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 the 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 it more practically useful. We show that the structural system enjoys desirable syntactic properties and we give its 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 \(\lambda\)-calculus famously analyses such context usage. Other systems go further. For example, bounded linear logic tracks the number of times a variable is used [5].

In software engineering, “context” provides more than just free-variable values. For example, in a distributed system, the context provides different resources that may be available on different devices (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 [13]. However, the formalization was restricted to tracking a class of whole-context properties where a term has just one coeffect. This limited the applications and precision of any analysis. For example, a whole-context liveness analysis marks the 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 a general coeffect type system (Section 3) and demonstrate two concrete uses – flat coeffect systems that tracks whole-context information and structural coeffect systems which tracks fine-grained per-variable information.
- 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 [13].
- We discuss the syntactic properties of the structural coeffect system (Section 4). It satisfies type preservation under \(\beta\)-reduction and \(\eta\)-expansion, allowing its use with both call-by-name and call-by-value languages. This important property distinguishes it from both effect systems and flat coeffects.
- We provide a denotational semantics, resisting 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 in programming that keep turning up. 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.
dered scalars, we write vector. So, a vector of length \( v \) includes a coeffect annotation \( \tau \) which is 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 \( C \) of scalars such as \( r_1, \ldots, r_n \in C \). A vector \( \mathbf{R} \) over \( C \) is a tuple \( (r_1, \ldots, r_n) \) of scalars. We use letters like \( R, S, T \) for vectors and \( r, s, t \) for scalars. We also say that a shape of a vector \( \mathbf{R} \) (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 on scalars into a scalar-vector one: \( s \otimes R = \langle s \otimes r_1, \ldots, s \otimes r_n \rangle \). Given two vectors \( R, S \) of the same shape, containing partially ordered scalars, we write \( R \preceq S \) for the pointwise extension of \( \leq \) on scalars. Finally, the associative operation \( \otimes \) concatenates vectors.

We note that an environment \( \Gamma \) containing \( n \) uniquely named, typed variables is also a vector, but we continue to write \( \langle \cdot \rangle \) 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 [5] restricts well-typed terms to polynomial-time algorithms. This is done by limiting the number of times a value (proposition) can be used. An assumption \( !A \) means that a variable can be used at most \( k \) times. We attach annotations to the whole context rather than individual assumptions and so a context \( \Gamma \) includes annotation \( \tau \) for the positions \( \langle k_1, \ldots, k_n \rangle \). This difference is further explained in Section [6].

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:

\[
\Gamma \vdash R \quad \Gamma, \sigma \vdash R(x(0)) \quad \Gamma, \sigma, \tau \vdash R(x) \quad \Gamma, \tau \vdash R(x) \quad \Gamma, \sigma, \tau \vdash R(s, t) \quad \Gamma, \tau \vdash R(s, t) \quad \Gamma \vdash \tau
\]

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

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

### Bounded linear coeffects

The system in Figure 1 extends the outlined idea into a simple calculus. Variable access (var) has a singleton context with a singleton coeffect vector \( 1 \). 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 \( \sigma \) as a latent coeffect. The remaining variables in \( \Gamma \) 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 \( \Gamma_2 \) \( S \)-times means that, in total, the variables in \( \Gamma_2 \) will be used \( (t + S) \)-times. Recall that \( t + S \) is a scalar multiplication of a vector. Meanwhile, the variables in \( \Gamma_1 \) 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 using 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 \( (x + y) (x + y) \) 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:

\[
\begin{align*}
&\Gamma, x, y : Z, v : \mathbb{Z} @ (1, 2) \vdash x + v + v : Z \\
&\quad \quad \quad \Gamma, x, y : Z, v : \mathbb{Z} @ (1, 2) \vdash (x' + y) : Z \\
&\quad \quad \quad \quad \quad \Gamma, x, y : Z, v : \mathbb{Z} @ (1, 2) \vdash x + v + v + v : Z
\end{align*}
\]

The first step performs scalar multiplication, producing the vector \( (1, 2, 2) \). 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 \( \beta \)-reduction.

### 2.3 Dataflow and data access

Dataflow languages such as Lucid [13] describe computations over streams. 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 \texttt{prev} \( e \) returns the previous value of \( e \), where \texttt{prev} (\texttt{prev} \( e \)) therefore returns the second past value.

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 in that it tracks a vector of numbers \( R \) as part of the context \( \Gamma \vdash R \). Here, coeffects represent the maximal number of past values \( \text{past value} (\text{causality depth}) \) required for a variable.

Weakening, exchange, abstraction and sub-co-effecting are the same as in bounded linear coeffects, but the remaining rules differ. In Figure 2 accessed variables \( \texttt{var} \) are annotated with 0 meaning that no past value is required (only the current one). The \texttt{(prev)} rule
creates 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 contracting, requiring s and t past values, then overall we need at most max(s, t) past values (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 e2 are needed which in turn requires S past values of its free variables Γ2. Thus, we need t + S past values of Γ2 to perform the call (e.g., we need 1 + S values to get 1 past value of the input σ, 2 + S values to get 2 past values of σ, etc.).

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

\[ \Gamma_1, y : \tau, z : \tau, \Gamma_2 \vdash R \times (s, t) \times Q \vdash e : \tau \]

(contr)

\[ \Gamma_1, x : \tau, \Gamma_2 \vdash R \times (max(s, t)) \times Q \vdash e[y, z \leftarrow x] : \tau \]

(app)

\[ \Gamma_1, \Gamma_2 \vdash R \vdash e_1 : \sigma \]

\[ \Gamma_1, \Gamma_2 \vdash e_2 : \sigma \]

(var)

\[ x : \tau \vdash (0) \vdash x : \tau \]

(prim)

\[ \Gamma \vdash R \vdash prev e : \tau \]

Figure 2: Type and coeffect system for dataflow caching

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 (param) (a system-specific rule). 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 [13]. In particular, the operations used in abstractions and application may differ (to accommodate over-approximation). We return to this in Section 3.4.

Example. Unlike structural coeffect systems, flat 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 \(\lambda x. \{?p_1, ?p_2\}\). There are nine possible type and coeffect derivations, two of which are:

\[ \{?p_1\} \vdash \{\ldots\} : \text{unit} \rightarrow \{\{?p_1, ?p_2\}\} \rightarrow Z \]

\[ \{?p_1\} \vdash \{?p_2\} \vdash \{\ldots\} : \text{unit} \rightarrow \{\{?p_1, ?p_2\}\} \rightarrow Z \]

In the first case, both parameters are dynamically scoped and have to be provided by the caller. In the second case, the parameter \(?p_1\) 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 – Haskell’s implicit parameters use it and it can be used to model resource rebinding in distributed systems such as [15].

2.5 Pathways to coefficients

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

Effect systems. Effect systems [4] track effectful operations of computations such as memory access or lock usage [3]. They are written as judgments \(\Gamma \vdash e : \tau \& \rho\) associating effects \(\rho\) 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.” [16]. This thunking is typically a \(\lambda\)-abstraction. Given an effectual expression \(e\), the function \(\lambda x. e\) is an effect-free value that delays all effects:

\[ (\text{abs}) \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \& \rho \]

\[ \Gamma \vdash \lambda x. e : \tau_1 \vdash \tau_2 \& \rho \vdash \emptyset \]

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 abstrac-
Categorical semantics. Moggi models effectful computations as functions of type $\tau_1 \to M\tau_2$ where $M$ is monad providing composition of effectful computations [8]. Wadler and Thiemann [19] link effect systems with monads using annotated monads $\tau_1 \to M\tau_2$ whose semantics has been provided by Katsumata [6].

Context-dependent computations require a different model. Uustalu and Vene [17] use functions $C\tau_1 \to \tau_2$ where $C$ is a comonad. Our earlier work [13] used indexed comonads with denotations $C^\triangleright\tau_1 \to \tau_2$ 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 [8]. 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.

Looking at context, Uustalu and Vene [17] follow the first approach (using a concrete comonad to model dataflow). Contextual-Modal Type Theory (CMIT) of Nanevski et al. [B] follows the latter approach, adding a comonad via the $\Box$ modality of modal $S4$ to the language. We focus on concrete languages using the first approach. A “coeffect meta-language” is an interesting future work.

Sub-structural and bunched types. 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.

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 [13] are flat (whole-context) systems. This section presents our primary contribution: the general coeffect calculus.

The calculus is parameterised by an algebraic structure of co-effects. 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 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 $\Gamma @ R \vdash e : \tau$:

- As contextual capabilities, the rules should be read bottom-up. The capabilities provided to a larger term are split between sub-terms; 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 is a function of the form $D_\kappa[\Gamma] \to [\tau]$ where $D_\kappa[\Gamma]$ encodes the contextual capabilities used to evaluate a term. Similarly functions have models of the form $D_\kappa[\sigma] \to [\tau]$ 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 comprised 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 $\sigma \to \tau$.

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. Coeffect scalar $(C, \oplus, \otimes, use, ign, \preceq)$ is a set $C$ together with elements $use, ign \in C$, relation $\preceq$ and binary operations $\otimes, \oplus$ such that $(C, \otimes, use)$ and $(C, \oplus, ign)$ are monoids and $(C, \preceq)$ is a pre-order. The following distributivity axioms are required:

\[
(r \oplus s) \otimes t = (r \otimes t) \oplus (s \otimes t)
\]

\[
t \otimes (r \oplus s) = (t \otimes r) \oplus (t \otimes s)
\]

The operation $\otimes$ 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 $\otimes$ and $\oplus$ do not always mean $\cdot$ and $+$.

The context annotations 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. We take inspiration from the work of Abbott et al. [1] which describes containers in terms of shapes and a set of positions in each shape.

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

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

**Definition 2.** A coeffect shape structure $(S, [\cdot], 0, 1)$ comprises a set $S$ with a binary operation $\circ$ on $S$ for shape composition, a mapping from contexts to shapes $[\Gamma] \in S$, and elements $0, 1 \in S$ such that $(S, \circ, 0)$ is a monoid. The elements $0$ and $1$ represent the shapes of empty and singleton free-variable contexts respectively.

As said earlier, we use two kinds of shape structures that describe the shape of vectors and the shape of trivial singleton container:

- Structural coeffect shape is defined as $(\mathbb{N}, [\cdot], +, 0, 1)$. We treat numbers as shape structures with $0 = \{\}$, $1 = \{\emptyset\}$, $2 = \{\emptyset, 1\}$, $3 = \{\emptyset, 1, 2\}$... (so that a number is a set of positions). The shape mapping $[\Gamma]$ returns the number of variables in $\Gamma$. 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 $R \in \mathbb{N}$ and assigns a coeffect scalar $R(i) \in C$ for each variable $x_i$ in the context.

- Flat coeffect shape is defined as $(\{\cdot\}, \text{const}, \circ, \ast, \ast)$ where $\ast \circ x = \ast$ and $\ast = \{\emptyset\}$. That is, there is a single shape $\ast$ with a single position and all free-variable contexts have the same singleton shape. Therefore, a coeffect annotation is drawn from $\mathbb{N}$ which is isomorphic to $C$ and so a coeffect scalar $r \in C$ is associated with every free-variable context.

Using a shape with no positions reduces our system to the simply-typed $\lambda$-calculus with no context annotations. Trees could be used to build a system akin to bunched typing [10].

**Coeffect algebra.** The coeffect calculus annotates judgments with shape-indexed (or, shaped) coeffects. The coeffect algebra structure combines a coeffect scalar and coeffect shape structures to define shaped coeffects and operations for combining these. In Section 2, shaped coeffects were combined with the tensor $\times$ in structural and $\ast$ in the implicit parameters example. To capture the examples so far and those described previously [13], we distinguish two operators for combining shaped coeffects.

**Definition 3.** Given a coeffect scalar $(C, \circ, \ast, \use, \ign, \less)$ and a coeffect shape $(S, [\cdot], 0, 1)$ a coeffect algebra extends the two structures with $([\cdot], \use, \ign)$ where $\use \in C^0$ is a coeffect annotation for the empty context and $\use \in C^0$ is families of operations that combine coeffect annotations indexed by shapes. That is $\forall n, m \in S$:

$$\forall m, n : C^m \times C^n \rightarrow C^{m \circ n}$$

A coeffect algebra induces the following two additional operations:

$$\langle - \rangle : C \rightarrow C^1$$

$$\langle x \rangle : \lambda \lambda.x$$

$$\langle - \rangle$$

lifts a scalar coeffect to a shaped coeffect indexed by the singleton context shape. The $\circ_m$ operation is a left multipication of a vector by 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 $\circ$.

The coeffects $\circ$ and $\circ_m$ combined shape coeffects associated with two contexts. For example, assume we have $\Gamma_1$ and $\Gamma_2$ with coeffects $R \in C^m$ and $S \in C^n$. In the structural system, the context shapes $m, n$ denote the number of variables in the two contexts. The combined context $\Gamma_1, \Gamma_2$ has a shape $m \circ n$ and the combined coeffects $R \times S, R \times S \in C^{m \circ n}$ are indexed by that shape.

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

<table>
<thead>
<tr>
<th>$\Gamma \vdash R \vdash e : \tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(const) $\bot \vdash e : \tau$</td>
</tr>
<tr>
<td>(var) $x : \sigma \vdash e : \tau$</td>
</tr>
<tr>
<td>(abs) $\Gamma, x : \sigma \vdash R \times e : \sigma \vdash \tau$</td>
</tr>
<tr>
<td>(app) $\Gamma_1 \vdash e_1 : \tau \vdash \sigma, \Gamma_2 \vdash e_2 : \tau$</td>
</tr>
<tr>
<td>(let) $\Gamma_1, \Gamma_2 \vdash R \times e \vdash \tau \vdash e_1 : \tau$, $\Gamma_2 \vdash e_2 : \sigma$</td>
</tr>
</tbody>
</table>

Figure 4: The general coeffect calculus

The difference is explained by the semantics (Section 5), where $R \times S$ 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): $R \times S$ 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 $\times$ in the rule assumptions and $\use$ in conclusions. For now, it suffices to use the bounded reuse intuition and read the operations as tensor products.

### 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 coeffects $R, S, T \in C^1$ and function types are annotated with coeffect scalars $r, s, t \in C$. The rules manipulate coeffect annotations using the operations provided by coeffect algebra $(\times, \use, \ign)$ and the derived constructs $(\langle - \rangle$ and $\circ$. Free-variable contexts $\Gamma$ are treated as vectors modulo duplicate of variables – the associativity is built-in. The order of variables matters, but can be changed using the structural rules. To make the system easier to follow, structural rules are expressed using a separate judgment.

**Typing rules.** Constants (const) and variable access (var) annotate the context with special values. Empty unused context is annotated with $\emptyset \in C^0$, while singleton context is annotated with $\use \in C^1$. Note that the shapes $0, 1$ match the shape of the variable contexts.

Lambda abstraction splits the context requirements using $\times$ into a coeffect $R$ and a coeffect $\langle s \rangle$ of a shape 1 (semantically, it merges capabilities provided by the declaration-site and call-site contexts).
In structural systems such as bounded reuse, \( x \) is not symmetric and so this gives us a coeffect associated with the bound variable.

The \((\text{app})\) rule follows the patterns seen earlier — it uses the scalar-vector multiplication \((t \otimes S)\) on coeffects associated with \( \Gamma_2 \). Using the syntactic reading, it then merges context requirements for \( \Gamma_1 \) and \( \Gamma_2 \) in the dual semantic reading, it splits the provided context into two parts passed to the sub-expressions.

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

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

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

The \((\text{weak})\) rule drops an ignored variable annotated with \((\text{ign})\) (compare with \((\text{var})\) annotated using \((\text{use})\)). The \((\text{exch})\) rule switches the values while \((\text{contr})\) combines them using \((\oplus)\) to represent sharing of the context. Finally, \((\text{sub})\) represents sub-coeffecting that can be applied (point-wise) to any individual coeffect.

### 3.4 Structural coeffects

The coeffect system uses a general notion of context shape, but it has been 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 (vectors) over a coeffect scalar. This means that the system keeps a vector of basic coeffect annotations — one for each variable. An empty context \((\text{e.g., in the (const) rule})\) is annotated with an empty vector.

**Definition 4.** Given a coeffect scalar \((C, \oplus, \odot, \text{use}, \text{ign}, \leq)\) a structural coeffect system has:

- Coeffect shape \( \{N, [], +, 0, 1\} \) formed by natural numbers
- Coeffect algebra \( (x, \epsilon) \) where \( x \) and \( \epsilon \) are shape-indexed versions of the binary operation and the unit of a free monoid over \( C \). That is \( x : C^n \times C^m \rightarrow C^{n+m} \) appends vectors (lists) and \( \epsilon : C^0 \) represents empty vectors (lists)

The definition is valid since the shape operations form a monoid \( (N, +, 0) \) 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 \( \{N, *, +, 1, 0, \leq\} \). As in the system of Fig. 1, \textup{used} variables are annotated with 1 and \textup{unused} with 0. Contraction adds the number of uses via \textup{via} and application (sequencing) multiplies the uses.

- Dataflow uses natural numbers (of past values), but differently: \( \{N, +, \text{MAX}, 0, 0, \leq\} \). Variables are initially annotated with 0 (and can be incremented using the \texttt{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 \textit{variable liveness}. The annotations are formed by \( C = \{D, L\} \) where \( L \) represents a \textit{live} (used) variable and \( D \) represents a \textit{dead} (unused) variable. The scalars are given by \( C = \{L, \land, L, D, \land, L\} \).

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

Structural liveness is a practically useful, precise version of an example from our earlier work, which was a flat system overapproximating liveness to the entire context [13].

### 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 \( \{\} \). In this setting, the context annotations \( C^* \) are equivalent to coeffect scalars \( C \).

In addition to the coeffect scalar structure, we also need to define \( x \) and \( \times \). Our examples of flat coeffects use \( \oplus \) (merging of scalar coeffects) for \( x \) (merging of shaped coeffect annotations). However, the \( x \) operation needs to be provided explicitly.

**Definition 5.** Given a coeffect scalar \((C, \oplus, \odot, \text{use}, \text{ign}, \leq)\) and a binary \( \land : C \times C \rightarrow C \) such that \((r \land s) \leq (r \lor s)\), we define:

- Flat coeffect shape \( \{\}, \text{const} \times, \ominus, \odot, \star \) where \( \star \circ \circ \circ \star = \star \)
- Flat coeffect algebra \((\land, \odot, \text{ign})\), \textit{i.e.}, the \( x = \odot \) and \( \perp = \text{ign} \) with the additional binary operation \( x = \land \).

The requirement \((r \land s) \leq (r \lor s)\) guarantees exchange and contraction preserve the coeffect of the assumption in the conclusion. Thus, flat coeffect calculi do not require substructural-style rules.

**Examples.** Implicit parameters are the prime example of a flat coeffect system, but other examples include rebindable resources [12] and Haskell type classes [11].

In the \textit{implicit parameters} system (Section 2.4), the coeffect scalars are sets of names with types \( C = \mathcal{P}(\text{Name} \times \text{Types}) \). Variables are always annotated with 0 and coeffects are combined or split using set union \( \cup \). Thus the system is given by coeffect scalar structure \((\mathcal{P}(\text{Name} \times \text{Types}), \cup, \cup, \emptyset, 0, 0)\) with \( \land = \emptyset \).

**Remark 6.** We previously described flat systems for liveness and dataflow [13]. Turning a structural system to flat requires finding \( \land \) that underapproximates the capabilities of combined contexts. For dataflow, this is given by the min function as \( \min(r, s) \leq \max(r, s) \).

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

\[
\Gamma, x : \sigma \vdash \min(r, s) + e : \tau
\to
\Gamma \vdash \lambda x. e : \sigma \rightarrow \tau
\]

Both the declaration-site and the call-site need to provide at least the number of past values required by the body. The overapproximation means that both \( r \) and \( s \) can be greater than actually required. For dataflow, we could annotate both contexts with the same coeffect, but that would require treating \( x \) 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 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 4.4 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 $e_1[x \leftarrow e_2]$, $\beta$-reduction and $\eta$-expansion, written as $\leadsto_\beta$ and $\leadsto_\eta$. Equality of terms $e_1$ and $e_2$, defined by a relation $\equiv$, requires the equality of their contexts, types, and coeffects, written $\Gamma \vdash e \equiv e' : \tau$.

### 4.1 Structural coeffect systems

For structural coeffect systems, recall that coeffects are vectors with $x = \times = \times$ (vector concatenation) and $\bot = \emptyset$ (the empty vector), thus coeffect annotations comprise the free monoid i.e., lists over the coeffect scalars (although we continue using the vector terminology). We first show substitution:

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

\[ \Gamma \vdash e \cdot \sigma \land \Gamma_1, x : \sigma, \Gamma_2 \vdash R_1 \times \langle \rho \rangle \times R_2 \vdash e_i : \tau \]

$\Rightarrow \Gamma_1, \Gamma_2 \vdash R_1 \times (r \oplus S) \times R_2 \vdash e_i[x \leftarrow e_s] : \tau$

**Proof.** By induction over the $e_i$ derivation using the free monoid structure $(\mathcal{C}, \times (\emptyset))$ and coeffect scalar axioms (full proof [12]).

Because of the vector (free monoid) structure, coeffects $R_1, R_2$, and $\langle \rho \rangle$ for the receiving term $e_i$ are uniquely associated with $\Gamma_1, \Gamma_2$, and $x$ respectively. Therefore, substituting $e_i$ (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 $\beta$-equality:

\[ \Gamma_1, x : \sigma @ R \times \langle r \rangle \vdash e_1 : \tau \]

\[ \Gamma_1 \vdash R \vdash \times e_1 : \sigma \Rightarrow \Gamma_2 \vdash S \vdash e_2 : \sigma \]

$\Rightarrow \Gamma_1, \Gamma_2 \vdash R \times (r @ S) \vdash \langle \lambda x.e \rangle e_1 \equiv e_1[x \leftarrow e_s] : \tau$

As a result, $\beta$-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 $\Gamma \vdash e : \tau$ and $e \leadsto e'$ then $\Gamma \vdash e' : \tau$.

**Proof.** Following from Lemma 7 and $\beta$-equality.

Structural coeffect systems also exhibit $\eta$-equality, therefore satisfying both local soundness and local completeness conditions set by Pfenning and Davies [14]. This means that abstraction does not introduce too much, and application does not eliminate too much.

\[ \Gamma \vdash e \cdot \sigma \Rightarrow \tau \]

\[ \Gamma, x : \sigma \cdot R \times (s @ \text{use}) \vdash e \cdot x : \sigma \]

\[ \Rightarrow \Gamma \vdash \lambda x.e \equiv e : \sigma \Rightarrow \tau \]

The last step uses the fact that $s @ \text{use} = (s @ \text{use}) = (s)$ arising from the monoid $(\mathcal{C}, \oplus, \text{use})$ of the scalar coeffect structure.

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

**What is a 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 we say that $e$ is a value if it has no immediate context requirements.

**Definition 9.** An expression $e$ is a value, written as $\text{val}(e)$ if $\text{val} \vdash e : \tau$ where $\text{val} : \mathcal{C}^\mathcal{T}$ is a coeffect indexed by the shape of $\Gamma$ that always returns $\text{val}$. Use that $\text{val} = \lambda n.\text{use}$.

**Call-by-value.** In call-by-value, a pure value is substituted for a variable. The right-hand side of an application is evaluated to a value before the $\beta$-reduction, but the discharging of coeffects prior to substitution is different for each concrete coeffect system.

Recall that a flat coeffect system consists of coeffect scalars $(\mathcal{C}, \oplus, \otimes, \text{use}, \text{ign}, \leq)$ together with a binary operation $\&$ on $\mathcal{C}$. The coeffect algebra is then defined as $(\mathcal{A}, \oplus, \&)$ where $\&$ and $\oplus$ represent splitting and merging of context-requirements, respectively.

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

\[ \Gamma \vdash \text{val} \vdash e_1 : \sigma \land \Gamma_1, x : \sigma, \Gamma_2 \vdash e_2 : \tau \]

$\Rightarrow \Gamma_1, \Gamma_2 \vdash e_2 \cdot e_1[x \leftarrow e_s] : \tau$

**Proof.** By induction over the type derivation, using the fact that both $x$ and $e_s$ are annotated with $\text{use}$.

**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 $\Gamma \vdash e$ to a value. Assuming $\leadsto_{cbv}$ is call-by-value reduction using the above definition of value:

**Theorem 11 (Call-by-value reduction).** In a flat coeffect system, if $\Gamma \vdash e : \tau$ and $e \leadsto_{cbv} e'$ then $\Gamma \vdash e' : \tau$.

**Proof.** A direct consequence of Lemma 10 using the flat coeffect system requirement $(r \& s) \leq (r \oplus s)$ to prove $\beta$-equality.

**Call-by-name.** The call-by-name strategy reduces a term $(\lambda x.e_1)$ when both sub-expressions may have contextual requirements. Assume that $\Gamma \vdash e_1 : \tau_2$ and $\Gamma, x : \tau_1 \vdash e_1 : \tau_1$.

We call a flat coeffect algebra top-pointed if $\text{use}$ is the greatest (top) coeffect scalar $\mathcal{C}$ and bottom-pointed if it is the smallest (bottom) coeffect scalar. Liveness analysis is an example of top-pointed coeffects as variables are annotated with $\mathcal{L}$ and $\mathcal{D} \leq \mathcal{L}$.

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

\[ \Gamma \vdash e_1 : \sigma \land \Gamma_1, x : \sigma, \Gamma_2 \vdash e_2 : \tau \]

$\Rightarrow \Gamma_1, \Gamma_2 \vdash e_2[x \leftarrow e_s] : \tau$

**Proof.** Using sub-coeffecting $(s \leq \text{use})$ 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 $s \leq use$).

In a bottom pointed coeffect system, substituting $e$ for $x$ increases the context requirements. However, if the system satisfies the strong condition that $\Delta = \emptyset = \emptyset$ then the context requirements arising from the substitution can be associated with the context $\Gamma$. As a result, substitution does not break soundness as in effect systems. The requirement $\Delta = \emptyset = \emptyset$ 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 $\{C, \circ, \oplus, use, \oplus, \leq\}$ and the $\wedge$ operator where $\wedge = \circ = \oplus$ is idempotent and commutative:

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

**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 the coeffects associated with a 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 $\frac{\text{--cbn}}{}$ is the standard $\text{call-by-name}$ reduction, the following theorem holds:

**Theorem 14** (Call-by-name reduction). In a coeffect system that satisfies the conditions for Lemma 12 or Lemma 13 if $\frac{\Gamma \times r \vdash e : \tau}{\Gamma \times r \vdash e' : \tau}$ then $\frac{\Gamma \times r \vdash e \rightarrow_{\text{cbn}} e'}{\Gamma \times r \vdash e' : \tau}$.

**Proof.** A direct consequence of Lemma 12 or Lemma 13.

5. **Semantics**

Coeffects provide a unified notion 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 the flat system, a similar result has already been shown in the second author’s dissertation [11]. The semantics is introduced in pieces:

- Section 5.1 describes the range and domain (signature) of the interpretation $\llbracket \cdot \rrbracket$, 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 [13] and the indexed structural comonad structure (new here).
- More structure is needed for 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 along with structural rules. The semantics is illustrated by executing an example bounded-reuse program in the semantics (Example 26).

- Section 5.5 shows our semantics sound with respect to the syntactic equational theory of Section 4. This uses the derivation of categorical structures for the semantics as lax homomorphisms between structure on the coeffect category $\mathbb{I}$ and the base $C$.

In this section, $C$, $\mathbb{D}$, $\mathbb{I}$ range over categories. The objects of a category $C$ are written $\text{obj}(C)$. The category of functors between $C$ and $\mathbb{D}$ is written $[C, \mathbb{D}]$. 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 $(C, \circ, \oplus, use, \oplus, \leq)$, coeffect shape $(\tau, \{\llbracket\cdot\rrbracket, \circ, \oplus, 0, 1\})$, and $(\times, x, \bot)$. An interpretation $\llbracket \cdot \rrbracket$ is given to types, free-variable contexts, and type and coeffect judgments, with a base Cartesian-closed category $C$ for denotations and a category $\mathbb{I}$ of scalar coeffects, where $\text{obj}(\mathbb{I}) = C$. Since $C$ is Cartesian-closed, we use the $\lambda$-calculus as the syntax for giving concrete definitions.

The interpretation $\llbracket \cdot \rrbracket$ 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 $C$, of the form:

$$\llbracket \Gamma \times R \vdash e : \tau \rrbracket : D^n(\llbracket \Gamma \rrbracket) \rightarrow [\llbracket \tau \rrbracket]$$

The interpretation is a morphism from an interpretation of the context $\Gamma$ to the interpretation of the result. The functor $D^n(\llbracket \Gamma \rrbracket)$ over the context 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 coproduct of functors $D^n$ for every possible shape $n \in S$ of free-variable context:

$$D : \Sigma_{n \times S} D^n$$

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

From a programming perspective, this functor defines a data structure that models the additional context provided to the program. The shape of this data structure depends on the coeffect annotation $I^n$. 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 $C$, that is $\llbracket \cdot \rrbracket : \text{obj}(C)$ where function types have the interpretation as exponentials:

$$\llbracket \sigma \rightarrow \tau \rrbracket = D^1(\llbracket \sigma \rrbracket) \Rightarrow [\llbracket \tau \rrbracket]$$

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

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

We define $\llbracket \cdot \rrbracket$ on free-variable context 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 $C$. For structural systems, the shape matches the number of variables and so the model is a value in the product category $C \times \ldots \times C$.

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

$$\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n \rrbracket = [\tau_1] \times \ldots \times [\tau_n]$$
Denotations of typing judgments in a flat coeffect system are thus of the form (where \( R \in I \)):
\[
[x_1 : \tau_1, \ldots, x_n : \tau_n @ R \vdash e : \tau] : D_R([\tau_1] \times \cdots \times [\tau_n]) \to [\tau]
\]

**Structural coeffects.** Recall that \( S = \mathbb{N} \) and \([I] = [\Gamma] \) (number of free variables), thus \([\Gamma] : obj(\mathbb{C}^I)\). 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:
\[
[x_1 : \tau_1, \ldots, x_n : \tau_n @ R \vdash e : \tau] : D_R([\tau_1] \times \cdots \times [\tau_n]) \to [\tau]
\]
where \( |R| = n \) and we use commas (instead of \( \times \)) to denote the product of categories. This means that \( D^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}] \) is a functor between an \( n \)-length vector of coeffects indices and an \( n \)-ary endofunctor. As mentioned, 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 \( \mathbb{C} = \mathbb{N} \) and shapes \( S = \mathbb{N} \). We model contexts by replicating the value of each variable exactly once for each use. This matches the model used by Girard et al. [5]. Contexts are described by \( B : \Sigma_n : n.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]) \), where for \( R = (r_1, \ldots, r_n) \):
\[
B^R(A_1, \ldots, A_n) = A_1^{r_1} \times \cdots \times A_n^{r_n}
\]

Thus each object in the free-variable context \( A \) is indexed by its associated coeffect \( r \). For the morphism mapping, \( f_1 : A_1 \to B_1 \) and \( a_i : A_i^{r_i} \) hence \( f_1 \circ a_i : B_1^{r_i} \) thus by \( \exp(1) \) can be read as a product of \( r_i \) copies of \( A_i \), e.g.:
\[
B^1_{0,0}(A, B, C) = A^1 \times B^0 \times C^2 = (A) \times 1 \times (C \times C)
\]

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

Its contexts are defined by \( I^* : \Sigma_{\text{name} \times \text{set}}.(\mathbb{I}^n \to [\text{set}^n, \text{set}]) \), which is equivalent to \( I \to \text{[set, set]} \) and defined as follows:
\[
I^*_A = A \times [R] \quad I^*_A f = \lambda(a, r).(f a, r)
\]
The interpretation \([R] \) maps a set of variable-type pairs to an object representing a set of variables-values pairs in \( \text{Set} \).

### 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 \( D^i \), where \( I^1 = \mathbb{I} \) and \( C^1 = C \) in both flat and structural systems and thus \( D^1 : \mathbb{I} \to [\mathbb{C}, \mathbb{C}] \). Composition of morphisms \( f : D^n_A \to B \) and \( g : D^n_B \to C \) is defined by an indexed comonad (which we previously introduced briefly [13]).

**Definition 17.** An indexed comonad comprises a strict monoidal category \((\mathbb{I}, \bullet, I)\) and a functor \( F : I \to [\mathbb{C}, \mathbb{C}] \) with two natural transformations (where we write \( (F R) A \) as \( F_R A \)):
\[
(\delta_{X,Y} A) : F_{(X \times Y)A} \to F_X (F_Y A) \quad (\varepsilon_1 A) : F_I A \to A
\]
where \( \delta \) is called comultiplication and \( \varepsilon \) is called counit. We require indexed analogues of the usual comonad axioms (cf. [17]):
\[
\begin{align*}
F_R &\xrightarrow{\delta_{R,T}} F_RF_R \\
&\xrightarrow{\varepsilon_1 F_R} F_I \\
F_RF_R &\xrightarrow{\delta_{R,S} F_T} F_RF_SF_T \\
&\xrightarrow{\varepsilon F_R} F_R \\
F_RF_T &\xrightarrow{\delta_{R,S} T} F_RF_SF_T \\
&\xrightarrow{\delta_R \delta_{S,T}} F_RF_S F_T \\
&\xrightarrow{\delta_{R,S} T} F_RF_S F_T
\end{align*}
\]

Note the indexed comonad comultiplication \( \delta \) for associativity.

**Example 18 (Bounded reuse).** \( B^i_R \) (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 \( I \) (with \( 1 \) and the bifunctor \( \star : I \times I \to I \)), with operations:
\[
e_1 = \lambda(1), a_1
\]
\[
\delta_{R,S} = \lambda(a_1, a_{R+S}), \quad ((a_1, a_{R+S}), (a_{R+S+1}, \ldots)), (a_{R+1} + \ldots, a_{R+S})
\]

Indexed comonads essentially model single-variable contexts. The count requires a single copy of the value from the context. The 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 \( D^n_R(A_1, \ldots, A_n) = (A_1 \times \mathbb{A}^{+1}) \times \cdots \times (A_n \times \mathbb{A}^{+n}) \), 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 \((\mathbb{P}(\text{Name} \times \text{Types}), \cup, \emptyset)\) of implicit parameters, \( I^* \) (Example [16]) there is an indexed comonad structure, with operations:
\[
e_0(a, \emptyset) \to a \quad \delta_{R,S}(a, \gamma) \to ((a, \gamma), \gamma)|_R
\]
where \( \gamma = \{(x, v) | (x, v) \in \gamma, (x, t) \in R\} \) filters incoming implicit parameters to those variable-value pairs where the variable is in the coeffect \( R \).

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 introduced the notion of **structural indexed comonads**.

**Definition 21.** A structural indexed comonad comprises a functor \( D : \Sigma_{n:S} . (\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]) \) where \((I, \bullet, I)\) is a strict monoidal category, \( I \in S \) and \( D^i : I^i \to [\mathbb{C}^i, \mathbb{C}] \) is an indexed comonad (with \( (\delta_{X,Y} A) : D_X \times Y A \to D_X D_Y A \) and \( (\varepsilon_1 A) : D_I A \to A \)) and a structural comultiplication natural transformation:
\[
(\delta_{X,Y} A) : D_X \times Y A \to D_1 D_Y A
\]
where \( A^n \in \mathbb{C} \) is an indexed comonad with \( i \) and \( \bullet \) lifting scalar coeffects to shaped coeffects (e.g., the scalar-vector version of \( \bullet \)). An analogous law to monoid left actions for unitality and associativity hold for structural comultiplication:
\[
e_1 \circ \delta_{X,Y} = id \quad \delta_{X,Y} \circ \delta_{Y,Z} = \delta_{X,Y} \circ \delta_{Z,Y}
\]

Note the indexed comonad comultiplication \( \delta \) for associativity.

Structural indexed comonads provide composition for morphisms \( f : D^n_A \to B \) and singleton-shaped \( g : D^n B \to C \):
\[
g \circ f = g \circ D^n f \circ \eta^n_S : D^n \circ S A^n \to C
\]

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 shapes (shown in the next section).

**Example 22** (Bounded reuse). \( B : \Sigma_{n,N} \Gamma^\alpha \rightarrow [C^n, C] \) has a structured indexed comonad structure with the indexed comonad \( B \) (Example 15) and the following structural comultiplication:

\[
\delta^n_{r,S} = \lambda((a^1_1, \ldots, a^n_1, \ldots), \ldots, (a^1_1, \ldots, a^n_1, \ldots))
\]

This does not provide enough structure for a full semantics of abstraction and coeffects. Core to the semantics of abstraction is the partitioning of the incoming context into singleton-shaped contexts. 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 structured indexed comonad collapses to a regular indexed comonad on \( D^1 \).

### 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 full semantics of the 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:

\[
\begin{align*}
\Gamma_1 \triangleright R \vdash e_1 & \quad \Gamma_2 \triangleright S \vdash e_2 & \quad (\text{abs}) \\
\Gamma_1, \Gamma_2 \triangleright R \times S (\triangleright e_1 \triangleright e_2) & \quad (\text{app})
\end{align*}
\]

This equation computes the calling context and parameter context denotation as shown in the (abs) rule in Figure 5 using the structures described in the previous sections.

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

The premise of \( (\text{abs}) \) takes a context of shape \( n \) \( \circ \) \( 1 \) with coeffects \( RX(s) \) and a free-variables context consisting of \( I \) and an additional variable \( x \). The denotation \( g : D^n_{RX(s)} \Gamma, v : \sigma \rightarrow [\tau] \) is pre-composed with \( m \), such that its context is obtained by merging the declaration-site context \( (\Gamma) \) and call-site context \( (\sigma) \):

\[
g \circ m_{n,1} \Gamma_{R,:} (\bullet D^n_{R}[\Gamma] \times D^m_{\langle \sigma \rangle}) \rightarrow [\tau]
\]

This is uncurred 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 (\( \text{app} \)) has two sub-expressions for the function and argument, with denotations requiring two distinct contexts:

\[
g_1 : D^n_{R}[\Gamma_1] \rightarrow (D^m_{\langle \sigma \rangle}) \quad g_2 : D^m_{S}[\Gamma_2] \rightarrow [\tau]
\]

The target of \( g_1 \) is an exponential object with singleton shape for the parameter of type \( \sigma \). To evaluate \( g_1 \) and \( g_2 \), the semantics of (\( \text{app} \)) splits the incoming context over \( \Gamma_1, \Gamma_2 \) using \( n \):

\[
D^{m}_{RX(n)} ([\Gamma_1] \times [\Gamma_2]) \rightarrow (D^m_{\langle \sigma \rangle}) \rightarrow [\tau]
\]

Since \( e_2 \) computes the parameter for function \( e_1 \), the denotation \( g_2 \) must be sequentially composed with the parameter part of \( g_1 \). Thus, the structured indexed comonad is used with \( g_2 \) to compute the correct context for the parameter of \( g_1 \):

\[
D^n_{m[\circ]} [\Gamma_2] \rightarrow D^m_{\langle \sigma \rangle} [\Gamma_2] \rightarrow D^m_{\langle \sigma \rangle} [\Gamma_2] \rightarrow D^n_{m[\circ]} [\Gamma_2]
\]

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

\[
D^n_{\bullet[RX]} [\Gamma_1] \times D^m_{\bullet[S]} [\Gamma_2] \rightarrow D^n_{\bullet[RX]} [\Gamma_1] \times D^m_{\langle \sigma \rangle} [\Gamma_2]
\]

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 (\( \text{app} \)) rule in Figure 5.

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

\[
\Delta_r \cdot s : D^m_{\langle \sigma \rangle} A \rightarrow D^n_{\langle \sigma \rangle} (A \times A)
\]

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

\[
f : Z \rightarrow Z, x : Z@\langle 2, 4 \rangle \vdash (\lambda z. z + z) (f x)
\]
The full semantics has (Soundness)

\[ g : D^c_{R,\mathcal{E}}(\sigma) \Rightarrow (\Delta_{\mathcal{E}}(\epsilon) \circ \sigma) \]

\[ \Gamma_0 \vdash \lambda x.e : \sigma \Rightarrow \tau \]
Proposition 30. A structural indexed comonad provides \( \delta_{B \times C}^{n,m} : D^1 \circ \hat{D}_B^{n,m} \leftarrow D^1 \circ \hat{D}_C^{n,m} \) which witnesses that \( D \) is a colax homomorphism between the following monoid left-actions for \( \langle \circ, \circ, \text{ign} \rangle \) and \( \langle [C, C], \circ, 1_C \rangle \):

\[
\begin{align*}
(r : I) \circ ((s_1, \ldots, s_n) : \Pi^n) & = (r \circ s_1, \ldots, r \circ s_n) : \Pi^n \\
(D^1 \hat{D} \circ [C, C]) \circ (D^1 \hat{D} \circ [C^n, C]) & = D^1 \circ \hat{D} \circ [C^n, C]
\end{align*}
\]

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

The lax and colax indexed monoidal operations \( n, m \times C \) follow a similar derivation but as lax and colax monoid homomorphisms since every semantic structure has a unique corresponding tactic equational theory. Consequently, semantic proofs correspond to syntactic proofs, modulo naturality laws and product/exponent laws in \( C \). This result holds in the general coeffect calculus and since every semantic structure has a unique corresponding structure on coeffect annotations (i.e., \( (C, \times) \) for sequential composition of unary denotations, \( (C, \times) \) for splitting contexts, \( (C, \times) \) for joining contexts).

Example 31. Section 4.1 showed \( \eta \)-equality for structural systems, which uses the properties \( 1 \times = \times = \times \times \times \times \) for structural systems and \( 2 \times (\text{use}) = (2 \times \text{use}) = (2 \times) \). The semantics here is sound with respect to \( \eta \)-equality, the proof of which uses the corresponding axioms \( (1) \hat{m}_{X,Y} \circ \hat{m}_{X,Y} = \text{id} \) and \( (2) D^1 \circ \text{use} \circ \delta_{X,Y} \) (structural indexed comonad unit law, Definition 21).

The accompanying technical report shows the full semantic proofs for \( \beta \eta \)-equality whose structure corresponds exactly to the syntactic proofs with the corresponding coherence conditions [13].

6. Related work

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

The (storage) rule for bounded linear logic explains the contextual requirements induced by proposition reuse [5]:

\[
\begin{align*}
\text{(storage)} & \quad \frac{\Gamma \vdash T \vdash A}{X \Gamma \vdash 1 \times A &}
\end{align*}
\]

where \( X \Gamma = (X \Gamma_1, \ldots, X \Gamma_n) \) is the scalar multiple of a vector. This rule is akin to the \( \delta^n \) operation of structural indexed comonads, indeed, we can model it exactly using \( \delta^n_{X,Y} \) and the lifting \( D^\gamma \).

In BLL, the modality \( !X \) is a constructor and may appear both on the left- and right-hand sides of \( \circ \). 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 and its term language is that of standard \( \lambda \)-calculus.

Previously we briefly introduced indexed comonads [13] without derivation. Here we derived indexed comonads as colax homomorphisms. This is dual to the parametric effect monad structure defined as a lax homomorphism [6]. Our semantics requires additional structure not needed for effects due to the asymmetry inherent in the \( \lambda \)-calculus.

The necessity modality \( ! \) in S4 logic corresponds to a comonad with lax monoidal functor structure \( m : \Box A \times \Box B \rightarrow \Box (A \times B) \) 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 \( n \) unlike in our approach. Our approach can be thought of as having a single \( \Box \) modality over the context which allows both flat whole-context dependence and structural per-variable dependence.

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 dependency 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.

References