Combining Effects and Coeffects via Grading

Marco Gaboardi
SUNY Buffalo, USA
gaboardi@buffalo.edu

Shin-ya Katsumata
Kyoto University, Japan
sinya@kurims.kyoto-u.ac.jp

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

Flavien Breuvart
Inria Sophia Antipolis, France
flavien.breuvart@inria.fr

Tarmo Uustalu
Institute of Cybernetics at TUT, Estonia
tarmo@cs.ioc.ee

Abstract

Effects and *coeffects* are two general, complementary aspects of program behaviour. They roughly correspond to computations which change the execution context (effects) versus computations which make demands on the context (coeffects). Effectful features include partiality, non-determinism, input-output, state, and exceptions. Coeffectful features include resource demands, variable access, notions of linearity, and data input requirements.

The effectful or coeffectful behaviour of a program can be captured and described via type-based analyses, with fine grained information provided by monoidal effect annotations and semiring coeffects. Various recent work has proposed models for such typed calculi in terms of *graded (strong) monads* for effects and *graded (monoidal) comonads* for coeffects.

Effects and coeffects have been studied separately so far, but in practice many computations are both effectful and coeffectful, *e.g.*, possibly throwing exceptions but with resource requirements. To remedy this, we introduce a new general calculus with a combined *effect-coeffect system*. This can describe both the *changes* and *requirements* that a program has on its context, as well as interactions between these effectful and coeffectful features of computation. The effect-coeffect system has a denotational model in terms of effect-graded monads and coeffect-graded comonads where interaction is expressed via the novel concept of *graded distributive laws*. This graded semantics unifies the syntactic type theory with the denotational model. We show that our calculus can be instantiated to describe in a natural way various different kinds of interaction between a program and its evaluation context.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords effects; coeffects; monads; comonads; distributive laws; grading; types; categorical semantics

ICFP'16, September 18 - 22, 2016, Nara, Japan

ACM 978-1-4503-4219-3/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2951913.2951939

1. Introduction

Pure, total functional programming languages are highly amenable to clear and concise semantic descriptions. This semantics aids both correct-by-construction programming and tools for reasoning about program properties. A pure program can be described as a mathematical object that is isolated from the real world. However, even in the most abstract setting, a program is hardly isolated. Instead it interacts with its evaluation context; *paradise is lost*.

The interaction of a program with its context can be described in several ways. For instance it can be described by recording the *changes* that a program performs on its context, *e.g.* the program can write to a memory cell or it can print a character on an output display. At the same time the interaction can also be expressed by recording the *requirements* that a program has with respect to its context, *e.g.* the program can require a given amount of memory or to read the input from some channel. These two aspects correspond to the view of a program as a *producer* and as a *consumer*.

Computational effects and monads The need to describe the interaction of a program with its context emerged early in pure functional programming. Indeed, basic operations like input-output are inconceivable in a program that runs in isolation. Most of the original efforts to understand the interaction of a program with its context focussed on input-output, stateful computations, non-determinism, and probabilistic behaviours. This leads to the distinction between pure and *effectful* computation. The diverse collection of interactions described above are often referred to as *computational effects*. For our presentation, we identify these with behaviours that *change* the execution context, or as *producer* effects.

Moggi showed that the semantics of sequential composition for various computational effects can be described uniformly via the structure of a *(strong) monad* [**?**]. It was then shown how to syntactically integrate monads into a typed calculus (the monadic metalanguage) [**?**] providing a way to describe and encapsulate concrete computational effects in languages like Haskell [**?**].

In parallel to this, *effect systems*—a class of static analysis augmenting a type system—were introduced to analyse various kinds of computational effect [**? ? ?**]. Effect systems track individual effectful operations. This gives a more fine grained view than monadic types, which indicate only the variety of effect taking place (*e.g.*, state effects) but not which effect operations are used.

These two strands of work on the analysis and semantics of effects were eventually unified, syntactically [**?**] with effect information annotating monadic types, and then semantically [**? ?**] by "grading" a monad with *effects*. This grading requires effect terms to have a (preordered) monoidal structure. *Effect-graded monads*[1] provide a semantics for effects, in the style of Moggi's monadic calculus, but where effects are explicitly tracked. This provides a denotational semantics describing computations in a refined way whilst also providing tools for program analysis in the types.

Coeffects and comonads Dual to the notion of *producer* effects, which change the evaluation context, are *consumer effects* which make demands on the context by requiring some *computational resource*. Computational resource requirements may be *intensional* in nature, such as memory or CPU usage; or requirements may be extensional, affecting the outcome of a computation. For example, requirements might be for a particular library version, hardware resource, service, or size and extent of a data structure.

Comonads (the categorical dual of monads) have been shown to describe a general class of resource-dependent computations and the requirements that a program has on the execution context. For instance, (*monoidal*) comonads are at the heart of the resource management mechanism embedded in Girard's Linear Logic via the ! modality [**?**] and give the semantics of context-dependent dataflow programs [**?**]. Similarly to monads, comonads provide a uniform semantics for consumer effects. However, they suffer also the same limitations: the abstraction layer provided by comonads gives only coarse-grained information on the resource requirements.

Dual to effect systems, for fine-grained effect information, are *coeffect systems* for resource requirements, which have been recently introduced [**? ? ? ? ? ?**]. For example, the reuse bounds in Bounded Linear Logic are an instance of a coeffect system which precisely tracks the usage requirements on variables. The name "coeffect" emphasizes the duality with traditional effect systems and the notion of resource consumption or context-dependent effects. The notion of a *coeffect-graded comonad*, dualising graded monads, has also been shown to unify fine-grained resource requirement analyses with a denotational model of resources.

Our contribution: effect-coeffect systems via distributivity The interaction of a program with its evaluation context is not always solely about producing a change or consuming a resource. Instead, programs make both demands on, and produce changes to, the context; computations are often both effectful and coeffectful. Motivated by this observation we propose an *effect-coeffect system*: a typed calculus combining effects and coeffects syntactically, and employing both effect-graded monads and coeffect-graded comonads in its semantics. We show that combining these two notions in one system captures a broad class of fine-grained interactions between a program and its context.

Moreover, changes to the evaluation context may depend on program requirements, and vice versa. That is, coeffects and effects may *interact*. To capture these interactions semantically requires the interaction of a graded monad and a graded comonad. A standard categorical technique for combining a (non-graded) monad and (non-graded) comonad uses a *distributive law* between them [**? ? ?**]. Inspired by this, we lift notions of distributive law to the graded setting. This grading induces a syntactic theory of interaction between effects and coeffects, captured by a *matched-pair* of operations which calculates an effect and a coeffect from a coeffect-effect pair.

We make the following contributions:

- a novel typed calculus with both a general effect system and coeffect system which may interact via a family of distributive laws; the calculus is parameterised by the algebraic structure for effects, coeffects, and their distributive interaction (Section 3),
- an equational theory for our calculus describing the interaction of these components from a syntactic perspective (Section 4),
- a categorical denotational semantics, introducing the notion of *graded distributive laws* between graded monads and comonads, giving a sound model of our calculus with respect to its equational theory (Section 5),
- various examples demonstrating how our calculus can be smoothly instantiated to describe different computational behaviours that result from the interaction of effectful and coeffectful computation (Section 6).

Section 7 discusses related work and Section 8 considers various possible avenues for further study. We begin by introducing and motivating the main components of our system with examples.

Our intention with this calculus is to give a strong and flexible starting point for building languages and designing semantics that clearly capture effect-coeffect interactions. We present a general system, setting out a design space of the choices for distributive laws, which provide the effect-coeffect interaction.

2. In Brief: Effects, Coeffects & their Interaction

To introduce effectful and coeffectful computations, their type-based analysis, and their graded models, we look first at *exceptions* as a classic example of effects (Section 2.1) and *reuse bounds* for variables as an example of coeffects (Section 2.2). We then combine exceptions and reuse bounds, giving an example of their interaction and an introduction to *graded distributive laws* (Section 2.3).

2.1 Effects and Graded Monads

Consider a language with a notion of global exception that interrupts the control-flow of a program and is uncaught. Exceptions are introduced to a program via an operation **throw**.

In a monadic metalanguage à la Moggi [**?**], exceptions are typed $\vdash \textbf{throw} : T\,\textbf{unit}$ where T is the type constructor of a monad and **unit** is the singleton unit type as the evaluation need not return a value. In the monadic metalanguage, the monad T comes with a term for its *unit* operation (often called *return*) and a term for monadic *composition* (based on the *multiplication* operation):

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \langle t \rangle : TA} \qquad \frac{\Gamma \vdash t_1 : TA \qquad \Gamma, x : A \vdash t_2 : TB}{\Gamma \vdash \textbf{let}\ \langle x \rangle = t_1\ \textbf{in}\ t_2 : TB}$$

If the evaluation of some term t *potentially* throws an exception, the soundness of the typing rules ensures that we will assign to t a type $\Gamma \vdash t : TA$. Seen from the converse perspective, given a term of type $\Gamma \vdash t : TA$ all we know is that the term t may throw an exception. Consider a situation where instead we have a program analysis that can determine whether a term definitely throws an exception, definitely does not throw an exception, or it is unknown what will happen. The analysis is thus three-valued: $\bot$ (meaning definite exception), $\top$ (no exceptions) and $?$ (statically unknown). The information from this analysis can be added explicitly to the type system. We can do this in a uniform way for a broad class of effectful computations and analyses by following **?**] and "grading" the monad[2] T with *effect annotations* that are elements of a preordered monoid[3] $(\mathcal{E}, \bullet, I)$, that is, with binary operation $\bullet : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ and the unit element $I \in \mathcal{E}$. Given an effect annotation $e \in \mathcal{E}$ (or *effect* for short), then an *effect graded monad* provides the indexed type T_e.

[1] We borrow this terminology from "graded algebra" to avoid confusion with "parametrised" or "indexed monad" terminology in different contexts.

[2] To be clear, we use "grading" as a verb here, but a specific graded monad is not the result of some "grading" transformation on a monad. Rather, the graded monad definition generalisations that of monads.

[3] We highlight effect annotation operations and elements in *orange* and coeffect annotation operations and elements in *blue*.

Thus to improve the type-level information in our example, we use a graded monad for exceptions with the discretely ordered monoid $(\{\bot, ?, \top\}, \bullet, \top)$, where $\bullet$ is defined:

$\bullet$	$\bot$	$?$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
$?$	$\bot$	$?$	$?$
$\top$	$\bot$	$?$	$\top$

i.e., $\bot$ is the absorbing element and $\top$ is the unit. The typing for **throw** becomes $\vdash$ **throw** $: T_{\bot}$**unit** since it is clear that the term definitely raises an exception.

Unit and composition for effectful computations is then provided by the following two rules for effect graded monads, which use the monoid structure on $\mathcal{E}$:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \langle t \rangle : T_{\top} A} \quad \frac{\Gamma \vdash t_1 : T_e A \qquad \Gamma, x : A \vdash t_2 : T_f B}{\Gamma \vdash \textbf{let}\ \langle x \rangle = t_1\ \textbf{in}\ t_2 : T_{e \bullet f} B}$$

The soundness of the typing rules now ensures that if the evaluation of t can potentially throw an exception then we will either assign to t a type $\Gamma \vdash t : T_{\bot} A$ or $\Gamma \vdash t : T_{?} A$. So, by using effect-graded monad typing we are able to recover from our program more information than in the classical monadic approach; the indices of the graded monad provide an effect system.

2.2 Coeffects and Graded Comonads

One way to understand coeffects is to view them as a generalisation of resource consumption control provided by the exponential modality ! of linear logic. This modality distinguishes terms that are evaluated exactly once from terms that can be evaluated an arbitrary number of times. If the evaluation of a term t requires the repeated evaluation of some free variable x, then we assign to x a type of the form $!A$. This expresses the *requirement* that x can be evaluated an arbitrary number of times.

The comonadic nature of ! is apparent from its typing rules:[4]

$$\text{der}\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : !A \vdash t : B} \quad \text{pr}\frac{!\Gamma \vdash t : B}{!\Gamma \vdash !t : !B}$$

In the linear logic literature, these two rules are usually referred to as *dereliction* and *promotion*, respectively. The first corresponds to the *counit* axiom $!A \to A$ of the comonad, while the second rule roughly internalizes the *comultiplication* $!A \to !!A$ of the comonad (combined with functoriality of !).

The ! modality has additional structure relating to the multiplicative connectives $(1, \otimes)$ of linear logic. Firstly, ! is has the additional structure of a *monoidal comonad* for managing the use of ! in the context with operations of type $1 \to !A$ and $!A \otimes !B \to !(A \otimes B)$ (called *0-monoidality* and *2-monoidality* respectively). Secondly, $!A$ admits structural rules via additional comonoidal structure, with *contraction* $!A \to !A \otimes !A$ and *weakening* $!A \to 1$ operations. These ensure that the ! modality satisfies the requirement of using a variable an arbitrary number of times.

Whilst the ! comonad expresses unrestricted (re)use, in many situations this requirement is too coarse grained. For instance, in resource analysis we are often interested in giving a *bound* on the number of times a variable is used. This can be used to express the computational complexity of a program. Unfortunately, the ! comonad alone is not enough for expressing bounds. A natural way to recover this information is by adding it explicitly to the type system. We can do this in a uniform way for a large class of coeffectful computations by following **? ? ?**] and using a *coeffect-graded comonad* which provides a type constructor D_r indexed by elements $r \in \mathcal{R}$ of a preordered semiring $(\mathcal{R}, \leq, 0, +, 1, *)$.

[4] These rules are not syntax directed and break the type preservation property. In the next sections, we will use a slightly different language with syntax directed typing rules that guarantee type preservation.

In the concrete case of resource analysis we take the standard natural numbers semiring $\mathbb{N}$, which gives an indexed ! modality corresponding to that of *bounded* linear logic [**?**]. In the rest of this paper, D is used for the graded comonad type constructor, but for this example we continue using !. Graded comonads provide graded dereliction and promotion:

$$\text{der}\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : !_1 A \vdash t : B} \quad \text{pr}\frac{!_{\vec{s}}\Gamma \vdash t : B}{!_{r * \vec{s}}\Gamma \vdash !t : !_r B}$$

The type $!_1$ for dereliction (corresponding to graded counit) is indexed by 1, that is the multiplicative unit for the semiring $\mathbb{N}$. The promotion rule uses the multiplication $*$ of the semiring, where we use the notation $!_{\vec{s}}\Gamma$ to mean that each variable assignment has the shape $x_i : !_{s_i} A_i$, and $!_{r*\vec{s}}\Gamma$ denotes that each variable assignment has the shape $x_i : !_{r * s_i} A_i$. A coeffect graded comonad enables more information about program requirements to be captured in types compared with using just the standard ! comonad.

We also need graded versions of contraction and weakening, which uses the additive structure of $\mathbb{N}$: graded contraction has type $!_{r+s} A \to !_r A \otimes !_s A$ and graded weakening $!_0 A \to 1$.

In our calculus, defined formally in Section 3, coeffect types on the left of typing judgments introduced by dereliction and promotion are written $[A]_r$ instead of $!_r A$. This notation is needed to distinguish coeffect requirements $[A]_r$ on a free-variable assumption from values of type D_r, which can be thought of as coeffect capabilities. Along with this distinction comes syntax for coeffectful substitution, composing a capability with a requirement:

$$\text{let}\frac{\Gamma \vdash t_1 : D_r A \qquad \Delta, x : [A]_r \vdash t_2 : B}{\Gamma + \Delta \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : B}$$

We subsequently write $[t]$ for the syntax of promotion (rather than $!t$ in the linear logic example above). Now, the soundness of the typing rules ensures that if in the evaluation of **let** $[x] = t_1$ **in** t_2 the term t_1 is evaluated n times, then we will assign to t_2 a type $\Delta, x : [A]_n \vdash t_2 : B$. In fact, by using the preorder we can assign type $\Delta, x : [A]_m \vdash t_2 : B$ where $n \leq m$.

Other example coeffects in this style include tracking secure information flow (which we use in Section 6.1), consumption-bounds in dataflow computations (see [**?**]), and fine-grained strictness information based on tracking the deconstructors applied to variables.

2.3 Effect and Coeffect Interaction; Graded Distributed Law

Effect and coeffect systems as presented above express different properties of a program's interaction with its context. We can combine them in a system where an effect-graded monad T_e, graded by the elements of an effect monoid $\mathcal{E}$, coexists with a coeffect-graded comonad D_r, graded by the elements of a coeffect semiring $\mathcal{R}$. For instance, combining the two examples above gives a system for exception tracking with reuse bound information.

In this system we can describe two kinds of composition involving effects and coeffects. The first is the coeffectful composition of the following two typed terms:

$$\Gamma \vdash t_1 : D_r T_e A \qquad \Delta, x : [T_e A]_r \vdash t_2 : B$$

In this situation we prioritize the reuse bound information r over the exception information e (coeffects are at the outer level). The second is the effect composition of the following two typed terms:

$$\Gamma \vdash t_1 : T_e D_r A \qquad \Delta, x : D_r A \vdash t_2 : T_{e'} B$$

In this situation we prioritize the exception information over the reuse bound information (effects are at the outer level).

Having only these two situations is unsatisfying because the graded monad and the graded comonad cannot interact, simply coexisting independently in the same world. Instead, we would also like to allow their interaction.

Consider the following two terms and their typings:

$$\Gamma \vdash t_1 : T_e A \qquad \Delta, x : [A]_r \vdash t_2 : B \tag{1}$$

In general, we would like to be able to compose these two computations with the effect and coeffect interacting. How can we do this? One answer is provided by a *distributive law* between the graded comonad and graded monad which captures an interaction between coeffects and effects. In the non-graded case, a distributive law of a comonad over a monad is an operation of type:

$$\mathbf{dist}_A : DTA \to TDA$$

This can be understood as taking a capability for an effectful computation and transforming it into an effectful computation of a capability. Our calculus provides a number of different possible graded distributive laws for graded comonads and monads that we will present in Section 3. For this example, we will use one specialised to the following type:

$$\mathbf{dist}_{r,e,A} : D_{\iota(r,e)} T_e A \to T_e D_r A \tag{2}$$

where $\iota : \mathcal{R} \times \mathcal{E} \to \mathcal{R}$ is a binary operation that describes the interaction between coeffects and effects, defined:

$$\iota(r, \bot) = 1 \qquad \iota(r, \top) = r \qquad \iota(r, ?) = r$$

Expanding the definition of ι, we have the following family of graded distributive law operations:

$$\mathbf{dist}_{r,\bot,A} : D_1 T_\bot A \to T_\bot D_r A$$
$$\mathbf{dist}_{r,\top,A} : D_r T_\top A \to T_\top D_r A$$
$$\mathbf{dist}_{r,?,A} : D_r T_? A \to T_? D_r A$$

The first case explains that, if an effectful computation is known to definitely throw an exception, then only one copy of that effectful computation is needed to satisfy any number of copies of A. This is because the flow of execution is interrupted by the exception, and so more copies of the exception are not needed. The other two explain that, if it is not known whether the computation throws an exception (or it definitely does not), then the coeffect is unchanged.

Using **dist** the original two terms in (1) can be composed:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash t_1 : T_e A}{[\Gamma]_1 \vdash t_1 : T_e A}}{[\Gamma]_{\iota(r,e)} \vdash [t_1] : D_{\iota(r,e)} T_e A}}{[\Gamma]_{\iota(r,e)} \vdash \mathbf{dist}[t_1] : T_e D_r A} \qquad \dfrac{\Delta, x : [A]_r \vdash t_2 : B}{\Delta, x : [A]_r \vdash \langle t_2 \rangle : T_1 B}}{[\Gamma]_{\iota(r,e)} + \Delta \vdash \mathbf{let}\ \langle [x] \rangle = \mathbf{dist}[t_1]\ \mathbf{in}\ \langle t_2 \rangle : T_e B}$$

where $\mathbf{let}\ \langle [x] \rangle = t\ \mathbf{in}\ t'$ is syntactic sugar for the two forms of effectful and coeffectful binding combined: $\mathbf{let}\ \langle z \rangle = t\ \mathbf{in}\ \mathbf{let}\ [x] = z\ \mathbf{in}\ t'$. Therefore, in the above composition, we see that the requirements of t_2 propagate towards the left-hand side, and are modified by the effect of t_1 which may reduce the requirements.

Compositional motivation The graded distributive law above (2) is a specialisation of a more general operation, of the form:

$$\mathbf{dist}_{r,e,A} : D_{\iota(r,e)} T_e A \to T_{\kappa(r,e)} D_r A$$

with a pair of functions $\iota : \mathcal{R} \times \mathcal{E} \to \mathcal{R}$ and $\kappa : \mathcal{R} \times \mathcal{E} \to \mathcal{E}$ which describe how effects can modulate coeffects, and vice versa. The specialised distributive law of eq. (2) had κ as right projection π_2.

This graded distributive law provides a way to compose effectful-coeffectful computations modelled as functions (more generally morphisms) of the form $D_r A \to T_e B$. In our semantics (Section 5) this is the interpretation of typing derivations proving judgments of the form $x : [A]_r \vdash t : T_e B$. The composition is defined:

$$\dfrac{D_s A \xrightarrow{g} T_e B \qquad D_r A \xrightarrow{h} T_f C}{D_{\iota(r,e)*s} A \xrightarrow{g^\dagger} D_{\iota(r,e)} T_e A \xrightarrow{\sigma} T_{\kappa(r,e)} D_r A \xrightarrow{h^*} T_{\kappa(r,e)*f} A}$$

where $-^\dagger$ is the extension operation of the graded comonad (essentially, *promotion*) and $-^*$ the extension operation of the graded monad. But this isn't the only way we could combine coeffects and effects—there are other forms of distributive law.

A different interaction and distributive law We now consider an alternate form of interaction between effects and coeffects in our example via a different law, which we compare via the types with the previous one in equation (2):

$$\textit{(previously)} \quad \mathbf{dist}_{r,e,A} : D_{\iota(r,e)} T_e A \to T_e D_r A$$
$$\textit{(alternate)} \quad \mathbf{dist}'_{r,e,A} : T_e D_r A \to D_{\iota(r,e)} T_e A$$

In the previous operation, coeffects are distributed over effects. The *information flow* for coeffects is from the right to left since the coeffect capability provided by the input parameter is $\iota(r, e)$, *i.e.*, calculated from the coeffect r on the output and the effect e (which is preserved from left-to-right). In the alternate rule, we see two immediate differences. Firstly, the order of T and D is changed—effects are now distributed over coeffects. Secondly, the information flow has changed, where the coeffect parameter r is provided by the coeffect capability of the input (left of the arrow).

In our exception/bounded-reuse example, the interesting case for $\mathbf{dist}'$ is when $e = \bot$. This specialises to the rule typed:

$$\mathbf{dist}'_{r,\bot,A} : T_\bot D_r A \to D_1 T_\bot A$$

The meaning is clear from the types, if we have a definitely failing computation then the r-copies of value A inside cannot be accessed and therefore we need only produce one copy of the exception.

The above shows two possible distributive laws, but there are more choices possible due to the two ways of ordering effects over coeffects and four different kinds of information flow relating to the position of r and e either on the input or output. In our framework, we thus provide eight different forms of distributive law, which we present in the next section.

3. Syntax and Type System

In order to show how to combine effects and coeffects in actual programs we consider a λ-calculus that combines effects in the style of Moggi's monadic metalanguage [?], with explicit terms for managing effects via monadic constructions, and coeffects in the style of the coeffect calculus from [?], with explicit terms for managing coeffects via comonadic constructions.

The syntax of the calculus is given in Figure 1. We identify four parts of the calculus: pure, effectful, coeffectful and distributive. The pure fragment corresponds to the standard terms of the λ-calculus. The effectful fragment includes the constructions for managing effects: the *unit* construct written $\langle t \rangle$ for lifting a term t to a trivially effectful computation and the construct $\mathbf{let}\ \langle x \rangle = t\ \mathbf{in}\ t$ for sequentially composing monadic, effectful computations (which we refer to as *letT*). The coeffectful fragment includes constructions for managing coeffects: the *promotion* construct $[t]$ induces requirements on the context (corresponding to comonadic *comultiplication*) and $\mathbf{let}\ [x] = t\ \mathbf{in}\ t$ for discharging coeffect requirements (referred to as *letD*). Finally, the distributive fragment includes a family of operations $\mathbf{dist}^\phi$ for the *distributive laws*, and a family of possibly effectful and/or coeffectful *operations* **op**.

$$
\begin{aligned}
t ::= {} & x \mid \lambda x.t \mid t\ t \mid [t] \mid \langle t \rangle \mid \mathbf{let}\ \langle x \rangle = t\ \mathbf{in}\ t \\
& \mid \mathbf{let}\ [x] = t\ \mathbf{in}\ t \mid \mathbf{dist}^\phi \mid \mathbf{op} && (\phi \in \mathrm{FMT}) \\
A, B, C ::= {} & o \mid A \to A \mid D_r A \mid T_e A && (e \in E) \\
\Gamma, \Delta ::= {} & \emptyset \mid x : A, \Gamma \mid x : [A]_r, \Gamma && (r \in R)
\end{aligned}
$$

Figure 1. Grammar for terms, types and typing environments.

The semantics of the calculus will be described by providing a syntactic equational theory in Section 4 and a categorical semantics in Section 5. We focus here on the type system for explicitly tracking effects and coeffects, which we now define formally.

3.1 Effects and Coeffects

The calculus is built upon the following data specifying effects, coeffects and their interactions.

Effects We follow the approach of **?**], identifying effects with elements of a preordered monoid.

A *preordered monoid* is a tuple $\mathcal{E} = (E, \leq, 1, \bullet)$ such that $(E, 1, \bullet)$ is a monoid, $(E, \leq)$ is a preordered set and $\bullet$ is monotone with respect to $\leq$ in each argument, *i.e.*, $e \leq f$ and $g \leq h$ implies $e \bullet g \leq f \bullet h$. The preorder $(E, \leq)$ of $\mathcal{E}$ is denoted by E. The order-opposite of $\mathcal{E}$ is again a preordered monoid, denoted by $\mathcal{E}^{op}$.

The calculus is parameterised by a preordered monoid $\mathcal{E}$, the *effect monoid*, whose elements are *effects* ranged over by e, f, g.

Coeffects Similarly, we follow the approach by **?**], **?**], and **?**], identifying coeffects with the elements of a semiring.

A *preordered semiring* is a tuple $\mathcal{R} = (R, \leq, 0, +, 1, *)$ where $(R, \leq)$ is a preordered set, $(R, 0, +, 1, *)$ is a semiring and $+, *$ are monotone with respect to $\leq$ in both arguments. The additive and multiplicative preordered monoids of $\mathcal{R}$ are denoted by $\mathcal{R}^+$ and $\mathcal{R}^*$, respectively. The latter is sometimes denoted by $\mathcal{R}$ when no confusion occurs. The preorder part $(R, \leq)$ of $\mathcal{R}$ is denoted by R. The order-opposite of $\mathcal{R}$ is again a preordered semiring, which is denoted by $\mathcal{R}^{op}$.

The calculus is parameterised by a preordered semiring $\mathcal{R}$, the *coeffect semiring*, with *coeffect* elements ranged over by r, s, t.

Note the asymmetry between effects and coeffects: coeffects are structured by a (preordered) semiring, whilst effects are structured instead only by a (preordered) monoid. This asymmetry arises naturally as a consequence of the λ-calculus typing judgments taking many inputs (free-variable assumptions) to a single output. Thus, the input structure "on the left" is richer, capturing multiple values, contrasting with the single output "on the right". Since coeffects are primarily a property of the input/context, they have a richer structure to match.

Distributive law format As we discussed in Section 2.3, there are several possibilities on the format of the distributive law, depending on the purpose of the calculus and the role of the graded (co)monadic types. To cover them systematically, we introduce a symbolic representation of all formats of the distributive law.

Definition 1. A *distributive law format* is an element ϕ in the eight-element set $\mathrm{FMT} = \{\textsc{ll}, \textsc{lr}, \textsc{rl}, \textsc{rr}\} \times \{\textsc{td}, \textsc{dt}\}$.

The elements $\textsc{td}$ and $\textsc{dt}$ express that the distributive law is either T-over-D (effects over coeffects) or D-over-T (coeffects over effects). The elements $\textsc{ll}, \textsc{lr}, \textsc{rl}, \textsc{rr}$ represent the position of T_e and D_r in a distributive law. We discuss this further in Section 3.2.1.

Effect-coeffect interaction by matched pairs A key novel part of our calculus is the presence of two operations ι and κ which combine the elements of the effect monoid $\mathcal{E}$ with those of the coeffect semiring $\mathcal{R}$. These operations must respect a particular structure to fit the distributive law we present in the next section. Interestingly, this structure corresponds to well-known structures from quantum groups and group theory: *matched pairs* and *Zappa-Szép products*. We first define the primitive form of matched pair.

Definition 2. Let $\mathcal{R}, \mathcal{E}$ be preordered monoids $(\mathcal{E}, \leq, 1, \bullet)$ and $(\mathcal{R}, \leq, 1, *)$. An $\mathcal{R}, \mathcal{E}$*-matched pair* [**?**] is a pair of monotone functions $\iota : R \times E \to R$ and $\kappa : R \times E \to E$ such that

$$\begin{array}{ll} \iota(r, 1) = r & \iota(r, e \bullet f) = \iota(\iota(r, e), f) \\ \iota(1, e) = 1 & \iota(r * s, e) = \iota(r, \kappa(s, e)) * \iota(s, e) \\ \kappa(1, e) = e & \kappa(r * s, e) = \kappa(r, \kappa(s, e)) \\ \kappa(r, 1) = 1 & \kappa(r, e \bullet f) = \kappa(r, e) \bullet \kappa(\iota(r, e), f) \end{array}$$

Upon this definition we define the concept of $\mathcal{R}, \mathcal{E}$-matched pair for a given distributive law format ϕ. Below, for a preordered monoid, by $\overline{\mathcal{E}}$ we mean $\mathcal{E}$'s reverse monoid, whose multiplication is given by $e \bullet_{\overline{\mathcal{E}}} f = f \bullet e$ and likewise for coeffects $\overline{\mathcal{R}}$ is the reverse monoid with $r *_{\overline{\mathcal{R}}} s = s * r$.

Definition 3. Let $\mathcal{R}, \mathcal{E}$ be preordered monoids, $\iota : R \times E \to R$ and $\kappa : R \times E \to E$ be monotone functions and ϕ be a distributive law format. We say that (ι, κ) is an $\mathcal{R}, \mathcal{E}$*-matched pair for the format* ϕ if the pair (ι, κ) is a $M\phi$-matched pair, where $M\phi$ is looked up from the following table:

$$M\phi = \left(\begin{array}{c|cccc} \phi & \textsc{ll} & \textsc{lr} & \textsc{rl} & \textsc{rr} \\ \hline \textsc{td} & \overline{\mathcal{R}}, \overline{\mathcal{E}} & \overline{\mathcal{R}}, \mathcal{E} & \mathcal{R}, \overline{\mathcal{E}} & \mathcal{R}, \mathcal{E} \\ \textsc{dt} & \mathcal{R}, \mathcal{E} & \mathcal{R}, \overline{\mathcal{E}} & \overline{\mathcal{R}}, \mathcal{E} & \overline{\mathcal{R}}, \overline{\mathcal{E}} \end{array} \right)$$

For instance, (ι, κ) is an $\mathcal{R}, \mathcal{E}$-matched pair for the format $(\textsc{rl}, \textsc{dt})$ if (ι, κ) is an $\overline{\mathcal{R}}, \mathcal{E}$-matched pair in the sense of Definition 2. The calculus is then parameterised by an $\mathcal{R}^{op}, \mathcal{E}$-matched pair (ι, κ) for the format ϕ chosen for the calculus, using the multiplicative preordered monoid $\mathcal{R}^*$ in the matched pair axioms (Definition 2).

To summarise, the parameters of our calculus comprise: (1) a coeffect semiring $\mathcal{R}$ and an effect monoid $\mathcal{E}$ (2) a distributive law format ϕ and an $\mathcal{R}^{op}, \mathcal{E}$-matched pair (ι, κ) for ϕ, and (3) operations $\mathbf{op} : A_{\mathbf{op}}$.

3.2 Type System

Typing judgments have the shape $\Gamma \vdash t : A$ where A is a *type* and Γ is a *typing environment*. The syntax of types and typing environments is described in Figure 1. Types comprise simple types built over base types o and an *effect graded monad* type constructor $T_e A$, graded over the effect e, and a *coeffect graded comonad* type constructor $D_r A$, graded over the coeffect r.

Typing environments comprise type assignments to variables. Environments are treated as sets, therefore an exchange rule (permuting the order of assignments) is implicit, and variables can only appear at most once in an environment. In the categorical semantics (Section 5), exchange is made explicit to model environments.

Environments comprise two kinds of type assignment: *linear* assignments of the shape $x : A$, and *discharged* assignments of the shape $x : [A]_r$ that are graded over a coeffect r. Discharged assignments have been introduced in some presentations of linear logic [**?**] as a technical artifact useful for implicitly managing variables in environments—without using explicit contraction and weakening rules. We write $[\Gamma]$ for an environment Γ which consists only of discharged assignments, and $[\Gamma]_r$ when all such discharged assignments have the same coeffect r.

Before introducing the type system, we lift coeffect operations $+$ and $*$ to typing environments as follows:

Definition 4 (Summing and scalar multiplication on environments)**.** We say that Γ, Δ are *summable*, if for any $x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$, there exists (necessarily unique) type A and $r, s \in R$ such that $\Gamma(x) = [A]_r$ and $\Delta(x) = [A]_s$. The *sum* $\Gamma + \Delta$ of two summable typing environments Γ, Δ is defined as follows:

$$\begin{array}{rcll} \emptyset + \Delta = \Delta & & \Gamma + \emptyset = \Gamma & \\ (x : A, \Gamma) + \Delta & = & x : A, (\Gamma + \Delta) & \text{if } x \notin FV(\Delta) \\ \Gamma + (x : A, \Delta) & = & x : A, (\Gamma + \Delta) & \text{if } x \notin FV(\Gamma) \\ x : [A]_r, \Gamma + x : [A]_s, \Delta & = & x : [A]_{r+s}, (\Gamma + \Delta) & \end{array}$$

$$\text{ax}\ \frac{}{x : A \vdash x : A} \qquad \text{sub}\ \frac{\Gamma \vdash t : A \quad \Gamma' <: \Gamma \quad A <: B}{\Gamma', [\Delta]_0 \vdash t : B}$$

$$\text{abs}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \text{app}\ \frac{\Gamma \vdash t : A \to B \quad \Delta \vdash t' : A}{\Gamma + \Delta \vdash t\, t' : B}$$

$$\text{unit}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \langle t \rangle : T_1 A} \qquad \text{letT}\ \frac{\Gamma \vdash t_1 : T_e A \quad \Delta, x : A \vdash t_2 : T_f B}{\Gamma + \Delta \vdash \textbf{let}\ \langle x \rangle = t_1\ \textbf{in}\ t_2 : T_{e \bullet f} B}$$

$$\text{der}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B} \qquad \text{pr}\ \frac{[\Gamma] \vdash t : B}{r{*}[\Gamma] \vdash [t] : D_r B}$$

$$\text{letD}\ \frac{\Gamma \vdash t_1 : D_r A \quad \Delta, x : [A]_r \vdash t_2 : B}{\Gamma + \Delta \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : B}$$

$$\vdash \textbf{dist}^{\phi} : F^{\phi}_{r,e} A \to G^{\phi}_{r,e} A \qquad\qquad \vdash \textbf{op} : A_{\mathsf{op}}$$

Figure 2. Typing rules: (a) pure, (b) effect rules, (c) coeffect rules, (d) distributivity and operations.

$$\text{s-ax}\ \frac{}{A <: A} \qquad \text{s-arr}\ \frac{A' <: A \quad B <: B'}{A \to B <: A' \to B'} \qquad \text{s-rf}\ \frac{}{\Gamma <: \Gamma}$$

$$\text{s-D}\ \frac{A <: A' \quad s \le r}{D_r A <: D_s A'} \qquad \text{s-d}\ \frac{A <: A' \quad s \le r}{[A]_r <: [A']_s}$$

$$\text{s-T}\ \frac{A <: A' \quad e \le f}{T_e A <: T_f A'} \qquad \text{s-c}\ \frac{\Gamma <: \Delta \quad B <: A}{\Gamma, x : B <: \Delta, x : A}$$

Figure 3. Subeffecting rules

ϕ	T over D (TD)		D over T (DT)	
	$F^{\phi}_{r,e}$	$G^{\phi}_{r,e}$	$F^{\phi}_{r,e}$	$G^{\phi}_{r,e}$
LL	$T_e \quad D_r$	$\to D_{\iota(r,e)} T_{\kappa(r,e)}$	$D_r \quad T_e$	$\to T_{\kappa(r,e)} D_{\iota(r,e)}$
RR	$T_{\kappa(r,e)} D_{\iota(r,e)}$	$\to D_r \quad T_e$	$D_{\iota(r,e)} T_{\kappa(r,e)}$	$\to T_e \quad D_r$
LR	$T_e \quad D_{\iota(r,e)}$	$\to D_r \quad T_{\kappa(r,e)}$	$D_{\iota(r,e)} T_e$	$\to T_{\kappa(r,e)} D_r$
RL	$T_{\kappa(r,e)} D_r$	$\to D_{\iota(r,e)} T_e$	$D_r \quad T_{\kappa(r,e)}$	$\to T_e \quad D_{\iota(r,e)}$

Figure 4. A zoo of distributive laws $\textbf{dist}^{\phi} : F^{\phi}_{r,e} a \to G^{\phi}_{r,e} A$

Multiplication $r{*}[\Gamma]$ of a coeffect r by an environment $[\Gamma]$ of discharged variable assignments (of the form $x_i : [A_i]_{s_i}$) is defined:

$$r * \emptyset = \emptyset \qquad r * (x : [A]_s, [\Gamma]) = x : [A]_{r*s}, r * [\Gamma]$$

With these operations, we now present the typing rules in Figure 2. There are four sets of rules: (a) pure rules, (b) effect rules, (c) coeffect rules, (d) distributivity and operations (Section 3.2.1).

(a) pure rules The pure fragment corresponds to the typing of the linear λ-calculus (without exponentials). It is worth noticing that, in the (app) rule, type environments are composed using the sum operation defined above. Moreover, we have a subtyping rule (sub) that permits casting to a super-effect in the type and to subcoeffects in the type environment. The subtyping relation on types and environments is defined in Figure 3. This relation is defined in terms of the effect monoid and coeffect semiring preorders, where it is covariant in effects whilst contravariant in coeffects (and thus in discharged assignments). It is extended to environments and the other types following the traditional structure of subtyping.

We can weaken and contract discharged variables at several places in a derivation tree. The strength of these operations depends on the coeffect semiring. *Weakening* of environments with 0-discharged variables is always permitted at the rule (sub). This limited form of weakening is analogous to the weakening of environments with !-types in linear logic. When 0 is the least element in the coeffect semiring $\mathcal{R}$, further application of (sub) allows us to weaken 0-discharged variables to r-discharged variables. Therefore in such a situation one can virtually weaken environments with arbitrary r-discharged variables. In the rules (app), (letT) and (letD), discharged variables in the environments of the subtrees are added up by the environment summation. When the coeffect addition $+$ is idempotent, the environment sum of Δ and Δ' now *contracts* the discharged variables that are common in Δ and Δ'.

(b) effect rules The effectful fragment corresponds to the standard rules for monads in the computational λ-calculus but with explicit effects. These rules are those presented by **?**]. The (unit) rule introduces the unit of the monad graded with the 1 effect, and (letT) composes effects via the $\bullet$ operation.

(c) coeffect rules Following **?**], the rules for coeffects correspond to a decoration of the rules for linear logic. The (der) rule introduces a trivial coeffect 1 on the left by discharging the assignment of a variable in the environment, and rule (pr) introduces coeffects on the right by introducing a graded coeffect comonad D_r on the type—this rule also requires the environment to consist only of discharged assumptions, and it multiplies the coeffects on the discharged formulas by r. Finally, we have a rule (letD) that replaces a variable x which is a discharged assignment with a comonadic term. This rule provides sequential composition of a computation that provides a coeffect capability with one that has a matching coeffect requirement.[5]

The (**op**) rule introduces operations to the language which may be effectful/coeffectful, of type A_{op} which can be a function, monadic, or comonadic type. Any function-typed operation can then be applied using the standard application rule.

Note on linearity and coeffects The connection between linear types and coeffects is established in recent work [**? ? ?**], where coeffects arise as an indexed generalisation of the exponential !. We follow this tradition. Hence, the comonadic fragment of our calculus reflects the constructions of linear logic. It is worth noting that each instantiation of $\mathcal{E}$ and $\mathcal{R}$ corresponds to a refinement of the simply typed lambda calculus. When the coeffect semiring $\mathcal{R}$ is instantiated with an idempotent addition operation we obtain an analysis that is not *quantitative* in the usual sense of linear logic. For example, Section 6.1 shows information flow coeffects which are non-quantitative as there is a lattice semiring for security labels with the addition $+$ as lattice join, which is thus idempotent.

3.2.1 Distributive Laws

Since the goal is for our calculus to be flexible in the interaction of effects and coeffects, the distributive law syntax $\textbf{dist}^{\phi}$ is parameterised by a distributive law format ϕ indicating which law to use (we omit ϕ when clear from the context, *e.g.*, from the typing).

Figure 4 defines our "zoo" of distributive laws which is derived systematically along the two axes which define formats ϕ. The horizontal axis defined over {TD, DT} is a binary choice between whether effects T are distributed over coeffects D or the converse, coeffects D over effects T. The vertical axis over {LL, RR, LR, RL} relates to the direction of information flow for effects/coeffects as enforced by the matched pair operations ι, κ. In the case where effects e are parameters on the left (*i.e.*, T_e appears to the left of an

[5] Notice that when the term t_1 in (letD) is a variable, then (letD) corresponds to introducing a coeffect graded comonad constructor on the left. The usual *dereliction* rule of linear logic for ! can be obtained by combining the rule (der) with the rule (letD) while *promotion* can be obtained by combining the rule (pr) with (several applications of) the rule (letD). Contraction and weakening are instead implicitly used in the management of environments.

arrow) and $\kappa(e, r)$ appears on the right then the flow of information with respect to effects is from the left, marked L$*$; conversely if T_e is on the right and $T_{\kappa(e,r)}$ on the left then effect information flows from the right, marked R$*$. The information flow for coeffects varies in the same fashion with D_r and $D_{\iota(e,r)}$ on either side marked by $*$L and $*$R. Definition 3 gave the axioms on $\mathcal{R}^{op}, \mathcal{E}$-matched pairs that are induced by the choice of ϕ.

For both effects and coeffects, the choice of the information flow direction collapses when ι and/or κ are projections.

Effectful-coeffectful let-binding We have chosen to include two different *let*-binding constructions: one for effects (letT) and one for coeffects (letD). In many concrete situations, it is however convenient to have a *let*-binding that is effectful and coeffectful at the same time. We introduce syntactic sugar for their composition:

$$(\textbf{let } \langle[x]\rangle = t_1 \textbf{ in } t_2) \;:=\; \textbf{let } \langle y\rangle = t_1 \textbf{ in let } [x] = y \textbf{ in } t_2$$

where y is fresh in t_2. The derived typing is:

$$\text{letTD}\;\frac{\Gamma \vdash t_1 : T_e D_r A \qquad \Delta, x : [A]_r \vdash t_2 : T_f B}{\Gamma + \Delta \vdash \textbf{let } \langle[x]\rangle = t_1 \textbf{ in } t_2 : T_{e \bullet f} B} \tag{3}$$

In the dual situation of a computation of type $D_r T_e$, the most useful composition of effectful and coeffectful **let** is:

$$(\textbf{let } [\langle x\rangle] = t_1 \textbf{ in } t_2) \;:=\; \textbf{let } [y] = t_1 \textbf{ in } [\textbf{let } \langle x\rangle = y \textbf{ in } t_2]$$

where again y is fresh in t_2. This gives the typing:

$$\text{letDT}\;\frac{\Gamma \vdash t_1 : D_r T_e A \qquad \Delta, x : A \vdash t_2 : T_f B}{\Gamma + \Delta \vdash \textbf{let } [\langle x\rangle] = t_1 \textbf{ in } t_2 : D_r T_{e \bullet f} B} \tag{4}$$

The form is a little different from (letTD), but can be understood as giving a way to compose effects underneath a coeffect capability.

4. Equational Theory

We equip our type system with a syntactic equational theory further refinable into a rewrite system (operational semantics). Section 5 shows the corresponding categorical semantics.

4.1 Substitution

In defining the equational theory $\equiv$ we consider $\beta\eta$-equality (as pairs of introduction-elimination and elimination-introduction rules) and associativity equalities for the pure, effectful, coeffectful, and distributive parts of our calculus. Some equations rely on the syntactic notion of (capture avoiding) substitution which for our calculus is the standard one for the λ-calculus and recursively defined over all other terms in a standard way.

We show that substitution is type preserving. Like several linear type systems, our calculus distinguishes two kinds of type assignments for variables: normal (linear) and discharged (coeffectful). We show that substitution is preserved when the variable to be substituted for belongs to either one of these assignments. Type-preservation for substitution of linearly-typed variables is given by the following lemma.

Lemma 1 (Linear substitution)**.** *Let* $\Gamma, x : A \vdash t_2 : B$ *and* $\Delta \vdash t_1 : A$*. Then,* $\Gamma + \Delta \vdash t_2[t_1/x] : B$

The proof employs the commutativity and associativity of $+$. The following lemma shows substitution is type-preserving when instead the substituted variable is a discharged assignment:

Lemma 2 (Coeffectful substitution)**.** *Let* $\Gamma, x : [A]_r \vdash t_2 : B$ *and* $[\Delta] \vdash t_1 : A$*. Then,* $\Gamma + r * [\Delta] \vdash t_2[t_1/x] : B$

In order to prove the substitution lemmas above, we use several of the algebraic properties of the coeffect semiring. In fact, the semiring structure emerges naturally by the requirements imposed by the substitution on the typing environments.

$$\begin{array}{rcll}
(\lambda x.t_2)t_1 & \equiv & t_2[t_1/x] & (\beta)\\
\lambda x.tx & \equiv & t & [x\#t]\ (\eta)\\[6pt]
\textbf{let } \langle x\rangle = \langle t_1\rangle \textbf{ in } t_2 & \equiv & t_2[t_1/x] & (\beta\text{-eff})\\
\textbf{let } \langle x\rangle = t_1 \textbf{ in } \langle x\rangle & \equiv & t_1 & (\eta\text{-eff})\\
(\textbf{let } \langle x\rangle = t_1 \textbf{ in let } \langle y\rangle = t_2 \textbf{ in } t_3) & & & (\text{assoc-eff})\\
& \equiv & \textbf{let } \langle y\rangle = (\textbf{let } \langle x\rangle = t_1 \textbf{ in } t_2) \textbf{ in } t_3 & [x\#t_3]\\[6pt]
\textbf{let } [x] = [t_1] \textbf{ in } t_2 & \equiv & t_2[t_1/x] & (\beta\text{-coeff})\\
\textbf{let } [x] = t_1 \textbf{ in } [x] & \equiv & t_1 & (\eta\text{-coeff})\\
\textbf{let } [x] = [t_1] \textbf{ in } [t_2] & \equiv & [\textbf{let } [x] = [t_1] \textbf{ in } t_2] & (\text{assoc-coeff})\\[6pt]
f\,(\textbf{let } [x] = t_1 \textbf{ in } t_2) & \equiv & \textbf{let } [x] = t_1 \textbf{ in } f\,t_2 & [x\#f]\ (\text{app}\leftrightarrow\text{letD})\\
\langle \textbf{let } [x] = t_1 \textbf{ in } t_2\rangle & \equiv & \textbf{let } [x] = t_1 \textbf{ in } \langle t_2\rangle & (\text{unit}\leftrightarrow\text{letD})
\end{array}$$

Figure 5. Equational theory for the effect-coeffect calculus

4.2 Equations

We now introduce the equational theory $\equiv$. This is defined by the set of rules given in Figure 5. To avoid variables in terms being unintentionally captured, we use the *freshness* predicate $\#$ to denote that a variable does not appear free or bound in a term.

The equational theory is defined over typing derivations, like the interpretation given in Section 5, but to keep the presentation compact we will describe it only on terms. The equational theory is well-defined in the sense that, if $t \equiv u$, then we can give to t and u the same type in the same environment. As in the simply-typed λ-calculus,this is true only under some additional assumptions on the typability of the different components of the rule, *e.g.*, the (η) rule in Figure 5 is well-defined only if we can assign to the term t a functional type $A \to B$. We omit here most of these assumptions for brevity, but highlight a few examples.

Pure fragment: The (β) rule is well defined following Lemma 1 under the assumptions $\Gamma, x : A \vdash t_2 : B$ and $\Delta \vdash t_1 : A$. The (η) rule follows under the assumption $\Gamma \vdash t : A \to B$.

Effectful fragment: The equational theory for the effectful fragment follows the standard one of the monadic metalanguage by **?**], and others [**? ?**], modulo grading. In particular, the (β*-eff*) rule relies on the left-unit axiom of the monoid, (η*-eff*) on the right-unit of the monoid, and (*assoc-eff*) on its associativity.

Coeffectful fragment: The equational theory for the coeffectful fragment is similar to the one presented by **? ?**] but with some adaptation due to the difference in syntax. We show here how to derive some of the rules. The fact that the (β-coeff) rule $\textbf{let } [x] = [t_1] \textbf{ in } t_2 \equiv t_2[t_1/x]$ is well-defined follows from the typing of its left-hand side:

$$\text{letD}\;\frac{\text{pr}\;\dfrac{[\Gamma] \vdash t_1 : A}{r*[\Gamma] \vdash [t_1] : D_r A} \qquad \Delta, x : [A]_r \vdash t_2 : B}{r*[\Gamma] + \Delta \vdash \textbf{let } [x] = [t_1] \textbf{ in } t_2 : B}$$

and from the coeffectful substitution (Lemma 2) for its right-hand side—assuming the premises of the derivation above. Similarly, (η-coeff) is well-defined by the following (partial) type derivation:

$$\text{letD}\;\frac{\Gamma \vdash t : D_r A \qquad \text{pr}\;\dfrac{x : [A]_1 \vdash x : A}{x : [A]_r \vdash [x] : D_r A}}{\Gamma \vdash (\textbf{let } [x] = t \textbf{ in } [x]) \equiv t : D_r A}$$

assuming $\Gamma \vdash t : D_r A$. The other rules are similarly well-typed using the properties of the coeffect semiring.

Distributive fragment: The equational theory for the distributive fragment splits into two sets of axioms depending on whether a TD or DT axiom is being used. These are shown in Figure 6. To be well-typed, these axioms rely on the properties of the $\mathcal{R}^{op}, \mathcal{E}$ matched pair (ι, κ) (see Definition 3).

$$
\begin{array}{rcll}
\mathbf{dist}\;[\langle t\rangle] & \equiv & \langle [t]\rangle & \text{(eff1-dist)}\\
\mathbf{let}\;\langle [x]\rangle = \mathbf{dist}\,t\;\mathbf{in}\;\langle x\rangle & \equiv & \mathbf{let}\;[x] = t\;\mathbf{in}\;x & \text{(coeff1-dist)}\\
\mathbf{let}\;\langle [x]\rangle = \mathbf{dist}[t]\;\mathbf{in}\;\langle [[x]]\rangle & \equiv & \mathbf{dist}\,[\mathbf{dist}\,[t]] & \text{(coeff}*\text{-dist)}\\
\mathbf{let}\;\langle x\rangle = \mathbf{dist}(t)\;\mathbf{in}\;\mathbf{dist}(x) & \equiv & & \text{(eff}\bullet\text{-dist)}\\
 & & \mathbf{dist}(\mathbf{let}\;[\langle x\rangle] = t\;\mathbf{in}\;x) &
\end{array}
$$

(a) Equations for DT distributive laws

$$
\begin{array}{rcll}
[\langle t\rangle] & \equiv & \mathbf{dist}\;\langle [t]\rangle & \text{(eff1-dist)}\\
\mathbf{let}\;\langle [x]\rangle = t\;\mathbf{in}\;\langle x\rangle & \equiv & \mathbf{let}\;[x] = \mathbf{dist}\,t\;\mathbf{in}\;x & \text{(coeff1-dist)}\\
\mathbf{let}\;[x] = \mathbf{dist}\,t\;\mathbf{in}\;[[x]] & \equiv & & \text{(coeff}*\text{-dist)}\\
\multicolumn{3}{r}{\mathbf{let}\;[y] = \mathbf{dist}(\mathbf{let}\;\langle [x]\rangle = t\;\mathbf{in}\;\langle [[x]]\rangle)\;\mathbf{in}\;[\mathbf{dist}\,y]} & \\
\mathbf{dist}(\mathbf{let}\;\langle x\rangle = t\;\mathbf{in}\;x) & \equiv & & \text{(eff}\bullet\text{-dist)}\\
\multicolumn{3}{r}{\mathbf{let}\;[\langle x\rangle] = \mathbf{dist}(\mathbf{let}\;\langle y\rangle = t\;\mathbf{in}\;\langle\mathbf{dist}\;y\rangle)\;\mathbf{in}\;x} &
\end{array}
$$

(b) Equations for TD distributive laws

Figure 6. Equational theory for **dist**

For example, the DT rule (eff1-dist) $\mathbf{dist}\;[\langle t\rangle] \equiv \langle [t]\rangle$ says that **dist** can swap any graded comonad D with the graded monadic unit $\langle - \rangle$. This is justified by the following derivation:

$$
\text{dist}\;\dfrac{\text{pr}\;\dfrac{\text{unit}\;\dfrac{[\Gamma]\vdash t : A}{[\Gamma]\vdash \langle t\rangle : T_1 A}}{\iota(r,1)*[\Gamma]\vdash [\langle t\rangle] : D_{\iota(r,1)}T_1 A}}{\iota(r,1)*[\Gamma]\vdash \mathbf{dist}^{(\text{LR,DT})}\;[\langle t\rangle]\equiv\langle [t]\rangle : T_{\kappa(r,1)}D_r A}\quad\text{(eff1-dist)}
$$

The typing of the two sides of the equation are equal since by the definition of a $\mathcal{R}^{op},\mathcal{E}$ matched pair we have $\iota(r,1) = r$ and $\kappa(r,1) = 1$. The rule thus says that a coeffect can be distributed over any trivial effect.

In the case of TD distributive laws, the order of effects and coeffects is flipped, leading to a mirrored axiom, with typing derivation:

$$
\text{dist}\;\dfrac{\text{unit}\;\dfrac{\text{pr}\;\dfrac{[\Gamma]\vdash t : A}{\iota(r,1)*[\Gamma]\vdash [t] : D_{\iota(r,1)}A}}{\iota(r,1)*[\Gamma]\vdash \langle [t]\rangle : T_1 D_{\iota(r,1)}A}}{\iota(r,1)*[\Gamma]\vdash \mathbf{dist}^{(\text{LR,TD})}\;\langle [t]\rangle\equiv [\langle t\rangle] : D_r T_{\kappa(r,1)}A}\quad\text{(eff1-dist)}
$$

The (eff1-dist) rule has a coeffectful counterpart in (coeff1-dist): $\mathbf{let}\;\langle [x]\rangle = \mathbf{dist}\,t\;\mathbf{in}\;\langle x\rangle \;\equiv\; \mathbf{let}\;[x] = t\;\mathbf{in}\;x$ (for DT distributions). This axiom is typed with the following (partial) derivation for its left-hand side:

$$
\text{letTD}\;\dfrac{\text{dist}\;\dfrac{\Gamma\vdash t : D_{\iota(1,e)}T_e A}{\Gamma\vdash \mathbf{dist}^{(\text{LR,DT})}\;t : T_{\kappa(1,e)}D_1 A}\qquad \text{unit}\;\dfrac{x : [A]_1\vdash x : A}{x : [A]_1\vdash \langle x\rangle : T_1 A}}{\Gamma\vdash \mathbf{let}\;\langle [x]\rangle = \mathbf{dist}^{(\text{LR,DT})}\;t\;\mathbf{in}\;\langle x\rangle : T_{\kappa(1,e)\bullet 1}A}
$$

and the judgment $\Gamma \vdash \mathbf{let}\;[x] = t\;\mathbf{in}\;x : T_e A$ for its right-hand side. Again, the two types are the same because by definition of $\mathcal{R}^{op},\mathcal{E}$ matched pair we have $\iota(1,e) = 1$ and $\kappa(1,e) = e$. So, this rule says that distributing a trivial coeffect over an effect then discharging it is equivalent to just discharging the trivial coeffect (since on the right-hand side, x is bound and discharged without any operations on it). The fact that the other rules are well-defined can be shown in a similar way.

In the case of (eff$\bullet$-dist) and (coeff$*$-dist), both equate a double use of **dist** (with some additional composition of effects/coeffects) with a single use. For example, each side of (eff$\bullet$-dist) for (DT, LL) takes a term $\Gamma \vdash t : D_r T_e T_f A$ and produces a term whose type is of the form $T_{\kappa(r,e\bullet f)}D_{\iota(r,e\bullet f)}A$ by either applying **dist** twice ($DTT \to TDT \to TTD$) and sequentially composing effects (the left-hand side of the rule) or sequentially composing effects then distributing once (right-hand side). Due to space limitations we omit the rest of the typings for the equational theory.

Subtyping: For every typing rule, there is a non-syntax directed equation stating that it commutes with (sub). That is, for an n-ary typing rule Ψ with premises provided by derivation trees $\pi_1 ... \pi_n$ then for each $i \in \{1, .., n\}$ there is an equation $\Psi(\pi_1, .., sub(\pi_i), .., \pi_n) \equiv sub(\Psi(\pi_1, .., \pi_i, .., \pi_n))$. We omit these here for brevity. The accompanying technical report [**?**] provides the full set of equations.

Before moving on, it is worth noting that by orienting the equations in Figures 5 and 6 we obtain an operational semantics for the operation-free calculus. Moreover, since the equational theory is well-defined with respect to typing, this operational semantics enjoys type preservation. We next introduce the denotational, categorical semantics before showing example instantiations in Section 6.

5. Categorical Semantics

We give a categorical semantics to our calculus, built upon its parameters. Recall that its parameters comprise (1) a coeffect semiring $\mathcal{R}$ and an effect monoid $\mathcal{E}$, (2) a distributive law format ϕ and an $\mathcal{R}^{op},\mathcal{E}$-matched pair (ι,κ) for ϕ, and (3) operations $\mathbf{op} : A_{\mathsf{op}}$.

Our calculus is based on the intuitionistic linear lambda calculus. We thus fix an underlying symmetric monoidal closed category $(\mathcal{C}, \mathbf{I}, \otimes, \multimap)$ which provides the semantics of functions, environments, and abstraction. To interpret the (co)effect-annotated types T_e, D_r and distributive laws we introduce the following structures:

1. An $\mathcal{E}$-graded strong monad T on $\mathcal{C}$ (Section 5.1).
2. An $\mathcal{R}^{op}$-graded exponential comonad D on $\mathcal{C}$ (Section 5.2).
3. An (ι,κ)-distributive law σ for each ϕ (Section 5.3).

5.1 Graded Strong Monads

We first review the primitive form of graded monads. By $[\mathcal{C},\mathcal{C}]$ we mean the category of endofunctors on $\mathcal{C}$ and natural transformations between them. We equip it with the strict monoidal structure given by the identity functor and the functor composition. Then an *$\mathcal{E}$-graded monad* on $\mathcal{C}$ is given by a lax monoidal functor of type $\mathcal{E} \to ([\mathcal{C},\mathcal{C}], \mathrm{Id}, \circ)$. This terse definition is expanded to the following concrete definition: an $\mathcal{E}$-graded monad on $\mathcal{C}$ consists of the following functor and natural transformations:

$$
\begin{array}{llrcl}
\text{Functor} & T : & E & \to & [\mathcal{C},\mathcal{C}]\\
\text{Unit} & \eta_A : & A & \to & T1A\\
\text{Multiplication} & \mu_{e,f,A} : & Te(TfA) & \to & T(e\bullet f)A
\end{array}
$$

making the following diagrams commute in $[\mathcal{C},\mathcal{C}]$:

$$
\begin{tikzcd}
Te \arrow[r, "\eta\circ Te"] \arrow[d, "Te\circ\eta"'] \arrow[dr, equal] & T1\circ Te \arrow[d, "\mu_{1,e}"] \\
Te\circ T1 \arrow[r, "\mu_{e,1}"'] & Te
\end{tikzcd}
\qquad
\begin{tikzcd}
Te\circ Tf\circ Tg \arrow[r, "Te\circ\mu_{f,g}"] \arrow[d, "\mu_{e,f}\circ Tg"'] & Te\circ T(f\bullet g) \arrow[d, "\mu_{e,fg}"] \\
T(e\bullet f)\circ Tg \arrow[r, "\mu_{ef,g}"'] & T(e\bullet f\bullet g)
\end{tikzcd}
$$

The primitive form of graded comonads are dually defined. In the model we write Tf instead of T_f (and similarly for coeffects) to make clear that (co)effect annotations are in fact object parameters.

Recall that interpreting the computational metalanguage using a monad [**?**] requires the extra structure of *tensorial strength* on the monad so that computations can be parameterised by environments. We adopt the same approach for graded monads. To extend them with tensorial strength, we first consider the category $[\mathcal{C},\mathcal{C}]_s$ of strong endofunctors and strong natural transformations between them. We equip it with the strict monoidal structure given by the identity functor and the functor composition. Then we define an *$\mathcal{E}$-graded strong monad* to be a lax monoidal functor of type $\mathcal{E} \to [\mathcal{C},\mathcal{C}]_s$. Concretely speaking, it is an $\mathcal{E}$-graded monad (above)

$$
\begin{tikzcd}
D(r*0) \arrow[r, equal] \arrow[d, "\delta"'] & D0 \arrow[dd, "w"] \\
Dr \circ D0 \arrow[d, "D\circ w"'] & \\
Dr \circ \dot{\mathbf{I}} & \dot{\mathbf{I}} \arrow[l, "m_{r,\mathbf{I}}"]
\end{tikzcd}
\quad
\begin{tikzcd}
D(t*(r+s)) \arrow[r, equal] \arrow[d, "\delta"'] & D(t*r+t*s) \arrow[d, "c"] \\
Dt \circ D(r+s) \arrow[d, "D\circ c"'] & D(t*r)\dot{\otimes}D(t*s) \arrow[d, "\delta\dot{\otimes}\delta"] \\
Dt \circ (Dr\dot{\otimes}Ds) & (Dt\circ Dr)\dot{\otimes}(Dt\circ Ds) \arrow[l, "m_{t,D,D}"]
\end{tikzcd}
\quad
\begin{tikzcd}
D(0*r) \arrow[r, equal] \arrow[d, "\delta"'] & D0 \arrow[dd, "w"] \\
D0 \circ Dr \arrow[d, "w\circ D"'] & \\
\dot{\mathbf{I}} \circ Dr \arrow[r, equal] & \dot{\mathbf{I}}
\end{tikzcd}
\quad
\begin{tikzcd}
D((r+s)*t) \arrow[r, equal] \arrow[d, "\delta"'] & D(r*t+s*t) \arrow[d, "c"] \\
D(r+s) \circ Dt \arrow[d, "c\circ D"'] & D(r*t)\dot{\otimes}D(s*t) \arrow[d, "\delta\dot{\otimes}\delta"] \\
(Dr\dot{\otimes}Ds)\circ Dt \arrow[r, equal] & (Dr\circ Dt)\dot{\otimes}(Ds\circ Dt)
\end{tikzcd}
$$

Figure 7. Diagrammatic axioms for semiring-graded comonads; (co)effect annotations in morphisms are omitted

together with a *tensorial strength* $st_{e,A,B} : A \otimes TeB \to Te(A \otimes B)$, which interacts with T, η, μ in a coherent way (with the usual strong monad axioms [**? ?**], modulo grading).

5.2 Semiring Graded Exponential Comonads

In our calculus, the weakening and contraction is allowed on discharged types $[A]_r$ in the context. To model these facilities, the primitive form of graded comonads is insufficient on its own. We need to give an additional structure describing the interaction between monoidal structure and the comonadic structure that is controlled by the coeffect semiring. This was given by **? ?**], which we introduce below.

By $\mathbf{SMon}_l[\mathcal{C}, \mathcal{C}]$ we mean the category of symmetric lax monoidal endofunctors on $\mathcal{C}$ and monoidal natural transformations between them. We equip it with the pointwise extension of the symmetric monoidal structure on $\mathcal{C}$. Namely, we give the following tensor unit and tensor product on $\mathbf{SMon}_l[\mathcal{C}, \mathcal{C}]$:

$$\dot{\mathbf{I}}A = \mathbf{I}, \quad (F\dot{\otimes}G)A = FA \otimes GA.$$

We give a general definition of an $\mathcal{R}$*-graded exponential comonad* on $\mathcal{C}$ for a preordered semiring $\mathcal{R}$. It consists of a symmetric colax monoidal functor

$$(D, w, c) : \mathcal{R}^+ \to (\mathbf{SMon}_l[\mathcal{C}, \mathcal{C}], \dot{\mathbf{I}}, \dot{\otimes})$$

and a colax monoidal functor

$$(D, \varepsilon, \delta) : \mathcal{R}^* \to (\mathbf{SMon}_l[\mathcal{C}, \mathcal{C}], \mathrm{Id}, \circ)$$

making the diagrams in Figure 7 commute.

A concrete definition of an $\mathcal{R}$-graded exponential comonad consists of the following functor and natural transformations:

Functor	$D :$	R	$\to$	$[\mathcal{C}, \mathcal{C}]$
0-Monoidality	$m_{r,\mathbf{I}} :$	$\mathbf{I}$	$\to$	$Dr\mathbf{I}$
2-Monoidality	$m_{r,A,B} :$	$DrA \otimes DrB$	$\to$	$Dr(A \otimes B)$
Weakening	$w_A :$	$D0A$	$\to$	$\mathbf{I}$
Contraction	$c_{r,s,A} :$	$D(r+s)A$	$\to$	$DrA \otimes DsA$
Dereliction	$\varepsilon_A :$	$D1A$	$\to$	A
Digging	$\delta_{r,s,A} :$	$D(r*s)A$	$\to$	$Dr(DsA)$

making a number of diagrams commute. When the semiring is trivial, it becomes a linear exponential comonad on $\mathcal{C}$.

The categorical semantics of the calculus whose coeffect semiring is $\mathcal{R}$ employs an $\mathcal{R}^{op}$-graded comonad rather than $\mathcal{R}$-graded one. This is because for each ordered coeffect pair $r \le s$ we would like to have the monoidal natural transformation $Ds \to Dr$ embodying the principle that "large also serves as a small". This contravariance also matches the subtyping rule in Figure 3.

5.3 Distributive Laws

A key part of our calculus is the family of distributive operations which are the direct counterpart of categorical graded distributive laws. They are graded generalisations of the classical distributive laws of a comonad D over a monad T [**?**] (and vice versa) and involve nontrivial interactions between two kinds of grading given by a matched pair. We first focus on one of eight variations.

Definition 5. Let $\mathcal{R} = (R, \le, 1, *)$ and $\mathcal{E} = (E, \le, 1, \bullet)$ be preordered monoids, D be an $\mathcal{R}$-graded comonad on a category $\mathcal{C}$, T be an $\mathcal{E}$-graded monad on $\mathcal{C}$, and (ι, κ) be an $\mathcal{R}, \mathcal{E}$-matched pair for the distributive law format (LL, DT). An (ι, κ)*-distributive law (for* (LL, DT)*)* is a natural transformation

$$\sigma_{r,e,A} : D\,r\,(T\,e\,A) \to T\,\kappa(r,e)\,(D\,\iota(r,e)\,A) \qquad (5)$$

satisfying four equational axioms displayed in Figure 8.

The reason why we impose the matched pair axioms (Def. 2) on effect-coeffect interactions ι, κ is the following. When we add gradings to the equational axioms of the classical (*i.e.*, non-graded) distributive law, both sides of equational axioms get different gradings, thus become incomparable. The matched pair axioms on ι, κ are introduced to resolve this mismatch. For instance, one of the equational axioms of the classical distributive law $\sigma : D \circ T \to T \circ D$ of a comonad D over a monad (T, η, μ) is: $\sigma_A \circ D\eta_A = \eta_{DA} : DA \to TDA$. When we add gradings to D, T, σ, the morphisms on each side of the equation have different gradings:

$$\begin{aligned} \sigma_{r,1,A} \circ Dr\eta_A &: DrA \to T\,\kappa(r,1)\,(D\,\iota(r,1)\,A) \\ \eta_{DrA} &: DrA \to T1(DrA). \end{aligned}$$

To equate them, we introduce two equalities $\kappa(r, 1) = 1$ and $\iota(r, 1) = r$, which are a part of Definition 2. Remaining axioms of matched pair are similarly derived.

Generalising Definition 5, we define distributive laws for arbitrary format ϕ with respect to a given matched pair for ϕ.

Definition 6. Let $\mathcal{R}$ and $\mathcal{E}$ be preordered monoids, D be an $\mathcal{R}$-graded comonad on a category $\mathcal{C}$, T be an $\mathcal{E}$-graded monad on $\mathcal{C}$, ϕ be a distributive law format, and (ι, κ) be an $\mathcal{R}, \mathcal{E}$-matched pair for ϕ. An (ι, κ)*-distributive law* (for ϕ) is a natural transformation $\sigma^\phi : F^\phi(r, e) \to G^\phi(r, e)$, where F^ϕ and G^ϕ are functors of type $\mathcal{R} \times \mathcal{E} \to [\mathcal{C}, \mathcal{C}]$ determined by the following table:

ϕ		$F^\phi(r,e)$	$G^\phi(r,e)$
LL	TD	$Te \circ Dr$	$D(\iota(r,e)) \circ T(\kappa(r,e))$
LR	TD	$Te \circ D(\iota(r,e))$	$Dr \circ T(\kappa(r,e))$
RL	TD	$T(\kappa(r,e)) \circ Dr$	$D(\iota(r,e)) \circ Te$
RR	TD	$T(\kappa(r,e)) \circ D(\iota(r,e))$	$Dr \circ Te$
LL	DT	$Dr \circ Te$	$T(\kappa(r,e)) \circ D(\iota(r,e))$
LR	DT	$D(\iota(r,e)) \circ Te$	$T(\kappa(r,e)) \circ Dr$
RL	DT	$Dr \circ T(\kappa(r,e))$	$Te \circ D(\iota(r,e))$
RR	DT	$D(\iota(r,e)) \circ T(\kappa(r,e))$	$Te \circ Dr$

Moreover, σ^ϕ should satisfy four equalities that are given by the diagrams similar to Figure 8.

5.4 Categorical Semantics

We have set-up the categorical structures we need to interpret the calculus. The interpretation translates type derivation trees to morphisms. The interpretation of types is standard. We fix an object $[\![o]\!]$ of the interpretation of the base type o. We then inductively extend this to the interpretation of all types by

$$[\![A \to B]\!] = [\![A]\!] \multimap [\![B]\!] \quad [\![D_r A]\!] = Dr[\![A]\!] \quad [\![T_e A]\!] = Te[\![A]\!].$$

$$\begin{tikzcd} Dr \arrow[r, "D\circ\eta"] \arrow[dr, "\eta\circ D"'] & Dr \circ T1 \arrow[d, "\sigma"] \\ & T1 \circ Dr \end{tikzcd}$$

by $\iota(r, 1) = r$
and $\kappa(r, 1) = 1$

$$\begin{tikzcd} Dr \circ Te \circ Tf \arrow[r, "\sigma\circ T"] \arrow[d, "D\circ\mu"'] & T(\kappa(r, e)) \circ D(\iota(r, e)) \circ Tf \arrow[r, "T\circ\sigma"] & T(\kappa(r, e)) \circ T(\kappa(\iota(r, e), f)) \circ D(\iota(r, e \bullet f)) \arrow[d, "\mu\circ D"] \\ Dr \circ T(e \bullet f) \arrow[rr, "\sigma"'] & & T(\kappa(r, e \bullet f)) \circ D(\iota(r, e \bullet f)) \end{tikzcd}$$

by $\iota(r, e \bullet f) = \iota(\iota(r, e), f)$ *and* $\kappa(r, e \bullet f) = \kappa(r, e) \bullet \kappa(\iota(r, e), f)$

$$\begin{tikzcd} D1 \circ Te \arrow[r, "\epsilon\circ T"] \arrow[d, "\sigma"'] & Te \\ Te \circ D1 \arrow[ur, "T\circ\epsilon"'] & \end{tikzcd}$$

by $\iota(1, e) = 1$
and $\kappa(1, e) = e$

$$\begin{tikzcd} D(r * s) \circ Te \arrow[rr, "\sigma"] \arrow[d, "\delta\circ T"'] & & T(\kappa(r * s, e)) \circ D(\iota(r * s, e)) \arrow[d, "T\circ\delta"] \\ Dr \circ Ds \circ Te \arrow[r, "D\circ\sigma"'] & Dr \circ T(\kappa(s, e)) \circ D(\iota(s, e)) \arrow[r, "\sigma\circ D"'] & T(\kappa(r * s, e)) \circ D(\iota(r, \kappa(s, e))) \circ D(\iota(s, e)) \end{tikzcd}$$

by $\iota(r * s, e) = \iota(r, \kappa(s, e)) * \iota(s, e)$ *and* $\kappa(r * s, e) = \kappa(r, \kappa(s, e))$

Figure 8. Axioms of distributive laws in the (LL-DT) format; (co)effect annotations in morphisms are omitted

$$\text{ax}\ \frac{}{\mathrm{id}_{\llbracket A\rrbracket} : \llbracket A\rrbracket \to \llbracket A\rrbracket} \qquad \text{abs}\ \frac{f : \llbracket \Gamma, x : A\rrbracket \to \llbracket B\rrbracket}{\lambda(f \circ M_{\Gamma,x:A}) : \llbracket \Gamma\rrbracket \to \llbracket A \to B\rrbracket} \qquad \text{app}\ \frac{f : \llbracket \Gamma\rrbracket \to \llbracket A \to B\rrbracket \quad g : \llbracket \Delta\rrbracket \to \llbracket A\rrbracket}{ev \circ (f \otimes g) \circ S_{\Gamma,\Delta} : \llbracket \Gamma + \Delta\rrbracket \to \llbracket B\rrbracket}$$

$$\text{sub}\ \frac{f : \llbracket \Gamma\rrbracket \to \llbracket A\rrbracket}{\llbracket A <: B\rrbracket \circ f \circ \llbracket \Gamma', [\Delta_0] <: \Gamma\rrbracket : \llbracket \Gamma', [\Delta]_0\rrbracket \to \llbracket B\rrbracket} \qquad \text{op}\ \frac{}{\llbracket \mathbf{op}\rrbracket : \mathbf{I} \to \llbracket A_{\mathsf{op}}\rrbracket} \qquad \text{dist}\ \frac{f = \lambda(\sigma^{\phi}_{r,e,\llbracket A\rrbracket} \circ r_{F^{\phi}(r,e)\llbracket A\rrbracket})}{f : \mathbf{I} \to (F^{\phi}(r, e)\llbracket A\rrbracket \multimap G^{\phi}(r, e)\llbracket A\rrbracket)}$$

$$\text{unit}\ \frac{f : \llbracket \Gamma\rrbracket \to \llbracket A\rrbracket}{\eta_{\llbracket A\rrbracket} \circ f : \llbracket \Gamma\rrbracket \to \llbracket T_1 A\rrbracket} \qquad \text{letT}\ \frac{f : \llbracket \Gamma\rrbracket \to \llbracket T_e A\rrbracket \quad g : \llbracket \Delta, x : A\rrbracket \to \llbracket T_{e'} B\rrbracket}{\mu_{e,e',\llbracket B\rrbracket} \circ Te(g \circ M_{\Delta,x:A}) \circ st_{e,\llbracket \Delta\rrbracket,\llbracket A\rrbracket} \circ (\llbracket \Delta\rrbracket \otimes f) \circ S_{\Delta,\Gamma}}$$

$$\text{pr}\ \frac{f : \llbracket [\Gamma]\rrbracket \to \llbracket B\rrbracket}{Drf \circ d_{r,[\Gamma]} : \llbracket r * [\Gamma]\rrbracket \to \llbracket DrB\rrbracket} \qquad \text{der}\ \frac{f : \llbracket \Gamma, x : A\rrbracket \to \llbracket B\rrbracket}{f \circ \epsilon_{(\Gamma,x:A)@x} : \llbracket \Gamma, x : [A]_1\rrbracket \to \llbracket B\rrbracket} \qquad \text{letD}\ \frac{f : \llbracket \Gamma\rrbracket \to \llbracket D_r A\rrbracket \quad g : \llbracket \Delta, x : [A]_r\rrbracket \to \llbracket B\rrbracket}{g \circ M_{\Delta,x:[A]_r} \circ (\llbracket \Delta\rrbracket \otimes f) \circ S_{\Delta,\Gamma} : \llbracket \Gamma + \Delta\rrbracket \to \llbracket B\rrbracket}$$

Figure 9. Categorical semantics of typing derivations

We also extend this interpretation to discharged types (which appear only inside typing environments) by $\llbracket [A]_r\rrbracket = Dr\llbracket A\rrbracket$

We next interpret the subtyping relations for types as morphisms. To the subtyping relation $A <: B$, we inductively assign a morphism $\llbracket A <: B\rrbracket : \llbracket A\rrbracket \to \llbracket B\rrbracket$ using the functoriality of each type constructor. As an example, we highlight the interpretation of the (s-D) subtyping rule. Note that D is $\mathcal{R}^{op}$-graded (with the opposite pre-order), thus:

$$\llbracket D_r A <: D_s A'\rrbracket = D(s \leq r)(\llbracket A <: A'\rrbracket) : Dr\llbracket A\rrbracket \to Ds\llbracket A'\rrbracket.$$

To interpret typing environments we assume an arbitrary linear order $<$ on variables. The interpretation $\llbracket \Gamma\rrbracket$ of a typing environment Γ is the tensor product $\llbracket \Gamma(x_1)\rrbracket \otimes \cdots \otimes \llbracket \Gamma(x_n)\rrbracket$ of the interpretation of types (including discharged ones), where $x_1, \cdots, x_n$ forms the $<$-sorted list of variables in $\mathrm{dom}(\Gamma)$. We then extend this interpretation to the subtyping relation between typing environments.

We introduce some additional auxiliary morphisms.

- Let Γ, Δ be summable typing environments. We define the *splitting* $S_{\Gamma,\Delta} : \llbracket \Gamma + \Delta\rrbracket \to \llbracket \Gamma\rrbracket \otimes \llbracket \Delta\rrbracket$ by a combination of the contraction $c_{r,s,A}$ and the symmetry of $\otimes$. The contraction is performed only on the types assigned to the variables $x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$. We note that when Γ, Δ are disjoint, $S_{\Gamma,\Delta}$ becomes a (permutation) isomorphism.
- For a typing environment Γ with $x \notin \mathrm{dom}(\Gamma)$, define $M_{\Gamma,x:A}$ to be $(S_{\Gamma,x:A})^{-1} : \llbracket \Gamma\rrbracket \otimes \llbracket A\rrbracket \to \llbracket \Gamma, x : A\rrbracket$.
- For $r \in R$ and a discharged environment $[\Gamma]$, define the *multi-comultiplication* $d_{r,[\Gamma]} : \llbracket r * [\Gamma]\rrbracket \to Dr\llbracket [\Gamma]\rrbracket$ to be the composite of the tensor product of comultiplications of the form $\delta_{r,s,A}$ followed by the $|\mathrm{dom}([\Gamma])|$-monoidality, which is a combination of 0- and 2-monoidality. For example, let $\Delta = x : [A]_{s_1}, y : [B]_{s_2}$ then multi-comultiplication $d_{r,\Delta}$ is defined:

$$\begin{tikzcd} D(r * s_1)A \otimes D(r * s_2)B \arrow[dr, "\delta_{r,s_1,A} \otimes \delta_{r,s_2,B}"'] & & Dr(Ds_1 A \otimes Ds_2 B) \\ & DrDs_1 A \otimes DrDs_2 B \arrow[ur, "m_{r,A,B}"'] & \end{tikzcd}$$

- For a typing environment Γ with $x \in \Gamma$ and $\Gamma(x) = [A]_1$, let $x_1, \cdots, x, \cdots, x_n$ be the $<$-sorted list of variables in $\mathrm{dom}(\Gamma)$. Then we define *multi-counit* $\epsilon_{\Gamma@x} : \llbracket \Gamma\rrbracket \to \llbracket A\rrbracket$ to be $\mathrm{id}_{\llbracket \Gamma(x_1)\rrbracket} \otimes \cdots \otimes \varepsilon_{\llbracket A\rrbracket} \otimes \cdots \otimes \mathrm{id}_{\llbracket \Gamma(x_n)\rrbracket}$ (dereliction/counit appears only at the position of x).
- For a discharged typing environment $[\Delta]_0$ having coeffect 0 only, let $x_1, \cdots, x_n$ be the $<$-sorted list of variables in it, and $[A_i]_0$ be the discharged type assigned to x_i. Then we define *multi-weakening* $w_{[\Delta]_0}$ by

$$\llbracket [\Delta]_0\rrbracket \xrightarrow{w_{\llbracket A_1\rrbracket} \otimes \cdots \otimes w_{\llbracket A_n\rrbracket}} \mathbf{I} \otimes \cdots \otimes \mathbf{I} \xrightarrow{\simeq} \mathbf{I}$$

- For environments $\Gamma' <: \Gamma$ we define the subtyping-and-weakening morphism $\llbracket \Gamma', [\Delta]_0 <: \Gamma\rrbracket$ as the composite:

$$\llbracket \Gamma', [\Delta]_0\rrbracket \xrightarrow{(\llbracket \Gamma' <: \Gamma\rrbracket \otimes w_{[\Delta]_0}) \circ S_{\Gamma',[\Delta]_0}} \llbracket \Gamma\rrbracket \otimes \mathbf{I} \xrightarrow{\simeq} \llbracket \Gamma\rrbracket$$

The core structures, along with these auxiliary definitions, then provides the interpretation for typing derivations as morphisms in $\mathcal{C}$. For this, we assume that a morphism $\llbracket \mathbf{op}\rrbracket : \mathbf{I} \to \llbracket A_{\mathsf{op}}\rrbracket$ is given for each operator symbol **op**. We then inductively interpret a derivation of $\Gamma \vdash t : A$ by the rules in Fig. 9. Each of these has a corresponding rule in Fig. 2 and represents a construction of a morphism along the typing rule. For instance, we read (abs) rule in Fig. 9 as "if we construct a morphism f alongside the derivation π

of $\Gamma, x : A \vdash t : B$, we construct the morphism $\lambda(f \circ M_{\Gamma,x:A})$ for the derivation π+(abs) of $\Gamma \vdash \lambda x \,.\, t : A \to B$". We can now state the main theorem of this section.

Theorem 1 (Soundness)**.** *Let π_1 and π_2 be derivations of $\Gamma \vdash t_i : A$ $(i = 1, 2)$, respectively. If $\pi_1 = \pi_2$ is derivable in the equational theory presented in Sect. 4, then $[\![\pi_1]\!] = [\![\pi_2]\!]$ holds. The technical report provides the full proof and auxiliary lemmas [**?**].*

6. Examples

We now present the details for two concrete instances of our calculus which model some new interesting computational behaviours. The first example combines a coeffect-graded comonad for information flow with an effect-graded monad for nondeterminism analysis. The second example combines a coeffect-graded comonad for exact resource analysis with an effect-graded monad for errors.

6.1 Combining Information Flow and Non-determinism

Information flow properties, such as tracking high- and low-security code/data, have been described by effect systems with a lattice of security levels, *e.g.* [**?**]. We argue that a coeffectful presentation is more natural, since information flow relates to variable use. As an example of combining information flow coeffects with effects, we pick non-determinism effects for the sake of variety. We instantiate the calculus of Section 3 with the following data:

Coeffect $\mathcal{R}$	A distributive lattice $(R, \leq, \bot, \vee, \top, \wedge)$
Effect $\mathcal{E}$	$(\{\mathrm{DET} \leq \mathrm{ND}\}, \mathrm{DET}, \bullet)$ where $x \bullet y = \mathrm{DET}$ iff $x = y = \mathrm{DET}$
Dist. law format	$(\mathrm{LL}, \mathrm{DT})$
ι	The first projection π_1
κ	The second projection π_2
Operation	$\mathbf{or}_A : T_{\mathrm{ND}}A \to T_{\mathrm{ND}}A \to T_{\mathrm{ND}}A$

The additive monoid of $\mathcal{R}$ is $(R, \vee, \bot)$ and multiplicative is $(R, \wedge, \top)$. Note that, compared to the traditional effect-based presentation of information flow, the lattice is inverted here for coeffects, matching their contravariant nature.

The graded comonadic type D_r plays the role of *information masking* corresponding to a (partial) view of an observer. That is, a computation typed $D_r A$ provides a value of type A only at security levels inside the downset $\downarrow r = \{r' \in R \mid r' \leq r\}$. Outside this downset, the type $D_r A$ is only observable as the singleton type 1.

The graded monadic type constructor T_e classifies whether the computation is definitely deterministic (DET) or possibly nondeterministic (ND). The order $\mathrm{DET} \leq \mathrm{ND}$ allows us to upcast deterministic computations to nondeterministic ones. The operation $\mathbf{or}_A$ is the nondeterministic choice operator. As the result is always nondeterministic, the result of the choice is classified as ND. The matched pair of this calculus is simply (π_1, π_2). Therefore the distributive laws we allow in this instantiation take the following form:

$$\mathbf{dist}^{(\mathrm{LL},\mathrm{DT})}_{r,e,A} : D_r T_e A \to T_e D_r A.$$

The semantics for this **dist**, given below in Def. 7, is that if a computation is observed at a security level r' that is not within $\downarrow r$ then any non-determinism at security levels within $\downarrow r$ is *masked*, *i.e.*, not visible. Otherwise, if a computation is observed at $r' \in \downarrow r$, then the non-determinism is available (not masked). The intended semantics of T_{DET} and T_{ND} are the identity functor and the nonempty powerset functor, respectively. These functors preserve terminal objects 1 which are used to model masked computations (see below), hence T_{DET} and T_{ND} commute with the masking functor $D\,r$ for each $r \in \mathcal{R}$. In the calculus we reflect this isomorphism via the distributive law $(\mathrm{LL}, \mathrm{DT})$, which transforms possibly-masked effectful computations to effectful possibly-masked computations.

Semantics We give a $\mathbf{Set}$-based interpretation of this calculus. The domain of the interpretation is $\mathbf{Set}^R$, the product category of R-fold copies of $\mathbf{Set}$. It is Cartesian closed, and the structure is given security-level-wise. The idea is that denotations of terms are computations that are indexed by security levels.

An object A of $\mathbf{Set}^R$ is an R-indexed family $\{A\,r\}_{r \in R}$ of sets. This family describes how a type A is observed depending on security levels. For instance, suppose that a set $B \in \mathbf{Set}$ corresponds to a base type (*e.g.*, natural numbers / booleans).

- The type where a B value is observable at each security level corresponds to the constant family K_B given by $K_B r = B$.
- The type where B values are available only at the security level inside $\downarrow r$ corresponds to the following family $P r B$:

$$(P r B) r' = \begin{cases} 1 & r' \not\leq r \\ B & r' \leq r \end{cases}$$

  Here 1 is the terminal object of $\mathbf{Set}$. This type reduces to the trivial data type (*i.e.* 1) when the security level is outside of $\downarrow r$.

A morphism from A to B in $\mathbf{Set}^R$ is an R-indexed family of functions $f_r : A\,r \to B\,r$. For natural transformations α between endofunctors on $\mathbf{Set}^R$ we write $\alpha_{A,r}$ as shorthand for $(\alpha_A)_r$ (the morphism in $\mathbf{Set}$). Such a morphism may be seen as a security-level dependent computation. We illustrate this situation by considering a morphism $f : P r B \to P r B$ in $\mathbf{Set}^R$. From the definition of $P r B$, f needs to be the following family:

$$f_{r'} = \begin{cases} \mathrm{id}_1 : 1 \to 1 & r' \not\leq r \\ f_{r'} : B \to B & r' \leq r \end{cases}$$

Thus f can perform some nontrivial computation over B inside the downset $\downarrow r$ of security levels, but outside $\downarrow r$, it performs nothing.

For each security level $r \in R$, we introduce the *masking functor* $D\,r : \mathbf{Set}^R \to \mathbf{Set}^R$. This functor takes an R-indexed family of sets, removes all the sets assigned to the security level outside of the downset $\downarrow r$, then fills the removed part with the terminal object $1 \in \mathbf{Set}$. The formal definition of Dr is the following.

$$(D\,r A)\,r' = \begin{cases} 1 & r' \not\leq r \\ A\,r' & r' \leq r \end{cases}, \quad (D\,r f)_{r'} = \begin{cases} \mathrm{id}_1 & r' \not\leq r \\ f_{r'} & r' \leq r \end{cases}$$

where $(DrA)\,r'$ is thus a functor in $\mathbf{Set}$. For instance, we have $DrK_B = PrB$. To extend $r \mapsto Dr$ to a functor, for each ordered pair $r \leq r'$ of security levels, we define a natural transformation $D(r \leq r') : D\,r' \to D\,r$ by

$$D(r \leq r')_{A,r''} = \begin{cases} !_{(D\,r'A)\,r''} & r'' \not\leq r \\ \mathrm{id}_{Ar''} & r'' \leq r (\leq r') \end{cases}$$

The join and meet of the security level poset $\mathcal{R}$ make the masking functor D a *graded exponential comonad.*

Theorem 2. *The assignment $r \mapsto Dr$ extends to an $\mathcal{R}^{op}$-graded exponential comonad over $\mathbf{Set}^R$.*

We next construct an $\mathcal{E}$-graded monad. Let us write $\mathbf{Mnd}(\mathbf{Set})$ for the category of monads over $\mathbf{Set}$ and monad morphisms between them. We define a functor $T' : \mathcal{E} \to \mathbf{Mnd}(\mathbf{Set})$ by

$$T'(\mathrm{DET}) = \mathrm{Id}, \quad T'(\mathrm{ND}) = P^+, \quad T'(\mathrm{DET} \leq \mathrm{ND}) = \eta$$

where P^+ is the nonempty powerset monad, and η is the unit of P^+, which is also a monad morphism from Id to P^+. Since $\mathcal{E}$ is a (two pointed) join semilattice, this extends to a graded monad (T', η', μ') on $\mathbf{Set}$ [**?** , Theorem 2.12]. We then further extend this pointwise to the graded monad (T, η, μ) on $\mathbf{Set}^R$:

$$(T\,e\,A)\,r = T'e\,(A\,r), \quad \eta_{A,r} = \eta'_{A\,r}\,, \quad \mu_{e,e',A,r} = \mu'_{e,e',(A\,r)}$$

The key part of this example is then the distributive law definition.

Definition 7. We define a (π_1, π_2)-distributive law $\sigma_{r,e} : DrTe \to TeDr$ for (LL, DT):

$$\sigma_{r,e,A,r'} = \begin{cases} T'(\text{DET} \le e)_1 \circ \eta'_1 & r' \not\le r \\ \mathrm{id}_{T'e(Ar')} & r' \le r \end{cases}$$

In the first case, if the observer's security level r' is *not* within $\downarrow r$, the domain and codomain of σ is observed as 1 and $T'e1$, respectively. The latter is always isomorphic to 1; thus σ connects them by the isomorphism. In the second case, where the observer's security level is within $\downarrow r$ we preserve the non-determinism exactly via the identity. Note that σ is an isomorphism at each component.

Finally, we interpret the non-deterministic choice operator by:

$$[\![\mathbf{or}_A]\!]_r : P^+([\![A]\!]r) \Rightarrow P^+([\![A]\!]r) \Rightarrow P^+([\![A]\!]r)$$
$$[\![\mathbf{or}_A]\!]_r(x)(y) = x \cup y$$

Theorem 3. *The pair* (π_1, π_2) *is a* $\mathcal{R}^{op}, \mathcal{E}$*-matched pair for* (LL, DT)*, and* σ *is a* (π_1, π_2)*-distributive law.*

The key point of this example is that the coeffect security grade r modulates the effect– a non-trivial effect-coeffect interaction. An alternate graded monad model might count the *degree* of non-determinism (similar to the work of **?**]).

6.2 Usage Analysis and Errors

Here we consider an example similar to that presented informally in Section 2. We instantiate our calculus with coeffects that track the exact number of times a variable is used and effects distinguishing computations with errors (*e.g.*, exceptions) ($\bot$) from the total, pure computations ($\top$). We have the following data:

Coeffect $\mathcal{R}$	$(\mathbb{N}, =, 0, +, 1, \times)$
Effect $\mathcal{E}$	$(\{\top, \bot\}, =, \top, \bullet)$ such that $e \bullet f = \top$ iff $e = f = \top$
Dist. law format	(LL, TD)
ι	$\iota(r, \bot) = 1$ and $\iota(r, \top) = r$
κ	The second projection π_2
Operation	$\mathbf{throw}_A : T_\bot A$

In this instantiation, the graded comonadic type $D_r A$ ($r \in \mathbb{N}$) describes values that can be used *exactly* r times to obtain values of the type A. In the semantics, we implement the graded comonadic type $D_r A$ by the symmetric tensor product A^r. The graded monad for this example classifies whether a program causes errors or not. Note that the ordering of the coeffects $\mathcal{R}$ and effects $\mathcal{E}$ here is equality, hence sub-(co)effecting is not useful in this instantiation.

The matched pair and the distributive law in this derived calculus give the following two components:

$$\mathbf{dist}_{r,\top,A} : T_\top D_r A \to D_r T_\top A \tag{6}$$
$$\mathbf{dist}_{r,\bot,A} : T_\bot D_r A \to D_1 T_\bot A \tag{7}$$

The first law explains that, if the computation is pure then the coeffect is unchanged. The second explains that if we know that we will definitely have an error, then only one copy of that effectful computation is needed. This is because execution flow is interrupted by the error, and so more copies of the error are not needed.

More concretely, the type $T_\top A$ consists only of pure computations of type A. We hence identify it with the type A itself. Then both sides of the first component of the distributive law (6) may be seen as the type $D_r A$. The intended behaviour of this component of the distributive law is the identity function. On the other hand, the type $T_\bot A$ consists only of erratic computations and they contain no value of type A; this applies when $A = D_r B$. Therefore any computation in $T_\bot D_r B$ can be safely casted as an element in the type $D_1 T_\bot B$ without changing its contents. This is the meaning of the second component of the distributive law (7).

Semantics A categorical semantics of this derived calculus can be given in a symmetric monoidal closed category $\mathcal{C}$ with a terminal object 1. We also assume that $\mathcal{C}$ has limits of functors from any one-object category, and the tensor product preserves these limits in each argument.

We interpret the graded comonadic type by the *symmetric tensor product*, which we sketch below. First, let S_r be the group of bijections on $r \in \mathbb{N}$ (as a finite cardinal number), and regard S_r as a one-object category. Each bijection $i \in S_r$ naturally induces a permutation morphism of type $A^{\otimes r} \to A^{\otimes r}$ for every $A \in \mathcal{C}$. We make this into a functor $S_{r,A} : S_r \to \mathcal{C}$, and let (A^r, π^r_A) be its limit. The object part of this limit is called the *symmetric tensor product* of A; see [**?**] for a similar calculation.

Theorem 4. *The mapping* $D : r, A \mapsto A^r$ *extends to an* $\mathcal{R}^{op}$*-graded comonad* D *on* $\mathcal{C}$*.*

Next, define K_1 to be the constant functor returning the terminal object 1. This functor has a unique strength. It satisfies

$$\mathrm{Id} \circ K_1 = K_1 \circ \mathrm{Id} = K_1 \circ K_1 = K_1.$$

Therefore the functor $T : E \to [\mathcal{C}, \mathcal{C}]$ given by $T\top = \mathrm{Id}$ and $T\bot = K_1$ is a strict monoidal functor of type $\mathcal{E} \to ([\mathcal{C}, \mathcal{C}], \mathrm{Id}, \circ)$, hence is an $\mathcal{E}$-graded strong monad. The exception-throwing operation is interpreted by the unique morphism to the terminal object: $[\![\mathbf{throw}_A]\!] = !_\mathbf{I} : \mathbf{I} \to [\![T\bot A]\!] = 1$.

We introduce the (ι, π_2)-distributive law $\sigma : Te \circ Dr \to D(\iota(r, e)) \circ Te$ for (LL, TD). We reflect the intuition of the distributive law described in the previous paragraph by

$$\sigma_{r,\top} = \mathrm{id}_{Dr}, \quad \sigma_{r,\bot} = \mathrm{id}_{K_1} : K_1 \to K_1.$$

Theorem 5. *The pair* (ι, π_2) *is an* $\mathcal{R}^{op}, \mathcal{E}$*-matched pair for* (LL, TD)*, and the above* σ *is a* (ι, π_2)*-distributive law.*

7. Related Work

Monads and effects, comonads and coeffects. Starting with the seminal work of **?**] effectful computations have often been structured by monads. The connection between monads and effectful computation has also provided a rich mathematical foundation for different concepts such as effectful operations [**?**] and effect handlers [**?**]. Comonads have also been used to structure computation. Fundamental in this direction has been the work by **? ?**] formulating several context-dependent programming models in terms of comonadic computations.

Monads and comonads together **?**] used comonads to structure dataflow computations with partial computations modelled monadically. As in our work, they interact monads and comonads via distributive laws, but they do not use any graded structure. They show how to implement some instances of the distributive laws in Haskell. **?**] used monad and comonads for describing respectively the extensional and intensional semantics for a language with computational effects. They used distributive laws to describe the intensional semantics of computational effects and they have shown several instances for concrete models. **?**] combined monads and comonads in the setting of categorical operational and denotational semantics. They provide also a categorical account of the different relationships that can be established between monads and comonads when looking at distributive laws. **?**] have used monads, comonads and distributive laws to give a categorical account of innocent strategies in game semantics. They have shown in particular that the combinatorics of the distributivity reflect one of the components involved in the composition of innocent strategies. None of these works uses grading. The novelty of our approach is in making the interaction between monads, comonads and distributive laws emerge in the type theory via grading.

In recent work, **?**] have studied a polarized calculus with both effects and resources. They do not consider graded monads and comonads but they investigate a calculus with effect and the resource structures (in the sense of linear logic exponentials); their model does not rely on distributive laws as effects and resources are treated orthogonally via an adjunction-based model. It will be interesting in future works to investigate whether the grading structure can be used in their context as well.

Zappa-Szép products appears in work on distributive laws between *directed container comonads* by Ahman and Uustalu [**?**]. In their work, directed containers have a monoid(-like) structure on shapes: container comonads can then be composed by a distributive law which has the operations and axioms of a Zappa-Szép product (at the value level). This structure also appears in their later work on *update monads* with a similar situation of a composition of two monoids [**?**]. The main difference with our work is that this structure emerges for us naturally as a result of the grading.

Indexing and grading The idea of refining monadic models of effects with some additional information has emerged quite naturally and has generated different notions such as: indexed monads [**?**], layered monads [**?**], parametrised monads [**?**], and parametric-effect monads [**?**] (which are graded monads). Particularly relevant for our work is the approach followed by **?**] and by **?**] who have established a bridge between monadic systems and effect type systems such as those in [**? ?**]. This approach has also been advocated by **?**] who proposes the notion of *productor* to describe general producer effect systems. Interestingly, he also mentions the notion of *consumptor*, as dual to productor, without working out the technical details for this notion. The effectful fragment of our calculus is inspired by these works and in particular it follows the mathematical formulation provided by **?**] and by **?**]. Various recent work employs graded monads for refining models of effectful computation *e.g.*, [**? ? ? ? ?**]. **?**] study the mathematical theory of graded monads in more depth.

A similar approach has been recently proposed for comonadic computations. Coeffect systems have been introduced to to structure context-dependent computations firstly proposed by **?**], including the semantic model of graded comonads. Coeffect systems have also emerged in the study of resource consumption following the approach of bounded linear logic. Indeed, the comonad of bounded linear logic can be generalized to a coeffect-graded comonad as shown by **?**], **?**], and Petricek et al. [**? ?**]. Moreover, the semantics of coeffect systems has been also studied by **?**] in the relational semantics of linear logic.

Our work is the first to combine these two directions to study the interaction between monadic and comonadic computations via graded distributive laws.

Recent work by **?**] uses resource bounds (in the style of bounded linear logic) in a dependently-typed context for explaining interactions between linearity and dependence. This includes a coeffect-like semiring structure in the type-system for fine grained tracking of variable usage. Similar structures in combination with dependent types à la Dependent ML [**?**] have been also used by **? ?**].

The Contextual Modal Type theory of **?**] appears to be a related example of a coeffect-graded necessitation modality, like our D_r, but graded by contexts of local scopes. Further work is to explore fitting CMTT into our coeffect framework. An early precursor to CMTT combines state effects with dynamic binding effects (which resemble coeffects) [**?**]. Exploring whether this is an instance of our effect-coeffect calculus is future work.

8. Conclusion and Future Directions

We presented a core calculus for effectful and coeffectful computation, where coeffects and effects may interact. Our semantics builds on recently established graded monad and graded (exponential) comonad models of effects and coeffects. We introduced graded distributive laws to model effect-coeffect interactions, with a design space of choices. This is a step towards a better understanding of how to combine effects and coeffects. There are many exciting directions for further study. We touch on some of them here.

Concrete semantics and operations One of the most interesting directions for future research is a general operational semantics for effect-coeffect interactions, and their operations. Previous work by **?**] showed how to design an operational semantics that collects effect information. Similarly, previous work on coeffects included an *instrumented* operational semantics collecting information on observable coeffect actions [**?**]. Both these have used the operational semantics to prove the soundness of effect and coeffect types, respectively. These works can be a source of inspiration for designing an operational semantics for our calculus instrumented with effect and coeffect observations to prove a general soundness result. This would be particularly useful for understanding the kind of operations we can add to our calculus.

Computational λ-calculus and coeffect-effect analysis Early effect systems (*e.g.* [**? ? ?**]) were defined for impure λ-calculus-like languages where any term may have side effects. Subsequently effect information e forms part of typing judgments $\Gamma \vdash t : A, e$. Relatedly, the *latent* effects of a function are recorded on the function type arrow, *e.g.* $A \xrightarrow{e} B$. Similarly, in early work on coeffect systems all terms are considered potentially coeffectful and thus coeffects form part of the typing judgment [**?**]. This contrasts with our approach where a pure λ-calculus fragment is extended with additional constructs for handling effects and coeffects (*e.g.*, the monadic metalanguage approach [**?**], *cf.* Haskell's **do**-notation). We believe the implicit style can be suitably integrated with the explicit style and the distributive behaviour of our calculus. This corresponds more closely to static analysis of effects and coeffects.

Categorical analysis of distributive laws Distributive laws between monads and comonads are equivalent to the liftings of one structure (monad or comonad) to the Kleisli / Eilenberg-Moore category of the other structure [**?**]. It is thus natural to extend this equivalence to the graded case. One possible direction of this extension is to relate graded distributive laws and the liftings of graded (co)monads to the Kleisli / Eilenberg-Moore categories of graded (co)monads introduced in [**?**].

Type checking We have yet to develop a type checking procedure for our calculus. Some exciting work has been done in this direction for both effects [**?**] and coeffects [**?**]. We plan to study a bidirectional re-characterisation of our system which goes towards a type checking procedure. We expect this to explain where it is necessary to insert explicit type signatures in a derivation (in Church style).

Acknowledgments

The authors thank the anonymous reviewers for their helpful comments. Orchard was supported by EPSRC grant EP/M026124/1 and EP/K011715/1 (whilst previously at Imperial College London), Katsumata by JSPS KAKENHI grant JP15K00014, Uustalu by Estonian Min. of Educ. and Res. grant IUT33-13 and Estonian Sci. Found. grant 9475. Gaboardi's work was done in part while at the University of Dundee, UK supported by EPSRC grant EP/M022358/1.

References

[] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL*. ACM, 1999.

[] D. Ahman and T. Uustalu. Distributive laws of directed containers. *Progress in Informatics*, 10:3–18, 2013.

[] D. Ahman and T. Uustalu. Update monads: cointerpreting directed containers. In *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES*, volume 13, pages 1–23, 2014.

[] R. Atkey. Parameterised notions of computation. *JFP*, 19(3-4), 2009.

[] N. Benton, A. Kennedy, M. Hofmann, and V. Nigam. Counting Successes: Effects and Transformations for Non-Deterministic Programs. In *A List of Successes That Can Change the World*. Springer, 2016.

[] F. Breuvart and M. Pagani. Modelling coeffects in the relational semantics of linear logic. In *CSL*, 2015.

[] S. Brookes and K. V. Stone. Monads and Comonads in Intensional Semantics. Technical report, Pittsburgh, PA, USA, 1993.

[] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *ESOP*, pages 351–370, 2014.

[] P. Curien, M. P. Fiore, and G. Munch-Maccagnoni. A theory of effects and resources: adjunction models and polarised calculi. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 44–56, 2016. doi: 10.1145/2837614.2837652.

[] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In IEEE LICS '11, pages 133–142, 2011.

[] A. Filinski. Representing layered monads. In *POPL*. ACM, 1999.

[] S. Fujii, S. Katsumata, and P.-A. Melliès. Towards a formal theory of graded monads. In *FoSSaCS*, volume 9634 of *LNCS*. Springer Berlin / Heidelberg, 2016.

[] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *POPL*, pages 357–370, 2013. URL `http://dl.acm.org/citation.cfm?id=2429113`.

[] D. R. Ghica and A. I. Smith. Bounded linear types in a resource semiring. In *ESOP*, pages 331–350, 2014.

[] J. Gibbons. Comprehending ringads. In *A List of Successes That Can Change the World*, pages 132–151. Springer, 2016.

[] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and func. prog.*, LFP '86, 1986. ISBN 0-89791-200-4.

[] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical computer science*, 97(1):1–66, 1992.

[] R. Harmer, M. Hyland, and P.-A. 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. doi: 10.1109/LICS.2007.14.

[] M. Hicks, G. M. Bierman, N. Guts, D. Leijen, and N. Swamy. Polymonadic programming. In P. Levy and N. Krishnaswami, editors, *MSFP*, volume 153 of *EPTCS*, pages 79–99, 2014.

[] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 303–310. ACM, 1991.

[] C. Kassel. *Quantum Groups*. Graduate Texts in Mathematics. Springer New York, 1994. ISBN 9780387943701.

[] S. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of POPL*, pages 633–646. ACM, 2014.

[] J. Lucassen and D. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM, 1988.

[] C. McBride. I got plenty onuttin. In *A List of Successes That Can Change the World*, pages 207–233. Springer, 2016.

[] P. Melliès, N. Tabareau, and C. Tasson. An explicit formula for the free exponential modality of linear logic. In *Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Part II*, pages 247–260, 2009. doi: 10.1007/978-3-642-02930-1_21.

[] P.-A. Melliès. Parametric monads and enriched adjunctions. Technical report, 2012. `http://www.pps.univ-paris-diderot.fr/~mellies/tensorial-logic/`.

[] S. Milius, D. Pattinson, and L. Schröder. Generic Trace Semantics and Graded Monads. In *Proceedings of 6th International Conference in Algebra and Coalgebra in Computer Science*, 2015.

[] 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. ISBN 0818619546.

[] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1): 55–92, 1991. doi: 10.1016/0890-5401(91)90052-4.

[] A. Mycroft, D. Orchard, and T. Petricek. Effect systems revisited – control-flow algebra and semantics. In *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, pages 1–32. Springer, 2016. doi: 10.1007/978-3-319-27810-0_1.

[] A. Nanevski. From dynamic binding to state via modal possibility. In *PPDP*, volume 27, pages 207–218, 2003.

[] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):23, 2008.

[] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[] D. Orchard. Programming Contextual Computations. *PhD dissertation, University of Cambridge*, 2014.

[] D. Orchard and T. Petricek. Embedding effect systems in haskell. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, pages 13–24. ACM, 2014.

[] D. Orchard, T. Petricek, and A. Mycroft. The semantic marriage of monads and effects. *arXiv preprint arXiv:1401.5391*, 2014.

[] T. Petricek. Context-Aware Programming Languaes. *Forthcoming PhD dissertation, University of Cambridge*, 2016.

[] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: Unified Static Analysis of Context-Dependence. In *ICALP (2)*, pages 385–397, 2013.

[] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 123–135. ACM, 2014. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628160.

[] G. Plotkin and J. Power. Adequacy for algebraic effects. In *FOSSACS*, volume 2030 of *LNCS*, 2001.

[] G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. ISBN 978-3-642-00589-3.

[] J. Power and H. Watanabe. Combining a monad and a comonad. *Theoretical Computer Science*, 280(1-2):137–162, 2002. doi: http://dx.doi.org/10.1016/S0304-3975(01)00024-X.

[] R. Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.

[] K. Terui. Light affine lambda calculus and polytime strong normalization. In LICS '01, pages 209–220. IEEE Computer Society, 2001.

[] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008. doi: http://dx.doi.org/10.1016/j.entcs.2008.05.029.

[] T. Uustalu and V. Vene. The Essence of Dataflow Programming. *LNCS*, 4164:135–167, November 2006.

[] 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. ISBN 0897914538.

[] P. Wadler. The marriage of effects and monads. In *ICFP*. ACM, 1998.

[] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.

[] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.