of tags is partially ordered. However, we believe that finding a categorical semantics for language with such rule is an open problem (*toposes*, which have *sub-object* classifiers, may be useful).

8 Conclusion

It is well known that effects can be captured using a monadic effect system. The main contribution of this paper was introducing the dual concept of a *comonadic coeffect system*, which can be used to capture a range of context-dependent properties. Using a *tagged comonad*, variable contexts are annotated with tags formed by a *monoid*. We used the system to track environments in a distributed language and to give types for dynamically-scoped implicit parameters.

Unlike the monadic effect system, there are multiple variants of the comonadic coeffect system. We described different variants that can be obtained by specifying how coeffects are combined when constructing a function. We showed that properties tracked by a monadic effect system can be equivalently tracked by a variant of the comonadic coeffect system.

Acknowledgements. We are grateful to Gavin Bierman, Peter Calvert, Iliano Cervesato, and Robin Message for useful comments and discussions. This work was supported in part by an EPSRC DTA and CHESS.

References

1. *Concise Oxford English Dictionary.* Oxford University Press, 2009.
2. R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19, 2009.
3. G. Bierman and V. De Paiva. On an intuitionistic modal logic. *Studia Logica*, 65(3):383–416, 2000.
4. N. Biri, D. Galmiche, and H. Poincar. A modal linear logic for distribution and mobility. In *In FLOC02 Workshop on Linear Logic*. Citeseer, 2002.
5. R. Blute, J. Cockett, and R. Seely. ! and?–Storage as tensorial strength. *Mathematical Structures in Computer Science*, 6(04):313–351, 1996.
6. S. Brookes and K. Stone. Monads and Comonads in Intensional Semantics. 1993.
7. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. FMCO '00.
8. K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. POPL '99, 1999.
9. C. Flanagan and M. Abadi. Types for Safe Locking. ESOP '99, 1999.

10. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of Conference on Programming Language Design and Implementation*, PLDI '03.
11. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and func. prog.*, LFP '86, 1986.
12. M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, pages 97–136, London, UK, 1995. Springer-Verlag.
13. P. Jouvelot and D. K. Gifford. Communication Effects for Message-Based Concurrency. Technical report, Massachusetts Institute of Technology, 1989.
14. R. B. Kieburtz. Codata and Comonads in Haskell, 1999.
15. O. Kiselyov and C.-c. Shan. Lightweight monadic regions. In *Proceedings of Symposium on Haskell*, Haskell '08, pages 1–12, 2008.
16. J. R. Lewis, M. B. Shields, E. Meijert, and J. Launchbury. Implicit parameters: dynamic scoping with static types. In *POPL00*, 2000.
17. S. Martini and A. Masini. On the fine structure of the exponential rule. In *Advances in Linear Logic*. Citeseer, 1993.
18. C. McBride. Kleisli arrows of outrageous fortune. Under consideration for publication in J. Funct. Program., 2011.
19. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.
20. T. Murphy, VII., K. Crary, and R. Harper. Type-safe distributed programming with ML5. TGC'07, pages 108–123, 2008.
21. V. Murphy et al. A Symmetric Modal Lambda Calculus for Distributed Computing. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 2004.
22. A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.
23. D. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: Declarative, Parallel Structured Grid Programming. In *DAMP '10*, pages 15–24, NY, USA, 2010. ACM.
24. S. Park. A modal language for the safety of mobile values. *Programming Languages and Systems*, pages 217–233, 2006.
25. T. Petricek, D. Orchard, and A. Mycroft. Coeffects: typing context-dependent computations using comonads (with appendices)[2], 2011.
26. S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.
27. F. Pfenning and H. Wong. On a Modal [lambda]-Calculus for S41. *Electronic Notes in Theoretical Computer Science*, 1:515–534, 1995.
28. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
29. T. Sans and I. Cervesato. QWeSST for Type-Safe Web Programming. In *Third International Workshop on Logics, Agents, and Mobility*, LAM'10, 2010.
30. D. Syme, A. Granicz, and A. Cisternino. *Expert F# 2.0*. Apress, 2010.
31. J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS'92.*, pages 162–173, 1994.
32. T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203:263–284, June 2008.
33. T. Uustalu and V. Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, Nov 2006.
34. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.
35. P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.

[2] `http://www.cl.cam.ac.uk/~tp322/papers/coeffects.html`