Coeffects: A calculus of context-dependent computation

Tomas Petricek Dominic Orchard Alan Mycroft
University of Cambridge
{firstname.lastname}@cl.cam.ac.uk

Abstract

The notion of *context* in functional languages no longer refers just to variables in scope. Context can capture additional properties of variables (usage patterns in linear logics; caching requirements in dataflow languages) as well as additional resources or properties of the execution environment (rebindable resources; platform version in a cross-platform application). The recently introduced notion of coeffects captures the latter, whole-context properties, but it failed to capture fine-grained per-variable properties.

We remedy this by developing a generalized coeffect system with annotations indexed by a coeffect *shape*. By instantiating a concrete shape, our system captures previously studied *flat* (whole-context) coeffects, but also *structural* (per-variable) coeffects, making coeffect analyses more useful. We show that the structural system enjoys desirable syntactic properties and we give a categorical semantics using extended notions of *indexed comonad*.

The examples presented in this paper are based on analysis of established language features (liveness, linear logics, dataflow, dynamic scoping) and we argue that such context-aware properties will also be useful for future development of languages for increasingly heterogeneous and distributed platforms.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Context; Types; Coeffects; Indexed comonads

1. Introduction

Context is important for defining meaning – not just in natural languages, but also in logics and programming languages. The standard notion of context in programming is an environment providing values for free variables. An open term with free variables is context dependent – its meaning depends on the free-variable context. The simply-typed λ-calculus famously analyses such context usage. Other systems go further. For example, bounded linear logic tracks the number of times a variable is used [7].

In software engineering, "context" often encompasses more than just free-variable values. For example, in a distributed system, the context provides resources that may be available on a particular device (*e.g.*, a database on a server or a GPS sensor on a phone).

In this paper, we develop a calculus for capturing various notions of context in programming. A key feature and contribution of the calculus is its *coeffect system* which provides a static analysis for contextual properties (coeffects). The system follows the style of type and effect systems, but captures a different class of properties. Another key contribution of the calculus is its semantics which can be smoothly instantiated for specific notions of context.

Coeffect systems were previously introduced as a generic analysis of context dependence which can be instantiated for various notions of context [15]. The formalization was restricted to tracking a class of *whole-context* properties where terms have just one coeffect. This limited the applications and precision of any analysis. For example, a whole-context liveness analysis marks the entire free-variable context as live (some variable may be used) or dead (no variable is used), but it cannot record liveness *per variable*.

We develop a more general system which captures both per-variable coeffects, which we call *structural*, and whole-context coeffects, which we call *flat*, and more. Our key contributions are:

- We present the *coeffect calculus* which augments the simply-typed λ-calculus with a general coeffect type system (Section 3). We demonstrate the two classes of flat (whole context) and structural (fine-grained, per variable) systems.
- We show practical examples, instantiating the calculus for structural systems capturing variable usage based on bounded linear logic, dataflow caching, and precise liveness analysis. We also instantiate the calculus to flat systems, building on and extending previous examples from [15].
- We discuss the syntactic properties of flat and structural variants of the coeffect calculus (Section 4). Notably, structural systems satisfy type preservation under both β-reduction and η-expansion, allowing their use with both call-by-name and call-by-value languages. This important property distinguishes structural coeffects from both effect systems and flat coeffects.
- We provide a denotational semantics, revisiting and extending the notion of *indexed comonads* to the structural setting (Section 5). We prove soundness by showing the correspondence between syntactic and semantic properties of coeffect systems.

Coeffects can be approached from multiple directions (Section 2.5) including syntactic (effect systems), semantic, and proof-theoretic. We emphasize the syntactic view, though we also outline a categorical semantics and note the interesting technical details.

2. Why coeffects matter

Coeffects are a way to describe notions of context that keep turning up in programming. To illustrate this, we overview three systems tracking contextual properties that motivate our general coeffect system. Two systems track per-variable properties (bounded linear logic and dataflow) and one tracks whole-context properties (implicit parameters). We start with some background and finish with a brief overview of the literature leading to coeffects.

2.1 Background, scalars and vectors

The λ-calculus is asymmetric – it maps a context with *multiple* variables to a *single* result. An expression with n free variables of types τ_i can be modelled by a function $\tau_1 \times \ldots \times \tau_n \to \tau$ with a product on the left, but a single value on the right. Effect systems attach effect annotations to the result τ. In coeffect systems, we attach coeffects to the context $\tau_1 \times \ldots \times \tau_n$ and we often (but not always) have one coeffect per variable. We call the overall coeffect a *vector* consisting of *scalar* coeffects. This asymmetry explains why coeffect systems are not trivially dual to effect systems.

It is useful to clarify how vectors are used in this paper. Suppose we have a set $\mathcal{C}$ of *scalars*. A vector R over $\mathcal{C}$ is a tuple $\langle r_1, \ldots, r_n \rangle$ of scalars. We use letters like R, S, T for vectors and r, s, t for scalars.[1] We also say that the *shape* of a vector $[R]$ (or more generally any container) is the set of *positions* in a vector. So, a vector of length n has shape $\{1, 2, \ldots, n\}$.

Just as in scalar-vector multiplication, we lift any binary operation $\bullet$ on scalars into a scalar-vector one: $s \bullet R = \langle s \bullet r_1, \ldots, s \bullet r_n \rangle$. Given two vectors R, S of the same shape, containing partially ordered scalars, we write $R \leq S$ for the pointwise extension of $\leq$ on scalars. Finally, the associative operation $\times$ concatenates vectors.

We note that an environment Γ containing n uniquely named, typed variables is also a vector, but we continue to write ',' for the product, so $\Gamma_1, x\!:\!\tau, \Gamma_2$ should be seen as $\Gamma_1 \times \langle x\!:\!\tau \rangle \times \Gamma_2$.

2.2 Bounded reuse

Bounded linear logic provides a modality that limits the number of times a proposition (variable) can be reused [7]. A type system corresponding to this logic can be used, for example, to restrict well-typed terms to polynomial-time algorithms. A proposition $!_k A$ means that A can be used at most k times. For uniformity with later notation, we write propositions A as τ. Our work attaches a vector of annotations to sets of assumptions, using the $@$ operator, *i.e.*, $\tau_1, ..., \tau_n @ \langle k_1, ..., k_n \rangle$, rather than writing bounds for each assumption as in $!_{k_1} A_1, ..., !_{k_n} A_n$.

Bounded linear logic includes explicit weakening and contraction rules that affect the multiplicity. Following the original logical style (but with our notation), these are written as:

$$\text{(weak)}\ \frac{\Gamma @ R \vdash \tau}{\Gamma, \tau_0 @ R \times \langle 0 \rangle \vdash \tau} \qquad \text{(contr)}\ \frac{\Gamma_1, \tau_0, \tau_0, \Gamma_2 @ R \times \langle s, t \rangle \times Q \vdash \tau}{\Gamma_1, \tau_0, \Gamma_2 @ R \times \langle s + t \rangle \times Q \vdash \tau}$$

The context $\Gamma @ R$ includes a *coeffect annotation* R which is a vector $\langle r_1, \ldots, r_n \rangle$ of the same length as Γ (a side-condition omitted for brevity). In weakening, unused propositions are annotated with 0 (no uses), while in contraction, multiple occurrences of a proposition are joined by adding the number of uses.

Bounded linear coeffects. The system in Figure 1 fleshes out the idea into a simple calculus. Variable access (*var*) has a singleton context with a singleton coeffect vector $\langle 1 \rangle$. Weakening (*weak*) extends the free-variable context with an unused variable and the coeffect with an associated scalar 0. Explicit contraction (*contr*) and exchange (*exch*) rules manipulate variables in the context and modify the annotations accordingly – adding the number of uses in contraction and switching vector elements in exchange.

For abstraction (*abs*), we know the number of uses of the parameter variable x and attach it to the function type $\tau_1 \xrightarrow{s} \tau_2$ as a *latent* coeffect. The remaining variables in Γ are annotated with the remaining coeffect vector R, specifying *immediate* coeffects.

Application (*app*) describes call-by-name evaluation. Applying a function that uses its parameter t-times to an argument that uses variables in Γ_2 S-times means that, in total, the variables in Γ_2 will

[1] For better readability, the paper distinguishes different structures using colours. However ignoring the colour does not introduce any ambiguity.

$$\text{(var)}\ \frac{}{x\!:\!\tau @ \langle 1 \rangle \vdash x : \tau} \qquad \text{(weak)}\ \frac{\Gamma @ R \vdash e : \tau}{\Gamma, x\!:\!\tau_0 @ R \times \langle 0 \rangle \vdash e : \tau}$$

$$\text{(sub)}\ \frac{\Gamma @ R \vdash e : \tau}{\Gamma @ R' \vdash e : \tau}\ (R \leq R') \qquad \text{(abs)}\ \frac{\Gamma, x\!:\!\tau_1 @ R \times \langle s \rangle \vdash e : \tau_2}{\Gamma @ R \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

$$\text{(app)}\ \frac{\Gamma_1 @ R \vdash e_1 : \tau_1 \xrightarrow{t} \tau_1 \quad \Gamma_2 @ S \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 @ R \times (t * S) \vdash e_1\, e_2 : \tau_2}$$

$$\text{(contr)}\ \frac{\Gamma_1, y\!:\!\tau_0, z\!:\!\tau_0, \Gamma_2 @ R \times \langle s, t \rangle \times Q \vdash e : \tau}{\Gamma_1, x\!:\!\tau_0, \Gamma_2 @ R \times \langle s + t \rangle \times Q \vdash e[z, y \leftarrow x] : \tau}$$

$$\text{(exch)}\ \frac{\Gamma_1, x\!:\!\tau_1, y\!:\!\tau_2, \Gamma_2 @ R \times \langle s, t \rangle \times Q \vdash e : \tau}{\Gamma_1, y\!:\!\tau_2, x\!:\!\tau_1, \Gamma_2 @ R \times \langle t, s \rangle \times Q \vdash e : \tau}$$

Figure 1: Bounded reuse: Type & coeffect system in the λ-calculus

be used $(t * S)$-times. Recall that $t * S$ is a scalar multiplication of a vector. Meanwhile, the variables in Γ_1 are used just R-times when reducing the expression e_1 to a function value.

Finally, the sub-coeffecting rule (*sub*) safely overapproximates the number of uses by the pointwise $\leq$ relation. We can view any variable as being used a greater number of times than it actually is.

Example. To demonstrate, consider a term $(\lambda v.x + v + v)\ (x + y)$. According to the call-by-name intuition, the variable x is used three times – once directly inside the function and twice via the variable v after substitution. Similarly, y is used twice. Eliding the derivation of the function body's coeffect, abstraction yields:

$$\text{(abs)}\ \frac{x\!:\!\mathbb{Z}, v : \mathbb{Z} @ \langle 1, 2 \rangle \vdash x + v + v : \mathbb{Z}}{x\!:\!\mathbb{Z} @ \langle 1 \rangle \vdash (\lambda v.x + v + v) : \mathbb{Z} \xrightarrow{2} \mathbb{Z}}$$

To avoid name clashes, we α-rename x to x' and later join x and x' using contraction. Assuming $(x' + y)$ is checked in a context that marks x' and y as used once, the application rule yields a judgment that is simplified as follows:

$$\text{(contr)}\ \frac{x\!:\!\mathbb{Z}, x'\!:\!\mathbb{Z}, y\!:\!\mathbb{Z} @ \langle 1 \rangle \times (2 * \langle 1, 1 \rangle) \vdash (\lambda v.x + v + v)\ (x' + y) : \mathbb{Z}}{x\!:\!\mathbb{Z}, x'\!:\!\mathbb{Z}, y\!:\!\mathbb{Z} @ \langle 1, 2, 2 \rangle \vdash (\lambda v.x + v + v)\ (x' + y) : \mathbb{Z}}$$
$$x\!:\!\mathbb{Z}, y\!:\!\mathbb{Z} @ \langle 3, 2 \rangle \vdash (\lambda v.x + v + v)\ (x + y) : \mathbb{Z}$$

The first step performs scalar multiplication, producing the vector $\langle 1, 2, 2 \rangle$. In the second step, we use contraction to join variables x and x' from the function and argument terms respectively.

It is worth pointing out that reduction by substitution yields $x + (x + y) + (x + y)$ which has the same coeffect as the original. We return to evaluation strategies in Section 4, and show that structural coeffect systems preserve types and coeffects under β-reduction.

2.3 Dataflow and data access

Dataflow languages, such as Lucid, describe computations over *streams* [20]. An expression is re-evaluated when new inputs are available (push) or when more output is demanded (pull). In causal dataflow, programs can access past values of a stream. We consider a language where **prev** e returns the previous value of e. In the language, **prev** (**prev** e) returns the second past value and so on.

An implementation of causal dataflow may cache past values of variables as an optimisation. The question is, how many past values should be cached? This can be approximated by a coeffect system.

Dataflow coeffects. The coeffect system for dataflow is similar to the one for bounded reuse, tracking a vector of natural numbers R as part of the context $\Gamma @ R$. Here, coeffects represent the maximal number of past values (*causality depth*) required for a variable.

$$\text{(contr)}\ \frac{\Gamma_1, y:\tau, z:\tau, \Gamma_2 @ R \times \langle s,t\rangle \times Q \vdash e : \tau}{\Gamma_1, x:\tau, \Gamma_2 @ R \times \langle \mathsf{max}(s,t)\rangle \times Q \vdash e[y, z \leftarrow x] : \tau}$$

$$\text{(app)}\ \frac{\Gamma_1 @ R \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma_2 @ S \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 @ R \times (t + S) \vdash e_1\ e_2 : \tau_2}$$

$$\text{(var)}\ \frac{}{x:\tau @ \langle 0\rangle \vdash x : \tau} \qquad \text{(prev)}\ \frac{\Gamma @ R \vdash e : \tau}{\Gamma @ 1 + R \vdash \mathbf{prev}\ e : \tau}$$

Figure 2: Type and coeffect system for dataflow caching

$$\text{(exch)}\ \frac{\Gamma_1, x:\tau_1, y:\tau_2, \Gamma_2 @ r \cup s \cup t \cup q \vdash e : \tau}{\Gamma_1, y:\tau_2, x:\tau_1, \Gamma_2 @ r \cup t \cup s \cup q \vdash e : \tau}$$

$$\text{(app)}\ \frac{\Gamma_1 @ r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_1 \qquad \Gamma_2 @ s \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 @ r \cup t \cup s \vdash e_1\ e_2 : \tau_2}$$

$$\text{(param)}\ \frac{}{()@\{?\mathsf{p} : \tau\} \vdash ?\mathsf{p} : \tau} \qquad \text{(abs)}\ \frac{\Gamma, x:\tau_1 @ r \cup s \vdash e : \tau_2}{\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

Figure 3: Type and coeffect system for implicit parameters

Weakening, exchange, abstraction and sub-coeffecting are the same as in bounded linear coeffects, but the remaining rules differ. In Figure 2, accessed variables (*var*) are annotated with 0 meaning that no past value is required (only the current one). The (*prev*) rule crates caching requirements – it increments the number of required values for all variables used in e using scalar-vector addition.

Application and contraction have the same structure as before, but use different operators. If two variables are contracted, requiring s and t past values, then at most $\mathsf{max}(s,t)$ past values are needed (*contr*). That is, two caches are combined with the maximum of the two requirements, which satisfy the smaller requirements. In (*app*), the function requires t past values of its parameter. This means t past values of e_2 are needed which in turn requires S past values of its free variables Γ_2. Thus, we need $t + S$ past values of Γ_2 to perform the call (*e.g.*, we need $1 + S$ values to get 1 past value of the input τ_1, $2 + S$ values to get 2 past values of τ_1, *etc.*).

Example. As an example, consider a function $\lambda x.\mathbf{prev}\ (y + x)$ applied to an argument $\mathbf{prev}\ (\mathbf{prev}\ y)$. The body of the function accesses the past value of two variables, one free and one bound:

$$\text{(abs)}\frac{y:\mathbb{Z}, x:\mathbb{Z}@\langle 1,1\rangle \vdash \mathbf{prev}\ (y + x) : \mathbb{Z}}{y:\mathbb{Z}@\langle 1\rangle \vdash \lambda x.\mathbf{prev}\ (y + x) : \mathbb{Z} \xrightarrow{1} \mathbb{Z}}$$

The expression always requires the previous value of y and adds it to a previous value of the parameter x. Evaluating the value of the argument $\mathbf{prev}\ (\mathbf{prev}\ y)$ requires two past values of y and so the overall requirement is 3 past values:

$$\begin{array}{l}\text{(app)}\ \dfrac{y:\mathbb{Z}@\langle 1\rangle \vdash \lambda x.\ \ldots \qquad y':\mathbb{Z}@\langle 2\rangle \vdash (\mathbf{prev}\ (\mathbf{prev}\ y')) : \mathbb{Z}}{\text{(contr)}\ \dfrac{y:\mathbb{Z}, y':\mathbb{Z}@\langle 1,3\rangle \vdash (\lambda x.\mathbf{prev}\ (y + x))\ (\mathbf{prev}\ (\mathbf{prev}\ y')) : \mathbb{Z}}{y:\mathbb{Z}@\langle 3\rangle \vdash (\lambda x.\mathbf{prev}\ (y + x))\ (\mathbf{prev}\ (\mathbf{prev}\ y)) : \mathbb{Z}}}\end{array}$$

The derivation uses (*app*) to get requirements $\langle 1,3\rangle$ and then (*contr*) to take the maximum, showing three past values are sufficient. Reducing the expression by substitution we get $\mathbf{prev}\ (y + (\mathbf{prev}\ (\mathbf{prev}\ y)))$. Semantically, this performs stream lookups $y[1] + y[3]$ where the indices are the number of enclosing **prev**s.

We previously used dataflow as an example of coeffects [15], but tracked caching requirements on the whole context. The system outlined here is more powerful and practically useful, with finer-grained coeffects tracking per-variable caching requirements.

2.4 Implicit parameters

As our third example, we revisit Haskell implicit parameters [9] used in our earlier coeffect work [15]. Implicit parameters are variables that mix aspects of dynamic and lexical scoping. Implicit parameters are a distinct syntactic category to variables and we write them as $?\mathsf{p}$. For simplicity, we omit let-binding for implicit parameters and focus just on tracking requirements.

Implicit parameters coeffects. Implicit parameters are a whole-context coeffect not linked to ordinary variables. We keep track of sets of implicit parameters that are required by an expression (and their types). For example $\Gamma@\{?\mathsf{p}_1 : \tau_1, \ldots, ?\mathsf{p}_n : \tau_n\}$ means that a context provides ordinary variables Γ and values for implicit parameters $?\mathsf{p}_i$. Unlike in the previous examples, we no longer need to distinguish between coeffects attached to variables (scalars) and coeffects attached to contexts (vectors), so we write r, s, t for both.

Despite the differences, the type system in Figure 3 follows the same structure as the earlier two examples. Context requirements are created when accessing an implicit parameter, in a system-specific rule (*param*). Structural rules (exchange, weaken, contract) do not affect the coeffects. For example, parameters are reordered in (*exch*), but this has no effect as set union $\cup$ is commutative.

In abstraction and application, the structural $\times$ operator (previously vector concatenation) becomes $\cup$. Sets of implicit parameters are not associated to individual variables and so they are unioned. The (*app*) rule uses $\cup$ to combine the implicit parameters required by the function with the requirements of the argument too.

We call this a *flat* coeffect system since coeffects have only one shape (there is no scalar/vector distinction). Other flat coeffect systems may use a richer structure [15]. In particular, the operations used in abstraction and application may differ (to accommodate over-approximation). We return to this in Section 3.5.

Example. Unlike structural (per-variable) coeffect systems, flat (whole-context) systems do not necessarily have principal coeffects. This arises from the (*abs*) rule which can freely split requirements between the function type and the declaring context. Consider a function $\lambda().?\mathsf{p}_1 + ?\mathsf{p}_2$. There are nine possible type and coeffect derivations, two of which are:

$$\emptyset@\{\} \vdash \lambda().?\mathsf{p}_1 + ?\mathsf{p}_2 : \mathsf{unit} \xrightarrow{\{?\mathsf{p}_1:\mathbb{Z}, ?\mathsf{p}_2:\mathbb{Z}\}} \mathbb{Z}$$
$$\emptyset@\{?\mathsf{p}_1 : \mathbb{Z}\} \vdash \lambda().?\mathsf{p}_1 + ?\mathsf{p}_2 : \mathsf{unit} \xrightarrow{\{?\mathsf{p}_2:\mathbb{Z}\}} \mathbb{Z}$$

In the first case, both parameters are dynamically scoped and have to be provided by the caller. In the second case, the parameter $?\mathsf{p}_1$ is available in the declaring scope and so it is (lexically) captured.

Although structural coeffects have more desirable syntactic properties, we aim to capture this non-principality too as it is practically useful – not only in Haskell's implicit parameters, but also in resource rebinding in distributed systems such as Acute [17].

2.5 Pathways to coeffects

This paper largely follows work on effect systems and their link to categorical semantics. We briefly review this and other directions leading to coeffects. An eager reader can return to this section later.

Effect systems. Effect systems [6] track effectful operations of computations such as memory access or lock usage [4]. They are written as judgments $\Gamma \vdash e : \tau\ \&\ \rho$ associating effects ρ with the result. Effect systems capture *output effects* where, as Tate puts it, *"all computations with [an] effect can be thunked as pure computations for a domain-specific notion of purity."* [18]. This thunking is typically a λ-abstraction. Given an effectful expression e, the function $\lambda x.e$ is an effect-free value that delays all effects:

$$\text{(abs)}\ \frac{\Gamma, x:\tau_1 \vdash e : \tau_2\ \&\ \rho}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\rho} \tau_2\ \&\ \emptyset}$$

Coeffects do not follow this pattern. In contrast to effect systems, context requirements cannot be easily "thunked" as pure values. Lambda abstraction can split context requirements between *immediate* and *latent* requirements. This is akin to how lambda abstraction splits a free-variable context into the bound parameter (call site) and the remaining free variables (declaration site).

Categorical semantics. Moggi models effectful computations as functions of type $\tau_1 \to M\,\tau_2$ where M is a monad providing composition of effectful computations [10]. Wadler and Thiemann link effect systems to monads via annotated monads $\tau_1 \to M^{\rho}\tau_2$ [21], whose semantics has been provided by Katsumata [8].

Context-dependent computations require a different model. Uustalu and Vene use functions $C\tau_1 \to \tau_2$ where C is a *comonad* [19]. Our earlier work [15] used indexed comonads with denotations $C^{r}\tau_1 \to \tau_2$ adding annotations akin to Wadler and Thiemann. In Section 5 we extend indexed comonads to capture the general coeffect systems of this paper, in the style of Katsumata.

Language and meta-language. Moggi uses monads in two systems [10]. In the first system, a monad is used to model an effectful language itself – the semantics of a language uses a specific monad. In the second system, monads are added as type constructors, together with syntax corresponding to *unit* and *bind* operations.

For context dependence, Uustalu and Vene follow the first approach using comonads for their semantics [19]. Contextual-Modal Type Theory (CMTT) of Nanevski *et al.* [11] follows the latter approach, adding a comonad to the language via the $\Box$ modality of modal S4. We focus on concrete languages using the first approach. A "coeffect meta-language" is an interesting future work.

Sub-structural systems Sub-structural type systems restrict how a context is used. This is achieved by removing some of the structural typing rules (weakening, contraction, exchange). As the bounded linear logic example (Section 2.2) shows, our system can be viewed as a generalization of sub-structural type systems.

3. The coeffect calculus

The three calculi shown in the previous section track two kinds of contextual properties: bounded reuse and dataflow are structural (per-variable) systems, and implicit parameters and our earlier coeffect systems [15] are flat (whole-context) systems. This section presents our primary contribution: the general coeffect calculus.

The calculus is parameterised by an algebraic structure of coeffects. To capture both structural and flat systems, coeffect annotations are indexed by a *shape*. In flat systems, the shape is a singleton set $\{*\}$ and so annotations are *scalar* values. Structural systems use shapes matching the number of variables in a free-variable context $\{1, \ldots, n\}$ and so annotations are *vectors*. However, the coeffect calculus could also use shapes describing trees and other structures.

3.1 Understanding coeffects: syntax and semantics

The coeffect calculus provides both an analysis of context dependence (its coeffect system) and a semantics for context (see Section 5). These two features of the calculus provide different perspectives on coeffect annotations R in a judgment $\Gamma @ R \vdash e : \tau$.

- Syntactically, coeffects model *contextual requirements* and may be overapproximated, so that more capabilities are required than necessary at runtime.
- Semantically, coeffects model *contextual capabilities* and behave like containers of capabilities, such that the semantics may throw away capabilities that will not be needed.

Thus there are two dual ways to understand coeffect annotations. Each perspective implies an alternate reading of the typing rules.

- As *contextual requirements*, the rules should be read top-down. The requirements of multiple sub-terms are *merged* and the requirements of a function body are *split* between immediate (declaration-site) and latent (call-site) coeffects.
- As *contextual capabilities*, the rules should be read bottom-up. The capabilities provided to a larger term are *split* between subterms; for functions, the capabilities of declaration-site and call-site are *merged* and passed to the body.

The reason for this asymmetry follows from the fact that context appears in a *negative position* in the model. In Section 5, the denotation of a judgment $\Gamma @ R \vdash e : \tau$ is a function of the form $\mathsf{D}_R[\![\Gamma]\!] \to [\![\tau]\!]$ where $\mathsf{D}_R[\![\Gamma]\!]$ encodes the contextual capabilities used to evaluate a term. Similarly a function $\tau_1 \xrightarrow{s} \tau_2$ has a model of the form $\mathsf{D}_s[\![\tau_1]\!] \to [\![\tau_2]\!]$ with additional contextual capabilities attached to the input.

3.2 Structure of coeffects

We describe the algebraic structure of coeffects in three steps. First, we define a *coeffect scalar* structure which defines the basic building blocks of coeffect information; then we define *coeffect shapes* which determines how coeffect scalar values are related to the free-variable context. Finally, we define the *coeffect algebra* which consists of shape-indexed coeffect scalar values.

For example, in bounded reuse the coeffect scalar structure comprise natural numbers $\mathbb{N}$ with $+$ and $*$ operators. The shape for bounded reuse is the length of the free-variable context and so the coeffect annotation is a vector of matching length. Finally, the coeffect algebra specifies how vectors are concatenated and split in abstraction and application.

In the coeffect system of the calculus, contexts are annotated with shape-indexed coeffects (*e.g.*, vectors) as in $\Gamma @ R \vdash e : \tau$. However, functions take just a single input parameter and so are annotated with scalar coeffect values as in $\sigma \xrightarrow{r} \tau$. From now on, we write σ for the *source* and τ for the *target* of function types.

Coeffect scalar. Coeffect scalar structures are equipped with two operations. In bounded reuse, those were $*$ for sequencing (in function application) and $+$ for context sharing (in contraction). Additional structure is needed for variable access and sub-coeffecting.

Definition 1. A *coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ comprises a set $\mathcal{C}$ together with elements $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$, relation $\leq$ and binary operations $\circledast, \oplus$ such that $(\mathcal{C}, \circledast, \mathsf{use})$ and $(\mathcal{C}, \oplus, \mathsf{ign})$ are monoids, $(\mathcal{C}, \leq)$ is a pre-order, and the following distributivity axioms hold:

$$(r \oplus s) \circledast t = (r \circledast t) \oplus (s \circledast t)$$
$$t \circledast (r \oplus s) = (t \circledast r) \oplus (t \circledast s)$$

The operation $\circledast$ must form a monoid with use to guarantee an underlying category in the semantics (Section 5). It models sequential composition with variable access (use) as the identity. The other element (ign) is used for variables that are not accessed. The operation $\oplus$ combines coeffects for contexts used in multiple places (contraction). The notation is inspired by the bounded reuse example, which uses coeffect scalar structure $(\mathbb{N}, *, +, 1, 0, \leq)$, but be aware that $\circledast$ and $\oplus$ are not always multiplication and addition.

Coeffect annotations R can be viewed as *containers* of scalar coeffects. For structural coeffects, the container is a vector, while for flat coeffects it is a trivial singleton container. The following definition takes inspiration from the work of Abbott *et al.* [1] which describes containers in terms of *shapes* and a set of *positions*.

Coeffect shapes. The coeffect system is parameterised by a set of shapes $\mathcal{S}$. A coeffect annotation is indexed by a shape $s \in \mathcal{S}$ calculated from the shape of the free-variable context Γ. The correspondence is not necessarily bijective. For example, flat coeffect systems have just a single shape $\mathcal{S} = \{\star\}$.

Thus, in the judgment $\Gamma @ R \vdash e : \tau$, the coeffect annotation R is drawn from the set of coeffect scalars $\mathcal{C}$ indexed by the shape of Γ. We write $s = [\Gamma]$ for the shape corresponding to Γ. We define shapes by a *set* of positions and so we can define $R \in s \to \mathcal{C}$ as a mapping from positions (defined by the shape) to scalar coeffects. We usually write this as the exponent $R \in \mathcal{C}^s$.

The set of shapes is equipped with an operation that combines shapes (when we combine variable contexts), an operation that computes shape from the free-variable contexts, and two special shapes in $\mathcal{S}$ representing empty context and singleton context.

Definition 2. A *coeffect shape* structure $(\mathcal{S}, [\text{–}], \diamond, \hat{0}, \hat{1})$ comprises a set $\mathcal{S}$ with a binary operation $\diamond$ on $\mathcal{S}$ for shape composition, a mapping from contexts to shapes $[\Gamma] \in \mathcal{S}$, and elements $\hat{0}, \hat{1} \in \mathcal{S}$ such that $(\mathcal{S}, \diamond, \hat{0})$ is a monoid and $[\text{–}]$ is partially specified on empty and singleton free-variable contexts by:

$$[\emptyset] = \hat{0} \qquad [v : \tau] = \hat{1}$$

This means that the elements $\hat{0}$ and $\hat{1}$ represent the shapes of empty and singleton free-variable contexts respectively. As said earlier, we use two kinds of shape structure:

- Structural coeffect shape is defined as $(\mathbb{N}, |\text{–}|, +, 0, 1)$. We treat numbers as sets $0 = \{\}, 1 = \{\emptyset\}, 2 = \{\emptyset, 1\}, 3 = \{\emptyset, 1, 2\} \ldots$ (so that a number is a set of positions). The shape mapping $|\Gamma|$ returns the number of variables in Γ. Empty and singleton contexts are annotated with 0 and 1, respectively, and shapes of combined contexts are added so that $|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|$. Therefore, a coeffect annotation is a *vector* $R \in \mathcal{C}^n$ and assigns a coeffect scalar $R(i) \in \mathcal{C}$ for each variable x_i in the context.
- Flat coeffect shape is defined as $(\{\star\}, \mathsf{star}, \diamond, \star, \star)$ where $\mathsf{star}(\Gamma) = \star$ and $\star \diamond \star = \star$ where $\star = \{\emptyset\}$. That is, there is a single shape $\star$ with a single position and all free-variable contexts have the same shape. Therefore, a coeffect annotation is drawn from $\mathcal{C}^\star$ which is isomorphic to $\mathcal{C}$ and so a coeffect scalar $r \in \mathcal{C}$ is associated with every free-variable context.

Using a shape with *no* positions reduces our system to the simply-typed λ-calculus with no context annotations. Trees can also be used to build a system akin to bunched typing [12].

Coeffect algebra. The coeffect calculus annotates judgments with shape-indexed, or *shaped*, coeffects. The *coeffect algebra* structure combines a coeffect scalar and coeffect shape structure to define shaped coeffects and operations for combining these. In Section 2, shaped coeffects were combined by the tensor $\times$ in structural examples and $\cup$ in the implicit parameters example. To capture the examples so far and those described previously [15], we distinguish two operators for combining shaped coeffects.

Definition 3. Given a coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ and a coeffect shape $(\mathcal{S}, [\text{–}], \diamond, \hat{0}, \hat{1})$ a *coeffect algebra* extends the two structures with $(\rtimes, \ltimes, \bot)$ where $\bot \in \mathcal{C}^{\hat{0}}$ is a coeffect annotation for the empty context and $\rtimes, \ltimes$ are families of operations that combine coeffect annotations indexed by shapes. That is $\forall n, m \in \mathcal{S}$:

$$\ltimes_{m,n}, \rtimes_{m,n} \; : \; \mathcal{C}^m \times \mathcal{C}^n \to \mathcal{C}^{m \diamond n}$$

A coeffect algebra induces the following two additional operations:

$$\begin{array}{ll} \langle \text{–} \rangle \; : \; \mathcal{C} \to \mathcal{C}^{\hat{1}} & \qquad \circledast_m \; : \; \mathcal{C} \times \mathcal{C}^m \to \mathcal{C}^m \\ \langle x \rangle \; = \; \lambda \hat{1}.x & \qquad r \circledast S \; = \; \lambda s. r \circledast (S(s)) \end{array}$$

$\langle \text{–} \rangle$ lifts a scalar coeffect to a shaped coeffect indexed by the singleton context shape. The $\circledast_m$ operation is a left multiplication of a vector by a scalar. As we always use lower-case for scalars and upper-case for vectors, using the same symbol is not ambiguous. We also tend to omit the subscript m and write just $\circledast$.

$\boxed{\Gamma @ R \vdash e : \tau}$

$$(\text{const}) \; \frac{}{() @ \bot \vdash c : \iota} \qquad (\text{var}) \; \frac{}{(x : \tau) @ \langle \mathsf{use} \rangle \vdash x : \tau}$$

$$(\text{abs}) \; \frac{\Gamma, x : \sigma @ R \ltimes \langle s \rangle \vdash e : \tau}{\Gamma @ R \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$$

$$(\text{app}) \; \frac{\Gamma_1 @ R \vdash e_1 : \sigma \xrightarrow{t} \tau \qquad \Gamma_2 @ S \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ R \ltimes (t \circledast S) \vdash e_1 \; e_2 : \tau}$$

$$(\text{let}) \; \frac{\Gamma_1 @ S \vdash e_1 : \sigma \qquad \Gamma_2, x : \sigma @ R \rtimes \langle t \rangle \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 @ R \ltimes (t \circledast S) \vdash \mathbf{let} \; x = e_1 \; \mathbf{in} \; e_2 : \tau}$$

$$(\text{ctx}) \; \frac{\Gamma @ R \vdash e : \tau \qquad \Gamma' @ R' \rightsquigarrow \Gamma @ R, \theta}{\Gamma' @ R' \vdash \theta e : \tau}$$

$\boxed{\Gamma' @ R' \rightsquigarrow \Gamma @ R, \theta}$

$$(\text{weak}) \qquad \Gamma, x : \tau @ R \ltimes \langle \mathsf{ign} \rangle \rightsquigarrow \Gamma @ R, \emptyset$$

$$(\text{exch}) \; \begin{array}{l} \Gamma_1, y : \sigma, x : \tau, \Gamma_2 @ R \ltimes \langle t \rangle \ltimes \langle s \rangle \ltimes Q \rightsquigarrow \\ \Gamma_1, x : \tau, y : \sigma, \Gamma_2 @ R \rtimes \langle s \rangle \rtimes \langle t \rangle \rtimes Q, \emptyset \end{array}$$

$$(\text{contr}) \; \begin{array}{l} \Gamma_1, x : \tau, \Gamma_2 @ R \ltimes \langle s \oplus t \rangle \ltimes Q \rightsquigarrow \\ \Gamma_1, y : \tau, z : \tau, \Gamma_2 @ R \rtimes \langle s \rangle \rtimes \langle t \rangle \rtimes Q, [y, z \mapsto x] \end{array}$$

$$(\text{sub}) \; \begin{array}{l} \Gamma_1, x : \tau, \Gamma_2 @ R \ltimes \langle s' \rangle \ltimes T \rightsquigarrow \\ \Gamma_1, x : \tau, \Gamma_2 @ R \rtimes \langle s \rangle \rtimes T, \emptyset \end{array} \qquad (s \leq s')$$

Figure 4: The general coeffect calculus

The operators $\ltimes$ and $\rtimes$ combine shaped coeffects associated with two contexts. For example, assume we have Γ_1 and Γ_2 with coeffects $R \in \mathcal{C}^m$ and $S \in \mathcal{C}^n$. In the structural system, the context shapes m, n denote the number of variables in the two contexts. The combined context Γ_1, Γ_2 has a shape $m \diamond n$ and the combined coeffects $R \rtimes S, R \ltimes S \in \mathcal{C}^{m \diamond n}$ are indexed by that shape.

For structural coeffect systems such as bounded reuse, both $\ltimes$ and $\rtimes$ are just the tensor product $\times$ of vectors. However, we need to distinguish them for flat coeffect systems discussed later.

The difference is explained by the semantics (Section 5), where $R \rtimes S$ is an annotation of the codomain of a morphism that merges the capabilities provided by two contexts (in the syntactic reading, splits the context requirements); $R \ltimes S$ is an annotation of the domain of a morphism that splits the capabilities of a single context into two parts (in the syntactic reading, merges their context requirements). Syntactically, this means that we always use $\rtimes$ in rule *premises* and $\ltimes$ in *conclusions*. For now, it suffices to use the bounded-reuse intuition and read the operations as tensor products.

The distinction between $\ltimes$ and $\rtimes$ provides flexibility to the calculus. For example, it is possible to instantiate the calculus such that structural rules are not permitted. In the case of flat and structural classes of system, different properties of $\ltimes$ and $\rtimes$ permit free use of structural rules. This is seen in the following sections.

3.3 General coeffect type system

In the previous section, we developed an algebraic structure capable of capturing different concrete context-dependent properties discussed in Section 2. Now, we use the structure to define the general coeffect calculus in Figure 4.

Coeffect annotations on free-variable contexts are shape-indexed, where for some shape $s \in \mathcal{S}$ then $R, S, T \in \mathcal{C}^s$. Function types are annotated with coeffects scalars $r, s, t \in \mathcal{C}$. The rules of Figure 4 manipulate coeffect annotations using the coeffect algebra

operations $(\barwedge, \leftthreetimes, \bot)$ and the derived constructs $\langle - \rangle$ and $\circledast$. Free-variable contexts Γ are treated as vectors modulo duplicate use of variables – associativity is built-in. Variable order matters, but can be changed using the structural rules. Structural rules are expressed using a helper relation, written $\rightsquigarrow$.

Typing rules. Constants (*const*) and variables (*var*) annotate the context with special values. The empty unused context is annotated with $\bot \in \mathcal{C}^{\hat{0}}$ and the singleton context with $\langle \mathsf{use} \rangle \in \mathcal{C}^{\hat{1}}$. Note that the shapes $\hat{0}, \hat{1}$ match the shape of the variable contexts.

Lambda abstraction *splits* the context requirements using $\barwedge$ into a coeffect R and a coeffect $\langle s \rangle$ of a shape $\hat{1}$ (semantically, it *merges* capabilities provided by the declaration-site and call-site contexts). In structural systems such as bounded reuse, this identifies coeffect associated with the bound variable, because $\barwedge$ is not commutative.

The (*app*) rule follows the patterns seen earlier – it uses the scalar-vector multiplication $(t \circledast S)$ of the coeffects S from the argument (associated with Γ_2) and the latent coeffect t of the function. Using the syntactic reading, it then *merges* context requirements for Γ_1 and Γ_2. In the dual semantic reading, it *splits* the provided context into two parts passed to the sub-expressions.

The typing of let-binding (*let*) corresponds to the typing of an expression $(\lambda x.e_2)\ e_1$. Syntactically, the context requirements are first split using $\barwedge$ and then re-combined using $\leftthreetimes$.

Structural rules. The coeffect-annotated context can be transformed using structural rules that are not syntax-directed. These are captured by (*ctx*), which uses a helper relation representing context transformations $\Gamma'@R' \rightsquigarrow \Gamma@R, \theta$. The rule models that a context used in the rule conclusion $\Gamma'@R'$ can be transformed to a context required by the premise $\Gamma@R$ (using the semantic bottom-up reading). In the rule, θ is a variable substitution generated by the transformation, which is used in the (*contr*) rule.

Exchange and contraction decompose and reconstruct coeffect annotations using $\barwedge_{m,n}$ (in assumption) and $\leftthreetimes_{m,n}$ (in conclusion). The shape subscripts are omitted, but we require the shapes to match using $m = [\Gamma_1]$ and $n = [\Gamma_2]$.

The (*weak*) rule drops an ignored variable annotated with $\langle \mathsf{ign} \rangle$ (compare with (*var*) annotated using $\langle \mathsf{use} \rangle$). The (*exch*) rule swaps variables/coeffects while (*contr*) combines coeffects using $\oplus$ to represent sharing of the context. Finally, (*sub*) represents subcoeffecting and can be applied (pointwise) to any scalar coeffect.

3.4 Structural coeffects

The coeffect system uses a general notion of context shape, but it was designed with structural and flat systems in mind. The structural system is new in this paper and so we look at it first.

Recall the coeffect shapes that characterise structural systems: the shape is formed by natural numbers (with addition) modelling the number of variables in the context. The coeffect algebra is therefore formed by the free monoid (lists/vectors) over a coeffect scalar. This means that the system keeps a vector of coeffect scalar annotations – one for each variable. An empty context (*e.g.*, in the (*const*) rule) is annotated with a zero-length vector.

Definition 4. Given a coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ a *structural coeffect system* has:

- Coeffect shape $(\mathbb{N}, |-|, +, 0, 1)$ formed by natural numbers
- Coeffect algebra $(\times, \times, \langle\rangle)$ where $\times$ and $\langle\rangle$ are shape-indexed versions of the binary operation and the unit of a free monoid over $\mathcal{C}$. That is $\times : \mathcal{C}^n \times \mathcal{C}^m \to \mathcal{C}^{n+m}$ appends vectors (lists) and $\langle\rangle : \mathcal{C}^0$ represents empty vectors (lists).

The definition is valid since the shape operations form a monoid $(\mathbb{N}, +, 0)$ and $[-]$ (calculating the length of a list) is a monoid homomorphism from the free monoid to the monoid of shapes.

Examples. Defining a concrete structural coeffect system is easy, we just provide the coeffect scalar structure and the rest is free.

- To recreate the system for bounded reuse, we use coeffect scalars formed by $(\mathbb{N}, *, +, 1, 0, \leq)$. As in the system of Figure 1, *used* variables are therefore annotated with 1 and *unused* with 0. Contraction adds the number of uses via $+$ and application (sequencing) multiplies the uses.
- *Dataflow* uses natural numbers (of past values), but differently: $(\mathbb{N}, +, \mathsf{max}, 0, 0, \leq)$. Variables are initially annotated with 0 (and can be incremented using the **prev** keyword). Annotations of a shared variable are combined by taking maximum (of past values needed) and sequencing uses $+$.
- Another use of the system is to track *variable liveness*. The annotations are formed by $\mathcal{C} = \{\mathsf{D}, \mathsf{L}\}$ where L represents a *live* (used) variable and D represents a *dead* (unused) variable. The coeffect scalar structure is $(\mathcal{C}, \sqcap, \sqcup, \mathsf{L}, \mathsf{D}, \sqsubseteq)$ where $\mathsf{D} \sqsubseteq \mathsf{L}$.

  In sequential composition ($\sqcap$), a variable is live only if it is required by both of the computations ($\mathsf{L} \sqcap \mathsf{L} = \mathsf{L}$), otherwise it is marked as dead (D). A computation is not evaluated if its result is not needed. A shared variable ($\sqcup$) is live if either of the uses is live ($\mathsf{D} \sqcup \mathsf{D} = \mathsf{D}$, otherwise L).

Structural liveness is a practically useful, precise version of an example from our earlier work, which was a flat system overapproximating liveness of the entire context [15]. Since $\barwedge = \leftthreetimes = \times$, structural rules (weaken, contract, exchange) are freely permitted, modifying the coeffects accordingly.

3.5 Flat coeffects

The same general coeffect system can be used to define systems that track whole-context coeffects as in the implicit parameters example (Section 2.4). Flat coeffect systems are characterised by a singleton set of shapes, such as $\{\star\}$. In this setting, the context annotations $\mathcal{C}^\star$ coincide with coeffect scalars $\mathcal{C}$.

In addition to the coeffect scalar structure, we also need to define $\leftthreetimes$ and $\barwedge$. Our examples of flat coeffects use $\oplus$ (merging of scalar coeffects) for $\leftthreetimes$ (merging of shaped coeffect annotations). However, the $\barwedge$ operation needs to be provided explicitly. Thus the general form of flat coeffect system is defined as follows.

Definition 5. Given a coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ and an operation $\wedge : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ such that $(r \wedge s) \leq (r \oplus s)$, we define:

- Flat coeffect shape $(\{\star\}, \mathit{const}\ \star, \diamond, \star, \star)$ where $\star \diamond \star = \star$
- Flat coeffect algebra $(\wedge, \oplus, \mathsf{ign})$, *i.e.*, the $\leftthreetimes = \oplus$ and $\bot = \mathsf{ign}$ with the additional binary operation $\barwedge = \wedge$.

The additional axiom $(r \wedge s) \leq (r \oplus s)$ is required for β-equality in flat systems (see later Theorem 11, Section 4.2).

If $\barwedge$ is idempotent, then structural rules (weaken, contract, exchange) are freely permitted since any flat coeffect annotation r can be expanded to $r \barwedge r$. This property holds for all examples here, hence structural rules can always be applied.

If $\leftthreetimes$ is also idempotent (as in all our examples), then exchange and contraction rules preserve the coeffects of the assumption in the conclusion. Otherwise $(r \wedge s) \leq (r \oplus s)$ means that exchange and contraction behave as the (*sub*) rule for subcoeffecting.

Examples. Implicit parameters are the prime example of a flat coeffect system, but other examples include rebindable resources [17] and Haskell type classes [13].

In the implicit parameters system (Section 2.4), coeffect scalars are sets of name-type pairs $\mathcal{C} = \mathcal{P}(\mathsf{Name} \times \mathsf{Type})$. Variables are annotated with $\emptyset$ and coeffects are combined or split (in the top-down reading for (*abs*)) using set union $\cup$. Thus, the coeffect scalar structure is $(\mathcal{P}(\mathsf{Name} \times \mathsf{Types}), \cup, \cup, \emptyset, \emptyset, \subseteq)$ with $\wedge = \cup$.

Remark 6. We previously described flat systems for liveness and dataflow [15]. Turning a structural system to a flat system requires finding $\wedge$ that underapproximates the capabilities of combined contexts. For dataflow, this is given by the min function, which satisfies the requirement because $\mathsf{min}(r,s) \leq \mathsf{max}(r,s)$.

In flat dataflow, we annotate the entire context with the maximal number of past elements required overall. We use the same coeffect scalars $(\mathbb{N}, +, \mathsf{max}, 0, 0, \leq)$ as in the structural version, but with $\wedge = \mathsf{min}$. Abstraction (which is the only rule using $\wedge$) becomes:

$$\text{(abs)}\ \frac{\Gamma, x:\sigma @ \mathsf{min}(r,s) \vdash e : \tau}{\Gamma @ r \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$$

Both the declaration-site and call-site must provide at least the number of past values required by the body. The overapproximation means both r and s can be greater than actually required. For dataflow, we could enforce that immediate and latent coeffects are identical, but that would require treating $\bar{\times}$ as a partial function.

4. Equational theory

Each of the concrete coeffect systems discussed in this paper has a different notion of context dependence, much like various effectful languages have different notions of effects (such as state or exceptions). However, there are common equational properties that hold for all (or some) of the systems we consider.

The equational theory in this section illuminates the axioms of coeffect algebra and the semantics of the calculus. We discuss syntactic substitution as it can form the basis for reduction in a concrete operational semantics. We consider structural and flat systems separately. This provides better insight into how the two systems work and differ. In particular, call-by-name evaluation is *coeffect preserving* for all structural, but only some flat systems.

The properties and proofs in this section are syntactic. In Section 5.5 we show that our denotational model of the coeffect calculus is sound with respect to the equational theory here.

We use standard syntactic substitution written as $e_1[x \leftarrow e_2]$, β-reduction and η-expansion, written as $\rightsquigarrow_\beta$ and $\rightsquigarrow_\eta$. Equality of terms e_1 and e_2, written as $\equiv$ is defined w.r.t their contexts, types and coeffects and is written $\Gamma @ R \vdash e_1 \equiv e_2 : \tau$.

4.1 Structural coeffect systems

For structural coeffect systems, recall that coeffects are vectors with $\times = \bar{\times} = \times$ (vector concatenation) and $\bot = \langle\rangle$ (the empty vector), thus coeffect annotations comprise the free monoid over scalars. We first show substitution:

Lemma 7 (Substitution lemma)**.** *In a structural coeffect calculus with a coeffect scalar structure* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$*:*

$$\Gamma @ S \vdash e_s : \sigma \ \ \wedge \ \ \Gamma_1, x:\sigma, \Gamma_2 @ R_1 \times \langle r \rangle \times R_2 \vdash e_r : \tau$$
$$\Rightarrow \ \Gamma_1, \Gamma, \Gamma_2 @ R_1 \times (r \circledast S) \times R_2 \vdash e_r[x \leftarrow e_s] : \tau$$

Proof. By induction over the derivation for e_r using the free monoid $(\mathcal{C}, \times, \langle\rangle)$ and coeffect scalar axioms (full proof [14]). □

Because of the vector (free monoid) structure, coeffects R_1, R_2, and $\langle r \rangle$ for the receiving term e_r are uniquely associated with Γ_1, Γ_2, and x respectively. Therefore, substituting e_s (which has coeffects S) for x introduces the context dependencies specified by S which are composed with the requirements r on x. Using the substitution lemma, we can demonstrate β-equality:

$$\frac{\dfrac{\Gamma_1, x:\sigma @ R \times \langle r \rangle \vdash e_1 : \tau}{\Gamma_1 @ R \vdash \lambda x.e_1 : \sigma \xrightarrow{r} \tau} \qquad \Gamma_2 @ S \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ R \times (r \circledast S) \vdash (\lambda x.e_1) e_2 \equiv e_1[x \leftarrow e_2] : \tau}$$

As a result, β-reduction preserves the type and coeffects of a term. This gives the following subject reduction property:

Theorem 8 (Subject reduction)**.** *In a structural coeffect calculus, if* $\Gamma @ R \vdash e : \tau$ *and* $e \rightsquigarrow_\beta e'$ *then* $\Gamma @ R \vdash e' : \tau$*.*

Proof. Following from Lemma 7 and β-equality. □

Structural coeffect systems also exhibit η-equality, therefore satisfying both the *local soundness* and *local completeness* conditions of Pfenning and Davies [16]. This means that abstraction does not introduce too much, and application does not eliminate too much.

$$\frac{\Gamma @ R \vdash e : \sigma \xrightarrow{s} \tau \qquad x : \sigma @ \langle \mathsf{use} \rangle \vdash x : \sigma}{\Gamma, x : \sigma @ R \times (s \circledast \langle \mathsf{use} \rangle) \vdash e\, x : \tau}$$
$$\Gamma @ R \vdash \lambda x.e\, x \equiv e : \sigma \xrightarrow{s} \tau$$

The last step uses the equalities $s \circledast \langle \mathsf{use} \rangle = \langle s \circledast \mathsf{use} \rangle = \langle s \rangle$ arising from the monoid $(\mathcal{C}, \circledast, \mathsf{use})$ of the scalar coeffect structure.

This highlights another difference between coeffects and effects, as η-equality does not hold for many notions of effect. For example, in a language with output effects, $e = (\mathsf{print}\ \texttt{"hi"}; (\lambda x.x))$ has different effects to its η-converted form $\lambda x.ex$ because the immediate effects of e are hidden by the purity of λ-abstraction. In the coeffect calculus, the (*abs*) rule allows immediate contextual requirements of e to "float outside" of the enclosing λ. Furthermore, the free monoid nature of $\times$in structural coeffect systems allows the exact immediate requirements of $\lambda x.ex$ to match those of e.

4.2 Flat coeffect systems

The equational theory for flat coeffect systems is somewhat similar to effect systems where (co)effects are not linked to individual variables. In effectful languages, substituting an effectful computation for y in $\lambda x.y$ changes the latent effect associated with the function.

Similarly, for some of the flat coeffect systems, substituting a context-dependent computation for y in $\lambda x.y$ adds latent context requirements to the function type. However, this is not the case for *all* flat coeffect systems – for example, call-by-name reduction preserves types and coeffects for the implicit parameters system (which makes it a suitable model for Haskell). For other systems, we first briefly consider call-by-value reduction.

Call-by-value. The notion of *value* in coeffect systems differs from the usual syntactic understanding. As discussed earlier, a function $(\lambda x.e)$ is not necessarily a value in coeffect calculi, because it may not delay all context requirements of e. Thus a syntactic value v is a value if it has no immediate context requirements.

Definition 9. A syntactic value v is a *pure value* if $\Gamma @ \mathsf{Val} \vdash v : \tau$ where $\mathsf{Val} : \mathcal{C}^{[\Gamma]}$ is a coeffect indexed by the shape of Γ that always returns use. That is $\mathsf{Val} = \lambda n.\mathsf{use}$.

In call-by-value, the right-hand side of an application is evaluated to a pure value, which is then substituted for a variable. However, the discharging of coeffects prior to substitution is different for each coeffect system.

Recall that a flat coeffect system consists of coeffect scalars $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ together with a binary operation $\wedge$ on $\mathcal{C}$ such that the coeffect algebra structure is $(\wedge, \oplus, \mathsf{ign})$.

Lemma 10 (Call-by-value substitution)**.** *In a flat coeffect calculus with coeffect scalars* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ *and the* $\wedge$ *operator:*

$$\Gamma @ \mathsf{VAL} \vdash e_s : \sigma \ \ \wedge \ \ \Gamma_1, x:\sigma, \Gamma_2 @ r \vdash e_r : \tau$$
$$\Rightarrow \ \ \Gamma_1, \Gamma, \Gamma_2 @ r \vdash e_r[x \leftarrow e_s] : \tau$$

Proof. By induction over the coeffect derivation, using the fact that both x and e_s are annotated with use. □

Lemma 10 holds for all flat coeffect systems, but it is weak. To use it, the operational semantics must provide a way of partially

evaluating a term with requirements R to a value. Assuming a call-by-value reduction $\rightsquigarrow_{\mathsf{cbv}}$, using the above definition of value:

Theorem 11 (Call-by-value subject reduction)**.** *In a flat coeffect calculus, if* $\Gamma@r \vdash e : \tau$ *and* $e \rightsquigarrow_{\mathsf{cbv}} e'$ *then* $\Gamma@r \vdash e' : \tau$.

Proof. A direct consequence of Lemma 10, using the flat coeffect system requirement $(r \wedge s) \leq (r \oplus s)$ to prove β-equality. □

Call-by-name. A term $(\lambda x.e_1)\ e_2$ can be β-reduced in the call-by-name strategy even if both sub-expressions have contextual requirements.

We call a flat coeffect algebra *top-pointed* if use (the coeffect of variable use) is the greatest (top) coeffect scalar $\mathcal{C}$ and *bottom-pointed* if it is the smallest (bottom) coeffect scalar with respect to the order $\leq$. Liveness analysis is an example of top-pointed coeffects as $\mathsf{use} = \mathsf{L}$ and $\mathsf{D} \leq \mathsf{L}$.

Lemma 12 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ *and the* $\wedge$ *operator:*

$$\Gamma@s \vdash e_s : \sigma \quad \wedge \quad \Gamma_1, x : \sigma, \Gamma_2@r \vdash e_r : \tau$$
$$\Rightarrow \quad \Gamma_1, \Gamma, \Gamma_2@r \vdash e_r[x \leftarrow e_s] : \tau$$

Proof. Using sub-coeffecting ($s \leq \mathsf{use}$) and Lemma 10. □

As variables are annotated with the top element use, we can substitute a term e_s for any variable and use sub-coeffecting to get the original typing (because $s \leq \mathsf{use}$).

In a bottom-pointed coeffect system, substituting e for x increases the context requirements. However, if the system satisfies the condition that $\wedge = \circledast = \oplus$ then the context requirements arising from the substitution can be associated with the context Γ. As a result, substitution does not break soundness as in effect systems. The requirement $\wedge = \circledast = \oplus$ holds for our implicit parameters example (all three operators are set union) and allows the following substitution lemma:

Lemma 13 (Bottom-pointed substitution)**.** *In a bottom-pointed flat coeffect calculus with* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$ *and the* $\wedge$ *operator where* $\wedge = \circledast = \oplus$ *is idempotent and commutative:*

$$\Gamma@s \vdash e_s : \sigma \quad \wedge \quad \Gamma_1, x : \sigma, \Gamma_2@r \vdash e_r : \tau$$
$$\Rightarrow \quad \Gamma_1, \Gamma, \Gamma_2@r \circledast s \vdash e_r[x \leftarrow e_s] : \tau$$

Proof. By induction over $\vdash$, using the idempotent, commutative monoid structure to keep s with the free-variable context. □

The structural system is precise enough to keep distinct coeffects associated with each concrete variable. The flat variant described here is flexible enough to let us always re-associate new context requirements with the free-variable context.

The two substitution lemmas show that the call-by-name evaluation strategy can be used for certain coeffect calculi, including liveness and implicit parameters. Assuming $\rightsquigarrow_{\mathsf{cbn}}$ is the standard call-by-name reduction, the following theorem holds:

Theorem 14 (Call-by-name subject reduction)**.** *In a flat coeffect system that satisfies the conditions for Lemma 12 or Lemma 13, if* $\Gamma@r \vdash e : \tau$ *and* $e \rightsquigarrow_{\mathsf{cbn}} e'$ *then* $\Gamma@r \vdash e' : \tau$.

Proof. Direct consequence of Lemma 12 or Lemma 13. □

5. Semantics

Coeffects provide a unified description of context dependence. In the previous sections, we used this to define a unified coeffect calculus. We now define a unified (categorical) semantics for the coeffect calculus. The semantics can be instantiated for different notions of context dependence and thus can model a wide range of context-aware languages (both for flat and structural systems).

We relate the semantics to the equational theory and show that it is sound with respect to term equality. For a variant of the flat system, a similar result has already been shown in the second author's PhD dissertation [13]. The semantics is introduced in pieces:

- Section 5.1 describes the signature (range and domain) of the interpretation $[\![-]\!]$, gives the interpretations for types and free-variable contexts (in flat and structural systems), and defines the signature of functors D which encode contexts.
- The first part of the semantics (Section 5.2) defines *sequential composition* of context-dependent computations via *indexed comonads* (introduced briefly in our previous work [15]) and the *indexed structural comonad* structure (new here).
- More structure is needed for the semantics of application and abstraction. Section 5.3 defines indexed monoidal operations for splitting and merging contexts. Concrete structures are given throughout for the semantics of the structural bounded reuse and flat implicit parameter systems.
- Section 5.4 puts the pieces together, defining the semantics of the coeffect calculus. The semantics is illustrated by executing an example bounded-reuse program (Example 26).
- Section 5.5 shows our semantics sound with respect to the syntactic equational theory of Section 4. This uses the derivation of the categorical structures for the semantics as *lax homomorphisms* between structure in a category of coeffect annotations $\mathbb{I}$ and the base category $\mathbb{C}$.

In this section, $\mathbb{C}, \mathbb{D}, \mathbb{I}$ range over categories. The objects of a category $\mathbb{C}$ are written $obj(\mathbb{C})$. The category of functors between $\mathbb{C}$ and $\mathbb{D}$ is written $[\mathbb{C}, \mathbb{D}]$. Exponential objects, representing function types in our model, are written in two ways, either B^A or $A \Rightarrow B$.

5.1 Interpreting contexts and judgments

The semantics is parameterised by a coeffect algebra, with scalar coeffects $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leq)$, coeffect shape $(\mathcal{S}, [-], \diamond, \hat{0}, \hat{1})$, and $(\bar{\times}, \bar{\times}, \bot)$. An interpretation $[\![-]\!]$ is given to types, free-variable contexts, and type and coeffect judgments, with a base Cartesian-closed category $\mathbb{C}$ for denotations and a category $\mathbb{I}$ of scalar coeffects, where $obj(\mathbb{I}) = \mathcal{C}$. Since $\mathbb{C}$ is Cartesian-closed, we use the λ-calculus as the syntax for giving concrete definitions in $\mathbb{C}$.

The interpretation $[\![-]\!]$ is parameterised by categorical structures which model a particular notion of context. The interpretation of free-variable contexts depends on shape, for which we give concrete definitions for flat and structural shapes.

Interpreting judgments. Type and coeffect judgments are interpreted (given denotations) as morphisms in $\mathbb{C}$, of the form:

$$[\![\Gamma@R \vdash e : \tau]\!] : \mathsf{D}_R^{[\Gamma]}[\![\Gamma]\!] \to [\![\tau]\!]$$

The interpretation is a morphism from an interpretation of the context Γ to the interpretation of the result. The functor $\mathsf{D}_R^{[\Gamma]}$ over Γ encodes the semantic notion of context and is indexed by the free-variable context shape $[\Gamma]$ and coeffect annotation R.

The structure D can be thought of as a dependent product of functors D^n over possible shapes $n \in \mathcal{S}$

$$\mathsf{D} : \Pi_{n:\mathcal{S}}.\mathsf{D}^n \quad \text{where} \quad \mathsf{D}^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]$$

For a fixed context shape n the functor $\mathsf{D}^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]$ maps an n-indexed coeffect (think positions) to a functor from a context $\mathbb{C}^n$ to an object in $\mathbb{C}$. That is, given a coeffect annotation (matching the shape of the context), we get a functor $\in [\mathbb{C}^n, \mathbb{C}]$.

From a programming perspective, this functor defines a data structure that models the additional context provided to the pro-

gram. The shape of this data structure depends on the coeffect annotation $\mathbb{I}^n$. For example, in bounded reuse, the annotation defines the number of values needed for each variable and the functor will be formed by lists of length matching the required number.

Types. Types are interpreted as objects of $\mathbb{C}$, that is $[\![\tau]\!] : obj(\mathbb{C})$ where function types have the interpretation as exponents:

$$[\![\sigma \xrightarrow{r} \tau]\!] = \mathsf{D}^{\hat{\mathbb{1}}}_{\langle r \rangle}[\![\sigma]\!] \Rightarrow [\![\tau]\!]$$

The parameter of a function is wrapped by a functor $\mathsf{D}^{\hat{\mathbb{1}}}_{\langle r \rangle}$ that defines a context with singleton shape $\hat{1}$, matching the single value that it contains. This interpretation is shared by all coeffect calculi.

Free-variable contexts. As described above, free-variable contexts Γ are given an interpretation as objects in $\mathbb{C}^{[\Gamma]}$. Thus, the interpretation of contexts is shape dependent.

We define $[\![-]\!]$ on free-variable contexts for structural and flat systems. For flat systems, there is only a single shape, so the interpretation is a product type inside the Cartesian-closed category $\mathbb{C}$. For structural systems, the shape matches the number of variables and so the model is a value in the product category $\mathbb{C} \times \ldots \times \mathbb{C}$.

Flat coeffects. Recall that $\mathcal{S} = \{\star\}$ and $[\Gamma] = \star$. Since the set of positions $\star$ is a singleton, then $\mathbb{C}^\star$ is isomorphic to $\mathbb{C}$. Therefore $[\![\Gamma]\!] : obj(\mathbb{C})$, which is defined as:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$$

Denotations of typing judgments in a flat coeffect system are thus of the form (where $r \in \mathbb{I}$):

$$[\![x_1 : \tau_1, .., x_n : \tau_n @ r \vdash e : \tau]\!] : \mathsf{D}^\star_r([\![\tau_1]\!] \times ... \times [\![\tau_n]\!]) \to [\![\tau]\!]$$

Structural coeffects. Recall that $\mathcal{S} = \mathbb{N}$ and $[\Gamma] = |\Gamma|$ (number of free variables), thus $[\![\Gamma]\!] : obj(\mathbb{C}^{|\Gamma|})$. This is defined similarly to the above, but instead of using products in $\mathbb{C}$, we use the product of categories. Thus, denotations have the form:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n @ R \vdash e : \tau]\!] : \mathsf{D}^n_R([\![\tau_1]\!], \ldots, [\![\tau_n]\!]) \to [\![\tau]\!]$$

where $|R| = n$ and we use commas (instead of $\times$) to denote the product of categories. This means that $\mathsf{D}^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]$ is a functor between an n-length vector of coeffects indices and an n-ary endofunctor. Thus, the key difference between the flat and structural interpretations of free-variable contexts is that flat uses products of objects in $\mathbb{C}$ and the structural uses products of $\mathbb{C}$ in the category of categories.

Example 15 (Bounded reuse)**.** Recall bounded reuse has coeffect scalars $\mathcal{C} = \mathbb{N}$ and shapes $\mathcal{S} = \mathbb{N}$. We model contexts by replicating the value of each variable so there is a value for each use. This matches the model used by Girard *et al.* [7]. Contexts are described by $\mathsf{B} : \Pi_{n:\mathbb{N}}.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$, where for $R = \langle r_1, \ldots, r_n \rangle$:

$$\mathsf{B}^n_R(A_1, \ldots, A_n) = A_1^{r_1} \times \ldots \times A_n^{r_n}$$
$$\mathsf{B}^n_R(f_1, \ \ldots, \ f_n) = \lambda\langle a_1, \ldots, a_n \rangle.\langle (f_1 \circ a_1), \ldots, (f_n \circ a_n) \rangle$$

Thus each object in the free-variable context A_i is exponentiated by its associated coeffect r_i. For the morphism mapping part, $f_i : A_i \to B_i$ and $a_i : {A_i}^{r_i}$, thus $(f_i \circ a_i) : {B_i}^{r_i}$. The exponent ${A_i}^{r_i}$ can be read as a product of r_i copies of A_i, *e.g.*:

$$\mathsf{B}^3_{1,0,2}(A, B, C) = A^1 \times B^0 \times C^2 = (A) \times 1 \times (C \times C)$$

Example 16 (Implicit parameters)**.** Recall the implicit parameter calculus with scalar coeffects as sets of names paired with types $\mathcal{C} = \mathcal{P}(\mathsf{Name} \times \mathsf{Types})$ and flat shape with singleton $\mathcal{S} = \{\star\}$.

Its contexts are defined by $\mathsf{I}^\star : \Pi_{n:\{*\}}.(\mathbb{I}^n \to [\mathbf{Set}^n, \mathbf{Set}])$ which is equivalent to $\mathbb{I} \to [\mathbf{Set}, \mathbf{Set}]$ and defined as follows:

$$\mathsf{I}^\star_R A = A \times [\![R]\!] \qquad \mathsf{I}^\star_R f = \lambda(a, r).(f\ a, r)$$

The interpretation $[\![R]\!]$ maps a set of variable-type pairs to an object representing a set of variable-value pairs in $\mathbf{Set}$.

5.2 Sequential composition

Following the usual categorical semantics approach, we require a notion of sequential composition for our denotations. We show first a special case for $\mathsf{D}^{\hat{\mathbb{1}}}$ (where $\mathbb{I}^{\hat{\mathbb{1}}} = \mathbb{I}$ and $\mathbb{C}^{\hat{\mathbb{1}}} = \mathbb{C}$) in both flat and structural systems[2] and thus $\mathsf{D}^{\hat{\mathbb{1}}} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$. Composition of morphisms $f : \mathsf{D}^{\hat{\mathbb{1}}}_S A \to B$ and $g : \mathsf{D}^{\hat{\mathbb{1}}}_R B \to C$ is defined by an *indexed comonad* (which we introduced previously [13, 15]).

Definition 17. An *indexed comonad* comprises a strict monoidal category $(\mathbb{I}, \bullet, I)$ and a functor $\mathsf{F} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ with two natural transformations (where we write $(\mathsf{F}\ R)\ A$ as $\mathsf{F}_R A$):

$$(\delta_{X,Y})_A : \mathsf{F}_{(X \bullet Y)} A \to \mathsf{F}_X(\mathsf{F}_Y A) \qquad (\varepsilon_I)_A : \mathsf{F}_I A \to A$$

where δ is called *comultiplication* and ε is called *counit*. We require indexed analogues of the usual comonad axioms (*cf.* [19]):

$$\begin{array}{ccc} \mathsf{F}_R & \xrightarrow{\delta_{R,I}} & \mathsf{F}_R\mathsf{F}_I \\ {\scriptstyle \delta_{I,R}}\downarrow & \overset{[C2]}{\searrow} & \downarrow{\scriptstyle \mathsf{F}_R\varepsilon_I} \\ \mathsf{F}_I\mathsf{F}_R & \xrightarrow[\varepsilon_I \mathsf{F}_R]{[C1]} & \mathsf{F}_R \end{array} \qquad \begin{array}{ccc} \mathsf{F}_{R \bullet S \bullet T} & \xrightarrow{\delta_{R \bullet S, T}} & \mathsf{F}_{R \bullet S}\mathsf{F}_T \\ {\scriptstyle \delta_{R, S \bullet T}}\downarrow & [C3] & \downarrow{\scriptstyle \delta_{R,S}\mathsf{F}_T} \\ \mathsf{F}_R\mathsf{F}_{S \bullet T} & \xrightarrow[\mathsf{F}_R \delta_{S,T}]{} & \mathsf{F}_R\mathsf{F}_S\mathsf{F}_T \end{array}$$

An indexed comonad $\mathsf{F} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ induces a notion of composition for all $f : \mathsf{F}_S A \to B$ and $g : \mathsf{F}_R B \to C$:

$$g \hat{\circ} f = g \circ \mathsf{F}_R f \circ \delta_{R,S} \ : \ \mathsf{F}_{R \bullet S} A \to C$$

with the identity $\hat{id}_A = (\varepsilon_I)_A : \mathsf{F}_I A \to A$ for all A. Thus indexed comonads induce a category which has the same objects as $\mathbb{C}$ and morphisms $\mathbb{C}_\mathsf{F}(A, B) = \bigcup_{R \in \mathbb{I}} \mathbb{C}(\mathsf{F}_R A, B)$. Note that an indexed comonad is not a family of (ordinary) comonads, because identity need only be defined for the functor F_I.

Therefore, if $\mathsf{D}^{\hat{\mathbb{1}}}$ is an indexed comonad, there is a notion of composition for denotations with a single coeffect index.

Example 18 (Bounded reuse)**.** $\mathsf{B}^{\hat{\mathbb{1}}}_R$ (Example 15) has an indexed comonad structure, where the monoid $(\mathbb{N}, *, 1)$ from the coeffect scalar for bounded reuse induces a monoidal category structure on $\mathbb{I}$ (with $1 : \mathbb{I}$ and the bifunctor $* : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$), with operations:

$$\varepsilon^{\hat{\mathbb{1}}}_1 = \lambda\langle a_1 \rangle.a_1$$
$$\delta^{\hat{\mathbb{1}}}_{R,S} = \lambda\langle a_1 ..., a_{RS} \rangle. \langle \langle a_1 ..., a_S \rangle, \langle a_{S+1}, ..., a_{S+S} \rangle, ..., \langle a_{(R-1)S+1}, ..., a_{RS} \rangle \rangle$$

Indexed comonads essentially model single-variable contexts. Counit here requires a single copy of the value from the context. Comultiplication splits R times S copies of a value into R copies of a context where each context contains just S copies of the value.

Remark 19. A semantics for dataflow coeffects is similar to bounded reuse with $\mathsf{D}^n_R(A_1, \ldots, A_n) = (A_1 \times {A_1}^{R_1}) \times ... \times (A_n \times {A_n}^{R_n})$, *i.e.*, each free-variable has an extra value representing the "current" value. A dataflow indexed comonad is similar to the above but with additive rather than multiplicative behaviour.

Example 20 (Implicit parameters)**.** For the coeffect scalar monoid $(\mathcal{P}(\mathsf{Name} \times \mathsf{Types}), \cup, \emptyset)$ of implicit parameters, $\mathsf{I}^\star$ (Example 16) has an indexed comonad structure, with operations:

$$\varepsilon_\emptyset = \lambda(a, \emptyset).a \qquad \delta_{R,S} = \lambda(a, \gamma).((a, \gamma|_S), \gamma|_R)$$

where $\gamma|_R = \{(x, v) \mid (x, v) \in \gamma, (x, t) \in R\}$ filters incoming implicit parameters to those variable-value pairs where the variable is in the coeffect R.

[2] Since $\hat{1} = 1$ in structural and $\hat{1} \cong 1$ in flat, *i.e.*, $\star$ is isomorphic to 1.

These two examples (which are new here) provide composition for context-dependent computations indexed by coeffects in a flat calculus. For structural coeffects, we need to compose morphisms which have more than a single coeffect annotation. For this, we introduce the new notion of *structural indexed comonads*.

Definition 21. A *structural indexed comonad* comprises a functor $\mathsf{D} : \Pi_{n:\mathcal{S}}.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$ where $(\mathbb{I}, \bullet, I)$ is a strict monoidal category, $\hat{1} \in \mathcal{S}$ which is terminal (*e.g.*, a singleton set), an indexed comonad over $\mathsf{D}^{\hat{1}} : \mathbb{I}^{\hat{1}} \to [\mathbb{C}^{\hat{1}}, \mathbb{C}]$ and a *structural comultiplication* natural transformation:

$$(\delta^n_{r,S})_{A^n} : \mathsf{D}^n_{r \bar{\bullet} S} A^n \to \mathsf{D}^{\hat{1}}_r \mathsf{D}^n_S A^n$$

where $A^n \in \mathbb{C}^n$, $r \in \mathbb{I}$, $S \in \mathbb{I}^n$ and $\bar{\bullet} : \mathbb{I} \times \mathbb{I}^n \to \mathbb{I}^n$ is the *monoid left action* that $\bullet$-lifts scalar coeffects to shaped coeffects (*i.e.*, the scalar-vector version of $\bullet$). Analogous laws to monoid left actions for unitality and associativity hold for structural comultiplication:

$$\begin{array}{ccc} \mathsf{D}^n_{I \bar{\bullet} R} & \xrightarrow{\delta^n_{I,R}} & \mathsf{D}^{\hat{1}}_I \mathsf{D}^n_R \\ & \searrow^{[\mathrm{SC1}]} & \downarrow{\scriptstyle \varepsilon_I \mathsf{D}^n_R} \\ & & \mathsf{D}^n_R \end{array} \qquad \begin{array}{ccc} \mathsf{D}^n_{r \bar{\bullet} (s \bar{\bullet} T)} & \xrightarrow{\delta^n_{r \bullet s, T}} & \mathsf{D}^{\hat{1}}_{r \bullet s} \mathsf{D}^n_T \\ \downarrow{\scriptstyle \delta^n_{r,s \bar{\bullet} T}} & [\mathrm{SC2}] & \downarrow{\scriptstyle \delta^{\hat{1}}_{r,s} \mathsf{D}^n_T} \\ \mathsf{D}^{\hat{1}}_r \mathsf{D}^n_{s \bar{\bullet} T} & \xrightarrow[\mathsf{D}^{\hat{1}}_r \delta^n_{s,T}]{} & \mathsf{D}^{\hat{1}}_r \mathsf{D}^{\hat{1}}_s \mathsf{D}^n_T \end{array} \qquad (1)$$

using axioms $I \bar{\bullet} R = R$ and $(r \bullet s) \bar{\bullet} T = r \bar{\bullet} (s \bar{\bullet} T)$ on coeffects respectively which are the monoid left action axioms for the scalar-vector application of $\bullet$. Note the use of indexed comonad comultiplication $\delta^{\hat{1}}$ for associativity [SC2].

Structural indexed comonads provide composition for morphisms $f : \mathsf{D}^n_S A^n \to B$ and singleton-shaped morphisms $g : \mathsf{D}^{\hat{1}}_r B \to C$:

$$g \hat{\circ} f = g \circ \mathsf{D}^{\hat{1}}_r f \circ \delta^n_{r,S} : \mathsf{D}^n_{r \bar{\bullet} S} A^n \to C$$

Note that this composition is asymmetric: the left morphism and right morphisms have different shapes. To compose morphisms which both have non-trivial context shapes requires additional structure for manipulating contexts (shown in the next section).

Example 22 (Bounded reuse)**.** $\mathsf{B} : \Pi_{n:\mathbb{N}}.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$ has a structural indexed comonad structure with the indexed comonad $\mathsf{B}^{\hat{1}}$ (Example 18) and the following structural comultiplication:

$$\begin{array}{l} \delta^n_{r,S} = \lambda(\langle a^1_1, \ldots, a^1_{r * S_1} \rangle, \ldots, \langle a^n_1, \ldots a^n_{r * S_n} \rangle). \\ (\ (\ \langle a^1_1, \ldots, a^1_{S_1} \rangle, \qquad\qquad \ldots, \langle a^n_1, \ldots, a^n_{S_n} \rangle), \\ \quad (\ \langle a^1_{(S_1+1)}, \ldots, a^1_{(S_1+1)+S_1} \rangle, \ldots, \langle a^n_{(S_n+1)}, \ldots, a^n_{(S_n+1)+S_n} \rangle), \\ \quad \ldots \\ \quad (\ \langle a^1_{(r-1)*s_1+1}, .., a^1_{r*S_1} \rangle, \quad \ldots, \langle a^n_{(r-1)*S_n+1}, \ldots, a^n_{r*S_n} \rangle)) \end{array}$$

The input is an n-variable context containing r times S_i copies of a^i for each variable. The output has r copies of a single n-variable context containing S_i copies of a^i for each variable. Thus, $\delta^n_{r,S}$ partitions the incoming context into r-sized contexts.

Note that in the case of the flat system, a structural indexed comonad collapses to a standard indexed comonad on $\mathsf{D}^{\hat{1}}$.

5.3 Splitting and merging contexts

Indexed comonads and structural indexed comonads give a semantics for sequential composition of contextual computations. However, this does not provide enough structure for a semantics of the full coeffect calculus. Core to the semantics of abstraction and application is the merging and splitting of contexts. Recall the free-variable contexts and coeffects in the *(abs)* and *(app)* rules:

$$(\mathrm{app})\ \frac{\Gamma_1 @ R \vdash e_1 \ldots \ \Gamma_2 @ S \vdash e_2 \ldots}{\Gamma_1, \Gamma_2 @ R \between (t \circledast S) \vdash e_1\ e_2 \ldots} \qquad (\mathrm{abs})\ \frac{\Gamma, x : \sigma @ R \between \langle s \rangle \vdash e \ldots}{\Gamma @ R \vdash \lambda x.e : \sigma \xrightarrow{s} \ldots}$$

Reading *(app)* bottom-up, the context of the application is split into two contexts for each subterm e_1 and e_2. Reading *(abs)* bottom-up, the context of the abstraction is merged with the singleton context of the parameter. Capturing these notions in the denotational semantics requires some additional structure.

A (non-indexed) comonadic semantics for the λ-calculus requires a *monoidal comonad* with operation $\mathsf{m}_{A,B} : \mathsf{F}A \times \mathsf{F}B \to \mathsf{F}(A \times B)$ [19]. Previously, we defined a similar operation for the semantics of a flat coeffect system, with an indexed monoidal operation $\mathsf{m}^{R,S}_{A,B}$ for merging contexts. Dually, contexts were split with $\mathsf{n}^{R,S}_{A,B}$ [15]. We used two operations for combining and splitting the coeffect annotations, respectively. Here we generalize these to shape-indexed versions using $\between$ and $\between$.

Definition 23. A functor $\mathsf{D} : \Pi_{n:\mathcal{S}}.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$ is an *indexed lax (semi)monoidal functor* and/or *colax (semi)monoidal functor* if it has the following natural transformations respectively:

$$\mathsf{m}^{n,m}_{R,S} : \mathsf{D}^n_R A \times \mathsf{D}^m_S B \to \mathsf{D}^{n \diamond m}_{R \between S}(A \times B)$$
$$\mathsf{n}^{n,m}_{R,S} : \mathsf{D}^{n \diamond m}_{R \between S}(A \times B) \to \mathsf{D}^n_R A \times \mathsf{D}^m_S B$$

satisfying associativity coherence conditions. In both, shape descriptions are combined by $\diamond$. The first operation models context merging and combines coeffects using $\between$. The second models context splitting, with $\between$ for the pre-split coeffect.

Example 24 (Bounded reuse)**.** For bounded reuse, B is an indexed lax and colax semimonoidal functor with the following operations:

$$\mathsf{m}^{n,m}_{R,S} = \lambda(\langle a_1, ..., a_n \rangle \times \langle b_1, ..., b_m \rangle).(\langle a_1, ..., a_n \rangle, \langle b_1, ..., b_m \rangle)$$
$$\mathsf{n}^{n,m}_{R,S} = \lambda(\langle a_1, ..., a_n \rangle, \langle b_1, ..., b_m \rangle).(\langle a_1, ..., a_n \rangle \times \langle b_1, ..., b_m \rangle)$$

Here $\mathsf{m}^{n,m}_{R,S}$ takes a pair of contexts and merges them simply by replacing the product in $\mathbb{C}$ which pairs the two arguments (written using $\times$) with products inside of B (written using tuple notation (x, y)). The operation $\mathsf{n}^{n,m}_{R,S}$ is the inverse.

Example 25 (Implicit parameters)**.** For implicit parameters, $\mathsf{I}^\star$ is an indexed lax and colax semimonoidal functor with operations:

$$\mathsf{m}^{\star,\star}_{R,S} = \lambda((a, \gamma_R), (b, \gamma_S)).((a, b), \gamma_R \cup \gamma_S)$$
$$\mathsf{n}^{\star,\star}_{R,S} = \lambda((a, b), \gamma).((a, \gamma|_R), (b, \gamma|_S))$$

As in Example 20, $\gamma|_R$ and $\gamma|_S$ restrict the set of implicit parameters γ to variable-value pairs for variables in R and S.

5.4 Putting it together

The semantics of the general coeffect calculus $[\![-]\!]$ is defined in Figure 5, using the structures described in the previous sections.

Core rules. The denotation in (*var*) maps a context of the singleton shape $\hat{1}$ containing just a single variable τ (with coeffect I) to a τ value using the counit operation.

The premise of (*abs*) takes a context of shape $n \diamond \hat{1}$ with coeffects $R \between \langle s \rangle$ and a free-variables context consisting of Γ and an additional variable x. The denotation $g : \mathsf{D}^{n \diamond \hat{1}}_{R \between \langle s \rangle}[\![\Gamma, x : \sigma]\!] \to [\![\tau]\!]$ is pre-composed with m, such that its context is obtained by merging the declaration-site context (Γ) and call-site context (σ):

$$g \circ \mathsf{m}^{n,\hat{1}}_{R,\langle s \rangle} : (\mathsf{D}^n_R [\![\Gamma]\!] \times \mathsf{D}^{\hat{1}}_{\langle s \rangle} [\![\sigma]\!]) \to [\![\tau]\!]$$

This is uncurried to give a denotation from a context to an exponential object representing the abstraction, where the singleton-shaped context becomes the source of the exponential.

The application rule (*app*) has two sub-expressions for the function and argument, with denotations requiring two distinct contexts:

$$g_1 : \mathsf{D}^n_R [\![\Gamma_1]\!] \to (\mathsf{D}^{\hat{1}}_{\langle t \rangle} [\![\sigma]\!] \Rightarrow [\![\tau]\!]) \qquad g_2 : \mathsf{D}^m_S [\![\Gamma_2]\!] \to [\![\sigma]\!]$$

$$\text{(var)}\ \frac{}{[\![x:\tau@\langle \mathsf{use}\rangle \vdash x:\tau]\!] = \varepsilon_I : \mathsf{D}^{\hat{1}}_I[\![x:\tau]\!] \to [\![\tau]\!]} \qquad \text{(abs)}\ \frac{[\![\Gamma, x:\sigma@R \rtimes \langle s\rangle \vdash e:\tau]\!] = g : \mathsf{D}^{n\diamond\hat{1}}_{R\rtimes\langle s\rangle}[\![\Gamma, x:\sigma]\!] \to [\![\tau]\!]}{[\![\Gamma@R \vdash \lambda x.e : \sigma \xrightarrow{s} \tau]\!] = \Lambda(g \circ \mathsf{m}^{n,\hat{1}}_{R,\langle s\rangle}) : \mathsf{D}^n_R[\![\Gamma]\!] \to (\mathsf{D}^{\hat{1}}_{\langle s\rangle}[\![\sigma]\!] \Rightarrow [\![\tau]\!])}$$

$$\text{(app)}\ \frac{[\![\Gamma_1@R \vdash e_1 : \sigma \xrightarrow{t} \tau]\!] = g_1 : \mathsf{D}^n_R[\![\Gamma_1]\!] \to (\mathsf{D}^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!] \Rightarrow [\![\tau]\!]) \qquad [\![\Gamma_2@S \vdash e_2 : \sigma]\!] = g_2 : \mathsf{D}^m_S[\![\Gamma_2]\!] \to [\![\sigma]\!]}{[\![\Gamma_1,\Gamma_2@R \rtimes (t \circledast S) \vdash e_1\, e_2 : \tau]\!] = \Lambda^{-1}g_1 \circ (\mathsf{id} \times (\mathsf{D}^{\hat{1}}_{\langle t\rangle}g_2 \circ \delta^m_{t,S})) \circ \mathsf{n}^{n,m}_{R,t\circledast S} : \mathsf{D}^{n\diamond m}_{R\rtimes(t\circledast S)}[\![\Gamma_1,\Gamma_2]\!] \to [\![\tau]\!]}$$

$$\text{(ctx)}\ \frac{[\![\Gamma@R \vdash e:\tau]\!] = f : \mathsf{D}^n_R[\![\Gamma]\!] \to [\![\tau]\!] \qquad [\![\Gamma'@R' \rightsquigarrow \Gamma@R, \emptyset]\!] = c : \mathsf{D}^m_{R'}[\![\Gamma']\!] \to \mathsf{D}^n_R[\![\Gamma]\!]}{[\![\Gamma'@R' \vdash e:\tau]\!] = f \circ c : \mathsf{D}^m_{R'}[\![\Gamma']\!] \to [\![\tau]\!]}$$

$$\text{(weak)}\quad [\![\Gamma, x:\tau@R \rtimes \langle \mathsf{ign}\rangle \rightsquigarrow \Gamma@R, \emptyset]\!] = \pi_1 \circ \mathsf{n}^{n,\hat{1}}_{R,\langle \mathsf{ign}\rangle} : \mathsf{D}^{n\diamond\hat{1}}_{R\rtimes\langle \mathsf{ign}\rangle}([\![\Gamma]\!] \times [\![\sigma]\!]) \to \mathsf{D}^n_R[\![\Gamma]\!]$$

$$\text{(contr)}\quad [\![\Gamma_1, x:\tau, \Gamma_2@R \rtimes \langle s \oplus t\rangle \rtimes Q \rightsquigarrow \Gamma_1, y:\tau, z:\tau, \Gamma_2@R \rtimes \langle s\rangle \times \langle t\rangle \times Q, [y,z \mapsto x]]\!] = \mathsf{m}^{n,\hat{1},m}_{R,\langle s\oplus t\rangle,Q} \circ (\mathsf{id} \times \Delta_{s,t} \times \mathsf{id}) \circ \mathsf{n}^{n,\hat{1},\hat{1},m}_{R,\langle s\rangle,\langle t\rangle,Q}$$

$$\text{(exch)}\quad [\![\Gamma_1, y:\sigma, x:\tau, \Gamma_2@R \rtimes \langle t\rangle \rtimes \langle s\rangle \rtimes Q \rightsquigarrow \Gamma_1, x:\tau, y:\sigma, \Gamma_2@R \rtimes \langle s\rangle \times \langle t\rangle \times Q, \emptyset]\!] = \mathsf{m}^{n,\hat{1},\hat{1},m}_{R,\langle s\rangle,\langle t\rangle,Q} \circ (\mathsf{id} \times \mathsf{swap} \times \mathsf{id}) \circ \mathsf{n}^{n,\hat{1},\hat{1},m}_{R,\langle t\rangle,\langle s\rangle,Q}$$

where $\mathsf{swap} : A \times B \to B \times A$ **and** Λ, Λ^{-1} denote currying and uncurrying respectively

Figure 5: Denotational semantics for the coeffect calculus

The target of g_1 is an exponential object with singleton shape for the parameter of type σ. To evaluate g_1 and g_2, the semantics of (*app*) splits the incoming context over Γ_1, Γ_2 using n:

$$\mathsf{D}^{n\diamond m}_{R\rtimes(t\circledast S)}([\![\Gamma_1]\!] \times [\![\Gamma_2]\!]) \xrightarrow{\mathsf{n}^{n,m}_{R,t\circledast S}} \mathsf{D}^n_R[\![\Gamma_1]\!] \times \mathsf{D}^m_{t\circledast S}[\![\Gamma_2]\!]$$

Since e_2 computes the argument for function e_1, the denotation g_2 is sequentially composed with the parameter part of g_1. Thus, the structural indexed comonad (where $\bullet = \circledast$) is used with g_2 to compute the correct context for the parameter of the function denotation g_1:

$$\mathsf{D}^m_{t\circledast S}[\![\Gamma_2]\!] \xrightarrow{\delta^m_{t,S}} \mathsf{D}^{\hat{1}}_{\langle t\rangle}\mathsf{D}^m_S[\![\Gamma_2]\!] \xrightarrow{\mathsf{D}^{\hat{1}}_{\langle t\rangle}g_2} \mathsf{D}^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!]$$

This is composed with the previous equation by lifting to the right-component of the product:

$$\mathsf{D}^n_R[\![\Gamma_1]\!] \times \mathsf{D}^m_{t\circledast S}[\![\Gamma_2]\!] \xrightarrow{id\times(\mathsf{D}^{\hat{1}}_{\langle t\rangle}g_2 \circ \delta^m_{t,S})} \mathsf{D}^n_R[\![\Gamma_1]\!] \times \mathsf{D}^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!]$$

This equation computes the calling context and parameter context for the function e_1, which is then composed with the uncurried g_1 denotation as shown in the (*app*) rule in Figure 5.

Structural rules. In Figure 5, (*ctx*) composes the denotation of an expression with a transformation c providing the semantic structural rules. The semantics of structural rules are defined by using $\mathsf{n}^{n,m}_{R,S}$ to split contexts, transforming the components, and merging the transformed contexts using $\mathsf{m}^{n,m}_{R,S}$. The (*contr*) rule uses an additional operation which duplicates a variable inside a context:

$$\Delta_{r,s} : \mathsf{D}^{\hat{1}}_{\langle r\oplus s\rangle}A \to \mathsf{D}^{\hat{1}\diamond\hat{1}}_{\langle r\rangle\rtimes\langle s\rangle}(A \times A)$$

Example 26. We demonstrate the semantics with a concrete example for the bounded reuse calculus. Consider the following term:

$$f : \mathbb{Z} \xrightarrow{2} \mathbb{Z}, x : \mathbb{Z}@\langle 2,4\rangle \vdash (\lambda z.z + z)\ (f\ x)$$

Let the denotation of the function body, prior to contraction, be $g = [\![x:\mathbb{Z}, y:\mathbb{Z}@\langle 1,1\rangle \vdash (+x)\, y : \mathbb{Z}]\!]$[3]. The example term's denotation is then constructed as follows:

$$[\![@\langle\rangle \vdash \lambda z.(+z)z : \mathbb{Z} \xrightarrow{2} \mathbb{Z}]\!] = \Lambda(g \circ \Delta_{1,1} \circ \mathsf{m}^{0,1}_{\langle\rangle,\langle 2\rangle}) \tag{2}$$

$$\begin{aligned}[\![f : \mathbb{Z} \xrightarrow{2} \mathbb{Z}, x : \mathbb{Z}@\langle 1,2\rangle \vdash fx : \mathbb{Z}]\!] &= \Lambda^{-1}\varepsilon_1 \circ (id \times (\mathsf{D}\varepsilon_1 \circ \delta^1_{2,\langle 1\rangle})) \circ \mathsf{n}^{1,1}_{\langle 1\rangle,\langle 2\rangle} \\ &= \Lambda^{-1}\varepsilon_1 \circ \mathsf{n}^{1,1}_{\langle 1\rangle,\langle 2\rangle} \end{aligned} \tag{3}$$

$$\begin{aligned}[\![f : \mathbb{Z} \xrightarrow{2} \mathbb{Z}, x : \mathbb{Z}@\langle 2,4\rangle \vdash (\lambda z.(+z)z)\ (fx) : \mathbb{Z}]\!] &= \Lambda^{-1}(2) \circ (id \times \mathsf{D}(3) \circ \delta^2_{2,\langle 2,1\rangle}) \circ \mathsf{n}^{0,2}_{\langle\rangle,\langle 4,2\rangle} \\ &= g \circ \Delta_{1,1} \circ \mathsf{m}^{0,1}_{\langle\rangle,\langle 2\rangle} \circ (id \times \mathsf{D}(3) \circ \delta^2_{2,\langle 2,1\rangle}) \circ \mathsf{n}^{0,2}_{\langle\rangle,\langle 4,2\rangle} \end{aligned} \tag{4}$$

where (3) and (4) are simplified. We "run" this semantics on an input, evaluating each step of the denotation as a function. We write context objects, *e.g.*, $\mathsf{D}^2_{\langle R,S\rangle}(A,B)$ as $\langle (a_1,...,a_R),(b_1,...,b_S)\rangle$ and products of contexts in $\mathbb{C}$, *e.g.*, $\mathsf{D}^n_R A \times \mathsf{D}^m_S B$, as $(a \times b)$.

$$\begin{aligned}
&\langle (f_1,f_2),(x_1,x_2,x_3,x_4)\rangle && : \mathsf{D}^2_{\langle 2,4\rangle}((\mathsf{D}^1_{\langle 2\rangle}\mathbb{Z} \Rightarrow \mathbb{Z}),\mathbb{Z}) \\
\xrightarrow{n^{0,2}_{\langle\rangle,\langle 2,4\rangle}} &\langle\rangle \times \langle (f_1,f_2),(x_1,x_2,x_3,x_4)\rangle && : \mathsf{D}^0_{\langle\rangle}1 \times \mathsf{D}^2_{\langle 2,4\rangle}(\textit{as above}) \\
\xrightarrow{id \times \delta^2_{2,\langle 1,2\rangle}} &\langle\rangle \times \langle \langle f_1,(x_1,x_2)\rangle, \langle f_2,(x_3,x_4)\rangle\rangle \\
& && : \mathsf{D}^0_{\langle\rangle}1 \times \mathsf{D}^1_{\langle 2\rangle}\mathsf{D}^2_{\langle 2,1\rangle}((\mathsf{D}^1_{\langle 2\rangle}\mathbb{Z} \Rightarrow \mathbb{Z}),\mathbb{Z}) \\
\xrightarrow{id \times \mathsf{Dn}^{1,1}_{\langle 1\rangle,\langle 2\rangle}} &\langle\rangle \times \langle \langle f_1\rangle \times \langle x_1,x_2\rangle, \langle f_2\rangle \times \langle x_3,x_4\rangle\rangle \\
& && : \mathsf{D}^0_{\langle\rangle}1 \times \mathsf{D}^1_{\langle 2\rangle}(\mathsf{D}^1_{\langle 1\rangle}(\mathsf{D}^1_{\langle 2\rangle}\mathbb{Z} \Rightarrow \mathbb{Z}) \times \mathsf{D}^1_{\langle 2\rangle}\mathbb{Z}) \\
\xrightarrow{id \times \mathsf{D}(\Lambda^{-1}\varepsilon_1)} &\langle\rangle \times \langle f_1\langle x_1,x_2\rangle, f_2\langle x_3,x_4\rangle\rangle && : \mathsf{D}^0_{\langle\rangle}1 \times \mathsf{D}^1_{\langle 2\rangle}\mathbb{Z} \\
\xrightarrow{\mathsf{m}^{0,1}_{\langle\rangle,\langle 1\rangle}} &\langle f_1\langle x_1,x_2\rangle, f_2\langle x_3,x_4\rangle\rangle && : \mathsf{D}^1_{\langle 2\rangle}\mathbb{Z} \\
\xrightarrow{\Delta_{1,1}} &\langle (f_1\langle x_1,x_2\rangle, f_2\langle x_3,x_4\rangle)\rangle && : \mathsf{D}^1_{\langle 1,1\rangle}(\mathbb{Z} \times \mathbb{Z}) \\
\xrightarrow{g} &[\![+]\!]\ \langle f_1\langle x_1,x_2\rangle\rangle\ \langle f_2\langle x_3,x_4\rangle\rangle && : \mathbb{Z}
\end{aligned}$$

5.5 Soundness, with respect to the equational theory

Our denotational semantics for the coeffect calculus is sound with respect to the equational theory of Section 4. That is:

Theorem 27 (Soundness)**.**

$$\Gamma@R \vdash e \equiv e' : \tau \Rightarrow [\![\Gamma@R \vdash e:\tau]\!] \equiv [\![\Gamma@R \vdash e' : \tau]\!]$$

Proof of this follows from an interesting result which we first unpack: determining whether $[\![\Gamma@R \vdash e_1 : \tau]\!] \equiv [\![\Gamma@S \vdash e_2 : \tau]\!]$ follows from a proof (on coeffect annotations) that $R = S$.

[3] The full semantics has $[\![+]\!] : \mathsf{D}^0_{\langle\rangle}1 \to (\mathsf{D}^1_1\mathbb{Z} \Rightarrow (\mathsf{D}^1_1\mathbb{Z} \Rightarrow \mathbb{Z}))$ as primitive and uses double application $(+\,e_1)\,e_2$.

Lemma 28. *Every coeffect algebra axiom corresponds to an axiom of one of the categorical structures introduced here (indexed (structural) comonad or indexed (co)lax monoidal functor).*

For example, the monoid axiom $X \circledast \mathsf{use} = X$ for scalar coeffects corresponds to indexed comonad axiom $\hat{\mathsf{D}}^{\mathbb{I}}_X \varepsilon_{\mathsf{use}} \circ \delta^{\mathbb{I}}_{X,\mathsf{use}} = id_{\hat{\mathsf{D}}^{\mathbb{I}}_X}$ (which requires the monoid axiom to hold). This lemma follows from our derivation of the indexed categorical structures here. They are not derived *ad hoc* but systematically as (lax) *homomorphisms* (structure-preserving maps) between the structure of coeffect annotations in $\mathbb{I}$ and the structure of denotations in $\mathbb{C}$.

Proposition 29. *An indexed comonad on* D *witnesses that* D *is a* colax monoid homomorphism *between the (strict) monoidal categories* $(\mathbb{I}, \bullet, I)$ *and* $([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$ *(endofunctor composition).*

Unpacking this, a *monoid homomorphism* maps between the underlying sets of two monoids, preserving the monoid structure of one into the other, *i.e.*, given monoids $(X, \bullet, I)$ and $(Y, \otimes, E)$ then a monoid homomorphism is a mapping $F : X \to Y$ such that:

$$FX \otimes FY \equiv F(X \bullet Y) \qquad E \equiv FI \tag{5}$$

The axioms of each monoid are preserved trivially by these equalities, *e.g.*, $FX \equiv F(X \bullet I) \equiv FX \otimes FI \equiv FX \otimes E \equiv FX$. A homomorphism is *lax* if the above equalities (5) are instead morphisms (which we say *witness* the homomorphism) and *colax* if these morphisms go in the opposite direction. Thus, a colax monoid homomorphism is witnessed by:

$$\delta : FX \otimes FY \leftarrow F(X \bullet Y) \qquad \varepsilon : E \leftarrow FI$$

Note our choice of morphism names. F no longer preserves the monoid axioms *up to equality* but has axioms on δ and ε, *e.g.*, $F(X \bullet I) \xrightarrow{\delta} FX \otimes FI \xrightarrow{id \otimes \varepsilon} FX \otimes E$ equals $FX \xrightarrow{id} FX$.

Our indexed comonad definition is equivalent to D being a colax homomorphism between strict monoidal category $(X, \bullet, I)$ and the monoidal category of $\mathbb{C}$-endofunctors $(Y, \otimes, E) = ([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$, with endofunctor composition $\circ$ and the trivial endofunctor $1_{\mathbb{C}}$; monoids are now at the level of categories. The indexed comonads axioms are the axioms of the colax homomorphism. Equivalently, D is a *colax monoidal functor*.

A similar approach is taken to deriving the remaining structures below, though we give less detail for brevity.

Proposition 30. *A structural indexed comonad provides* $\delta^n_{R,S} : \hat{\mathsf{D}}^{\mathbb{I}}_r \mathsf{D}^n_S A^n \leftarrow \mathsf{D}^n_{r \circledast S} A^n$ *which is a family of morphisms (indexed by shapes* n*) witnessing that* D *is a colax homomorphism between the following* monoid left-actions*:* $(\mathbb{I}^n, \circledast)$ *for* $(\mathbb{I}, \circledast, \mathsf{ign})$ *and* $(\mathbb{C}^n, \mathbb{C}], \hat{\circ})$ *for* $([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$*, defined:*

$$\begin{array}{rcl} (r : \mathbb{I}) \circledast (\langle s_1, ..., s_n \rangle : \mathbb{I}^n) & = & \langle r \circledast s_1, ..., r \circledast s_n \rangle : \mathbb{I}^n \\ (\hat{\mathsf{D}}^{\mathbb{I}}_r : [\mathbb{C},\mathbb{C}]) \;\hat{\circ}\; (\mathsf{D}^n_S : [\mathbb{C}^n,\mathbb{C}]) & = & \hat{\mathsf{D}}^{\mathbb{I}}_r \circ \mathsf{D}^n_S : [\mathbb{C}^n,\mathbb{C}] \end{array}$$

The axioms are the lax versions of the monoid left-action laws.

The lax and colax indexed monoidal operations $\mathsf{m}^{n,m}_{R,S}$ and $\mathsf{n}^{n,m}_{R,S}$ follow a similar derivation but as lax and colax monoid homomorphisms between composite monoids on coeffect annotations and shapes and $\times$ in $\mathbb{C}$. The details are elided here.

Returning to soundness, our semantics is therefore defined in terms of structures whose axioms correspond to axioms of the syntactic equational theory. Consequently, semantic proofs correspond to syntactic proofs, modulo naturality laws and product/exponent laws in $\mathbb{C}$. This result holds in the general coeffect calculus and semantics since every semantic structure has a unique corresponding structure on coeffect annotations (*i.e.*, $(\mathcal{C}, \circledast, \mathsf{use})$ for sequential composition of unary denotations, $(\mathcal{C}, \curlywedge)$ for splitting contexts, $(\mathcal{C}, \curlyvee)$ for joining contexts).

Example 31. Section 4.1 showed η-equality for structural systems, which uses the properties (1) $\curlywedge = \curlyvee = \times$ for structural systems and (2) $s \circledast \langle \mathsf{use} \rangle = \langle s \circledast \mathsf{use} \rangle = \langle s \rangle$. The semantics here is sound with respect to η-equality; the proof uses the corresponding axioms (1) $\mathsf{n}^{n,m}_{R,S} \circ \mathsf{m}^{n,m}_{R,S} = id$ and (2) $\varepsilon_{\mathsf{use}} \mathsf{D}^m_s \circ \delta^m_{\mathsf{use},s} = id$ (structural indexed comonad unit law [SC1], Definition 21).

The full semantic proofs of $\beta\eta$-equality then correspond to syntactic proofs on coeffect annotations. For brevity, we omit the full proofs here.

6. Related work

We expand on the overview of related work in Section 2.5.

Bounded reuse. The (*storage*) rule for bounded linear logic explains the contextual requirements induced by proposition reuse [7]:

$$(\text{storage}) \frac{!_{\overline{Y}} \Gamma \vdash A}{!_{X\overline{Y}} \Gamma \vdash !_X A}$$

where $X\overline{Y} = \langle XY_1, .., XY_n \rangle$ is the scalar multiple of a vector. This rule is akin to the δ^n operation of structural indexed comonads, indeed, we can model it exactly using $\delta^n_{X,\bar{Y}}$ and the lifting D^1_X.

In BLL, the modality $!_X$ is a constructor and may appear both on the left- and right-hand sides of $\vdash$. In this paper, reuse bounds annotate typing rules, thus there is no constructor corresponding to bounded reuse in the language; reuse bounds are meta-level. Our choice to work at the meta-level means that the coeffect calculus provides a unified analysis and semantics to different notions of context; its term language is that of the standard λ-calculus.

Semantics. Previously we briefly introduced indexed comonads [15] without derivation. Here we derived indexed comonads as colax homomorphisms. This is dual to the *parametric effect monad* structure defined as a lax homomorphism [8]. Our semantics requires additional structure not needed for effects due to the asymmetry inherent in the λ-calculus.

The necessity modality $\Box$ in S4 logic corresponds to a comonad with lax monoidal functor structure $\mathsf{m} : \Box A \times \Box B \to \Box(A \times B)$. Bierman and de Paiva [2] defined a term language corresponding to a natural deduction S4, where contexts contain sequences of $\Box$-wrapped assumptions $x_1 : \Box A_1, \ldots x_n : \Box A_n$. Modelling these judgments does not require a context-splitting operation unlike in our approach, which uses the n operation of the form $\mathsf{n} : \Box(A \times B) \to \Box A \times \Box B$. Our approach can be thought of as having a single $\Box$ modality over the context which can represent both flat whole-context dependence and structural per-variable dependence.

Coeffect-like calculi Recent works have also developed coeffect systems, following related approaches. Brunel, Gaboardi, Mazza, and Zdancewic derive a kind of general structural coeffect system, taking inspiration from bounded linear logic [3]. Their work provides an operational semantics and proves soundness of its coeffect system with respect to the semantics. Their coeffect system provides a coeffect-indexed !-modality as a type constructor in the language, and explicit coeffect-inducing expressions for a particular notion of coeffect.

This differs to our approach where coeffects are implicit: they are not attached to a type constructor and can be introduced through variable use, *e.g.* the (*var*) rule. The coeffect systems of Brunel *et al.* allow "local coeffects", which we called structural (per-variable), but not "global coeffects" (flat or per-context). Our previous work provided just global/flat (per context) coeffects. Our new system here reconciles both kinds into one system.

There are a number of encouraging similarities in the approach of Brunel *et al.* Their semantics is similar (even isomorphic in some parts) to ours, defining an indexed comonad like structure

in terms of a *positive action* $\bullet : \mathcal{S} \times \mathcal{A} \to \mathcal{A}$ on a monoidal category of coeffect annotations $\mathcal{S}$ (and base category $\mathcal{A}$) and an *exponential action* providing related operations to our indexed monoidal operations m and n.

There has been some other related bounded linear logic work, by Ghica and Smith, with "resource-aware types" annotated by a semiring of resource bounds [5]. This allows reuse bounds for BLL to be tracked, as well as other kinds of concurrency information. A categorical semantics is provided which is similar in style to that of Brunel *et al.* and which loosely resembles our indexed comonad approach (but does not require the additional indexed monoidal structures we used here). Included in their work, which this paper lacks, is a procedure for type-inference (using a decision procedure on semirings). Future work for us is to adopt a similar approach for inference of coeffects in our system.

Both the works of Brunel *et al.* and Ghica *et al.* use a semiring structure for annotations. Our scalar coeffect structure is similar: it is a semiring without commutativity of the $+$ operation (although all our examples here have a commutative $+$) and without the absorption law for multiplication (which only some of our systems have).

Future work is to unify the approaches of this paper and the coeffect systems of Brunel *et al.* and Ghica *et al.* Initial comparisons show several similarities, suggesting that unification is plausible.

7. Conclusions

In this paper, we looked at two forms of context-dependence analysis – *flat* coeffect systems that track whole-context requirements (such as implicit parameters, resources, or platform version) and *structural* coeffects that track per-variable requirements (such as usage or data access patterns). The newly introduced structural system makes applications such as liveness, bounded reuse, and dataflow analysis (from our earlier work) practically useful. With the move towards cross-platform systems running in diverse environments, analysing context dependence is vital for reasoning and compilation. The coeffect calculus provides a foundation for further study, similar to the type-and-effect discipline.

We presented the system together with its syntactic equational theory and categorical semantics. The equational theory is presented in order to explain how the systems work, but it also provides a basis for an operational semantics for concrete systems. Exploring these, and their connection to the denotational semantics, is further work.

Acknowledgments

Thanks are due to Marco Gaboardi, Dan Ghica, Marcelo Fiore, and Tarmo Uustalu for discussions on this work. We thank the anonymous reviewers for their comments and suggestions. This work was supported by CHESS.

References

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

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

[3] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic. A core quantitative coeffect calculus. In *Proceedings of ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014.

[4] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Prooceedings of PLDI*, pages 338–349. ACM, 2003.

[5] D. R. Ghica and A. I. Smith. Bounded linear types in a resource semiring. In *Proceedings of ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 331–350. Springer, 2014.

[6] 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.

[7] 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.

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

[9] J. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *Proceedings of POPL*, page 118, 2000.

[10] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991. ISSN 0890-5401.

[11] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):23:1–23:49, June 2008.

[12] P. O'Hearn. On bunched typing. *J. Funct. Program.*, 13(4):747–796, July 2003. ISSN 0956-7968.

[13] D. Orchard. Programming contextual computations (PhD dissertation). Technical Report UCAM-CL-TR-854, University of Cambridge, Computer Laboratory, 2014. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-854.pdf`.

[14] T. Petricek. Context-aware programming languages (PhD thesis), 2014. Forthcoming.

[15] T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: Unified static analysis of context-dependence. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2013.

[16] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, 2001.

[17] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *J. Funct. Program.*, 17(4-5):547–612, July 2007.

[18] R. Tate. The sequential semantics of producer effect systems. In *Proceedings of POPL 2013*, pages 15–26, 2013.

[19] T. Uustalu and V. Vene. Comonadic notions of computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008. .

[20] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985. ISBN 0-12-729650-6.

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