Analysing context dependence in programs

The tale of two coeffect calculi

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

Abstract

The concept of a *coeffect system* has been recently introduced to analyse context-dependence in programs. It has been shown to capture various contextual properties such as liveness, caching requirements in causal dataflow and dynamic variable scoping. Although the existing work provides a useful mechanism for tracking overall properties of the context, it has limited use for contextual properties associated with individual variables in the context, such as liveness.

We remedy this limitation by developing *structural* coeffects that captures fine-grained information about the usage of free variables. This captures additional instances of contextual properties and also makes the analysis of liveness and dataflow computations more precise and practically useful.

Since the work on coeffects is recent, we revisit the development here. We present the existing system as the *flat coeffect calculus* and further develop its syntactic and semantic properties. We then define the *structural coeffect calculus* and analyse its syntax and semantics. For contextual properties of variables, the structural coeffect calculus has desirable syntactic properties that are lacking in flat coeffect calculus. Finally, we extend *indexed comonads*, which models coeffects, to a structural variant.

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

General Terms Languages, Theory

Keywords Context, Indexed Comonads, Liveness, dataflow

1. Introduction

Modern applications behave differently depending on the environment or *context* in which they execute. Such context may be of different forms. Traditionally, it refers to free-variable typing assumptions. In a distributed system, additional resources available at certain nodes (*e.g.* a database or GPS sensor) can be viewed as contextual requirements associated with the free-variable assumptions. Alternatively, contextual requirements may relate to specific variables. For example, the liveness of a variable or access patterns, such as the number of past values in causal dataflow.

As described by Petricek *et al.* [19], context dependence is a form of impurity related to the *input* of a program rather than its *output* as in the notion of side effects. Thus, context dependence cannot be captured by effect systems and their semantics does not fit categorical models based on monads.

This can be understood by considering how input and output impurity interact with abstraction. In a recent formalization, Tate identifies a common property of effect systems *"all computations with [an] effect can be thunked as pure computations for a domain-specific notion of purity."* [22] The thunking usually refers to lambda abstraction; given an effectful expression e, the function $\lambda x.e$ is effect free value (thunk) that delays all effects.

Context-dependent computations do not follow this pattern. Indeed, lambda abstraction splits the free-variable context of an expression between variables defined at the *declaration site* (the lexical scope) and those provided by the *call site* (the parameter). Similarly, additional resources in a distributed system can be satisfied by both the declaration site (by capturing a remote reference) and the call site (by resource rebinding) [21]. In this case, the resource requirements are split freely. For contextual properties associated with variables (*e.g.* liveness or dataflow), the splitting tracks free variables – requirements associated with a *bound* variable should be satisfied by the call site, while those associated with *free* variables should be provided by the declaration site.

The previously described notion of a *coeffect system* tracks contextual properties of programs, associating the information with the entire context [19]. This approach was demonstrated for various notions of context-dependence, including implicit parameters, liveness, and dataflow access. These coeffect systems are however unable to track contextual properties relating to specific variables in the context. For this reason, liveness and dataflow coeffects were approximated over the entire context, for example, marking the whole context as live, when just one variable in the context is live.

The primary contribution of this paper is the introduction of the *structural coeffect calculus*, which provides a mechanism for tracking contextual properties attached to variables. We also revisit the existing work which we term the *flat coeffect calculus*. The two calculi can be seen as complementary, which we elucidate with examples that are natural in each. Our key contributions are:

- We revisit the *flat coeffect calculus* (Section 2), making its laws more precise, developing its syntactic properties, and giving reduction strategies (Section 2.5). We give semantics of Haskell-style implicit parameters and dataflow (Section 3).
- We introduce the *structural coeffect calculus* (Section 4), show how it captures precise analysis of liveness and dataflow (Section 4.3) and present its syntactic properties, such as type preservation under β-reduction and η-expansion (Section 4.4).
- The semantics of flat coeffect calculus is based on *indexed comonads*. We provide additional examples to guide the intuition (Section 3.4). From this basis, we give a semantics for structural coeffect calculus (Section 5), with examples.

$$\text{(app)}\ \frac{\Gamma \mathbin{@} t \vdash e_2 : \tau_1 \qquad \Gamma \mathbin{@} r \vdash e_1 : s \Rightarrow \tau_1 \to \tau_2}{\Gamma \mathbin{@} r \cup s \cup t \vdash e_1\ e_2 : \tau_2} \qquad \text{(impl)}\ \frac{\mathsf{?p} : \tau \in r}{\Gamma \mathbin{@} r \vdash \mathsf{?p} : \tau}$$

$$\text{(fun)}\ \frac{\Gamma, x : \tau_1 \mathbin{@} r \cup s \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : s \Rightarrow \tau_1 \to \tau_2} \qquad \text{(var)}\ \frac{x : \tau \in \Gamma}{\Gamma \mathbin{@} r \vdash x : \tau}$$

Figure 1. Type system for implicit parameters [19]

Section 1.1 shows a number of ad hoc type systems tracking contextual properties and Section 1.2 discusses existing literature leading towards coeffects. The rest of the paper develops two coeffect calculi: the flat and structural coeffect calculi. Both have an associated type systems and semantics which are parametric with respect to their notion of context. We instantiate the calculi for various contextual program features.

1.1 Why coeffects matter

Resources or meta-data provided by execution environments are typical examples of *flat coeffects*. We look at dynamically scoped implicit parameters, type-classes [9, 27] and dataflow [25]. Flat coeffects can also capture many aspects of distributed programming, including cross-compilation and rebindable resources [2, 4, 12, 21].

Context-dependent properties associated with variables are captured by *structural coeffects*. Our main motivation is variable liveness and caching in dataflow computations, but we give an example motivated by array stencil computations [18].

Implicit parameters. Implicit parameters are variables supporting dynamic scoping [9]. They can be used to parameterise a computation (involving, say, a complex chain of calls) without the effort of rewriting to pass arguments explicitly as additional parameters of all involved functions. We consider a variant of the lambda calculus with implicit parameters ?p and expressions **letdyn** $\mathsf{?p} = e_1$ **in** e_2 that evaluate e_2 in a context containing ?p.

The scoping of implicit parameters is not exclusively dynamic. Consider the following function that does some pre-processing then returns a function that builds a formatted string based on two implicit parameters ?wid and ?size:

$$\begin{array}{l}\textbf{let}\ \mathsf{format} = \lambda \mathsf{str}\ \to\ \textbf{let}\ \mathsf{lines} = \mathsf{formatLines\ str\ ?wid}\ \textbf{in} \\ \qquad\qquad\qquad\qquad (\lambda \mathsf{rest}\ \to\ \mathsf{append\ lines\ rest\ ?wid\ ?size})\end{array}$$

The body of the outer function accesses ?wid, and thus it requires a context of implicit parameters $\{\mathsf{?wid} : \mathsf{Int}\}$. The nested function also uses ?wid and additionally ?size. In a dynamic scoping, both of those would have to be defined when the returned function is called. However, implicit parameters can be also captured lexically.

As a result, the nested function may capture the value of ?wid and require just ?size or it may not capture the value and require both parameters. This corresponds to two possible Haskell types:

$$\begin{array}{l}(\mathsf{?wid} :: \mathsf{Int}) \Rightarrow \mathsf{Str} \to ((\mathsf{?size} :: \mathsf{Int}) \Rightarrow \mathsf{Str} \to \mathsf{Str}) \\ (\mathsf{?wid} :: \mathsf{Int}) \Rightarrow \mathsf{Str} \to ((\mathsf{?wid} :: \mathsf{Int}, \mathsf{?size} :: \mathsf{Int}) \Rightarrow \mathsf{Str} \to \mathsf{Str})\end{array}$$

The first type allows calling the function as follows:

$$\begin{array}{l}\textbf{let}\ \mathsf{formatHello} = (\textbf{letdyn}\ \mathsf{?wid} = 5\ \textbf{in}\ \mathsf{format}\ \texttt{"Hello"})\ \textbf{in} \\ \textbf{letdyn}\ \mathsf{?size} = 10\ \textbf{in}\ \mathsf{formatHello}\ \texttt{"world"}\end{array}$$

Figure 1 describes a type system for a simply-typed lambda calculus with implicit parameters, with judgements $\Gamma \mathbin{@} r \vdash e : \tau$ where r ranges over sets of implicit parameters with their types, such as $\{\mathsf{?size} : \mathsf{Int}\}$. The judgement means that, in a variable context Γ with implicit parameters r, an expression e has a type τ. The non-uniqueness of typing follows from the (*fun*) rule, where the context requirements of the body ($r \cup s$) are satisfied by a union of implicit parameters available in the current lexical scope (r) and the current dynamic scope (s). This is the key defining property of coeffects that distinguishes them from effect systems.

$$\text{(var)}\ \frac{x : \tau \in \Gamma}{\Gamma \mathbin{@} 0 \vdash x : \tau} \qquad \text{(prev)}\ \frac{\Gamma \mathbin{@} m \vdash e : \tau}{\Gamma \mathbin{@} m + 1 \vdash \textbf{prev}\ e : \tau}$$

$$\text{(app)}\ \frac{\Gamma \mathbin{@} n \vdash e_2 : \tau_1 \qquad \Gamma \mathbin{@} m \vdash e_1 : \tau_1 \mathbin{@} p \to \tau_2}{\Gamma \mathbin{@} max(m, p + n) \vdash e_1\ e_2 : \tau_2}$$

$$\text{(fun)}\ \frac{\Gamma, x : \tau_1 \mathbin{@} m \vdash e : \tau_2}{\Gamma \mathbin{@} m \vdash \lambda x.e : \tau_1 \mathbin{@} m \to \tau_2}$$

Figure 2. Type system for data flow [19]

Type classes. Implicit parameters are related to *type classes* [27], which provide overloading in Haskell. When an expressions uses an overloaded operation (e.g. numeric operators) a constraint is placed on the context in which the operation is used. For example:

$$\begin{array}{l}\mathsf{twoTimes}\ ::\ \mathsf{Num}\ \tau \Rightarrow \tau \to \tau \\ \mathsf{twoTimes}\ x = x + x\end{array}$$

The constraint $\mathsf{Num}\ a$ arises from the use of the $+$ operator. From the implementation perspective, this means that the function takes a dictionary that defines $+$. Thus the type $\mathsf{Num}\ \tau \Rightarrow \tau \to \tau$ can be viewed as $\mathsf{Num}_\tau \times \tau \to \tau$. Implicit parameters work in exactly the same way – they are passed around as hidden parameters.

Type classes and implicit parameters show two important points about context-dependent properties. 1) they are associated with a *scope* (e.g. function body) 2) they are associated (as extra hidden parameters) with function input and must be provided by the caller.

Dataflow computations. The (*app*) rule for implicit parameters resembles effect systems. It unions context requirements of both sub-expressions and the function. As an example with more complex structure, we show a system for calculating caching requirements of dataflow computations (modelled as streams of values).

Figure 2 describes a type system, based on dataflow coeffects [19], that calculates how many past values must the context provide. This enables an efficient implementation by caching previously computed values that are needed in the future.

In this causal dataflow language [25], the **prev** e syntax gets the previous value of an expression e. In the following example, counter generates positive integers and tick flips between 0 and 1:

$$\begin{array}{l}(\ \textbf{if}\ \ (\textbf{prev}\ \mathsf{tick}) = 0\ \textbf{then}\ (\lambda x \to \textbf{prev}\ x) \\ \quad \textbf{else}\ (\lambda x \to x)\ )\quad (\textbf{prev}\ \mathsf{counter})\end{array}$$

The function on the left-hand side returns the previous value of the argument every other step. Applied to a stream $\langle \ldots, 4, 3, 2, 1 \rangle$ (where 1 is the "current" value) it yields the stream $\langle \ldots, 4, 4, 2, 2 \rangle$. To obtain the function, we need one past value from the context (for **prev** tick). The returned function needs either none or one past value (thus a subtyping rule is required to type it as requiring one past value).

The function is called with **prev** counter as an argument, meaning that the result is either the first or second past element. Given $\mathsf{counter} = \langle \ldots, 5, 4, 3, 2, 1 \rangle$, the argument is $\langle \ldots, 5, 4, 3, 2 \rangle$ and so the result is a stream $\langle \ldots, 5, 5, 3, 3 \rangle$.

How many past values does the whole expression need? The argument requires one past value and the function one additional (**prev** (**prev** counter)), adding to 2. To obtain the function, one past value is needed. Overall we need the maximum of what is needed to evaluate the function and what is needed by passing the argument to the function: $max(1, 2) = 2$. This is captured by (*app*) in Figure 2.

The (*fun*) rule again does not allow delaying of context requirements. If the body requires m past values, these have to be available both from the outer context and from the argument provided by the

$$\text{(let)}\ \frac{(\Gamma_1, v:\tau_1)\ @\ r \times s \vdash e_1 : \tau_2 \qquad \Gamma_2\ @\ t \vdash e_2 : \tau_1}{(\Gamma_1, \Gamma_2)\ @\ r \times max(s,t) \vdash \textbf{let}\ x = e_2\ \textbf{in}\ e_1 : \tau_2}$$

$$\text{(contr)}\ \frac{x:\tau, y:\tau\ @\ r \times s \vdash e : \tau'}{z:\tau\ @\ max(r,s) \vdash e[x \leftarrow y][x \leftarrow z] : \tau}$$

$$\text{(fun)}\ \frac{\Gamma, x:\tau_1\ @\ r \times s \vdash e : \tau_2}{\Gamma\ @\ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2} \qquad \text{(var)}\ \frac{}{x:\tau\ @\ 0 \vdash x : \tau}$$

Figure 3. Structural coeffect types for stencil computations

caller. Later, we make the rule more flexible to make it compatible with (*fun*) for implicit parameters that *splits* requirements.

Stencil programming. In the type system for dataflow, we associated single caching requirements with the entire context Γ. The system loses information about the specific variable whose past value is needed. As a result (*fun*) duplicates the requirements rather than splitting them between immediate and latent requirements.

The *structural coeffect calculus* solves this limitation by tracking requirements per variable. The structure of annotations mirror the structure of variables – a context $x:\tau, y:\tau$ is annotated with $r \times s$ where the annotations r and s belong to x and y, respectively.

As an example, consider stencil programming over arrays where the syntax a_n accesses an array a with relative offset n (for now, assume n is a non-negative constant). A judgment may state:

$$(a : \mathsf{Arr}, b : \mathsf{Arr})\ @\ (5 \times 10) \vdash a_5 + b_{10} : \mathbb{N}$$

The coeffect annotation 5×10 denotes that we need values at offsets at most 5 and 10 for a and b. If we substitute c for both a and b, we need to combine the two annotations:

$$(c : \mathsf{Arr})\ @\ max(5,10) \vdash c_5 + c_{10} : \mathsf{nat}$$

This step requires explicit *structural* rules. Any manipulation on the variable context must manipulate coeffect annotations accordingly. To substitute c for a and b we used the (*contr*) rule from Figure 3. Before going further, consider the typing of let binding:

$$\textbf{let}\ \mathsf{c} = (\textbf{if}\ \mathsf{random}()\ \textbf{then}\ \mathsf{a}\ \textbf{else}\ \mathsf{b})\ \textbf{in}\ a_7 + c_3$$

The expression defines a variable c which may be assigned either a or b. The variable a may be used directly (at offset 7) or indirectly via c. The expression assigned to c uses variables a and b directly (the (*var*) rule) so its context is $(a{:}\mathsf{Arr}, b{:}\mathsf{Arr})\ @\ 0 \times 0$. The typing context of the body is $(a{:}\mathsf{Arr}, c{:}\mathsf{Arr})\ @\ 7 \times 3$. The let-binding then has the following derivation:

$$\frac{(a : \mathsf{Arr}, c : \mathsf{Arr})\ @\ 7 \times 3 \vdash a_7 + c_3 : \mathsf{nat} \qquad (a : \mathsf{Arr}, b : \mathsf{Arr})\ @\ 0 \times 0 \vdash \textbf{if} \ldots \textbf{then} \ldots \textbf{else} \ldots : \mathsf{nat}}{(a{:}\mathsf{Arr}, (a{:}\mathsf{Arr}, b{:}\mathsf{Arr}))\ @\ 7 \times (3 + (0 \times 0)) \vdash (\ldots) : \mathsf{nat}}$$

The $+$ operation can be applied pointwise (it distributes over $\times$), to get $7 \times (3 \times 3)$. For the overall requirements, we need to merge the two occurrences of a, resulting in 7 and 3 for a and b, respectively.

Figure 3 shows the (*let*), which follows as a combination of (*app*) and (*fun*) shown later. This is possible because function abstraction can now associate context requirements of the *bound variable* with the function (latent coeffect) and remaining context requirements with the outer context (immediate coeffects).

Although the context structure may appear complicated, it can be simplified and the user is not exposed to it directly. For more information, see Section 4.5.

1.2 Pathways to coeffects

The problem of integrating context-dependent properties with programming languages can be approached from various directions.

$$\text{(exch)}\ \frac{\Gamma, x:\tau_1, y:\tau_2 \vdash e : \gamma}{\Gamma, y:\tau_2, x:\tau_1 \vdash e : \gamma} \qquad \text{(weak)}\ \frac{\Gamma, \Delta \vdash e : \gamma}{\Gamma, x:\tau_1, \Delta \vdash e : \gamma}$$

$$\text{(contr)}\ \frac{\Gamma, x:\tau_1, y:\tau_1, \Delta \vdash e : \gamma}{\Gamma, x:\tau_1, \Delta \vdash e[y \leftarrow x] : \gamma}$$

Figure 4. Exchange, weakening and contraction typing rules

To give additional background, we start with a brief overview of related approaches. An eager reader can return to this section later.

Effect systems. Effect systems [7, 10], introduced by Gifford and Lucassen track effectful operations of computations such as memory access or lock usage [5]. They are usually written as judgments of of the form $\Gamma \vdash e : \tau, \sigma$ associating effects σ with the result. As discussed earlier, lambda abstraction delays all effects:

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

To distinguish coeffects, we associate context-dependent properties with the context writing $\Gamma @ r \vdash e : \tau$. In contrast to effect systems, coeffects cannot be delayed. Lambda abstraction splits context requirements between *immediate* and *latent*.

Language semantics. Moggi models the semantics of effectful computations using monads [11]. A computation is given a denotation as a function $\tau_1 \to M\tau_2$ where M is monad, which provides composition of effectful computations. Wadler and Thiemann link effect systems with monads using annotated monadic types $\tau_1 \to M^\sigma \tau_2$ [28]. Finally, Atkey describes *parameterised monads* that carry information about effects such as $M^\sigma \tau$ [1].

Context-dependent computations require a different model. Uustalu and Vene use functions $C\tau_1 \to \tau_2$ where C is a *comonad* which encode the context [23]. Examples include dataflow [24], stencil computations [18], I/O and interoperability [8].

Petricek et al. introduce *indexed comonads* to capture information about contextual properties in the type of computations such as $C^r \tau_1 \to \tau_2$ [19]. We revisit and improve this model (Section 2) and extend it (Section 4), to allow tracking of per-variable information.

Language and meta-language. Moggi uses monads in two systems [11]. In the first system, a monad is used to model an effectful language itself. This means that the semantics of a language is given in terms of one specific monad. In the second system, monads are added as type constructors, together with language constructs corresponding to operations of a monad (*unit*, *bind*).

Considering contextual properties, Uustalu and Vene [23] follow the first approach (using a concrete comonad to model dataflow). Contextual-Modal Type Theory (CMTT) of Nanevski et al. [15] follows the latter approach, adding a comonad via the $\Box$ modality of modal S4 to the language. We focus on concrete context-dependent languages and so we follow the first approach.

Sub-structural and bunched types. Sub-structural type systems control how variables are used. For example, the linear type system introduced by Wadler guarantees that a variable is used at most once [26]. This is achieved by treating variable contexts as a *list* rather than *set*. Sub-structural type systems remove one of the *structural* rules in Figure 4 that manipulate the variable context (exchange, weakening and contraction). Relatedly, *bunched typing* treats variable contexts as *trees* with different kinds of nodes [16]. It allows structural rules only on certain types of nodes.

Typing rules for our structural coeffect calculus are based on structural rules. This allows us to keep variable structure synchronized with the structure of annotations. For example, exchange rule swaps variables and turns an annotation $r \times s$ to $s \times r$.

2. Flat coeffect calculus

This section revisits the flat coeffect calculus λ_{fc} [19], which provides a basis for the structural coeffect calculus that we introduce in Section 4. We make a number of improvements – we cover more standard language features and require additional laws which allows a more useful equational theory. The formulation is general, so as not to restrict the possible uses unnecessarily, but is specialised enough to provide useful properties.

The flat coeffect calculus maintains a single annotation of the contextual requirements associated with the variable context. It can be instantiated to capture implicit parameters, the system for dataflow and simple liveness analysis.

The expressions are those of the λ-calculus with *let* binding; assuming T ranges over base types, types are extended as follows:

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\ \tau & ::= & T \mid \tau_1 @ r \to \tau_2 \end{array}$$

where T ranges over type constants. We discuss pairs and recursion in Sections 2.4 and 4.5. The type $\tau_1 @ r \to \tau_2$ represents a function from τ_1 to τ_2 that requires additional context r. It can be viewed as a pure function that takes τ_1 *with* context r. Note that $\tau @ r$ does not exist as syntactic value (although it has a semantic meaning discussed in Section 3). The annotations r are formed by an algebraic structure discussed next.

2.1 Flat coeffect algebra

The typing rules for implicit parameters in Section 1.1 used set union to split and combine contextual requirements. However, dataflow showed that we need more structure. It used three different operations ($+$, *min* and *max*). We generalize the examples using the following structure (operations are allowed to coincide).

Definition 1. *A* flat coeffect algebra $(S, \oplus, \wedge, \vee, 0, \bot)$ *is a set* S *with elements* $0, \bot$ *and operations* $\oplus, \wedge, \vee$ *such that* $\forall r, s, t \in S$*:*

$(S, \oplus, 0)$ *is a monoid*

$$r \oplus (s \oplus t) = (r \oplus s) \oplus t \qquad 0 \oplus r = r = r \oplus 0$$

$(S, \vee, \bot)$ *is a bounded semilattice and* $\wedge$ *is an associative and idempotent binary operation*

$$\begin{array}{lll} r = r \vee \bot & r \wedge (s \wedge t) = (r \wedge s) \wedge t & r \wedge r = r \\ r \vee s = s \vee r & r \vee (s \vee t) = (r \vee s) \vee t & r \vee r = r \end{array}$$

with a distributivity law $(r \vee s) \oplus t = (r \oplus t) \vee (s \oplus t)$
and a partial order $r \leq s \iff r \vee s = s$

The operation $\oplus$ represents *sequential composition* (e.g., passing arguments to a function). It forms a monoid with 0 which represents variable access. The monoid guarantees a category structure for the semantics in Section 3.

Dataflow computations use *max* to denote that the *combined* context requirements of an expression with multiple sub-expressions (an application) require the maximum of context requirements of the sub-expressions (dually, when the context is provided during a call, it is *split* between the two sub-expressions). This is the meaning of $\vee$. The $\wedge$ operation is the opposite of $\vee$. It represents *splitting* of context requirements (or, dually, *combining* of *provided* context). This is needed to provide additional flexibility in lambda abstraction, which is utilized by implicit parameters.

The operation $\vee$ forms a semilattice in order to provide a well defined partial order $\leq$. Similarly, associativity and idempotence for $\wedge$ guarantees reasonable properties for functions (Section 2.4).

2.2 Flat coeffect types

The typing rules in Figure 5 follow the earlier examples and are written as judgements $\Gamma @ r \vdash e : \tau$. Variable access and constants

$$\text{(app)}\ \frac{\Gamma @ r \vdash e_1 : \tau_1 @ s \to \tau_2 \qquad \Gamma @ t \vdash e_2 : \tau_1}{\Gamma @ r \vee (t \oplus s) \vdash e_1\ e_2 : \tau_2}$$

$$\text{(fun)}\ \frac{(\Gamma, x : \tau_1) @ r \wedge s \vdash e : \tau_2}{\Gamma @ r \vdash \lambda x.e : \tau_1 @ s \to \tau_2} \qquad \text{(var)}\ \frac{x : \tau \in \Gamma}{\Gamma @ 0 \vdash x : \tau}$$

$$\text{(let)}\ \frac{\Gamma @ s \vdash e_1 : \tau_1 \qquad (\Gamma, x : \tau_1) @ r \vdash e_2 : \tau_2}{\Gamma @ r \vee (s \oplus r) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$\text{(sub)}\ \frac{\Gamma @ s \vdash e : \tau}{\Gamma @ r \vdash e : \tau}\ (s \leq r) \qquad \text{(const)}\ \frac{\iota \in \mathbb{N}}{\Gamma @ \bot \vdash \iota : \mathbb{N}}$$

Figure 5. Type system for the flat coeffect language λ_{fc}

introduce special context requirements 0 and $\bot$, respectively. The sub-coeffecting rule (*sub*) directly uses the partial order $\leq$.

The (*app*) rule has the same structure as the rule for dataflow in Figure 2. This is also compatible with (*app*) for resources when we take both $\oplus$ and $\vee$ to be $\cup$. The (*fun*) rule splits the context requirements of the body ($r \wedge s$) between r associated with the declaration site and s associated with the call site (via the context-dependent function type $\tau_1 @ s \to \tau_2$). The next section shows that this is more general than the previous rule for dataflow in Section 1.1, which duplicated the context requirements of the body.

Finally, the typing of the (*let*) rule is a special case of a type and coeffect derivation for the expression $(\lambda x.e_1)\ e_2$ relying on the fact that $\wedge$ is idempotent[1] This gives more intuitive coeffects in case where the additional flexibility of lambda abstraction is not needed, because all context requirements need to be satisfied by the current scope. The annotation $r \vee (s \oplus r)$ can be simplified for all our concrete examples of coeffects.

2.3 Examples of flat coeffects

The above rules are shared by all flat coeffect systems. A concrete system needs to provide an instance of flat coeffect algebra $(S, \oplus, \wedge, \vee, 0, \bot)$ and rules for any additional language constructs.

Example 1 (Implicit parameters)**.** For implicit parameters, all three operations of the flat coeffect algebra coincide. Accessing a variable or constant creates no context requirements. More formally, given a set of names $\mathcal{N}$ and primitive types $\mathcal{T}$, the flat coeffect algebra for implicit parameters is formed by $(\mathcal{P}(\mathcal{N} \times \mathcal{T}), \cup, \cup, \cup, \emptyset, \emptyset)$. The ordering $\leq$ is subset ordering $\subseteq$. This leads to the (*app*) rule that unions requirements of both sub-expressions and the function and (*fun*) rule that (freely) splits required implicit parameters. Context requirements are introduced by the (*impl*) rule in Figure 1) that provides typing for the ?param expression.

Example 2 (Dataflow)**.** The flat coeffect algebra for dataflow computations is formed by $(\mathbb{N}, +, min, max, 0, 0)$ with $\leq$ being the standard ordering on $\mathbb{N}$. The numbers represent the number of required past values; 0 means that only current value is needed. For simplicity, we annotate constants with 0 too, although we can be more precise by combining the system with liveness (below).

The general (*fun*) rule differs from the one shown earlier. The body requires $min(r, s)$ past elements, meaning that both declaration-site and call-site need to provide at least the elements required by body (but can provide more). For dataflow, the earlier rule (that duplicates requirements) is *more precise*, but this is not true for all coeffect systems in general. Finally, in the (*let*) rule, the annotation $max(r, r + s)$ becomes just $r + s$.

[1] This design is similar to the ML value restriction, but it works the other way round. Our let binding is more restrictive rather than more general.

Example 3 (Liveness)**.** For liveness, the context is annotated with D when it is *dead* (no variables are accessed) and L if it is *live* (some variables may be used). This example distinguishes between constants, annotated as D, and variables annotated as L.

The flat coeffect algebra is formed by $(\{\mathsf{L},\mathsf{D}\},\sqcup,\sqcap,\sqcup,\mathsf{L},\mathsf{D})$ with $\mathsf{D} \leq \mathsf{L}$ where $\sqcup$ is disjunction (or) and $\sqcap$ is conjunction (and), where D is treated as false and L as true.

The (*fun*) rule follows similar reasoning as for dataflow. If the body is live, both declaration site and call site are marked as live; if the body is dead, they can (but do not have to) be marked as dead. The (*app*) rule can be best understood in terms of its semantics (Section 3). Briefly, the expression $e_1\ e_2$ requires the context if:

- the expression e_1 requires context (to produce a function) *or*
- the function uses its argument (has a type $\tau_1@\mathsf{L} \to \tau_2$) *and* the context is needed to evaluate e_2 (if the function does not need argument, e_2 is not evaluated; if e_2 does not need the context, it can be evaluated and the function called without the context)

Flat system for liveness tracks liveness of the *entire* variable context. It is interesting for semantic reasons (Section 3.2), but has a limited use. This is solved by the structural variant (Section 4.3).

2.4 Syntactic properties and extensions

The notion of context (just like the notion of effect) differs for every context-dependent language. However, a number of syntactic properties hold for all coeffect systems or for coeffect systems satisfying certain additional conditions[2].

Properties of lambda abstraction. Recall that $(S,\wedge)$ is a semilattice. The idempotence means that the context required by function body can be required from both declaration site and call site:

$$\text{(idfun)}\ \frac{(\Gamma, x:\tau_1)@r \wedge r \vdash e : \tau_2}{\Gamma@r \vdash \lambda x.e : \tau_1@r \to \tau_2}$$

It is easy to check that using this derivation in the typing of $(\lambda x.e_1)\ e_2$ gives the let binding rule shown in Figure 5.

The derivation is always valid, but it is too restrictive for implicit parameters. For coeffects where $r \wedge s \leq r$ (liveness and dataflow) the (*idfun*) rule could be used instead of (*fun*) removing the non-determinism of context requirement splitting. If we include subtyping rule (*typ*) that allows treating $\tau_1@r \to \tau_2$ as $\tau_1@s \to \tau_2$ whenever $r \leq s$, then the original (*fun*) is derivable as follows:

$$\begin{array}{rl}\text{(idfun)} & \dfrac{(\Gamma, x:\tau_1)@r \wedge s \vdash e : \tau_2}{\Gamma@r \wedge s \vdash \lambda x.e : \tau_1@r \wedge s \to \tau_2}\ , (r \leq r \wedge s) \\ \text{(sub)} & \\ \text{(typ)} & \dfrac{\Gamma@r \vdash \lambda x.e : \tau_1@r \wedge s \to \tau_2}{\Gamma@r \vdash \lambda x.e : \tau_1@s \to \tau_2}\ (r \leq r \wedge s)\end{array}$$

The $\wedge$ operation is associative, but it is not required to be symmetric ($r \wedge s = s \wedge r$). For many systems, this is the case, guaranteeing the following. Assuming that r', s', t' is a permutation of r, s, t:

$$\frac{(\Gamma, x:\tau_1, y:\tau_2)@r \wedge s \wedge t \vdash e : \tau_3}{\Gamma@r' \vdash \lambda x.\lambda y.e : \tau_1@s' \to \tau_2@t' \to \tau_3}$$

Intuitively, this means that the context requirements of a function with multiple arguments can be split arbitrarily between the declaration site and (multiple) call sites. In other words, it does not matter how the context requirements are satisfied.

Properties of constants and pairs. The flat coeffect calculus annotates constants with $\bot$ (which is needed, e.g. for liveness). To support pairs, we add the following rule for the constructor (e_1, e_2):

$$\text{(pair)}\ \frac{\Gamma@r \vdash e_1 : \tau_1 \qquad \Gamma@s \vdash e_2 : \tau_2}{\Gamma@r \vee s \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

[2] This section extends the §4 of [19] by considering properties of functions and pairs and revisits subject reduction by requiring more informative laws.

The operator $\vee$ combines context requirements of the two subexpressions. It models the case when the available context is split and passed to two independent expressions. The rule for projection is uninteresting, because π_i can be viewed as a pure function.

In a language with a constant $() : \mathsf{unit}$ and pairs, we would expect the following isomorphisms to hold:

$$(e_1,(e_2,e_3)) \simeq ((e_1,e_2),e_3) \quad (e_1,e_2) \simeq (e_2,e_1) \quad ((),e) \simeq e$$

The coeffect is not a part of the type of pairs, so the types are isomorphic. The isomorphism of the context requirements is guaranteed by the fact that $(S,\vee,\bot)$ is a bounded semilattice using associativity, symmetry and boundedness, respectively.

2.5 Syntactic reduction

The operational semantics for each concrete instance of the coeffect calculus may define evaluation differently, depending on the notion of context. In this section, we show that calculi satisfying certain additional laws (including liveness and implicit parameters) can use syntactic substitution as the basis of evaluation.

Call-by-value. In effectful languages, the call-by-value strategy substitutes pure *values* for variables. The reduction from effectful *computations* to values is specific to each concrete kind of effect.

The notion of *value* for context-dependent computations differs. A function $(\lambda x.e)$ is not necessarily a pure value, 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 ($\Gamma@0 \vdash e : \tau$). Assuming $e[x \leftarrow e']$ is the standard, capture-avoiding substitution:

Lemma 1 (Call-by-value substitution)**.** *If* $\Gamma@0 \vdash e' : \tau'$ *and* $\Gamma, x : \tau'@r \vdash e : \tau$ *then* $\Gamma@r \vdash e[x \leftarrow e'] : \tau$.

Proof. Both e' and x have coeffect annotation $\bot$. □

Lemma 1 holds for all coeffect calculi, but it is weak. To use it, the operational semantics must provide a way of capturing the context and (partially) evaluating any term with requirements $\Gamma@r$ to a term with no requirements $\Gamma@0$. Assuming e is a value if $\Gamma@0 \vdash e : \tau$ and $\to_{\mathsf{cbv}}$ is the corresponding call-by-value reduction:

Theorem 1 (Call-by-value reduction)**.** *In a coeffect system where* $r \wedge s \leq r \vee s$*, if* $\Gamma@r \vdash e : \tau$ *and* $e \to_{\mathsf{cbv}} e'$ *then* $\Gamma@r \vdash e' : \tau$.

Proof. A direct consequence of Lemma 1; the condition that $r \wedge s \leq r \vee s$ is required when $(\lambda x.e_1)\ e_2 \to_{\mathsf{cbv}} e_2[x \leftarrow e_1]$. □

Call-by-name. The call-by-name strategy is not sound for languages with effect typing. For example, a function $\lambda x.y$ has a type with empty effects, but substituting an effectful computation for y would change the type of the function. In contrast, coeffect calculi that satisfy certain additional conditions can use the call-by-name strategy, which provides another pathway to operational semantics.

The call-by-name strategy reduces a term $(\lambda e_1)\ e_2$ where both sub-expressions require some context. Assume that $\Gamma@s \vdash e_2 : \tau_2$ and $\Gamma, x : \tau_1@r \vdash e_1 : \tau_1$. The context requirements of the reduced term $e_1[x \leftarrow e_2]$ should be the same as the context requirements of **let** $x = e_2$ **in** e_1. Thus the context requirements of the result in the substitution lemma should be $r \vee (s \oplus r)$.

We call a flat coeffect algebra *top pointed* if 0 is the top element of the lattice $(S,\vee,\bot)$ and *bottom pointed* if 0 is the bottom element ($0 = \bot$). Liveness analysis is an example of top pointed coeffect system (variables are annotated with L and $\mathsf{D} \leq \mathsf{L}$).

Lemma 2 (Call-by-name substitution)**.** *For a top pointed coeffect system where* $\forall r, s.\ r \leq r \vee (s \oplus r)$ *if* $\Gamma@s \vdash e' : \tau'$ *and* $\Gamma, x : \tau'@r \vdash e : \tau$ *then* $\Gamma@r \vee (s \oplus r) \vdash e[x \leftarrow e'] : \tau$.

Proof. By rule induction over $\vdash$ □

As variables are annotated with the top element 0, we can substitute a term e' for any variable and use sub-coeffecting to get the original typing (because $s \leq 0$). The additional condition, which holds for all our examples, allows treating the resulting expression with requirements r as an expression with requirements $r \vee (s \oplus r)$.

In a bottom pointed coeffect system, substituting e for x increases the context requirements. However, if the system satisfies the strong condition that $\wedge = \oplus$ 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 $\wedge = \oplus$ holds for our implicit parameters example and allows the following substitution lemma:

Lemma 3 (Call-by-name substitution)**.** *For a coeffect calculus with bottom pointed flat coeffect algebra where also $\vee = \oplus$, if $\Gamma @ s \vdash e' : \tau'$ and $\Gamma, x : \tau' @ r \vdash e : \tau$ then $\Gamma @ r \vdash e[x \leftarrow e'] : \tau$.*

Proof. By rule induction over $\vdash$ □

The two substitution lemmas show that the call-by-name evaluation strategy can easily be used for certain coeffect calculi. In particular, we could use it to give operational semantics for liveness and implicit parameters. Assuming $\rightarrow_{\mathsf{cbn}}$ is the standard call-by-name reduction, the following subject reduction theorem holds:

Theorem 2 (Call-by-name reduction)**.** *In a coeffect system that satisfies the conditions for Lemma 2 or Lemma 3, if $\Gamma @ r \vdash e : \tau$ and $e \rightarrow_{\mathsf{cbn}} e'$ then $\Gamma @ r \vdash e' : \tau$.*

Proof. A direct consequence of Lemma 2 or Lemma 3. □

3. Semantics of flat coeffects

Petricek *et al.* introduced *indexed comonads* to structure the semantics of the flat coeffect calculus. We recall the definitions and semantics here, giving further detail and examples (they give only the indexed option comonad, shown in Section 3.2). This forms the basis of the structural coeffect calculus semantics (Section 5).

3.1 Comonads

Comonads structure the semantics of various context-dependent computations [17, 23]. The essential insight is that contextual computations can be modelled by functions (or, more generally, morphisms) of type $\mathsf{D}A \rightarrow B$. Here D encodes additional context over A values which is passed into the function. The comonad structure provides composition for functions of this type.

Definition 2. *A* comonad *on a category $\mathcal{C}$, comprises an object mapping $\mathsf{D} : |\mathcal{C}| \rightarrow |\mathcal{C}|$ and two operations: the* unit *natural transformation $\varepsilon_A : \mathsf{D}A \rightarrow A$ and the* extension *operation which maps a morphism $k : \mathsf{D}A \rightarrow B$ to $k^\dagger : \mathsf{D}A \rightarrow \mathsf{D}B$, satisfying the coherence conditions: $\varepsilon^\dagger = id$, $\varepsilon \circ f^\dagger = f$ and $g^\dagger \circ f^\dagger = (g \circ f^\dagger)^\dagger$.*

A composition operation $\hat{\circ}$ for all $f : \mathsf{D}A \rightarrow B$ and $g : \mathsf{D}B \rightarrow C$ is then defined: $g \,\hat{\circ}\, f = g \circ f^\dagger$. Following from the comonad axioms, this composition is associative and has ε as the identity.

Example (Product comonad)**.** The object mapping $\mathsf{D}A = A \times X$ for some fixed X is a comonad with operations:

$$\varepsilon(a, x) = a \qquad f^\dagger(a, x) = (f(a, x), x)$$

This comonad can be used to model an immutable global environment available throughout a computation, where ε ignores the environment and $f^\dagger$ passes the incoming environment to f and also returns it with the result of f to pass to the next computation. Linked to the previous example of the implicit parameters, a product comonad might be seen as providing the semantics for a set of implicit parameters that is uniform throughout a computation, with no account of the specific requirements of each subcomputation.

Contextual λ-calculi semantics A categorical semantics for a (typed) contextual calculus can be given denotations $[\![\Gamma \vdash e : \tau]\!] : \mathsf{D}[\![\Gamma]\!] \rightarrow [\![\tau]\!] \in \mathcal{C}$, for a comonad D, and with some suitable interpretation for Γ (usually products). For a contextual λ-calculus such a semantics requires the additional structure of a *monoidal comonad* [23]. The additional structure provides the operation of a *lax monoidal functor* $\mathsf{m}_{A,B} : \mathsf{D}A \times \mathsf{D}B \rightarrow \mathsf{D}(A \times B)$ for "merging" two contexts in the semantics of abstraction. For the product comonad $\mathsf{D}A = A \times X$, a definition for $\mathsf{m}_{A,B}$ requires X to have binary operation $\oplus$ so that: $\mathsf{m}((a, x), (b, y)) = ((a, b), x \oplus y)$. This gives a semantics for abstraction where the context available to a function body is the combination of that available at the declaration site and call site. We discuss this additional structure further in the next subsection for the indexed semantics of flat coeffects.

Non-uniform context propagation The product comonad provides a semantics for a calculus with an implicit, global environment, which may consist of some set of global parameters. It is likely that the subcomputations of programs in this calculus will not all require the full global environment, but will each have a more precise requirement.

Consider two denotations $f : A \times X \rightarrow B$ and a $g : B \times Y \rightarrow C$ which require an X and Y context. We can imagine a composition $\hat{\circ}$ which composes f and g whilst also merging their requirements such that $g \,\hat{\circ}\, f : A \times (X \times Y) \rightarrow C$. This composition does not however arise from a comonad, where the context is uniform throughout, but is instead provided by the structure of an *indexed comonad* [19].

3.2 Indexed comonads

Indexed comonads generalise comonads over an indexed family of endofunctors (which can be written itself as a functor $\mathsf{D} : \mathcal{I} \rightarrow [\mathcal{C}, \mathcal{C}]$ between a category of indices $\mathcal{I}$ and a category of endofunctors). We will write indices to D as superscripts, *e.g.*, D^r.

Definition 3. *For a monoid $(I, \oplus, 0)$, an* indexed comonad *comprises an indexed family of endofunctors D with natural transformation $\varepsilon_0 : \mathsf{D}^0 A \rightarrow A$ and an indexed family of mappings $(-)^\dagger_{r,s}$ mapping from $f : \mathsf{D}^s A \rightarrow B$ to $f^\dagger_{r,s} : \mathsf{D}^{r \oplus s} A \rightarrow \mathsf{D}^r B$ satisfying analogous laws to a comonad: ${\varepsilon_0}^\dagger{}_{0,0} = id$, $\varepsilon_0 \circ f^\dagger_{s,0} = f$ and $g^\dagger_{t,s} \circ f^\dagger_{t \oplus s,r} = (g \circ f^\dagger_{s,r})^\dagger_{t,s \oplus r}$ [19].*

(note, where we previously indexed natural transformations with their parameters, we omit these to avoid cluttering the indices).
Analogously to comonads, indexed comonads define a notion of composition, where for all $f : \mathsf{D}^s A \rightarrow B$ and $g : \mathsf{D}^r B \rightarrow C$, then $g \,\hat{\circ}\, f = g \circ f^\dagger_{r,s} : \mathsf{D}^{r \oplus s} A \rightarrow C$. By the indexed comonad axioms, this composition is associative and has ε_0 as the identity.

Example 1 (Indexed product)**.** For the monoid $(\mathcal{P}(X), \cup, \emptyset)$, there is an indexed product comonad $\mathsf{D}^X A = A \times X$ with:

$$\begin{array}{lll} \varepsilon_0 & (a, \emptyset) & = a \\ f^\dagger_{r,s} & (a, x) & = (f(a, x \backslash (r \backslash s)), x \backslash (s \backslash r)) \end{array}$$

Given some denotation $f : \mathsf{D}^s A \rightarrow B$ requiring the set s in its context, the incoming context of $r \cup s$ is divided into the s subset (by $x \backslash (r \backslash s)$) which is passed to f and the r subset (by $x \backslash (s \backslash r)$) which is returned along with the result of f. The indices represent the contextual requirements of the computation in its denotation.

Example 2 (Indexed option)**.** [19] The *indexed option comonad* is defined for a monoid $(\{\mathsf{D}, \mathsf{L}\}, \sqcap, \mathsf{L})$ where $\sqcap$ is conjunction (see Section 2.3) with $\mathsf{D}^{\mathsf{L}} A = A$ and $\mathsf{D}^{\mathsf{D}} A = 1$ (where 1 is the unit type inhabited by $()$). The operations are defined:

$$\varepsilon_{\mathsf{L}} A = A \qquad f^\dagger_{r,\mathsf{L}} x = f\, x \qquad f^\dagger_{r,\mathsf{D}} x = ()$$

Combinators in the semantics:

(clone)	$\Delta : A \to A \times A$	
(curry)	$\Lambda f : A \to (B \to C)$	$\forall f : (A \times B) \to C$
(uncurry)	$\Lambda^{-1} f : (A \times B) \to C$	$\forall f : A \to (B \to C)$
(pair)	$\langle f, g \rangle : A \to (B \times C)$	$\forall f : A \to B, g : A \to C$
($\times$)	$f \times g : A \times X \to B \times Y$	$\forall f : A \to B, g : X \to Y$

$$[\![T]\!] = T \qquad [\![\sigma @ r \to \tau]\!] = \mathsf{D}^r [\![\sigma]\!] \to [\![\tau]\!]$$

$$\text{(app)}\frac{\begin{array}{c}[\![\Gamma @ r \vdash e_1 : \sigma @ s \to \tau]\!] = k_1 : \mathsf{D}^r \Gamma \to (\mathsf{D}^s \sigma \to \tau) \\ [\![\Gamma @ t \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}^t \Gamma \to \sigma\end{array}}{\begin{array}{c}[\![\Gamma @ r \vee (s \oplus t) \vdash e_1\, e_2 : \tau]\!] = \\ (\Lambda^{-1}\ k_1) \circ (id \times k_2^\dagger) \circ \mathsf{s}_{r,s \oplus t} \circ \mathsf{D}\Delta : \mathsf{D}^{r \vee (s \oplus t)} \Gamma \to \tau\end{array}}$$

$$\frac{[\![\Gamma @ r \wedge s, x : \sigma \vdash e : \tau]\!] = k : \mathsf{D}^{r \wedge s}(\Gamma \times \sigma) \to \tau}{[\![\Gamma @ r \vdash \lambda x.e : \sigma @ s \to \tau]\!] = \Lambda\,(k \circ \mathsf{m}_{r,s}) : \mathsf{D}^r \Gamma \to (\mathsf{D}^s \sigma \to \tau)}$$

$$\text{(var)}\frac{v : \tau \in \Gamma}{[\![\Gamma @ 0 \vdash v : \tau]\!] = \pi_i \circ \varepsilon_0 : \mathsf{D}^0 \Gamma \to \tau}$$

$$\text{(sub)}\ \frac{[\![\Gamma @ s \vdash e : \tau]\!] = k : \mathsf{D}^s \Gamma \to \tau}{[\![\Gamma @ r \vdash e : \tau]\!] = k \circ \iota_{r,s} : \mathsf{D}^r \Gamma \to \tau}\ (s \leq r)$$

Figure 6. Categorical semantics of the flat coeffect calculus

This indexed comonad models the semantics of the liveness coeffect system, where $\mathsf{D}^{\mathsf{L}} A$ models a live context and D^{D} a dead context, which does not contain a value. The operation $(-)^\dagger_{r,s}$ can be seen as an implementation of dead code elimination. It only uses x if f requires it and it only calls f if the computation is composed with another whose context is live.

Here, D^r is analogous to a data type $\mathsf{Maybe}\ A = A + 1$. Note, $\mathsf{Maybe}\ A$ cannot be a comonad as $\varepsilon\ ()$ is undefined. This is not an issue for indexed comonads where ε is defined only for $\mathsf{D}^{\mathsf{L}} A = A$.

Indexed monoidal comonads Some additional structure to indexed comonads is required to give the semantics of abstract and application, similar to the notion of a monoidal comonad for a (non-indexed) comonadic semantics. *Indexed lax monoidal* and *colax monoidal* natural transformations are required for merging and splitting contexts, where indices are composed with binary operations $\vee$ and $\wedge$ (which will be taken from a coeffect algebra):

$$\mathsf{m}_{r,s} : \mathsf{D}^r A \times \mathsf{D}^s B \to \mathsf{D}^{r \wedge s} A \times B$$
$$\mathsf{s}_{r,s} : \mathsf{D}^{r \vee s} A \times B \to \mathsf{D}^r A \times \mathsf{D}^s B$$

The $\mathsf{m}_{r,s}$ operation gives the semantics of abstraction where contexts are merged (see (*abs*), Figure 6). On indices, $\wedge$ indicates that some information from the parameters may be lost (by taking the greatest-lower bound of the two incoming contexts).

For the semantics of application (see (*app*), Figure 6), the dual operation $\mathsf{s}_{r,s}$ splits a context between two subterms. Depending on the particular notion of context-dependence that is being modelled, this may be a left-, right-, or full-inverse of $\mathsf{m}_{r,s}$ if $\wedge = \vee$, (or for $r = s$) giving different equational theories.

3.3 Semantics

Figure 6 gives the semantics of the flat coeffect calculus with coeffect algebra $(S, \oplus, \wedge, \vee, 0, \bot)$ in terms of an indexed monoidal comonad, with denotations $[\![\Gamma @ r \vdash e : \tau]\!] : \mathsf{D}^r [\![\Gamma]\!] \to [\![\tau]\!]$. The operations of the coeffect algebra are used to combine indices in the operations ($\oplus$ for $(-)^\dagger_{r,s}$, 0 for ε_0, $\vee$ for $\mathsf{s}_{r,S}$, and $\wedge$ for $\mathsf{m}_{r,s}$). Thus the semantics maps coeffect annotations map directly to indices in the denotation, as stated by the following soundness theorem.

Theorem 3. *(Soundness) If* $\Gamma @ r \vdash e : \tau$ *then* $[\![\Gamma @ r \vdash e : \tau]\!] : \mathsf{D}^r [\![\Gamma]\!] \to [\![\tau]\!]$

Proof. By induction over the definition of coeffect judgments. □

Sub-coeffecting The semantics of sub-coeffecting requires an *indexed comonad morphism* $\iota_{r,s} : \mathsf{D}^r A \to \mathsf{D}^s A$ where $s \leq r$ with respect to the coeffect lattice $(S, \vee)$. See (*sub*) in Figure 6.

3.4 Semantics for our example calculi

Indexed comonads generalise comonads. This means that any comonad can be turned into an indexed comonad, but not all indexed comonads are also comonads. Some existing comonads can be also annotated with non-trivial indices.

Lemma 4. *Any comonad is an indexed comonad with the flat coeffect algebra* $(\{1\}, *, *, *, 1, 1)$ *that consists of a trivial singleton monoid where* $1 * 1 = 1$*, where* ε_0 *and* $(-)^\dagger_{r,s}$ *are the operations of a (non-indexed) comonad. A (semi) monoidal comonad [23] is required to obtain a non-indexed variant of the* m *operation. Finally,* $\mathsf{s}_{1,1} = \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle$*, and* $\iota_{1,1}$ *is the identity.*

Example 1 (Implicit parameters)**.** The semantics of the coeffect calculus for dynamically-scoped implicit parameters, with the coeffect algebra $(\mathcal{P}(\mathsf{Name}), \cup, \cup, \cup, \emptyset, \emptyset)$ is provided by the indexed comonad $\mathsf{D}^r A = A \times (r \to \tau)$ where $r \in \mathcal{P}(\mathsf{Name})$:

$$f^\dagger_{r,s}\ :\ \mathsf{D}^{r \cup s} A \to \mathsf{D}^s B \qquad \varepsilon : \mathsf{D}^\emptyset A \to A$$
$$f^\dagger_{r,s}(a, g) = (f(a, g|_r), g|_s) \qquad \varepsilon(a, \emptyset) = a$$

$$\mathsf{m}_{r,s} : \mathsf{D}^r A \times \mathsf{D}^s B \to \mathsf{D}^{(r \cup s)}(A \times B)$$
$$\mathsf{m}_{r,s}((a_1, g_1), (a_2, g_2)) = ((a_1, a_2), g_1|_{r \setminus s} \uplus g_2)$$

$$\mathsf{s}_{r,s} : \mathsf{D}^{(r \cup s)}(A \times B) \to \mathsf{D}^r A \times \mathsf{D}^s B$$
$$\mathsf{s}_{r,s}((a_1, a_2), g) = ((a_1, g|_r), (a_1, g|_s))$$

$$\iota_{r,s}\ :\ \mathsf{D}^r A \to \mathsf{D}^s A$$
$$\iota_{r,s}(a, g) = (a, g|_s) \quad (\text{if } \leq r)$$

In $f^\dagger_{r,s}$, the function g is defined on resource names $r \cup s$. It uses function restriction $g|_r$ and $g|_s$ to get functions that are defined on resources required by f and the result, respectively.

When merging contexts, $\mathsf{m}_{r,s}$ uses a disjoint union $\uplus$ to combine a function $g_1|_{r \setminus s}$ (defined for resources in r excluding those in s) and a function g_2 (defined for resources s). This definition is not symmetric as it prefers resources defined in the second context. As a result, the caller of a function can rebind parameter values.

Indexed comonads describe the context structure more precisely than comonads. A comonad $\mathsf{D}A = A \times (\mathsf{Name} \rightharpoonup r)$ needs to use a *partial* function on all names. Indexed comonads guarantees that all required implicit parameters are available.

Example 2 (Causal dataflow)**.** Uustalu and Vene [23] model causal dataflow computations using the non-empty list comonad $\mathsf{NEList}\ A = A \times (1 + \mathsf{NEList}\ A)$. Our refined coeffect model uses an indexed comonad $\mathsf{D}^n A = A \times \ldots \times A$ comprising the current value A (always available) followed by n past elements, together with a flat coeffect algebra $(\mathbb{N}, +, max, min, 0, 0)$:

$$\varepsilon\ :\ \mathsf{D}^0 A \to A \qquad \varepsilon(a_0) = a_0$$

$$f^\dagger_{m,n}\ :\ \mathsf{D}^{m+n} A \to \mathsf{D}^n B$$
$$f^\dagger_{m,n}\langle a_0, \ldots, a_{m+n} \rangle = (f\langle a_0, \ldots, a_m \rangle), \ldots, f\langle a_n, \ldots, a_{m+n} \rangle)$$

$$\mathsf{m}_{m,n} : \mathsf{D}^m A \times \mathsf{D}^n B \to \mathsf{D}^{min(m,n)}(A \times B)$$
$$\mathsf{m}_{m,n}(\langle a_0, \ldots, a_m \rangle, \langle b_0, \ldots, b_m \rangle) =$$
$$\langle (a_0, b_0), \ldots, (a_{min(m,n)}, b_{min(m,n)}) \rangle$$

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

$$\text{(var)}\ \frac{}{(x{:}\tau)@0 \vdash x : \tau} \qquad \text{(const)}\ \frac{\iota \in \mathbb{N}}{()@\bot \vdash \iota : \mathbb{N}}$$

$$\text{(fun)}\ \frac{(\Gamma, x{:}\tau_1)@r \times s \vdash e : \tau_2 \qquad x \notin \Gamma}{\Gamma@r \vdash \lambda x.e : \tau_1@s \to \tau_2}$$

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

$$\text{(let)}\ \frac{(\Gamma_1, x{:}\tau_1)@r \times s \vdash e_1 : \tau_2 \qquad \Gamma_2@t \vdash e_2 : \tau_1}{(\Gamma_1, \Gamma_2)@r \times (s \oplus t) \vdash \textbf{let}\ x = e_2\ \textbf{in}\ e_1 : \tau_2}$$

$$\text{(ctx)}\ \frac{\Gamma@r \vdash e : \tau \qquad \Gamma@r \rightsquigarrow \Gamma'@r'}{\Gamma'@r' \vdash e : \tau}$$

$\boxed{\Gamma@r \rightsquigarrow \Gamma'@r'}$

$$\text{(nest)}\ \frac{\Gamma_1@r \rightsquigarrow \Gamma_1'@r'}{(\Gamma_1, \Gamma_2)@r \times s \rightsquigarrow (\Gamma_1', \Gamma_2)@r' \times s}$$

$$\text{(swap)}\quad (\Gamma_1, \Gamma_2)@s \times r \rightsquigarrow (\Gamma_2, \Gamma_1)@r \times s$$

$$\text{(assoc)}\quad ((\Gamma_1, \Gamma_2), \Gamma_3)@(r \times s) \times t \rightsquigarrow (\Gamma_1, (\Gamma_2, \Gamma_3))@r \times (s \times t)$$

$$\text{(contr)}\quad (x{:}\tau, x{:}\tau)@r \times s \rightsquigarrow (x{:}\tau)@r \vee s$$

$$\text{(sub)}\quad \Gamma@s \rightsquigarrow \Gamma@r \quad (\text{when } s \le r)$$

$$\text{(empty)}\quad \Gamma@r \rightsquigarrow (\Gamma, ())@r \times \bot$$

$$\text{(weak)}\quad \Gamma@r \rightsquigarrow (\Gamma, x{:}\tau)@r \times \bot$$

Figure 7. Type/coeffect rules for the structural coeffect calculus

$$\mathsf{s}_{m,n} : \mathsf{D}^{max(m,n)}(A \times B) \to \mathsf{D}^m A \times \mathsf{D}^m B$$
$$\mathsf{s}_{m,n}\langle (a_0, b_0), \ldots, (a_{max(m,n)}, b_{max(m,n)})\rangle = \langle \langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_m\rangle\rangle$$

$$\iota_{m,n}\langle a_0, \ldots, a_m\rangle = \langle a_0, \ldots, a_n\rangle \qquad (\text{if } n \le m)$$

The reader is invited to check that the number of required past elements matches the indices. When combining two lists, $\mathsf{m}_{m,n}$ behaves as *zip* and produces a list that has the length of the shorter argument. When splitting a list, $\mathsf{s}_{m,n}$ needs the maximum of the required lengths. Finally, $(-)^{\dagger}_{m,n}$ needs $m + n$ elements in order to generate n past results of f, which itself requires m past values.

4. Structural coeffect calculus

The *flat* coeffect calculus λ_{fc} captures the essence of context dependence, but it has limited practical applicability. The analysis of liveness and dataflow, needs to track contextual information about individual variables rather than about the entire variable context.

In this section, we develop the *structural* coeffect calculus λ_{sc}. It provides practical liveness analysis, improves dataflow analysis (in particular, recursive definitions do not affect caching of the entire context, but only of a single variables) and can be used for other applications such as stencil computations outlined in Section 1.1.

The terms and types of the structural coeffect calculus are the same as for the flat variant, but the structure of indices differs.

4.1 Structural coeffect algebra

As outlined in Section 1.1, the structural coeffect calculus annotates multi-variable contexts, such as $x{:}\tau_1, y{:}\tau_2$, with an index that has a corresponding structure, such as $r \times s$.

The operation $\times$ replaces $\wedge$ in lambda abstraction. The difference is that $\times$ is not provided by a concrete instance of the calculus. Instead, it is a free structure that has only minimal interaction with other operators of the coeffect algebra. This change implies other changes in the type system that are explained later.

Definition 4. Structural coeffect algebra $(S, \oplus, \vee, 0, \bot)$ *is a set* S *with elements* $0, \bot$ *and binary* $\oplus, \vee$ *such that* $\forall r, s, t \in S$*:*

$(S, \oplus, 0)$ *is a monoid*

$$r \oplus (s \oplus t) = (r \oplus s) \oplus t \qquad 0 \oplus r = r = r \oplus 0$$

$(S, \vee, \bot)$ *is a bounded semilattice*

$$r \vee s = s \vee r \quad r \vee (s \vee t) = (r \vee s) \vee t \qquad r \vee r = r$$
$$\bot \vee r = r = r \vee \bot$$

$(S, \times, \bot)$ *is the free structure arising from the laws*

$$r \vee (s \times t) = (r \vee s) \times (r \vee t) \qquad \bot \times r = r$$
$$(s \times t) \vee r = (s \vee r) \times (t \vee r) \qquad r \times \bot = r$$

with a distributivity law $(r \vee s) \oplus t = (r \oplus t) \vee (s \oplus t)$
and a partial order $r \le s \iff r \vee s = s$

As previously, 0 and $\bot$ denote variable access and an unused context (e.g. constant), respectively; $\oplus$ forms a monoid with 0 to permit the indexed comonad structure in the semantics.

Rather than requiring an operation to split context requirements (e.g. *min* for dataflow or $\cup$ for implicit parameters) the $\times$ operation is used again. The tensor $\times$ interacts with the rest of the structure in only two ways – it has $\bot$ as unit and the $\vee$ operation distributes over it. Note that $\times$ is *not* associative or symmetric (the right distributivity is not strictly needed, but we include it for symmetry).

The operation $\vee$ is still required, because it is used when passing the same context to multiple sub-expressions that share a variable.

4.2 Structural coeffect system

Figure 7 gives the type and coeffect rules of the structural coeffect calculus. As hinted in Section 1.1, the key difference between the flat and structural calculi is that the type system of the structural includes explicit *structural* rules (abstracted as the $\rightsquigarrow$ relation in Figure 7). Unlike linear or affine typing, we do not omit some of the rules. Instead, we extend them to perform corresponding transformations on both Γ and the coeffect annotation. This way, the system maintains the association between variables and their contextual properties.

As in linear systems, the variable context Γ is not a simple set. We use a tree structure (similar to contexts in bunched typing [16]):

$$\Gamma ::= () \mid (x{:}\tau) \mid (\Gamma, \Gamma)$$

The syntax $()$ represents an empty context, so contexts are binary trees where leaves are either variables or empty contexts. For example, $((), (x, ()))@\bot \times (r \times \bot)$ is a context with two empty leaves annotated with $\bot$ and a variable x annotated with r.

The empty contexts are required in lambda abstraction and let binding where we need to split a context. Given a context $x{:}\tau$, we can first turn it into $((), x{:}\tau)$ and then split it into $()$ and $x{:}\tau$.

Standard rules. The rules in Figure 7 differ from the flat coeffect calculus rules in that sub-expressions have to use distinct parts of the variable context. (They can still share variables thanks to the structural *contraction* rule.)

The (*var*) rule annotates a leaf variable context with the unit coeffect 0 and (*const*) annotates an empty leaf context with $\bot$. The (*fun*) rule splits the context of the body into the bound variable and other (potentially empty) context. It attaches the coeffect associated with the bound variable to the resulting function type $\tau_1@s \to \tau_2$.

The (*app*) rule combines two parts of context Γ_1, and Γ_2. The variables in Γ_1 are only used in e_1, so the annotation of the left

branch remains r. The right branch is annotated with $s \oplus t$, because it is first used to evaluate e_1 (requiring t) and then the function (requiring s). Recall how this works for *let* binding in stencil computations in Section 1.1 – the requirement t needs to be propagated and combined with requirements of individual variables in Γ_2. Finally, the (*let*) rule in structural coeffects is derived from function abstraction and application (discussed further in Section 4.5).

Compared with flat coeffects, $\times$ replaces $\wedge$ in (*fun*) and $\vee$ in (*app*) (where contexts are merged and split). In the flat calculus, these operations lose information (in an intuitive, information-theoretical sense), but here, since $\times$ is a free structure, it preserves the information about contexts when they are merged/split.

Structural rules. The (*ctx*) rule is not syntax-directed and represents a transformation of the context structure. It uses a helper judgement $\mathsf{D}^{r_1}\Gamma_1 \rightsquigarrow \mathsf{D}^{r_2}\Gamma_2$ which states that the context $\mathsf{D}^{r_1}\Gamma_1$ can be transformed to a context $\mathsf{D}^{r_2}\Gamma_2$.

Thanks to (*nest*), the transformation can be applied to any part of the context. Note that it is sufficient to allow the transformation of the left part of the context; the right part can be transformed using (*swap*). Together with (*assoc*), these rules allow arbitrary reordering of context variables.

The (*contr*) rule is needed when a variable is used repeatedly. It merges the context requirements of the two variables using $\vee$. Note that the structural calculus has two notions of *merging* – one for repeated variables and one (invertible) for independent variables (for example, in application). The structural version of sub-coeffecting (*sub*) is applied to individual variables.

Finally, the rules (*empty*) and (*weak*) add an empty context and an unused variable, respectively. They are needed to type functions that use only their parameter (e.g., $\lambda x.x$) and functions that do not use their parameter (e.g., $\lambda x.y$). In both cases, the unused context is annotated with $\bot$. We demonstrate them in the next section.

4.3 Examples of structural coeffects

Flat coeffect algebra whose contextual information arise from variable use (such as liveness and dataflow) can be easily adapted to the structural setting. We adapt here liveness and dataflow, giving more precise coeffect analyses of these two calculi.

Example 1 (Structural liveness)**.** The structural coeffect algebra for liveness is formed by $(\{\mathsf{L}, \mathsf{D}\}, \sqcup, \sqcup, \mathsf{L}, \mathsf{D})$ where $r \sqcup s = \mathsf{L}$ if and only if $r = s = \mathsf{L}$. The $\sqcap$ operation is no longer needed.

The liveness analysis based on flat coeffects is very limited as it marks the context as dead only when all free variables are dead. For example, $\lambda x.y$ is marked as requiring its argument, even though it is actually dead. Structural coeffects remove this limitation:

$$\frac{x{:}\tau_1@\mathsf{L} \vdash x : \tau_1 \qquad x{:}\tau_1@\mathsf{L} \rightsquigarrow (x{:}\tau_1, y{:}\tau_2)@\mathsf{L} \times \mathsf{D}}{\dfrac{(x{:}\tau_1, y{:}\tau_2)@\mathsf{L} \times \mathsf{D} \vdash x : \tau_1}{x{:}\tau_1@\mathsf{L} \vdash \lambda y.x : \tau_2@\mathsf{D} \to \tau_1}}$$

The derivation uses (*weak*) to add an unused variable y (marked as $\bot = \mathsf{D}$) and then abstracts over it. Thanks to its structural nature, the calculus maintains that y is unused and thus the input of the resulting function is marked as dead.

Example 2 (Structural dataflow)**.** The structural coeffect calculus for dataflow is similar to liveness. It has coeffect algebra $(\mathbb{N}, +, max, 0, 0)$ where *max* is used only by the contraction rule (*contr*). The following example shows a coeffect/type derivation for **prev** $x + (x + y)$. The coeffect rule for **prev** (not shown) increments the coeffect associated with all variables in the context:

$$\dfrac{\dfrac{\dfrac{\dfrac{(x{:}\tau)@0 \vdash x : \tau}{(x{:}\tau)@1 \vdash \textbf{prev}\ x : \tau} \qquad \dfrac{(\ldots)}{(x{:}\tau, y{:}\tau)@0 \times 0 \vdash x + y : \tau}}{(x{:}\tau, (x{:}\tau, y{:}\tau))@1 \times (0 \times 0) \vdash (\textbf{prev}\ x) + (x + y) : \tau}}{((x{:}\tau, x{:}\tau), y{:}\tau)@(1 \times 0) \times 0) \vdash (\textbf{prev}\ x) + (x + y) : \tau}}{(x{:}\tau, y{:}\tau)@max(0, 1) \times 0 \vdash (\textbf{prev}\ x) + (x + y) : \tau}$$

For simplicity, we treat the $+$ expression as a primitive rather than function application. It follows the pattern of requiring two separate variable contexts for the sub-expression, so we obtain a single context with two occurrences of $x : \tau$. After using (*assoc*), they can be merged using (*contr*) and so the final judgement requires 1 past value for x and no past values for y.

4.4 Syntactic reduction

We start our discussion of syntactic properties of the structural coeffect calculus by discussing syntactic reduction. As previously, each concrete instance of the calculus needs to provide its own notion of context and evaluation.

The difference from flat coeffects is that structural coeffect calculi admit full β-reduction without additional requirements and so it can serve as a basis for operational semantics for all structural coeffect languages, allowing both call-by-value and call-by-name strategies. It also admits η-expansion, satisfying both *local soundness* and *local completeness* required by Pfenning and Davies [20].

Substitution and holes. The substitution lemma for structural coeffects replaces a variable (leaf) with a context (subtree) while also replacing the corresponding part of the coeffect annotation. More formally, we define contexts with a *hole*, written $\Delta[-]$ for a free-variable context and coeffect annotation with a corresponding hole in each:

$$\begin{array}{c}()@\bot \in \Delta[-] \\ (-)@- \in \Delta[-] \\ (x{:}\tau)@r \in \Delta[-]\end{array} \qquad \frac{\Gamma_i@r_i \in \Delta[-]}{(\Gamma_1, \Gamma_2)@r_1 \times r_2 \in \Delta[-]}$$

Assuming we have a context with hole $\Gamma@r \in \Delta[-]$, the hole filling operation $\Gamma[\Gamma']@r[r']$ fills the hole in the variable context with Γ' and the corresponding coeffect tag hole with r':

$$\begin{array}{rcl}()[\Gamma']@1[r'] & = & ()@1 \\ (-)[\Gamma']@-[r'] & = & (\Gamma')@r' \\ (x{:}\tau)[\Gamma']@r[r'] & = & (x{:}\tau)@r \\ (\Gamma_1, \Gamma_2)[\Gamma']@r_1 \times r_2[r'] & = & (\Gamma_1', \Gamma_2')@r_1' \times r_2' \\ & & \text{where } \Gamma_i'@r_i' = \Gamma_i[\Gamma']@r_i[r']\end{array}$$

The substitution lemma for structural coeffects uses holes to express that a single variable x is replaced with (a sub-tree) Γ' and an annotation s at a corresponding location is replaced with $s' \oplus s$:

Lemma 5 (Substitution lemma)**.** *If* $\Gamma[x : \tau]@r[s] \vdash e : \tau'$ *and* $\Gamma'@s' \vdash e' : \tau'$ *then* $\Gamma[\Gamma']@r[s \oplus s'] \vdash e[v \leftarrow e'] : \tau$.

Proof. By rule induction over $\vdash$ and $\rightsquigarrow$, using properties of the structural coeffect algebra $(S, \oplus, \vee, 0, \bot)$. The induction over $\rightsquigarrow$ requires a stronger induction hypothesis (with multiple holes) to handle the (*contr*) case. □

Reduction and expansion. As already mentioned, the structural coeffect calculus admits full β-reduction, without requiring any additional laws about the coeffect algebra. Assume that $\to_\beta$ reduces a redex $(\lambda x.e_1)\ e_2$ to $e_1[x \leftarrow e_2]$ anywhere in a term:

Theorem 4 (Subject reduction)**.** *In a structural coeffect calculus, if* $\Gamma@r \vdash e_1 : \tau$ *and* $e_1 \to_\beta e_2$ *then* $\Gamma@r \vdash e_2 : \tau$.

Proof. Direct consequence of Lemma 5. □

The η-expansion transforms a term f of a function type to a term $\lambda x.f\ x$. In structural coeffect system, it preserves the type and context requirements of the function, guaranteeing *local completeness* [20]. Assuming that $\rightarrow_\eta$ performs η-expansion anywhere in a term:

Theorem 5 (Subject expansion)**.** *In a structural coeffect calculus, if* $\Gamma@r \vdash e_1 : \tau$ *and* $e_1 \rightarrow_\eta e_2$ *then* $\Gamma@r \vdash e_2 : \tau$.

Proof. By induction, using the fact that if $\Gamma@r \vdash f : \tau_1@s \rightarrow \tau_2$ then $\Gamma@r \vdash \lambda x.fx : \tau_1@s \rightarrow \tau_2$, which is proved by:

$$\dfrac{\dfrac{\Gamma@r \vdash f : \tau_1@s \rightarrow \tau_2 \qquad (x : \tau_1)@0 \vdash x : \tau_1}{(\Gamma, x : \tau_1)@r \times (s \oplus 0) \vdash f\ x : \tau_2}}{\Gamma@r \vdash \lambda x.fx : \tau_1@s \rightarrow \tau_2}$$

using the property that $s \oplus 0 = s$ in the last step. □

Note that these proofs are all at the syntax-level (types/coeffects), rather than at the denotation-level. A companion report, providing the corresponding semantic proofs is in preparation.

4.5 Syntactic properties and extensions

In this section, we consider other important syntactic properties of the structural coeffect calculus and we briefly discuss extending the calculus with recursion.

Let *binding*. Flat coeffect calculus includes a restrictive form of *let* binding. In the structural variant, this is not needed. *Let* binding is equivalent to a function abstraction followed by application.

Lemma 6 (Definition of *let* binding)**.** $\Gamma@r \vdash (\lambda x.e_2)\ e_1 : \tau_2$ *holds if and only if* $\Gamma@r \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2$ *holds.*

Proof. The premises and conclusions of the following typing derivation for $(\lambda x.e_2)\ e_1$ correspond with the typing rule (*let*):

$$\dfrac{\dfrac{(\Gamma_1, x : \tau_1)@r \times s \vdash e_2 : \tau_2 \qquad v \notin \Gamma_1}{\Gamma_1@r \vdash \lambda x.e_2 : \tau_1@s \rightarrow \tau_2} \qquad \Gamma_2@t \vdash e_1 : \tau_1}{(\Gamma_1, \Gamma_2)@r \times (s \oplus t) \vdash (\lambda x.e_2)\ e_1 : \tau_2}$$ □

Normalization. For implementing languages based on structural coeffects, it is worth noting that the tree structure of contexts with coeffect annotation can be simplified. We present it using tree structure only because it simplifies the formalism.

The coeffect annotation can always be represented as a tree of $\times$ nodes with a structure that matches with the structure of the context. The $\oplus$ operator distributes over $\times$ and $\vee$ is only introduced in (*contr*), attached to a single variable. Thus, the tree structure can be stored as a flat list of variables.

This gives a way to efficiently represent the context. It also means that the context structure is not exposed to the developer. Typing errors can always be associated with individual variables.

Recursion. Another important practical concern is adding recursive computations. For space reasons we only provide a brief sketch. Assume the language includes a term **fix** e representing a fixed point of a function e. The typing rule requires a unary operation *fix* to the coeffect algebra. Then we can use the following rule for both flat and structural coeffect calculi:

$$\text{(fix)}\ \dfrac{\Gamma@r \vdash e : \tau@s \rightarrow \tau}{\Gamma@(\mathit{fix}(s) \oplus \mathit{fix}(r)) \vdash \textbf{fix}\ e : \tau}$$

The *fix* operation differs for each specific calculus. For dataflow it returns a special value ∞ if the argument is greater than zero.

For flat coeffects, this means that the entire context is marked as ∞ whenever (impure) recursion is used. In contrast, for structural coeffects, only variables that are actually used recursively are marked as ∞. This means that structural coeffect system allows a useful optimisation, even in the presence of recursion.

5. Semantics of structural coeffects

Similarly to the flat coeffect calculus, the semantics of the structural coeffect calculus is captured by an indexed comonad with an additional structure. In the structural variant, coeffect annotations match the structure of the free-variable context, and vice versa. To this end, free-variable contexts must be modelled by a structure distinct from ordinary products, which we denote $-\hat{\times}-$. The $\hat{\times}$ operator is a (bi)functor, but is not a product in the categorical sense. For example, it is not possible to transform $\mathsf{D}^{r\times s}(A\hat{\times}B)$ into $\mathsf{D}^{r\times s}A$ (which could be done using the functor D^r and projection π_1 for ordinary products). Consequently, the additional operations for merging and splitting contexts differ to those in the flat semantics.

5.1 Structural indexed monoidal comonads

Given a structural coeffect algebra $(S, \oplus, \vee, 0, \bot)$, the structural semantics requires an indexed comonad D^r with lax monoidal and colax monoidal operations between $\times$ and $\hat{\times}$:

$$\begin{aligned}\mathsf{m}_{r,s} &: \mathsf{D}^r A \times \mathsf{D}^s B \rightarrow \mathsf{D}^{(r\times s)}(A\hat{\times}B)\\ \mathsf{s}_{r,s} &: \mathsf{D}^{(r\times s)}(A\hat{\times}B) \rightarrow \mathsf{D}^r A \times \mathsf{D}^s B\end{aligned}$$

along with the natural transformation $\hat{\Delta}$ (used for contraction) and a family of mappings $\iota_{r,s}$ for all r, s such that $s \leq r$:

$$\begin{aligned}\hat{\Delta}_{r,s} &: \mathsf{D}^{r\vee s}A \rightarrow \mathsf{D}^r A \times \mathsf{D}^s A\\ \iota_{r,s} &: \mathsf{D}^r A \rightarrow \mathsf{D}^s A\end{aligned}$$

The type of the $\mathsf{m}_{r,s}$ operation differs from the flat version in two ways. Firstly, it combines tags using $\times$ instead of $\vee$, which corresponds to the fact that the variable context consists of two parts (a tree node). Secondly, the resulting context is modelled as $A\hat{\times}B$ instead of $A\times B$. That is, $\mathsf{m}_{r,s}$ and $\mathsf{s}_{r,s}$ provide homomorphism between usual products and $\hat{\times}$. Finally, the operation $\iota_{r,s}$ can be used to define contraction as $\hat{\Delta}_{r,s} = \langle \iota_{(r\vee s),r}, \iota_{(r\vee s),s}\rangle$, but we do not use this as default to better capture the intuition behind *splitting*.

5.2 Semantics

For clarity, we now assume the normalization discussed in the previous section and treat free-variable context as a list. Given $(x_1 : \tau_1, \ldots, x_n : \tau_n)@r_1 \times \ldots \times r_n \vdash e : \tau$, the denotation is:

$$[\![\Gamma\ @\ r_1 \times \ldots \times r_n \vdash e : \tau]\!] : \mathsf{D}^{(r_1\times\ldots\times r_n)}(\tau_1\hat{\times}\ldots\hat{\times}\tau_n) \rightarrow \tau$$

The semantics of the calculus is then defined by induction over coeffect judgments in Figure 8. Before looking at examples in Section 5.3, we finish our discussion of the semantics.

Standard rules Compared to the flat semantics (Figure 6), there are a number of notable differences. Firstly, the rule (*var*) is now interpreted as ϵ_0 without the need for projection π_i. When accessing a variable, the context contains only the accessed variable. The (*fun*) rule has the same structure, but combines coeffect annotations using $\times$ rather than $\vee$ (which is a result of the change of type signature in $\mathsf{m}_{r,s}$).

The rule (*app*) does not first duplicate the free-variable context via $\mathsf{D}\Delta$ as in the flat semantics, but instead uses the $\mathsf{s}_{s,(r\vee t)}$ operation straight away to split the context. This makes the system structural – expressions use disjunctive parts of the context – and also explains why the composed coeffect annotation is $s \times (r \vee t)$.

Structural rules The only non-syntax-directed rule from flat coeffects (*sub*) is now generalized to a number of rules that manipulate the context. Denotations $[\![\mathsf{D}^{r_2}\Gamma_2 \rightsquigarrow \mathsf{D}^{r_1}\Gamma_1]\!]$ are morphisms that transforms the context denotation $\mathsf{D}^{r_1}\Gamma_1$ to the denotation $\mathsf{D}^{r_2}\Gamma_2$. The direction may look reversed, but it is actually right – context transformation proceeds in the opposite order than the typing derivation. The semantics of (*ctx*) then becomes a simple composition of the transformation and original morphism k.

$$\text{(var)}\quad [\![v : \tau @ 0 \vdash v : \tau]\!] = \varepsilon_0 : \mathsf{D}^0\tau \to \tau$$

$$\text{(app)}\frac{[\![\Gamma_1 @ r \vdash e_1 : \sigma @ s \to \tau]\!] = k_1 : \mathsf{D}^r\Gamma_1 \to (\mathsf{D}^s\sigma \to \tau) \qquad [\![\Gamma_2 @ t \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}^t\Gamma_2 \to \sigma}{[\![\Gamma_1, \Gamma_2 @ r \times (s \oplus t) \vdash e_1\, e_2 : \tau]\!] = (\Lambda^{-1}\, k_1) \circ (id \times k_2^\dagger) \circ \mathsf{s}_{r,s\oplus t} : \mathsf{D}^{r\times(s\oplus t)}(\Gamma_1 \hat{\times} \Gamma_2) \to \tau}$$

$$\text{(fun)}\frac{[\![\Gamma @ r \times s, x : \sigma \vdash e : \tau]\!] = k : \mathsf{D}^{r\times s}(\Gamma \hat{\times} \sigma) \to \tau}{[\![\Gamma @ r \vdash \lambda x.e : \sigma @ s \to \tau]\!] = \Lambda\,(k \circ \mathsf{m}_{r,s}) : \mathsf{D}^r\Gamma \to (\mathsf{D}^s\sigma \to \tau)} \qquad \text{(ctxt)}\frac{[\![\Gamma @ r \vdash e : \tau]\!] = k : \mathsf{D}^r\Gamma \to \tau}{[\![\Gamma' @ r' \vdash e : \tau]\!] = k \circ [\![\Gamma @ r \rightsquigarrow \Gamma' @ r']\!] : \mathsf{D}^r\Gamma' \to \tau}$$

$$\boxed{[\![\Gamma' @ r' \rightsquigarrow \Gamma @ r]\!] : \mathsf{D}^{r'}\Gamma' \to \mathsf{D}^r\Gamma}$$

$$\begin{array}{lrcl}
\text{(nest)} & [\![(\Gamma_1, \Gamma_2) @ r \times s \rightsquigarrow (\Gamma_1', \Gamma_2) @ r' \times s]\!] & = & \mathsf{m}_{r,s} \circ ([\![\Gamma_1 @ r \rightsquigarrow \Gamma_1' @ r']\!] \times id) \circ \mathsf{s}_{r',s} \\
\text{(swap)} & [\![(\Gamma_2, \Gamma_1) @ s \times r \rightsquigarrow (\Gamma_1, \Gamma_2) @ r \times s]\!] & = & \mathsf{m}_{s,r} \circ \mathsf{swap} \circ \mathsf{s}_{r,s} \\
\text{(assoc)} & [\![((\Gamma_1, \Gamma_2), \Gamma_3) @ (r \times s) \times t \rightsquigarrow (\Gamma_1, (\Gamma_2, \Gamma_3)) @ r \times (s \times t)]\!] & = & \mathsf{m}_{r\times s,t} \circ (\mathsf{m}_{r,s} \times id) \circ \mathsf{assoc}_1 \circ (\mathsf{id} \times \mathsf{s}_{s,t}) \circ \mathsf{s}_{r,s\times t} \\
\text{(contr)} & [\![(x : \tau, x : \tau) @ r \times s \rightsquigarrow (x : \tau) @ r \vee s]\!] & = & \mathsf{m}_{r,s} \circ \hat{\Delta}_{r,s} \\
\text{(sub)} & [\![\Gamma @ s \rightsquigarrow \Gamma @ r]\!] & = & \iota_{r,s} \\
\text{(empty)} & [\![\Gamma @ r \rightsquigarrow (\Gamma, ()) @ r \times \bot]\!] & = & \pi_1 \circ \mathsf{s}_{r,1} \\
\text{(weak)} & [\![\Gamma @ r \rightsquigarrow (\Gamma, x : \tau) @ r \times \bot]\!] & = & \pi_1 \circ \mathsf{s}_{r,0}
\end{array}$$

where $\mathsf{assoc}_{A,B,C} : (A \times (B \times C)) \to ((A \times B) \times C)$ and $\mathsf{swap}_{A,B} : A \times B \to B \times A$

Figure 8. Categorical semantics for λ_{sc}

The context manipulation rules work as follows:

- The (*nest*), (*swap*) and (*assoc*) rules all use $\mathsf{s}_{r,s}$ to split the context into a product(s) of contexts, perform some operation on the product and finally reconstruct the context using $\mathsf{m}_{r,s}$.
- The semantics of (*contr*) needs to duplicate the variable value using $\hat{\times}$. This is done using the special form of duplication that splits coeffects $\hat{\Delta}$ and then merging the two newly produced contexts ($\mathsf{m}_{r,s}$ is the only operation that produces $\hat{\times}$). Note that we cannot simply duplicate the value under $\mathsf{D}^{r\vee s}$. That gives $\mathsf{D}^{r\vee s}(\tau \times \tau)$ which represents a single variable of a pair type.
- The (*sub*) rule interprets sub-coeffecting on a single-variable context using the natural transformation $\iota_{r,s}$.
- Finally, the (*empty*) and (*weak*) rules have the same semantics. They both split the context and discard one part (containing either an unused variable or an empty context).

5.3 Example structural indexed comonads

We give a concrete semantics of the structural coeffect calculus for dataflow and liveness by defining the relevant categorical structures. Assuming a, b are terms of type A and B, respectively, we write $[a, b]$ for a term of type $A \hat{\times} B$.

The semantics is simplified by two observations 1) for an object $D^r A$ there is a bijection between the $\times$ structure in r and $\hat{\times}$ in A (*i.e.*, they match) 2) objects constructed with $\hat{\times}$ exist only in *negative* positions in a morphism (directly to the left of an arrow), and do not appear directly on the right of an arrow.

As a result, the mapping $(\mathsf{D}^s A \to B) \to \mathsf{D}^{r\oplus s}A \to \mathsf{D}^r B$ never contains $\hat{\times}$ in B, and therefore the coeffect r never contains the $\times$ coeffect combinator. It should be noted then that there are denotations in our semantics that do not have a corresponding term in the structural coeffect calculus. Restricting the semantics to more closely match the term language is left as a future work.

Example 1 (Dataflow)**.** Recall that dataflow has structural coeffect algebra $(\mathbb{N}, +, max, 0, 0)$. The indexed object mapping D for indices which comprise $\hat{\times}$ is defined:

$$\mathsf{D}^{(r_1\times\ldots\times r_n)}(A_1 \hat{\times} \ldots \hat{\times} A_n) = (A_1 \times \underbrace{A_1 \times \ldots \times A_1}_{r_1}) \hat{\times} \ldots \hat{\times} (A_n \times \underbrace{A_n \times \ldots \times A_n}_{r_n})$$

For $n = 1$, this corresponds to the model of flat dataflow. This family of object mappings permits the following the indexed comonad structure:

$$\begin{array}{l}
\varepsilon_0\langle a_0\rangle = a_0 \\
f^\dagger_{r,s_1\ldots s_n}[\langle a_{0,0}, \ldots, a_{0,s_1+r}\rangle, \ldots, \langle a_{n,0}, \ldots, a_{n,s_n+r}\rangle] = \\
\quad \langle\, f[\langle a_{0,0}, \ldots, a_{0,s_1}\rangle, \ldots, \langle a_{n,0}, \ldots, a_{n,s_n}\rangle], \ldots, \\
\quad\; f[\langle a_{0,r}, \ldots, a_{0,(s_1+r)}\rangle, \ldots, \langle a_{n,r}, \ldots, a_{n,(s_n+r)}\rangle]\rangle
\end{array}$$

The ε_0 operation is the same as for the flat dataflow comonad (it is only ever called on single-variable contexts with a single value); for $(-)^\dagger_{r,s}$ the annotation r does not contain $\times$ and $s = s_1 \times \ldots \times s_n$. That is, we take n streams as input, with demands $s_1, \ldots, s_n$ for elements of each respectively. Since the output requires r elements, each input stream must have $s_1 + r, \ldots, s_n + r$ elements respectively to generate the r output elements.

The definitions for the merge and split operations are simple mappings between regular products $\times$ and $\hat{\times}$:

$$\mathsf{m}_{r,s}(a, b) = [a, b] \qquad \mathsf{s}_{r,s}[a, b] = (a, b)$$

Example 2 (Liveness)**.** Recall that liveness has structural coeffect algebra $(\{\mathsf{L}, \mathsf{D}\}, \sqcup, \sqcup, \mathsf{L}, \mathsf{D})$. We define here a *structural option* indexed comonad, The indexed object mapping D is defined similarly to the dataflow in terms of $\hat{\times}$, where we compose a number of option indexed comonad structures, written as D':

$$\mathsf{D}^{(r_1\times\ldots\times r_n)}(A_1 \hat{\times} \ldots \hat{\times} A_n) = \mathsf{D}'^{r_1}A_1 \hat{\times} \ldots \hat{\times} \mathsf{D}'^{r_n}A_n$$

The indexed comonad operations are defined as follows:

$$\begin{array}{l}
\varepsilon_{\mathsf{L}}\; x = x \\
f^\dagger_{\mathsf{D},s_1\ldots s_n}[x_1, \ldots, x_n] = () \\
f^\dagger_{\mathsf{L},s_1\ldots s_n}[x_1, \ldots, x_n] = f\,[x_1, \ldots, x_n]
\end{array}$$

Aside from the $\hat{\times}$ structures inside, this has exactly the same definition as for the usual option indexed comonad. Thus, if the outgoing

coeffect is dead, then the unit value is returned immediately, otherwise f is called with the incoming parameters (where $x_i = ()$ if $s_i = \mathsf{D}$). The merge and split operations are defined exactly as in the dataflow semantics.

6. Related and future work

Flat and structural coeffect systems fill the gap between (non-indexed) comonads of Uustalu and Vene [23] and effect systems with various forms of (indexed, or precise) semantics [1, 22, 28].

Linear and partial coeffects. The structural coeffect calculus is inspired by sub-structural type systems, but it does not subsume them. For example, affine systems discard the *contraction* rule which cannot be directly expressed in our system.

One approach is to allow *partial* operations in coeffect algebra. An affine system can then be obtained by defining $\vee$ only for $\bot \vee \bot$. Similarly, in flat coeffects, the $\wedge$ operation could be restricted to enforce certain splittings of contextual requirements. Allowing only $\bot \wedge r$ gives a variant of implicit parameters with purely dynamic scoping. Another approach would be to generalize coeffect algebras to a more general structure akin to the *effectors* of Tate [22].

Coeffect meta-language. We use (indexed) comonads to give the *semantics* of flat and structural calculi. Comonads are known to provide a model for the $\Box$ modality of S4 modal logic [3]). Previous work based on modal S4 [6, 13–15] uses the *meta-language* style, where $\Box$ is a type constructor and terms are extended with constructs embedding axioms of the logic (comonad operations).

In contrast, we build a calculi that models specific context-dependent languages or features (e.g., Lucid [25], implicit parameters [9]), where comonads/modalities appear only in the semantics. However, using flat and structural variants of indexed comonads to define a coeffect meta-language is an interesting future work.

In such system, a (structural) denotation $\mathsf{D}^{(r \times s)}(A \hat{\times} B) \to C$ could be expressed as $\mathsf{D}^r A \times \mathsf{D}^s B \to C$. However, the structure $\mathsf{D}^{(r \times s)}$ might include additional contextual information that cannot be split between individual variables.

Unified contextual calculi. In the introduction, we identified three notions of context – pure (variable context), flat (attached to the entire context) and structural (attached to variables). The flat and structural calculi capture the latter two. An interesting future work is to define a single calculus that captures all three notions.

7. Summary and conclusions

In programming language theory, context usually refers to free variables of an expression, but two other notions of context are common in programming for which we have told our tale of two calculi. The *flat coeffect calculus* captures contextual properties associated with the entire context, such as implicit parameters, and the *structural coeffect calculus* captures contextual properties associated with individual variables, such as liveness.

In this paper, we revisited existing work on (flat) coeffects. We further developed its syntactic properties, reduction strategies, and gave a categorical semantics to Haskell-style implicit parameters and data-flow computations. From this we developed the structural variant. For contextual properties associated with variables, such as liveness and data-flow, this gives a calculus with a more precise coeffect analysis and desirable syntactic properties.

With the rise of heterogeneous distributed computations, contextual properties of programs (input impurities) are becoming equally important to program effects (output impurities). Our work gives the foundation for developing types and semantics for contextual programming languages and language features, providing the counterpart to well established work on effect systems and monads.

References

[1] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.

[2] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. ICFP, 2003.

[3] G. M. Bierman and V. C. V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65:2000, 2001.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. FMCO Symposium, 2006.

[5] C. Flanagan and M. Abadi. Types for Safe Locking. ESOP, 1999.

[6] M. Gabbay and A. Nanevski. Denotation of syntax and metaprogramming in contextual modal type theory (CMTT). *CoRR*, abs/1202.0904, 2012.

[7] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. LFP, 1986.

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

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

[10] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of POPL*, pages 47–57, 1988.

[11] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93: 55–92, July 1991. ISSN 0890-5401.

[12] T. Murphy, VII., K. Crary, and R. Harper. Type-safe distributed programming with ML5. TGC'07, pages 108–123, 2008.

[13] T. Murphy VII, K. Crary, and R. Harper. Distributed control flow with classical modal logic. In *Proceedings of CSL*, pages 51–69, 2005.

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

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

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

[17] D. Orchard. Programming contextual computations. PhD thesis, University of Cambridge, 2013. (To appear).

[18] D. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: Declarative, Parallel Structured Grid Programming. In *DAMP 2010*, pages 15–24.

[19] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: Unified static analysis of context-dependence. ICALP '13, 2013.

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

[21] 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.

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

[23] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203:263–284, June 2008.

[24] T. Uustalu and V. Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, Nov 2006.

[25] 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.

[26] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.

[27] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of POPL*, pages 60–76, 1989.

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