Coeffects: Unified static analysis of context-dependence

Tomas Petricek, Dominic Orchard and Alan Mycroft

University of Cambridge, UK
{tp322,dao29,am}@cl.cam.ac.uk

Abstract. Monadic effect systems provide a unified way of tracking effects of computations, but there is no unified mechanism for tracking how computations rely on the environment in which they are executed. This is becoming an important problem for modern software – we need to track where distributed computations run, which resources a program uses and how they use other capabilities of the environment.

We consider three examples of context-dependence analysis: *liveness* analysis, tracking the use of *implicit parameters* (similar to tracking of *resource usage* in distributed computation), and calculating caching requirements for *dataflow* programs. Informed by these cases, we present a unified calculus for tracking context dependence in functional languages together with a categorical semantics based on *indexed comonads*. We believe that indexed comonads are the right foundation for constructing context-aware languages and type systems and that following an approach akin to monads can lead to a widespread use of the concept.

Modern applications run in diverse environments – such as mobile phones or the cloud – that provide additional resources and meta-data about provenance and security. For correct execution of such programs, it is often more important to understand how they *depend* on the environment than how they *affect* it.

Understanding how programs affect their environment is a well studied area: *effect systems* [13] provide a static analysis of effects and *monads* [8] provide a unified semantics to different notions of effect. Wadler and Thiemann unify the two approaches [17], *indexing* a monad with effect information, and showing that the propagation of effects in an effect system matches the semantic propagation of effects in the monadic approach.

No such unified mechanism exists for tracking the context requirements. We use the term *coeffect* for such contextual program properties. Notions of context have been previously captured using *comonads* [14] (the dual of monads) and by languages derived from *modal logic* [12,9], but these approaches do not capture many useful examples which motivate our work. We build mainly on the former comonadic direction (§3) and discuss the modal logic approach later (§5).

We extend a simply-typed lambda calculus with a *coeffect system* based on comonads, replicating the successful approach of effect systems and monads.

Examples of coeffects. We present three examples that do not fit the traditional approach of effect systems and have not been considered using the modal logic perspective, but can be captured as coeffect systems (§1) – the tracking of implicit dynamically-scoped parameters (or resources), analysis of variable liveness, and tracking the number of required past values in dataflow computations.

Coeffect calculus. Informed by the examples, we identify a general algebraic structure for coeffects. From this, we define a general *coeffect calculus* that unifies the motivating examples (§2) and discuss its syntactic properties (§4).

Indexed comonads. Our categorical semantics (§3) extends the work of Uustalu and Vene [14]. By adding annotations, we generalize comonads to *indexed comonads*, which capture notions of computation not captured by ordinary comonads.

1 Motivation

Effect systems, introduced by Gifford and Lucassen [5], track *effects* of computations, such as memory access or message-based communication [6]. Their approach augments typing judgments with effect information: $\Gamma \vdash e : \tau, F$. In Moggi's semantics, well-typed terms $\Gamma \vdash e : \tau$ are mapped to morphisms $[\![\Gamma]\!] \to M[\![\tau]\!]$ where M encodes effects and has the structure of a monad [8]. Wadler and Thiemann annotate monads with effect information, written M^F [17].

In contrast to the analysis of effects, our analysis of *context-dependence* differs in the treatment of lambda abstraction. Wadler and Thiemann explain that "*in the rule for abstraction, the effect is empty because evaluation immediately returns the function, with no side effects. The effect on the function arrow is the same as the effect for the function body, because applying the function will have the same side effects as evaluating the body*" [17]. We instead consider systems where λ-abstraction places *requirements* on both the *call-site* (latent requirements) and *declaration-site* (immediate requirements), resulting in different program properties. We informally discuss three examples that demonstrate how contextual requirements propagate. Section 2 unifies these in a single calculus.

We write coeffect judgements $C^s\Gamma \vdash e : \tau$ where the coeffect annotation s associates context requirements with the free-variable context Γ. Function types have the form $C^s\tau_1 \to \tau_2$ associating *latent* coeffects s with the parameter. The $C^s\Gamma$ syntax and $C^s\tau$ types are a result of the indexed comonadic semantics (§3).

Implicit parameters and resources. Implicit parameters [7] are *dynamically-scoped* variables. They can be used to parameterize a computation without propagating arguments explicitly through a chain of calls and are part of the context in which expressions evaluate. As correctly expected [7], they can be modelled by comonads. Rebindable resources in distributed computations (*e.g.*, a local clock) follow a similar pattern, but we discuss implicit parameters for simplicity.

The following function prints a number using implicit parameters ?culture (determining the decimal mark) and ?format (the number of decimal places):

$$\lambda n.\mathsf{printNumber}\ n\ \mathsf{?culture}\ \mathsf{?format}$$

Figure 1 shows a type-and-coeffect system tracking the set of an expression's implicit parameters. For simplicity here, all implicit parameters have type ρ.

Context requirements are created in (*access*), while (*var*) requires no implicit parameters; (*app*) combines requirements of both sub-expressions as well as the latent requirements of the function.

$$(var)\ \frac{x : \tau \in \Gamma}{C^{\emptyset}\Gamma \vdash x : \tau} \qquad (app)\ \frac{C^{r}\Gamma \vdash e_1 : C^{t}\tau_1 \to \tau_2 \qquad C^{s}\Gamma \vdash e_2 : \tau_1}{C^{r \cup s \cup t}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(access)\ \frac{}{C^{\{?\mathsf{a}\}}\Gamma \vdash ?\mathsf{a} : \rho} \qquad (abs)\ \frac{C^{r \cup s}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^{r}\Gamma \vdash \lambda x.e : C^{s}\tau_1 \to \tau_2}$$

Fig. 1. Selected coeffect rules for implicit parameters

The (*abs*) rule is where the example differs from effect systems. Function bodies can access the union of the parameters (or resources) available at the declaration-site ($C^{r}\Gamma$) and at the call-site ($C^{s}\tau_1$). Two of the nine permissible judgements for the above example are:

$$C^{\emptyset}\Gamma \vdash (\ldots) : C^{\{?\mathsf{culture},?\mathsf{format}\}}\mathsf{int} \to \mathsf{string}$$
$$C^{\{?\mathsf{culture},?\mathsf{format}\}}\Gamma \vdash (\ldots) : C^{\{?\mathsf{format}\}}\mathsf{int} \to \mathsf{string}$$

The coeffect system infers multiple, *i.e.* non-principal, coeffects for functions. Different judgments are desirable depending on how a function is used. In the first case, both parameters have to be provided by the caller. In the second, both are available at declaration-site, but ?format may be rebound (the precise meaning is provided by the semantics, discussed in §3).

Implicit parameters can be captured by the *reader* monad, where parameters are associated with the function codomain $M^{\emptyset}(\mathsf{int} \to M^{\{?\mathsf{culture},?\mathsf{format}\}}\mathsf{string})$, modelling only the first case. Whilst the reader monad can be extended to model rebinding, the next example cannot be structured by *any* monad.

Liveness analysis. Liveness analysis detects whether a free variable of an expression may be used (*live*) or whether it is definitely not used (*dead*). A compiler can remove bindings to dead variables as the result is never used.

We start with a restricted analysis and briefly mention how to make it practical later (§5). The restricted form is interesting theoretically as it gives rise to the *indexed partiality comonad* (§3), which is a basic but instructive example.

The coeffect system in Fig. 2 detects whether all free variables are dead ($C^{\mathsf{D}}\Gamma$) or whether at least one variable is live ($C^{\mathsf{L}}\Gamma$). Variable use (*var*) is annotated with L and constants with D, *i.e.*, if $c \in \mathbb{N}$ then $C^{\mathsf{D}}\Gamma \vdash c : \mathsf{int}$. A dead context may be marked as live by letting $\mathsf{D} \sqsubseteq \mathsf{L}$ and adding sub-coeffecting (§2).

The (*app*) rule can be understood by discussing its semantics. Consider semantic functions f, g, h annotated by r, s, t respectively. The *sequential composition* $g \circ f$ is live in its parameter only when both f and g are live. In the coeffect semantics, f is not evaluated if g ignores its parameter (regardless of evaluation order). Thus, $g \circ f$ is annotated by conjunction $r \sqcap s$ (where $\mathsf{L} \sqcap \mathsf{L} = \mathsf{L}$). A *pointwise composition* of g and h, passing the same parameter to both, is live in its parameter if either g or h is live (*i.e.*, disjunction $s \sqcup t$). Application uses both compositions, thus Γ is live if it is needed by e_1 *or* by the function *and* by e_2.

$$(var)\ \frac{x : \tau \in \Gamma}{C^{\mathsf{L}}\Gamma \vdash x : \tau} \qquad (app)\ \frac{C^{r}\Gamma \vdash e_1 : C^{t}\tau_1 \to \tau_2 \qquad C^{s}\Gamma \vdash e_2 : \tau_1}{C^{r \sqcup (s \sqcap t)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

Fig. 2. Selected coeffect rules for liveness analysis

$$(var)\ \frac{x : \tau \in \Gamma}{C^0\Gamma \vdash x : \tau} \qquad (app)\ \frac{C^m\Gamma \vdash e_1 : C^p\tau_1 \to \tau_2 \quad C^n\Gamma \vdash e_2 : \tau_1}{C^{max(m,n+p)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(prev)\ \frac{C^n\Gamma \vdash e : \tau}{C^{n+1}\Gamma \vdash \textbf{prev}\ e : \tau} \qquad (abs)\ \frac{C^{min(m,n)}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^m\Gamma \vdash \lambda x.e : C^n\tau_1 \to \tau_2}$$

Fig. 3. Selected coeffect rules for causal data flow

An (*abs*) rule (not shown) compatible with the structure in Fig. 1 combines the context annotations using $\sqcap$. Thus, if the body uses some variables, both the function argument and the context of the declaration-site are marked as live.

The coeffect system thus provides a call-by-name-style semantics, where redundant computations are omitted. Liveness cannot be modelled using monads with denotations $\tau_1 \to M^r\tau_2$. In call-by-value languages, the argument τ_1 is always evaluated. Using indexed comonads (§3), we model liveness as a morphism $C^r\tau_1 \to \tau_2$ where C^r is the parametric type $\mathsf{Maybe}\ \tau = \tau + 1$ (which contains a value τ when $r = \mathsf{L}$ and does not contain value when $r = \mathsf{D}$).

Efficient dataflow. Dataflow languages (*e.g.*, Lucid [16]) declaratively describe computations over streams. In *causal* data flow, programs may access past values. In this setting, a function $\tau_1 \to \tau_2$ becomes a function from a list of historical values $[\tau_1] \to \tau_2$. A coeffect system here tracks how many past values to cache.

Figure 3 annotates contexts with an integer specifying the maximum number of required past values. The current value is always present, so (*var*) is annotated with 0. The expression **prev** e gets the previous value of stream e and requires one additional past value (*prev*); *e.g.* **prev** (**prev** e) requires 2 past values.

The (*app*) rule follows the same intuition as for liveness. Sequential composition adds the tags (the first function needs $n + p$ past values to produce p past inputs for the second function); passing the context to two subcomputations requires the maximum number of the elements required by the two subcomputations. The (*abs*) rule for data-flow needs a distinct operator – *min* – therefore, the declaration-site and call-site must each provide at least the number of past values required by the function body (as the body may use variables coming from the declaration-site as well as the argument).

Soundness follows from our categorical model (§3). Uustalu and Vene model causal dataflow by a non-empty list comonad $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$ [14]. However, this model leads to (inefficient) unbounded lists of past elements. The coeffect system above infers a (sound) over-approximation of the number of required past elements and so fixed-length lists may be used instead.

2 Generalized coeffect calculus

The previous three examples exhibit a number of commonalities. We capture these in the *coeffect calculus*. We do not overly restrict the calculus to allow for notions of context-dependent computations not discussed above.

The syntax of our calculus is that of the simply-typed lambda calculus (where v ranges over variables, T over base types, and r over coeffect annotations):

$$e ::= v \mid \lambda v.e \mid e_1\ e_2 \qquad\qquad \tau ::= \mathsf{T} \mid \tau_1 \to \tau_2 \mid C^r \tau$$

The type $C^r \tau$ captures values of type τ in a context specified by the annotation r. This type appears only on the left-hand side of a function arrow $C^r \tau_1 \to \tau_2$. In the semantics, C^r corresponds to some data type (*e.g.*, List or Maybe). Extensions such as explicit *let*-binding are discussed later (§4).

The coeffect tags r, that were demonstrated in the previous section, can be generalized to a structure with three binary operators and a particular element.

Definition 1. *A* coeffect algebra $(S, \oplus, \vee, \wedge, \mathsf{e})$ *is a set* S *with an element* $\mathsf{e} \in S$*, a semi-lattice* $(S, \vee)$*, a monoid* $(S, \oplus, \mathsf{e})$*, and a binary* $\wedge$*. That is,* $\forall r, s, t \in S$*:*

$$r \oplus (s \oplus t) = (r \oplus s) \oplus t \qquad \mathsf{e} \oplus r = r = r \oplus \mathsf{e} \qquad \text{(monoid)}$$

$$r \vee s = s \vee r \qquad r \vee (s \vee t) = (r \vee s) \vee t \qquad r \vee r = r \qquad \text{(semi-lattice)}$$

The generalized coeffect calculus captures the three motivating examples (§1), where some operators of the coeffect algebra may coincide.

The $\oplus$ operator represents *sequential* composition; guided by the categorical model (§3), we require it to form a monoid with e. The operator $\vee$ corresponds to merging of context requirements in *pointwise composition* and the semi-lattice $(S, \vee)$ defines a partial order: $r \leq s$ when $r \vee s = s$. This ordering implies a sub-coeffecting rule. The coeffect e is often the top or bottom of the lattice.

The $\wedge$ operator corresponds to splitting requirements of a function body between the call- and definition-site. This operator is unrestricted in the general system, though it has additional properties in *some* coeffects systems, *e.g.*, semi-lattice structure on $\wedge$. Possibly these laws should hold for all coeffect systems, but we start with as few laws as possible to avoid limiting possible uses of the calculus. We consider constrained variants with useful properties later (§4).

Implicit parameters use sets of names $S = \mathcal{P}(\mathsf{Id})$ as tags with union $\cup$ for all three operators. Variable use is annotated with $\mathsf{e} = \emptyset$ and $\leq$ is subset ordering.

Liveness uses a two point lattice $S = \{\mathsf{D}, \mathsf{L}\}$ where $\mathsf{D} \sqsubseteq \mathsf{L}$. Variables are annotated with the top element $\mathsf{e} = \mathsf{L}$ and constants with bottom D. The $\vee$ operation is $\sqcup$ (join) and $\wedge$ and $\oplus$ are both $\sqcap$ (meet).

Dataflow tags are natural numbers $S = \mathbb{N}$ and operations $\vee, \wedge$ and $\oplus$ correspond to *max*, *min* and $+$, respectively. Variable use is annotated with $\mathsf{e} = 0$ and the order $\leq$ is the standard ordering of natural numbers.

Coeffect typing rules. Figure 4 shows the rules of the coeffect calculus, given some coeffect algebra $(S, \oplus, \vee, \wedge, \mathsf{e})$. The context required by a variable (*var*) is annotated with e. The sub-coeffecting rule (*sub*) allows the contextual requirements of an expression to be generalized.

The (*abs*) rule checks the body of the function in a context $r \wedge s$, which is a combination of the coeffects available in the context r where the function

$$(var)\ \frac{x : \tau \in \Gamma}{C^{\mathsf{e}}\Gamma \vdash x : \tau} \qquad (app)\ \frac{C^{r}\Gamma \vdash e_1 : C^{t}\tau_1 \to \tau_2 \qquad C^{s}\Gamma \vdash e_2 : \tau_1}{C^{r \vee (s \oplus t)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(sub)\ \frac{C^{s}\Gamma \vdash e : \tau}{C^{r}\Gamma \vdash e : \tau}\ (s \leq r) \qquad (abs)\ \frac{C^{r \wedge s}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^{r}\Gamma \vdash \lambda x.e : C^{s}\tau_1 \to \tau_2}$$

Fig. 4. Type and coeffect system for the coeffect calculus

is defined and in a context s provided by the caller of the function. Note that none of the judgements create a *value* of type $C^{r}\tau$. This type appears only immediately to the left of an arrow $C^{r}\tau_1 \to \tau_2$.

In function application (*app*), context requirements of both expressions and the function are combined as previously: the pointwise composition $\vee$ is used to combine the coeffect r of the expression representing a function and the coeffects of the argument, sequentially composed with the coeffects of the function: $s \oplus t$.

For space reasons, we omit recursion. We note that this would require adding coeffect variables and extending the coeffect algebra with a fixed-point operation.

3 Coeffect semantics using indexed comonads

The approach of *categorical semantics* interprets terms as morphisms in some category. For typed calculi, typing judgments $x_1 : \tau_1 \ldots x_n : \tau_n \vdash e : \tau$ are usually mapped to morphisms $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau]\!]$. Moggi showed the semantics of various effectful computations can be captured generally using the (*strong*) *monad* structure [8]. Dually, Uustalu and Vene showed that (*monoidal*) *comonads* capture various kinds of context-dependent computation [14].

We extend Uustalu and Vene's approach to give a semantics for the coeffect calculus by generalising comonads to *indexed comonads*. We emphasise semantic intuition and abbreviate the categorical foundations for space reasons.

Indexed comonads. Uustalu and Vene's approach interprets well-typed terms as morphisms $C(\tau_1 \times \ldots \times \tau_n) \to \tau$, where C encodes contexts and has a comonad structure [14]. Indexed comonads comprise a *family* of object mappings C^{r} indexed by a coeffect r describing the contextual requirements satisfied by the encoded context. We interpret judgments $C^{r}(x_1 : \tau_1, \ldots, x_n : \tau_n) \vdash e : \tau$ as morphisms $C^{r}([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$.

The indexed comonad structure provides a notion of composition for computations with different contextual requirements.

Definition 2. *Given a monoid* $(S, \oplus, \mathsf{e})$ *with binary operator* $\oplus$ *and unit* e*, an* indexed comonad *over a category* $\mathcal{C}$ *comprises a family of object mappings* C^{r} *where for all* $r \in S$ *and* $A \in \mathrm{obj}(\mathcal{C})$ *then* $C^{r}A \in \mathrm{obj}(\mathcal{C})$ *and:*

- *a natural transformation* $\varepsilon_A : C^{\mathsf{e}}A \to A$*, called the* counit*;*
- *a family of mappings* $(-)^{\dagger}_{r,s}$ *from morphisms* $C^{r}A \to B$ *to morphisms* $C^{r \oplus s}A \to C^{s}B$ *in* $\mathcal{C}$*, natural in* A, B*, called* coextend*;*

such that for all $f : C^r \tau_1 \to \tau_2$ *and* $g : C^s \tau_2 \to \tau_3$ *the following equations hold:*

$$\varepsilon \circ f^\dagger_{r,\mathsf{e}} = f \qquad (\varepsilon)^\dagger_{\mathsf{e},r} = \mathsf{id} \qquad (g \circ f^\dagger_{r,s})^\dagger_{(r \oplus s),t} = g^\dagger_{s,t} \circ f^\dagger_{r,(s \oplus t)}$$

The *coextend* operation gives rise to an associative composition operation for computations with contextual requirements (with *counit* as the identity):

$$\hat{\circ} : (C^r \tau_1 \to \tau_2) \to (C^s \tau_2 \to \tau_3) \to (C^{r \oplus s} \tau_1 \to \tau_3) \qquad g \,\hat{\circ}\, f = g \circ f^\dagger_{r,s}$$

The composition $\hat{\circ}$ best expresses the intention of indexed comonads: contextual requirements of the composed functions are combined. The properties of the composition follow from the indexed comonad laws and the monoid $(S, \oplus, \mathsf{e})$.

Example 1. Indexed comonads are analogous to comonads (in coKleisli form), but with the additional monoidal structure on indices. Indeed, comonads are a special case of indexed comonads with a trivial singleton monoid, *e.g.*, $(\{1\}, *, 1)$ with $1 * 1 = 1$ where C^1 is the underlying functor of the comonad and ε and $(-)^\dagger_{1,1}$ are the usual comonad operations. However, as demonstrated next, not all indexed comonads are derived from ordinary comonads.

Example 2. The *indexed partiality comonad* encodes free-variable contexts of a computation which are either *live* or *dead* (*i.e.*, have *liveness* coeffects) with the monoid $(\{\mathsf{D}, \mathsf{L}\}, \sqcap, \mathsf{L})$, where $C^{\mathsf{L}} A = A$ encodes live contexts and $C^{\mathsf{D}} A = 1$ encodes dead contexts, where 1 is the unit type inhabited by a single value $()$. The *counit* operation $\varepsilon : C^{\mathsf{L}} A \to A$ and *coextend* operations $f^\dagger_{r,s} : C^{r \sqcap s} A \to C^s B$ (for all $f : C^r A \to B$), are defined:

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

The indexed family C^r here is analogous to the non-indexed Maybe (or *option*) data type $\mathsf{Maybe}\ A = A + 1$. This type does not permit a comonad structure since $\varepsilon : \mathsf{Maybe}\ A \to A$ is undefined at $(\mathsf{inr}\,())$. For the indexed comonad, ε need only be defined for $C^{\mathsf{L}} A = A$. Thus, indexed comonads capture a broader range of contextual notions of computation than comonads.

Moreover, indexed comonads are not restricted by the *shape preservation* property of comonads [11]: that a coextended function cannot change the *shape* of the context. For example, in the second case above $f^\dagger_{\mathsf{D},\mathsf{L}} : C^{\mathsf{D}} A \to C^{\mathsf{L}} B$ where the shape changes from 1 (empty context) to B (available context).

Monoidal indexed comonads. Indexed comonads provide a semantics to sequential composition, but additional structure is needed for the semantics of the full coeffect calculus. Uustalu and Vene [14] additionally require a (*lax semi-*) *monoidal comonad* structure, which provides a monoidal operation $\mathsf{m} : CA \times CB \to C(A \times B)$ for merging contexts (used in the semantics of abstraction).

The semantics of the coeffect calculus requires an indexed lax semi-monoidal structure for combining contexts *as well as* an indexed *colax* monoidal structure for *splitting* contexts. These are provided by two families of morphisms (given a coeffect algebra with $\vee$ and $\wedge$):

$$
\begin{aligned}
[\![C^r \Gamma \vdash \lambda x.e : C^s \tau_1 \to \tau_2]\!] &= \mathit{curry}\,([\![C^{r \wedge s}(\Gamma, x : \tau_1) \vdash e : \tau_2]\!] \circ \mathsf{m}_{r,s}) \\
[\![C^{r \vee (s \oplus t)} \Gamma \vdash e_1\ e_2 : \tau]\!] &= (\mathit{uncurry}\,[\![C^r \Gamma \vdash e_1 : C^t \tau_1 \to \tau_2]\!]) \circ \\
&\qquad (\mathit{id} \times [\![C^s \Gamma \vdash e_2 : \tau_1]\!]^{\dagger}_{s,t}) \circ \mathsf{n}_{r,s \oplus t} \circ C^{r \vee (s \oplus t)} \Delta \\
[\![C^{\mathsf{e}} \Gamma \vdash x_i : \tau_i]\!] &= \pi_i \circ \varepsilon
\end{aligned}
$$

Fig. 5. Categorical semantics for the coeffect calculus

- $\mathsf{m}_{r,s} : C^r A \times C^s B \to C^{(r \wedge s)}(A \times B)$ natural in A, B;
- $\mathsf{n}_{r,s} : C^{(r \vee s)}(A \times B) \to C^r A \times C^s B$ natural in A, B;

The $\mathsf{m}_{r,s}$ operation merges contextual computations with tags combined by $\wedge$ (greatest lower-bound), elucidating the behaviour of $\mathsf{m}_{r,s}$: that merging may result in the loss of some parts of the contexts r and s.

The $\mathsf{n}_{r,s}$ operation splits context-dependent computations and thus the contextual requirements. To obtain coeffects r and s, the input needs to provide *at least* r and s, so the tags are combined using the $\vee$ operator (least upper-bound).

For the sake of brevity, we elide the indexed versions of the laws required by Uustalu and Vene (*e.g.*, most importantly, merging and splitting are associative).

Example 3. For the indexed partiality comonad, given the liveness coeffect algebra $(\{\mathsf{D}, \mathsf{L}\}, \sqcap, \sqcup, \sqcap, \mathsf{L})$, the additional lax/colax monoidal operations are:

$$
\begin{array}{lll}
\mathsf{m}_{\mathsf{L},\mathsf{L}}(x,y) = (x,y) & \mathsf{n}_{\mathsf{D},\mathsf{D}}\ () \quad = ((),()) & \mathsf{n}_{\mathsf{D},\mathsf{L}}(x,y) = ((),y) \\
\mathsf{m}_{\mathsf{r},\mathsf{s}}\ (x,y) = () & \mathsf{n}_{\mathsf{L},\mathsf{D}}(x,y) = (x,()) & \mathsf{n}_{\mathsf{L},\mathsf{L}}(x,y) = (x,y)
\end{array}
$$

Example 4. Uustalu and Vene model causal dataflow computations using the non-empty list comonad $\mathsf{NEList}\ A = A \times (1 + \mathsf{NEList}\ A)$ [14]. Whilst this comonad implies a trivial indexed comonad, we define an indexed comonad with integer indices for the number of past values demanded of the context.

We define $C^n A = A \times (A \times \ldots \times A)$ where the first A is the current (always available) value, followed by a finite product of n past values. The definition of the operations is a straightforward extension of the work of Uustalu and Vene.

Categorical Semantics. Figure 5 shows the categorical semantics of the coeffect calculus using additional operations π_i for projection of the i^{th} element of a product, usual *curry* and *uncurry* operations, and $\Delta : A \to A \times A$ duplicating a value. While C^r is a family of object mappings, it is promoted to a family of functors with the derived morphism mapping $C^r(f) = (f \circ \varepsilon)^{\dagger}_{\mathsf{e},r}$.

The semantics of variable use and abstraction are the same as in Uustalu and Vene's semantics, modulo coeffects. Abstraction uses $\mathsf{m}_{r,s}$ to merge the outer context with the argument context for the context of the function body. The indices of e for ε and r, s for $\mathsf{m}_{r,s}$ match the coeffects of the terms. The semantics of application is more complex. It first duplicates the free-variable values inside the context and then splits this context using $\mathsf{n}_{r,s \oplus t}$. The two contexts (with different coeffects) are passed to the two sub-expressions, where the argument subexpression, passed a context $(s \oplus t)$, is coextended to produce

a context t which is passed into the parameter of the function subexpression (*cf.* given $f : A \to (B \to C)$, $g : A \to B$, then $\mathit{uncurry}\, f \circ (id \times g) \circ \Delta : A \to C$).

A semantics for sub-coeffecting is omitted, but may be provided by an operation $\iota_{r,s} : C^r A \to C^s A$ natural in A, for all $r, s \in S$ where $s \leq r$, which transforms a value $C^r A$ to $C^s A$ by ignoring some of the encoded context.

4 Syntax-based equational theory

The operational semantics of every context-dependent language here differs as the notion of context is always different. However, for coeffect calculi satisfying certain conditions we can define a universal equational theory. This suggests a pathway to an operational semantics for two out of our three examples (the notion of context for data-flow is more complex).

In a pure λ-calculus, β- and η-equality for functions (also called *local soundness* and *completeness* respectively [12]) describe how pairs of abstraction and application can be eliminated: $(\lambda x.e_2)e_1 \equiv_\beta e_1[x \leftarrow e_2]$ and $(\lambda x.e\, x) \equiv_\eta e$. The β-equality rule, using the usual Barendregt convention of syntactic substitution, implies a *reduction*, giving part of an operational semantics for the calculus.

The call-by-name evaluation strategy modelled by β-reduction is not suitable for impure calculi therefore a restricted β rule, corresponding to call-by-value, is used, *i.e.* $(\lambda x.e_2)v \equiv e_2[x \leftarrow v]$. Such reduction can be encoded by a *let*-binding term, $\textsf{let}\ x = e_1\ \textsf{in}\ e_2$, which corresponds to sequential composition of two computations, where the resulting pure value of e_1 is substituted into e_2 [4,8].

For an equational theory of coeffects, consider first a notion of *let*-binding equivalent to $(\lambda x.e_2)\ e_1$, which has the following type and coeffect rule:

$$\frac{C^s \Gamma \vdash e_1 : \tau_1 \qquad C^{r_1 \wedge r_2}(\Gamma, x : \tau_1) \vdash e_2 : \tau_2}{C^{r_1 \vee (r_2 \oplus s)} \Gamma \vdash \textsf{let}\ x = e_1\ \textsf{in}\ e_2 : \tau_2} \tag{1}$$

For our examples, $\wedge$ is idempotent (*i.e.*, $r \wedge r = r$) implying a simpler rule:

$$\frac{C^s \Gamma \vdash e_1 : \tau_1 \qquad C^{r}(\Gamma, x : \tau_1) \vdash e_2 : \tau_2}{C^{r \vee (r \oplus s)} \Gamma \vdash \textsf{let}\ x = e_1\ \textsf{in}\ e_2 : \tau_2} \tag{2}$$

For our examples (but not necessarily *all* coeffect systems), this defines a more "precise" coeffect with respect to $\leq$ where $r \vee (r \oplus s) \leq r_1 \vee (r_2 \oplus s)$.

This rule removes the non-principality of the first rule (*i.e.*, multiple possible typings). However, using idempotency to split coeffects in abstraction would remove additional flexibility needed by the implicit parameters example.

The coeffect $r \vee (r \oplus s)$ can also be simplified for all our examples, leading to more intuitive rules – for implicit parameters $r \cup (r \cup s) = r \cup s$; for liveness we get that $r \sqcup (r \sqcap s) = r$ and for dataflow we obtain $\mathit{max}(r, r + s) = r + s$.

Our calculus can be extended with *let*-binding and (2). However, we also consider the cases when a syntactic substitution $e_2[x \leftarrow e_1]$ has the coeffects specified by the above rule (2) and prove *subject reduction* theorem for certain coeffect calculi. We consider two common special cases when the coeffect of

variables e is the greatest ($\top$) or least ($\bot$) element of the semi-lattice $(S, \vee)$ and derive additional properties that hold about the coeffect algebra:

Lemma 1 (Substitution). *Given $C^r(\Gamma, x : \tau_2) \vdash e_1 : \tau_1$ and $C^s\Gamma \vdash e_2 : \tau_2$ then $C^{r \vee (r \oplus s)}\Gamma \vdash e_2[x \leftarrow e_1] : \tau_1$ if the coeffect algebra satisfies the conditions that e is either the greatest or least element of the semi-lattice, $\oplus = \wedge$, and $\oplus$ distributes over $\vee$, i.e., $X \oplus (Y \vee Z) = (X \oplus Y) \vee (X \oplus Z)$.*

Proof. By induction over $\vdash$, using the laws (§2) and additional assumptions. □

Assuming $\rightarrow_\beta$ is the usual call-by-name reduction, the following theorem models the evaluation of coeffect calculi with coeffect algebra that satisfies the above requirements. We do not consider *call-by-value*, because our calculus does not have a notion of *value*, unless explicitly provided by *let*-binding (even a function "value" $\lambda x.e$ may have immediate contextual requirements).

Theorem 1 (Subject reduction). *For a coeffect calculus, satisfying the conditions of Lemma 1, if $C^r\Gamma \vdash e : \tau$ and $e \rightarrow_\beta e'$ then $C^r\Gamma \vdash e' : \tau$.*

Proof. A direct consequence of Lemma 1. □

The above theorem holds for both the liveness and resources examples, but not for dataflow. In the case of liveness, e is the greatest element ($r \vee \mathsf{e} = \mathsf{e}$); in the case of resources, e is the *least* element ($r \vee \mathsf{e} = r$) and the proof relies on the fact that additional context requirements can be placed at the context $C^r\Gamma$ (without affecting the type of function when substituted under λ-abstraction).

However, the coeffect calculus also captures context-dependence in languages with more complex evaluation strategies than *call-by-name* reduction based on syntactic substitution. In particular, syntactic substitution does not provide a suitable evaluation for dataflow (because a substituted expression needs to capture the context of the original scope).

Nevertheless, the above results show that – unlike effects – context-dependent properties can be integrated with *call-by-name* languages. Our work also provides a model of existing work, namely Haskell implicit parameters [7].

5 Related and further work

This paper follows the approaches of effect systems [5,13,17] and categorical semantics based on monads and comonads [8,14]. Syntactically, *coeffects* differ from *effects* in that they model systems where λ-abstraction may split contextual requirements between the declaration-site and call-site.

Our *indexed (monoidal) comonads* (§3) fill the gap between (non-indexed) *(monoidal) comonads* of Uustalu and Vene [14] and indexed monads of Atkey [2], Wadler and Thiemann [17]. Interestingly, *indexed* comonads are *more general* than comonads, capturing more notions of context-dependence (§1).

Comonads and modal logics. Bierman and de Paiva [3] model the $\Box$ modality of an intuitionistic S4 modal logic using monoidal comonads, which links our calculus to modal logics. This link can be materialized in two ways.

Pfenning et al. and Nanevski et al. derive term languages using the Curry-Howard correspondence [12,3,9], building a *metalanguage* (akin to Moggi's monadic metalanguage [8]) that includes $\Box$ as a type constructor. For example, in [12], the modal type $\Box\tau$ represents closed terms. In contrast, the *semantic* approach uses monads or comonads *only* in the semantics. This has been employed by Uustalu and Vene and (again) Moggi [8,14]. We follow the semantic approach.

Nanevski et al. extend an S4 term language to a *contextual* modal type theory (CMTT) [9]. The *context* is a set of variables required by a computation, which makes CMTT useful for meta-programming and staged computations. Our contextual types are indexed by a coeffect algebra, which is more general and can capture variable contexts, but also integers, two-point lattices, *etc.*.

The work on CMTT suggests two extensions to coeffects. The first is developing the logical foundations. We briefly considered special cases of our system that permits local soundness in §4; local completeness can be treated similarly. The second is developing a coeffect *metalanguage*. The use of coeffect algebras provides an additional flexibility over CMTT, allowing a wider range of applications via a richer metalanguage.

Relating effects and coeffects. The difference between effects and coeffects is mainly in the (*abs*) rule. While the semantic models (monads vs. comonads) are different, they can be extended to obtain equivalent syntactic rules. To allow splitting of implicit parameters in lambda abstraction, the reader monad needs an operation that eagerly performs some effects of a function: $(\tau_1 \to M^{r \oplus s} \tau_2) \to M^r(\tau_1 \to M^s \tau_2)$. To obtain a pure lambda abstraction for coeffects, we need to restrict the $\mathsf{m}_{r,s}$ operation of indexed comonads, so that the first parameter is annotated with e (meaning no effects): $C^{\mathsf{e}} A \times C^r B \to C^r(A \times B)$.

Structural coeffects. To make the liveness analysis practical, we need to associate information with individual variables (rather than the entire context). We can generalize the calculus from this paper by adding a product operation $\times$ to the coeffect algebra. A variable context $x : \tau_1, y : \tau_2, z : \tau_3$ is then annotated with $r \times s \times t$ where each component of the tag corresponds to a single variable. The system is then extended with structural rules such as:

$$(abs)\frac{C^{r \times s}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^r \Gamma \vdash \lambda x.e : C^s \tau_1 \to \tau_2} \quad (contr)\frac{C^{r \times s}(x : \tau_1, y : \tau_1) \vdash e : \tau_2}{C^{r \vee s}(z : \tau_1) \vdash e[x \leftarrow z][y \leftarrow z] : \tau_2}$$

The context requirements associated with function are exactly those linked to the specific variable of the lambda abstraction. Rules such as contraction manipulate variables and perform a corresponding operation on the indices.

The structural coeffect system is related to bunched typing [10] (but generalizes it by adding indices). We are currently investigating how to use structural coeffects to capture fine-grained context-dependence properties such as secure information flow [15] or, more generally, those captured by the dependency core calculus [1].

6 Conclusions

We examined three simple calculi with associated coeffect systems (liveness analysis, implicit parameters, and dataflow analysis). These were unified in the *coeffect calculus*, providing a general coeffect system parameterised by an algebraic structure describing propagation of context requirements throughout a program.

We model the semantics of the coeffect calculus using the *indexed (monoidal) comonad* structure – a novel structure, which is more powerful than (monoidal) comonads. Indices of the indexed comonad operations manifest the semantic propagation of context so that the propagation of information in the general coeffect type system corresponds exactly to the semantic propagation of context in our categorical model.

We consider the analysis of context to be essential, not least for the examples here but also given increasingly rich and diverse distributed systems.

Acknowledgements. We thank Gavin Bierman, Tarmo Uustalu, Varmo Vene and reviewers of earlier drafts. This work was supported by the EPSRC and CHESS.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of POPL*, 1999.
2. R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19, 2009.
3. G. M. Bierman and V. C. V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65:2000, 2001.
4. A. Filinski. Monads in action. In *Proceedings of POPL*, 2010.
5. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of Conference on LISP and func. prog.*, LFP '86, 1986.
6. P. Jouvelot and D. K. Gifford. Communication Effects for Message-Based Concurrency. Technical report, Massachusetts Institute of Technology, 1989.
7. J. R. Lewis, M. B. Shields, E. Meijert, and J. Launchbury. Implicit parameters: dynamic scoping with static types. In *Proceedings of POPL*, POPL '00, 2000.
8. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.
9. A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):23:1–23:49, June 2008.
10. P. O'Hearn. On bunched typing. *J. Funct. Program.*, 13(4):747–796, July 2003.
11. D. Orchard and A. Mycroft. A Notation for Comonads. In *Post-Proceedings of IFL'12*, LNCS. Springer Berlin / Heidelberg, 2012.
12. F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical. Structures in Comp. Sci.*, 11(4):511–540, Aug. 2001.
13. J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS'92.*, pages 162–173, 1994.
14. T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203:263–284, June 2008.
15. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.
16. W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language.* Academic Press Professional, Inc., San Diego, CA, USA, 1985.
17. P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.