CONTENTS

1 FLAT COEFFECT CALCULUS

Successful programming language abstractions need to generalize a wide range of recurring problems while capturing the key commonalities. These two aims are typically in opposition – more general abstractions are less powerful, while less general abstractions cannot be used as often.

In the previous chapter, we outlined a number of systems that capture how computations access the environment in which they are executed. We identified two kinds of systems – *flat* capturing whole-context properties and *structural* capturing per-variable properties. As we show in Chapter 3, the systems can be unified using a single abstraction. This is useful when implementing and composing the systems, but such abstraction is *less powerful* – i. e. its generality hides useful properties that we can see when we consider the systems separately. For this reason, this and the next chapter discusses *flat* and *structural* systems separately.

1.1 INTRODUCTION

In the previous chapter, we looked at three important examples of systems that track whole-context properties. The type systems for whole-context liveness (Section **??**) and whole-context data-flow (Section **??**) have a very similar structure – their lambda abstraction duplicates the requirements and their application arises from the combination of *sequential* and *point-wise* composition.

The system for tracking of implicit parameters (Section **??**), and similar systems for rebindable resources, differ in two ways. In lambda abstraction, they split the context requirements between the declaration-site and the call-site and they use only a single operator on the indices, typically $\cup$.

1.1.1 *Contributions*

All of the examples are practically useful and important and so we want to be able to capture all of them. Despite the differences, the systems can fit the same framework. The contributions of this chapter are as follows:

- We present a *flat coeffect calculus* with a type system that is parameterized by a *flat coeffect algebra* and can be instantiated to obtain all of the three examples discussed (Section 1.2).
- We give the equational theory of the calculus and discuss type-preservation for call-by-name and call-by-value reduction (Section 1.4). We also extend the calculus with subtyping and pairs (Section 1.5).
- We present the semantics of the calculus in terms of *indexed comonads*, which is a generalization of comonads, a category-theoretical dual of monads (Section 1.3). The semantics provides deeper insight into how (and why) the calculus works.

1.1.2 *Related work*

The development in this chapter can be seen as a counterpart to the well-known development of *effect systems* [29] and the use of *monads* [50] in programming languages. The syntax and type system of the flat coeffect calculus follows similar style as effect systems [48, 79], but differs in the structure, as explained in the previous chapter, most importantly in lambda abstraction.

Wadler and Thiemann famously show a correspondence between effect systems to monads [95], relating effectful functions $\tau_1 \xrightarrow{\sigma} \tau_2$ to monadic computations $\tau_1 \to M^{\sigma}\tau_2$. In this chapter, we show a similar correspondence between *coeffect systems* and *comonads*. However, due to the asymmetry of λ-calculus, this is not a simple mechanical dualization.

The main purpose of the comonadic semantics presented in this chapter is to provide a semantic motivation for the flat coeffect calculus. The semantics is inspired by the work of Uustalu and Vene [85] who present the semantics of contextual computations (mainly for data-flow) in terms of comonadic functions $C\tau_1 \to \tau_2$. Our *indexed comonads* annotate the structure with information about the required context, i. e. $C^{\sigma}\tau_1 \to \tau_2$. This is similar to the recent work on *parameterized monads* by Katsumata [40].

1.2 FLAT COEFFECT CALCULUS

The flat coeffect calculus is defined in terms of *flat coeffect algebra*, which defines the structure of context annotations, such as r, s, t. These can be sets of implicit parameters, integers or other values. The expressions of the calculus are those of the λ-calculus with *let* binding; assuming T ranges over base types, the types of the calculus are defined as follows:

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

We discuss subtyping and pairs in Section 1.5. The type $\tau_1 \xrightarrow{r} \tau_2$ 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* or *wrapped in* a context r.

In the categorical semantics, the function $\tau_1 \xrightarrow{r} \tau_2$ is modelled by a morphism $C^{r}\tau_1 \to \tau_2$. However, the object C^{r} does not exist as a syntactical value. This is because we use comonads to define the *semantics* rather than *embedding* them into the language as in the meta-language approaches (the distinction between the two approaches has been discussed in detail in Section **??**). The annotations r are formed by an algebraic structure discussed next.

1.2.1 *Reconciling lambda abstraction*

Recall the lambda abstraction rules for the implicit parameters system (annotating the context with sets of required parameters) and the data-flow system (annotating the context with the number of past required values):

$$(\textit{abs-imp})\ \frac{\Gamma, x : \tau_1\ @\ r \cup s \vdash e : \tau_2}{\Gamma\ @\ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2} \qquad (\textit{abs-df})\ \frac{\Gamma, x : \tau_1\ @\ n \vdash e : \tau_2}{\Gamma\ @\ n \vdash \lambda x.e : \tau_1 \xrightarrow{n} \tau_2}$$

In order to capture both systems using a single calculus, we need a way of unifying the two systems. For the data-flow system, this can be achieved by over-approximating the number of required past elements:

$$(\textit{abs-min})\quad \frac{\Gamma, x : \tau_1 \mathbin{@} \min(n, m) \vdash e : \tau_2}{\Gamma \mathbin{@} n \vdash \lambda x.e : \tau_1 \xrightarrow{m} \tau_2}$$

The rule (*abs-df*) is admissible in a system that includes the (*abs-min*) rule. If we include sub-typing rule (on annotations of functions) and sub-coeffecting rule (on annotations of contexts), then the reverse is also true – because $min(n, m) \leqslant m$ and $min(n, m) \leqslant n$.

1.2.2 *Flat coeffect algebra*

To make the flat coeffect system general enough, the algebra consists of three operations. Two of them, $\circledast$ and $\oplus$, represent the *sequential* and *point-wise* composition, respectively and the third one, $\wedge$ represents context *merging*. The term merging should be understood semantically – the operation models what happens when the semantics of lambda abstraction combines context available at the declaration-site and the call-site.

In addition to the three operations, we also require two special values used to annotate variable access and constant access and a relation that defines the ordering.

Definition 1. *A* flat coeffect algebra $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, relation* $\leqslant$ *and binary operations* $\circledast, \oplus, \wedge$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *and* $(\mathcal{C}, \oplus, \mathsf{ign})$ *are monoids,* $(\mathcal{C}, \leqslant)$ *is a pre-order and* $(\mathcal{C}, \wedge)$ *is a band (idempotent semigroup). That is, for all* $r, s, t \in \mathcal{C}$*:*

$$\begin{array}{rrr}
r \circledast (s \circledast t) = (r \circledast s) \circledast t & \mathsf{use} \circledast r = r = r \circledast \mathsf{use} & \text{(monoid)} \\
r \oplus (s \oplus t) = (r \oplus s) \oplus t & \mathsf{ign} \oplus r = r = r \oplus \mathsf{ign} & \text{(monoid)} \\
r \wedge (s \wedge t) = (r \wedge s) \wedge t & r \wedge r = r & \text{(band)} \\
\text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t & t \leqslant t & \text{(pre-order)}
\end{array}$$

In addition, the following distributivity axioms hold:

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

In two of the three systems, some of the operators of the flat coeffect algebra coincide, but the data-flow system requires all three. Similarly, the two special elements also coincide in some, but not all systems. The required laws are motivated by the aim to capture common properties of the three examples, without unnecessarily restricting the system:

- The monoid $(\mathcal{C}, \circledast, \mathsf{use})$ represents *sequential* composition of (semantic) functions. The laws of a monoid are required in order to form a category structure in the categorical model (Section 1.3).
- The monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ represents *point-wise* composition, i. e. the case when the same context is passed to multiple (independent) computations. The monoid laws guarantee that usual syntactic transformations on tuples and the unit value (Section 1.5) preserve the coeffect.
- For the $\wedge$ operation, we require associativity and idempotence. The idempotence requirement makes it possible to duplicate the coeffects and place the same requirement on both call-site and declaration-site,

i. e. it makes the (*abs-df*) rule admissible. In some cases, the operator forms a monoid with the unit being the greatest element of the set.

It is worth noting that the operators $\oplus$ and $\wedge$ are dual in some of the systems. For example, in data-flow computations, they are *max* and *min* respectively. However, this duality does not hold for implicit parameters. Using the syntactic reading, they represent *merging* and *splitting* of context requirements – in the (*abs*) rule, $\wedge$ appears in the assumption and the combined context requirements of the body are split between two positions in the conclusions; in the (*app*) rule, $\oplus$ appears in the conclusion and combines two context requirements from the assumptions.

ORDERING. The flat coeffect algebra requires a pre-order relation $\leqslant$, which is used to define sub-coeffecting rule of the type system. When the monoid $(\mathcal{C}, \oplus, \mathtt{ign})$ is idempotent and commutative monoid (semi-lattice), the $\leqslant$ relation can be defined in terms of $\oplus$ as:

$$r \leqslant s \iff r \oplus s = s$$

This definition is consistent with all three examples that motivate flat coeffect calculus, but it cannot be used with the structural coeffects (where it fails for the bounded reuse calculus) and so we choose not to use it.

Furthermore, the `use` coeffect is often the top (greatest) or the bottom (smallest) element of the semi-lattice, but not in general. As discussed in Section 1.4, when this is the case, we are able to prove certain properties of the calculus.

1.2.3 *Understanding flat coeffects*

Before looking at the type system in Figure 1, let us clarify how the rules should be understood. The coeffect calculus provides both analysis of context dependence (type system) and semantics for context (how it is propagated). These two aspects provide different ways of reading the judgements $\Gamma\,@\,r \vdash e : \tau$ and the typing rules used to define it.

- **Analysis of context dependence.** Syntactically, coeffect annotations r model *context requirements*. This means we can over-approximate them and require more than is actually needed at runtime.

  Syntactically, the typing rules should be read top-down. In (*app*), the context requirements of multiple assumptions are *merged*; in (*abs*), they are split between the declaration-site and the call-site.

- **Semantics of context passing.** Semantically, coeffect annotations r model *contextual capabilities*. This means that we can throw away capabilities, if a sub-expression requires fewer than we currently have.

  Semantically, the typing rules should be read bottom-up. In application, the capabilities provided to the term $e_1\ e_2$ are *split* between the two sub-expressions; in abstraction, the capabilities provided by the call-site and declaration-site are *merged* and passed to the body.

The reason for this asymmetry follows from the fact that the context appears in a *negative position* in the semantic model (Section 1.3). It means that we need to be careful about using the words *split* and *merge*, because they can be read as meaning opposite things. To disambiguate, we always use the term *context requirements* when using the syntactic view and *context capabilities* or just *available context* when using the semantic view.

$$(\textit{var})\ \frac{x : \tau \in \Gamma}{\Gamma\,@\,\mathsf{use} \vdash x : \tau}$$

$$(\textit{const})\ \frac{c : \tau \in \Delta}{\Gamma\,@\,\mathsf{ign} \vdash c : \tau}$$

$$(\textit{sub})\ \frac{\Gamma\,@\,r' \vdash e : \tau}{\Gamma\,@\,r \vdash e : \tau} \qquad (r' \leqslant r)$$

$$(\textit{app})\ \frac{\Gamma\,@\,r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma\,@\,s \vdash e_2 : \tau_1}{\Gamma\,@\,r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2}$$

$$(\textit{abs})\ \frac{\Gamma, x : \tau_1\,@\,r \wedge s \vdash e : \tau_2}{\Gamma\,@\,r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

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

Figure 1: Type system for the flat coeffect calculus

1.2.4 *Flat coeffect types*

The type system for flat coeffect calculus is shown in Figure 1. Variables (*var*) and constants (*const*) are annotated with special values provided by the coeffect algebra. Following the top-down syntactic reading, the (*sub*) rule allows us to treat an expression with fewer context requirements as an expression with more context requirements.

The (*abs*) rule is defined as discussed in Section 1.2.1. The body is annotated with context requirements $r \wedge s$, which are then split between the context-requirements on the declaration-site r and context-requirements on the call-site s. Examples of the $\wedge$ operator are discussed in the next section.

In function application (*app*), context requirements of both expressions and the function are combined as discussed in Chapter **??**. The pointwise composition $\oplus$ is used to combine the context requirements of the expression representing a function r and the context requirements of the argument, sequentially composed with the context-requirements of the function $s \circledast t$.

The type system also includes a rule for let-binding. The rule is *not* equivalent to the derivation for $(\lambda x.e_2)\ e_1$, but it represents one admissible typing derivation. We return to let-binding after looking a number of examples. Additional constructs such as pairs are covered in Section 1.5.

1.2.5 *Examples of flat coeffects*

The flat coeffect calculus generalizes the flat systems discussed in Section **??** of the previous chapter. We can instantiate it to a specific use just by providing a flat coeffect algebra. The following summary defines the systems for implicit parameters, liveness and data-flow. For the latter two, we obtain more general (but compatible) rule for lambda abstraction.

Example 1 (Implicit parameters)**.** *Assuming* Id *is a set of implicit parameter names, the flat coeffect algebra is formed by* $(\mathcal{P}(\mathsf{Id}), \cup, \cup, \cup, \emptyset, \emptyset, \subseteq)$.

For simplicity, we assume that all parameters have the same type ρ and so the annotations only track sets of names. The definition uses set union for all three operations. Both variables and constants are are annotated with $\emptyset$ and

the ordering is defined by $\subseteq$. The definition satisfies the flat coeffect algebra laws because $(S, \cup, \emptyset)$ is an idempotent, commutative monoid. The system has a single additional typing rule for accessing the value of a parameter:

$$(\textit{param}) \quad \frac{?p \in c}{\Gamma \circledast c \vdash ?p : \rho}$$

The rule specifies that the accessed parameter $?p$ needs to be in the set of required parameters c. As discussed earlier, we use the same type ρ for all parameters, but it is easy to define an extension tracking set of parameters with type annotations.

Example 2 (Liveness)**.** *Let* $L = \{L, D\}$ *be a two-point lattice such that* $D \sqsubseteq L$ *with a join* $\sqcup$ *and meet* $\sqcap$*. The flat coeffect algebra for liveness is then formed by* $(L, \sqcap, \sqcup, \sqcap, L, D, \sqsubseteq)$*.*

As in Section **??**, sequential composition $\circledast$ is modelled by the meet operation $\sqcap$ and point-wise composition $\oplus$ is modelled by join $\sqcup$. Two-point lattice is a commutative, idempotent monoid. The distributivity $(r \sqcup s) \sqcap t = (r \sqcap t) \sqcup (s \sqcap t)$ does not hold for *every* lattice, but it trivially holds for a two-point lattice used here.

The definition uses join $\sqcup$ for the $\wedge$ operator that is used by lambda abstraction. This means that, when the body is live L, both declaration-site and call-site are marked as live L. When the body is dead D, the declaration-site and call-site can be marked as dead D, or as live L, which is less precise, but permissible over-approximation, which could otherwise be achieved via sub-typing.

Example 3 (Data-flow)**.** *In data-flow, context is annotated with natural numbers and the flat coeffect algebra is formed by* $(\mathbb{N}, +, \mathit{max}, \mathit{min}, 0, 0, \leqslant)$*.*

As discussed earlier, sequential composition $\circledast$ is represented by $+$ and point-wise composition $\oplus$ uses *max*. For data-flow, we need a third separate operator for lambda abstraction. Annotating the body with $\mathit{min}(r, s)$ ensures that both call-site and declaration-site annotations are equal or greater than the annotation of the body. As with liveness, this allows over-approximation.

As required by the laws, $(\mathbb{N}, +, 0)$ and $(\mathbb{N}, \mathit{max}, 0)$ form monoids and $(\mathbb{N}, \mathit{min})$ forms a band. Note that data-flow is our first example where $+$ is not idempotent. The distributivity laws require the following to be the case: $\mathit{max}(r, s) + t = \mathit{max}(r + t, s + t)$, which is easy to see. Finally, a simple data-flow language includes an additional rule for `prev`:

$$(\textit{prev}) \quad \frac{\Gamma \circledast c \vdash e : \tau}{\Gamma \circledast c + 1 \vdash \texttt{prev}\ e : \tau}$$

As a further example that was not covered earlier, it is also possible to combine liveness analysis and data-flow. In the above calculus, 0 denotes that we require current value, but no previous values. However, for constants, we do not even need the current value.

Example 4 (Optimized data-flow)**.** *In optimized data-flow, context is annotated with natural numbers extended with the* $\bot$ *element, that is* $\mathbb{N}_\bot = \mathbb{N} \cup \{\bot\}$ *such that* $\forall n \in \mathbb{N}. \bot \leqslant n$*. The flat coeffect algebra is* $(\mathbb{N}_\bot, +, \mathit{max}, \mathit{min}, 0, \bot, \leqslant)$ *where* $m + n$ *is* $\bot$ *whenever* $m = \bot$ *or* $n = \bot$ *and* min, max *treat* $\bot$ *as the least element.*

Note that $(\mathbb{N}_\bot, +, 0)$ is a monoid for the extended definition of $+$, $(\mathbb{N}, \mathit{max}, \bot)$ is also a monoid and $(\mathbb{N}, \mathit{min})$ is a band. The required distributivity laws also holds for this algebra.

1.2.6 *Typing of let binding*

Recall the (*let*) rule in Figure 1. It annotates the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ with context requirements $s \oplus (s \circledast r)$. This is a special case of typing of an expression $(\lambda x.e_2)\ e_1$, using the idempotence of $\wedge$ as follows:

$$(app)\ \frac{\Gamma \circledast r \vdash e_1 : \tau_1 \qquad \dfrac{\Gamma, x : \tau_1 \circledast s \vdash e_2 : \tau_2}{\Gamma \circledast s \vdash \lambda x.e_2 : \tau_1 \xrightarrow{s} \tau_2}\ (abs)}{\Gamma \circledast s \oplus (s \circledast r) \vdash (\lambda x.e_2)\ e_1 : \tau_2}$$

This design decision is similar to ML value restriction, but it works the other way round. Our *let* binding is more restrictive rather than more general. The choice is motivated by the fact that the typing obtained using the special rule for let-binding is more precise (with respect to sub-coeffecting) for all the examples considered in this chapter. Table 1 shows how the coeffect annotations are simplified for our examples.

	Definition	**Simplified**
Implicit parameters	$s \cup (s \cup r)$	$s \cup r$
Liveness	$s \sqcap (s \sqcup r)$	s
Data-flow	$max(s, s + r)$	$s + r$

Table 1: Simplified annotation for let binding in sample flat calculi

The simplified annotations directly follow from the definitions of particular flat coeffect algebras. It is perhaps somewhat unexpected that the annotation can be simplified in different ways for different examples.

To see that the simplified annotations are *better*, assume that we used arbitrary splitting $s = s_1 \wedge s_2$ rather than idempotence. The "Definition" column would use s_1 and s_2 for the first and second s, respectively. The corresponding simplified annotation (using idempotence) would have $s_1 \wedge s_2$ instead of s. For all our systems, the simplified annotation (on the right) is more precise than the original (on the left):

$$\begin{array}{rcll} s_1 \cup (s_2 \cup r) & \supseteq & (s_1 \cup s_2) \cup r & \text{(implicit parameters)} \\ s_1 \sqcap (s_2 \sqcup r) & \sqsupseteq & (s_1 \sqcap s_2) & \text{(liveness)} \\ max(s_1, s_2 + r) & \geqslant & min(s_1, s_2) + r & \text{(data-flow)} \end{array}$$

The inequality cannot be proved from other properties of the flat coeffect algebra. To make the flat coeffect system as general as possible, we do not *in general* require it as an additional axiom, although the above examples provide reasonable basis for requiring that the specialized annotation for let binding is the least possible annotation for the expression $(\lambda x.e_2)\ e_1$.

1.3 CATEGORICAL MOTIVATION

The type system of flat coeffect calculus arises as a generalization of the examples discussed in Chapter **??**, but we can also obtain it by looking at the categorical semantics of context-dependent computations. This is a direction that we explore in this section. Although the development presented here is interesting in its own, our main focus is *using* categorical semantics to motivate and explain the design of flat coeffect calculus.

1.3.1 *Categorical semantics*

As discussed in Section **??**, categorical semantics interprets terms as morphisms in some category. For typed calculi, the semantics defined by $[\![-]\!]$ usually interprets typing judgements $x_1 : \tau_1 \ldots x_n : \tau_n \vdash e : \tau$ as morphisms $[\![\tau_1 \times \ldots \times \tau_n]\!] \to [\![\tau]\!]$.

As a best known example, Moggi [50] showed that the semantics of various effectful computations can be captured uniformly using the (*strong*) *monad* structure. In that approach, computations are interpreted as $\tau_1 \times \ldots \times \tau_n \to M\tau$ for some monad M. For example, $M\alpha = \alpha \cup \{\bot\}$ models partiality (maybe monad), $M\alpha = \mathcal{P}(\alpha)$ models non-determinism (list monad) and $M\alpha = (\alpha \times S)^S$ models side-effects (state monad). Here, the structure of a strong monad provides necessary "plumbing" for composing monadic computations.

Following similar approach to Moggi, Uustalu and Vene [85] showed that (*monoidal*) *comonads* uniformly capture the semantics of various kinds of context-dependent computations [85]. For example, data-flow computations over non-empty lists $\mathsf{NEList}\,\alpha = \alpha + (\alpha \times \mathsf{NEList}\,\alpha)$ are modelled using the non-empty list comonad.

The monadic and comonadic model outlined here represents at most a binary analysis of effects or context-dependence. A function $\tau_1 \to \tau_2$ performs *no* effects (requires no context) whereas $\tau_1 \to M\tau_2$ performs *some* effects and $C\tau_1 \to \tau_2$ requires *some* context. In the next section, we introduce *indexed comonads*, which provide a more precise analysis and let us model computations with context requirements r as functions $C^r\tau_1 \to \tau_2$ using an *indexed comonad* C^r.

1.3.2 *Introducing comonads*

In category theory, *comonad* is a dual of *monad*. Informally, we get a comonad by taking a monad and "reversing the arrows". More formally, one of the equivalent definitions of comonad looks as follows:

Definition 2. *A* comonad *over a category* $\mathcal{C}$ *is a triple* $(C, \mathsf{counit}, \mathsf{cobind})$ *where:*

- C *is a mapping on objects (types)* $C : \mathcal{C} \to \mathcal{C}$
- counit *is a mapping* $C\alpha \to \alpha$
- cobind *is a mapping* $(C\alpha \to \beta) \to (C\alpha \to C\beta)$

such that, for all $f : C\alpha \to \beta$ *and* $g : C\beta \to \gamma$*:*

$$\mathsf{cobind}\ \mathsf{counit} = \mathsf{id} \qquad \textit{(left identity)}$$

$$\mathsf{counit} \circ \mathsf{cobind}\ f = f \qquad \textit{(right identity)}$$

$$\mathsf{cobind}\ (g \circ \mathsf{cobind}\ f) = (\mathsf{cobind}\ g) \circ (\mathsf{cobind}\ f) \qquad \textit{(associativity)}$$

From the functional programming perspective, we can see C as a parametric data type such as NEList. The counit operations extracts a value α from a value that carries additional context $C\alpha$. The cobind operation turns a context-dependent function $C\alpha \to \beta$ into a function that takes a value with context, applies the context-dependent function to value(s) in the context and then propagates the context.

As mentioned earlier, Uustalu and Vene [85] use comonads to model data-flow computations. They describe infinite (coinductive) streams and non-empty lists as example comonads.

Example 5 (Non-empty list). *A non-empty list is a recursive data-type defined as* $\mathsf{NEList}\ \alpha = \alpha + (\alpha \times \mathsf{NEList}\ \alpha)$. *We write* `inl` *and* `inr` *for constructors of the left and right cases, respectively. The type* NEList *forms a comonad together with the following* counit *and* cobind *mappings:*

$$\begin{aligned}
\mathsf{counit}\ l &= h && \text{when } l = \texttt{inl}\ h \\
\mathsf{counit}\ l &= h && \text{when } l = \texttt{inr}\ (h, t) \\
\\
\mathsf{cobind}\ f\ l &= \texttt{inl}\ (f\ l) && \text{when } l = \texttt{inl}\ h \\
\mathsf{cobind}\ f\ l &= \texttt{inr}\ (f\ l,\ \mathsf{cobind}\ f\ t) && \text{when } l = \texttt{inr}\ (h, t)
\end{aligned}$$

The counit operation returns the head of the non-empty list. Note that it is crucial that the list is *non-empty*, because we always need to be able to obtain a value. The cobind defined here returns a list of the same length as the original where, for each element, the function f is applied on a *suffix* list starting from the element. Using a simplified notation for list, the result of applying cobind to a function that sums elements of a list gives the following behaviour:

$$\mathsf{cobind}\ \mathsf{sum}\ (7,6,5,4,3,2,1,0) = (28,21,15,10,6,3,1,0)$$

The fact that the function f is applied to a *suffix* is important in order to satisfy the *left identity* law, which requires that $\mathsf{cobind}\ \mathsf{counit}\ l = l$.

It is also interesting to examine some data types that do *not* form a comonad. As already mentioned, list $\mathsf{List}\ \alpha = 1 + (\alpha \times \mathsf{List}\ \alpha)$ is not a comonad, because the counit operation is not defined for the value `inl` $()$. Similarly, the Maybe data type defined as $1 + \alpha$ is not a comonad for the same reason. However, if we consider flat coeffect calculus for liveness, it appears natural to model computations as function $\mathsf{Maybe}\ \tau_1 \to \tau_2$. To use such model, we first need to generalise comonads to *indexed comonads*.

1.3.3 *Generalising to indexed comonads*

The flat coeffect algebra includes a monoid $(\mathcal{C}, \circledast, \mathsf{use})$, which defines the behaviour of sequential composition, where the annotation use represents a variable access. An indexed comonad is formed by a data type (object mapping) $C^r\,\alpha$ where the annotation r determines what context is required.

Definition 3. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with binary operator* $\circledast$ *and unit* use, *an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{r,s})$ *where:*

- C^r *for all* $r \in \mathcal{C}$ *is a family of object mappings*
- $\mathsf{counit}_{\mathsf{use}}$ *is a mapping* $C^{\mathsf{use}}\alpha \to \alpha$
- $\mathsf{cobind}_{r,s}$ *is a mapping* $(C^r\alpha \to \beta) \to (C^{r \circledast s}\alpha \to C^s\beta)$

such that, for all $f : C^r\alpha \to \beta$ *and* $g : C^s\beta \to \gamma$ *and the identity* $\mathsf{id}_s : C^s\alpha \to C^s\alpha$*:*

$$\mathsf{cobind}_{\mathsf{use},s}\ \mathsf{counit}_{\mathsf{use}} = \mathsf{id} \qquad \textit{(left identity)}$$

$$\mathsf{counit}_{\mathsf{use}} \circ \mathsf{cobind}_{r,\mathsf{use}}\ f = f \qquad \textit{(right identity)}$$

$$\mathsf{cobind}_{r \circledast s,t}\ (g \circ \mathsf{cobind}_{r,s}\ f) = (\mathsf{cobind}_{s,t}\ g) \circ (\mathsf{cobind}_{r,s \circledast t}\ f) \qquad \textit{(associativity)}$$

Rather than defining a single mapping C, we are now defining a family of mappings C^r indexed by a monoid structure. Similarly, the operation $\mathsf{cobind}_{r,s}$ operation is now also formed by a *family* of mappings for different pairs of indices r, s. To be fully precise, cobind is a family of natural transformations and we should include α, β as indices, writing $\mathsf{cobind}^{\alpha,\beta}_{r,s}$. For the purpose of this thesis, it is sufficient to treat cobind as a family of mappings or, when it does not introduce ambiguity, view it as a single mapping.

The counit operation is not defined for all $r \in \mathcal{C}$, but only for the unit use. We still include the unit as an index writing $\mathsf{counit}_{\mathsf{use}}$, but this is merely for symmetry. Crucially, this means that the operation is defined only for some special contexts.

If we look at the indices in the laws, we can see that the left and right identity require use to be the unit of $\circledast$. Similarly, the associativity law implies the associativity of the $\circledast$ operator.

The category that models sequential composition is formed by the unit arrow counit together with the (associative) composition operation that composes computations with contextual requirements as follows:

$$
\begin{aligned}
&- \mathbin{\hat{\circ}} - \;:\; (C^{r}\tau_1 \to \tau_2) \to (C^{s}\tau_2 \to \tau_3) \to (C^{r \circledast s}\tau_1 \to \tau_3) \\
&g \mathbin{\hat{\circ}} f \;=\; g \circ (\mathsf{cobind}_{r,s}\, f)
\end{aligned}
$$

The composition $\hat{\circ}$ best expresses the intention of indexed comonads. Given two functions with contextual requirements r and s, their composition is a function that requires $r \circledast, s$. The contextual requirements propagate *backwards* and are attached to the input of the composed function.

EXAMPLES. Any comonad can be turned into an indexed comonad using a trivial monoid. However, indexed comonads are more general and can be used with other data types, including indexed Maybe.

Example 6 (Comonads)**.** *Any comonad* C *is an indexed comonad with an index provided by a trivial monoid* $(\{1\}, *, 1)$ *where* $1 * 1 = 1$ *and* C^1 *is the underlying mapping* C *of the original comonad. The operations* counit_1 *and* $\mathsf{cobind}_{1,1}$ *are defined by the operations* counit *and* cobind *of the comonad.*

Example 7 (Indexed option)**.** *The indexed option comonad is defined over a monoid* $(\{L, D\}, \sqcup, L)$ *where* $\sqcup$ *is defined as earlier, i.e.* $L = r \sqcup s \iff r = s = L$*. Assuming* 1 *is the unit type inhabited by* $()$*, the mappings are defined as follows:*

$$
\begin{aligned}
&C^{L}\alpha = \alpha && \mathsf{cobind}_{r,s} \;:\; (C^{r}\alpha \to \beta) \to (C^{r \sqcup s}\alpha \to C^{s}\beta) \\
&C^{D}\alpha = 1 && \mathsf{cobind}_{L,L}\; f\; x \;=\; f\; x \\
& && \mathsf{cobind}_{L,D}\; f\; () \;=\; () \\
&\mathsf{counit}_{L} : C^{L}\alpha \to \alpha && \mathsf{cobind}_{D,L}\; f\; () \;=\; f\; () \\
&\mathsf{counit}_{L}\; v = v && \mathsf{cobind}_{D,D}\; f\; () \;=\; ()
\end{aligned}
$$

The indexed option comonad models the semantics of the liveness coeffect system discussed in **??**, where $C^{L}\alpha = \alpha$ models a live context and $C^{D}\alpha = 1$ models a dead context which does not contain a value. The counit operation extracts a value from a live context; cobind can be seen as an implementation of dead code elimination. The definition only evaluates f when the result is marked as live and is thus required, and it only accesses x if the function f requires its input.

The indexed family C^{r} in the above example is analogous to the Maybe (or option) data type $\mathsf{Maybe}\,\alpha = 1 + \alpha$. As mentioned earlier, this type does not permit (non-indexed) comonad structure, because counit $()$ is not defined. This is not a problem with indexed comonads, because counit only needs to be defined on live context.

Example 8 (Indexed product)**.** *The semantics of implicit parameters is modelled by an indexed product comonad. We use a monoid* $(\mathcal{P}(\mathsf{Id}), \cup, \emptyset)$ *where* Id *is the set of (implicit parameter) names. As previously, all parameters have the type* ρ*. The data type* $C^{r}\alpha = \alpha \times (r \to R)$ *represents a value* α *together with a function that*

associates a parameter value ρ *with every implicit parameter name in* $r \subseteq \mathsf{Id}$. *The cobind and counit operations are defined as:*

$$\begin{array}{ll} \mathsf{counit}_{\emptyset} : C^{\emptyset}\alpha \to \alpha & \mathsf{cobind}_{r,s} \ : \ (C^{r}\alpha \to \beta) \to (C^{r\cup s}\alpha \to C^{s}\beta) \\ \mathsf{counit}_{\emptyset}\ (a, g) = a & \mathsf{cobind}_{r,s}\ f\ (a, g) = (f(a, g|_{r}), g|_{s}) \end{array}$$

The definition of counit simply ignores the function and returns the value in the context. The cobind operation uses the restriction operation $f|_{r}$, which we already defined when discussing semantics of implicit parameters in Section **??** (indeed, cobind here captures an essential part of the semantics).

The function g in cobind is defined on the union of the implicit parameters, i. e. $r \cup s \to \rho$. When passing it to f, we restrict it to just r and when returning it as a result, we restrict it to s.

1.3.4 *Properties and related notions*

We discuss additional examples in Section 1.3.5, after we look at the remaining structure that is needed to define the semantics of flat coeffect calculus. Before doing so, we discuss additional properties and categorical structures that have been proposed mainly in the context of monads and effects and are related to indexed comonads.

SHAPE PRESERVATION. Ordinary comonads have the *shape preservation* property [58]. Intuitively, this means that the shape of the additional context does not change during the computation. For example, in the NEList comonad, the length of the list stays the same after applying cobind.

Indexed comonads are not restricted by this property of comonads. For example, given the indexed product monad, in the computation $\mathsf{cobind}_{r,s}\, f$ above, the shape of the context changes from containing implicit parameters $r \cup s$ to containing just implicit parameters s.

FAMILIES OF MONADS. When linking effect systems and monads, Wadler and Thiemann [50] propose a *family of monads* as the categorical structure. The dual structure, *family of comonads*, is defined as follows.

Definition 4. *A* family of comonads *is formed by triples* $(C^{r}, \mathsf{cobind}_{r}, \mathsf{counit}_{r})$ *for all* r *such that each triple forms a comonad. Given* r, r' *such that* $r \leqslant r'$, *there is also a mapping* $\iota_{r',r} : C^{r'} \to C^{r}$ *satisfying certain coherence conditions.*

Family of comonads is more restrictive than indexed comonad, because each of the data types needs to form a comonad separately. For example, our indexed option does not form a family of comonads (again, because counit is not defined on $C^{D}\alpha = 1$). However, given a family of comonads and indices such that $r \leqslant r \circledast s$, we can define an indexed comonad. Briefly, to define $\mathsf{cobind}_{r,s}$ of an indexed comonad, we use $\mathsf{cobind}_{r\circledast s}$ from the family, together with two lifting operations: $\iota_{r\circledast s,r}$ and $\iota_{r\circledast s,s}$.

PARAMETERIC EFFECT MONADS. Parametric effect monads introduced by Katsumata [40] (independently to our indexed comonads) are closely related to our definition. Although presented in a more general categorical framework (and using monads), the model defines unit operation only on the unit of a monoid and bind operation composes effect annotations using the provided monoidal structure.

1.3.5 *Flat indexed comonads*

Indexed comonads model the semantics of sequential composition, but additional structure is needed to model the semantics of the flat coeffect calculus. This is where the duality between monads and comonads can no longer help us, because context is propagated differently than effects in lambda abstraction and application.

Whereas Moggi [50] requires *strong* monad to model effectful λ-calculus, Uustalu and Vene [85] require *lax semi-monoidal* comonad to model λ-calculus with contextual properties. The structure requires a monoidal operation:

$$\mathsf{m} : \mathsf{C}\,\alpha \times \mathsf{C}\,\beta \to \mathsf{C}\,(\alpha \times \beta)$$

The m operation is needed in the semantics of lambda abstraction. It represents merging of contexts and is used to merge the context of the declaration-site (containing free variables) and the call-site (containing bound variable). For example, for implicit parameters, this combines the additional parameters defined in the two contexts.

The semantics of flat coeffect calculus requires operations for *merging*, but also for *splitting* of contexts. These are provided by *lax* and *oplax* monoidal structures. In addition, we also need a lifting operation (similar to ι from Definition 4) to model sub-coeffecting.

Definition 5. *Given a flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, *an* flat indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$ *and* $\mathsf{lift}_{r',r}$ *where:*

- $\mathsf{merge}_{r,s}$ *is a family of mappings* $\mathsf{C}^r\,\alpha \times \mathsf{C}^s\,\beta \to \mathsf{C}^{r \wedge s}\,(\alpha \times \beta)$
- $\mathsf{split}_{r,s}$ *is a family of mappings* $\mathsf{C}^{r \oplus s}\,(\alpha \times \beta) \to \mathsf{C}^r\,\alpha \times \mathsf{C}^s\,\beta$
- $\mathsf{lift}_{r',r}$ *is a family of mappings* $\mathsf{C}^{r'}\,\alpha \to \mathsf{C}^r\,\alpha$ *for all* r', r *such that* $r \leqslant r'$

The $\mathsf{merge}_{r,s}$ operation is the most interesting one. Given two comonadic values with additional contexts specified by r and s, it combineds them into a single value with additional context $r \wedge s$. The $\wedge$ operation often represents *greatest lower bound*[1], elucidating the fact that merging may result in the loss of some parts of the contexts r and s. We look at examples of this operation in the next section.

The $\mathsf{split}_{r,s}$ operation splits a single comonadic value (containing a tuple) into two separate values. Note that this does not simply duplicate the value, because the additional context is also split. To obtain coeffects r and s, the input needs to provide *at least* r and s, so the tags are combined using the $\oplus$, which is often the *least upper-bound*[1].

Finally, $\mathsf{lift}_{r',r}$ is a family of operations that "forget" some part of a context. This models the sub-coeffecting operation and lets us, for example, forget some of the available implicit parameters, or turn a live context (containing a value) into a dead context (empty).

ALTERNATIVE DEFINITION. Although we do not require this as a general law, in all our systems, it is the case that $r \leqslant r \oplus s$ and $s \leqslant r \oplus s$. This allows a simpler definition of *indexed flat comonad* by expressing the split operation in terms of the lifting (sub-coeffecting) as follows:

$$\begin{aligned}\mathsf{map}_r\ \mathsf{f} &= \mathsf{cobind}_{r,r}\ (\mathsf{f} \circ \mathsf{counit}_{\mathsf{use}})\\ \mathsf{split}_{r,s}\ \mathsf{c} &= (\mathsf{map}_r\ \mathsf{fst}\ (\mathsf{lift}_{r \oplus s,r}\ \mathsf{c}), \mathsf{map}_s\ \mathsf{snd}\ (\mathsf{lift}_{r \oplus s,s}\ \mathsf{c}))\end{aligned}$$

1 The $\wedge$ and $\oplus$ operations are the greatest and least upper bounds for the liveness and data-flow examples, but not for implicit parameters. However, they are useful as an informal analogy.

The map_r operation is the mapping on functions that corresponds to the object mapping C^r. The definition is dual to the standard definition of map for monads in terms of bind and unit. The functions fst and snd are first and second projections from a two-element pair. To define the $\mathsf{split}_{r,s}$ operation, we duplicate the argument c, then use lifting to throw away additional parts of the context and then transform the values in the context.

This alternative is valid for our examples, but we do not use it for two reasons. Firstly, it requires duplication of the value c, which is not required elsewhere in our model. So, using explicit split, our model could be embedded in a linear or affine model. Secondly, it is similar to the definition that is needed for structural coeffects in Chapter 2 and so it makes the connection between the two system easier to see.

EXAMPLES. All examples of *indexed comonads* discussed in Section 1.3.3 can be extended into *flat indexed comonads*.

Example 9 (Monoidal comonads)**.** *Just like indexed comonads generalise comonads, the additional structure of flat indexed comonads generalises symmetric semimonoidal comonads of Uustalu and Vene [85]. The flat coeffect algebra is defined as* $(\{1\}, *, *, *, 1, 1, =)$ *where* $1 * 1 = 1$ *and* $1 = 1$*. The additional operation* $\mathsf{merge}_{1,1}$ *is provided by the monoidal operation called* m *by Uustalu and Vene. The* $\mathsf{split}_{1,1}$ *operation is defined by duplication and* $\mathsf{lift}_{1,1}$ *is the identity function.*

Example 10 (Indexed option)**.** *Flat coeffect algebra for liveness defines* $\oplus$ *and* $\wedge$ *as* $\sqcup$ *and* $\sqcap$*, respectively and specifies that* $\mathsf{D} \sqsubseteq \mathsf{L}$*. Recall also that the object mapping is defined as* $C^{\mathsf{L}}\alpha = \alpha$ *and* $C^{\mathsf{D}}\alpha = 1$*. The additional operations of a flat indexed comonad are defined as follows:*

$$\begin{aligned}
\mathsf{merge}_{\mathsf{L},\mathsf{L}}\ (a, b) &= (a, b) & \mathsf{split}_{\mathsf{L},\mathsf{L}}\ (a, b) &= (a, b) \\
\mathsf{merge}_{\mathsf{L},\mathsf{D}}\ (a, ()) &= () & \mathsf{split}_{\mathsf{L},\mathsf{D}}\ (a, b) &= (a, ()) \\
\mathsf{merge}_{\mathsf{D},\mathsf{L}}\ ((), b) &= () & \mathsf{split}_{\mathsf{D},\mathsf{L}}\ (a, b) &= ((), b) \\
\mathsf{merge}_{\mathsf{D},\mathsf{D}}\ ((), ()) &= () & \mathsf{split}_{\mathsf{D},\mathsf{D}}\ () &= ((), ())
\end{aligned}
\qquad
\begin{aligned}
\mathsf{lift}_{\mathsf{L},\mathsf{D}}\ v &= () \\
\mathsf{lift}_{\mathsf{L},\mathsf{L}}\ v &= v \\
\mathsf{lift}_{\mathsf{D},\mathsf{D}}\ () &= ()
\end{aligned}$$

Without the indexing, the merge operations implements *zip* on option values, returning an option only when both values are present. The behaviour of the split operation is partly determined by the indices. When the input is *dead*, both values have to be dead (this is also the only solution of $\mathsf{D} = r \sqcap \mathsf{D}$), but when the input is *live*, the operation can perform implicit sub-coeffecting and drop one of the values.

Explicit sub-coeffecting using the (*sub*) rule is modelled by the lift operation. This can turn a *live* value v into a dead value $()$, or it can behave as identity. The behaviour is, again, determined by the index.

Example 11 (Indexed product)**.** *For implicit parameters, both* $\wedge$ *and* $\oplus$ *are the* $\cup$ *operation and the relation* $\leqslant$ *is formed by the subset relation* $\subseteq$*. Recall that the data type* $C^r\alpha$ *is* $\alpha \times (r \to \mathsf{R})$ *where* R *is some representation of a parameter value. The additional operations are defined as:*

$$\begin{aligned}
\mathsf{split}_{r,s}\ ((a, b), g) &= ((a, g|_r), (b, g|_s)) \\
\mathsf{merge}_{r,s}\ ((a, f), (b, g)) &= ((a, b), f \uplus g) \\
\mathsf{lift}_{r',r}\ (a, g) &= (a, g|_r)
\end{aligned}
\qquad \text{where } f \uplus g = f|_{dom(f) \setminus dom(g)} \cup g$$

The split operation splits the tuple and restricts the function (representing available implicit parameters) to the required sub-sets. This corresponds to the definition in terms of lift, which performs just the restriction. The merge

operation is more interesting. It uses $\uplus$ operation that we defined when introducing implicit parameters in Section **??**. It merges the values, preferring the definitions from the right-hand side (call-site) over left-hand side (declaration-site). Thus the operation is not symmetric.

Example 12 (Indexed list)**.** *Our last example provides the semantics of data-flow computations. The flat coeffect algebra is formed by* $(\mathbb{N}, +, max, min, 0, 0, \leqslant)$*. In a non-indexed version, the semantics is provided by a non-empty list. In the indexed semantics, the index represents the length of the storing past values. The data type is then a pair of the current value, followed by* n *past values. The mappings that form the flat indexed comonad are defined as follows:*

$$\mathsf{counit}_0\langle a_0\rangle = a_0 \qquad\qquad C^n\alpha = \underbrace{\alpha \times \ldots \times \alpha}_{(n+1)-\text{times}}$$

$$\mathsf{cobind}_{m,n}\ f\langle a_0, \ldots a_{m+n}\rangle = \langle f\langle a_0, \ldots, a_m\rangle, \ldots, f\langle a_n, \ldots, a_{m+n}\rangle\rangle$$

$$\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) = \langle (a_0, b_0), \ldots, (a_{min(m,n)}, b_{min(m,n)})\rangle$$

$$\mathsf{split}_{m,n}\langle (a_0, b_0), \ldots, (a_{max(m,n)}, b_{max(m,n)})\rangle = (\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle)$$

$$\mathsf{lift}_{n',n}\langle a_0, \ldots, a_{n'}\rangle = \langle a_0, \ldots, a_n\rangle \qquad (\text{when } n \leqslant n')$$

The reader is invited to check that the number of required past elements in each of the mappings matches the number specified by the indices. The index specifies the number of *past* elements and so the list always contains at least one value. Thus counit returns the element of a singleton list.

The $\mathsf{cobind}_{m,n}$ operation requires $m + n$ elements in order to generate n past results of the f function, which itself requires m past values. When combining two lists, $\mathsf{merge}_{m,n}$ behaves as *zip* and produces a list that has the length of the shorter argument. When splitting a list, $\mathsf{split}_{m,n}$ needs the maximum of the required lengths. Finally, the lifting operation just drops some number of elements from a list.

1.3.6 *Semantics of flat calculus*

In Section **??**, we defined the semantics of concrete (flat) context-dependent computations including implicit parameters, liveness and data-flow. Using the *flat indexed comonad* structure, we can now define a single uniform semantics that is capable capturing of all our examples, as well as other computations that can be modelled by the structure.

CONTEXTS AND FUNCTIONS. The modelling of contexts and functions generalizes the earlier concrete examples. We use the family of mappings C^r as an (indexed) data-type that wraps the product of free variables of the context and the arguments of functions:

$$\begin{aligned} [\![x_1 : \tau_1, \ldots, x_n : \tau_n \circledast r \vdash e : \tau]\!] &: C^r(\tau_1 \times \ldots \times \tau_n) \to \tau \\ [\![\tau_1 \xrightarrow{r} \tau_2]\!] &= C^r\tau_1 \to \tau_2 \end{aligned}$$

EXPRESSIONS. The definition of the semantics is shown in Figure 2. For readability, we write the definitions in a simple programming language nota-

$$[\![\Gamma \circledast \mathsf{use} \vdash x_i : \tau_i]\!]\ ctx = \pi_i\ (\mathsf{counit}_{\mathsf{use}}\ ctx) \qquad (var)$$

$$[\![\Gamma \circledast \mathsf{ign} \vdash c_i : \tau]\!]\ ctx = \delta\ (c_i) \qquad (const)$$

$$[\![\Gamma \circledast r \vdash e : \tau]\!]\ ctx = \qquad (sub)$$
$$[\![\Gamma \circledast r' \vdash e : \tau]\!]\ (\mathsf{lift}_{r,r'}\ ctx) \qquad (\text{when } r' \leqslant r)$$

$$[\![\Gamma \circledast r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!]\ ctx = \lambda v. \qquad (abs)$$
$$[\![\Gamma, x : \tau_1 \circledast r \wedge s \vdash e : \tau_2]\!]\ (\mathsf{merge}_{r,s}\ (ctx, v))$$

$$[\![\Gamma \circledast r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2]\!]\ ctx = \qquad (app)$$
$$\mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{r,s\circledast t}\ (\mathsf{map}_{r\oplus(s\circledast t)}\ (\lambda x.(x,x))\ ctx)$$
$$\mathsf{in}\ [\![\Gamma \circledast r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]\ ctx_1\ (\mathsf{cobind}_{s,t}\ [\![\Gamma \circledast s \vdash e_2 : \tau_1]\!]\ ctx_2)$$

Figure 2: Categorical semantics of the flat coeffect calculus

tion as opposed to the point-free categorical style. However, it can be equally written using just the operations of flat indexed comonad together with i^{th} projection from a tuple represented by π_i, *curry* and *uncurry*, function composition, value duplication ($\Delta : A \to A \times A$) and function pairing (given $f : A \to B$ and $g : C \to D$ then $f \times g : A \times C \to B \times D$). These operations can be provided by e. g. a Cartesian Closed Category.

The semantics of variable access and abstraction are the same as in the semantics of Uustalu and Vene [85], modulo the indexing. The semantics of variable access (*var*) uses $\mathsf{counit}_{\mathsf{use}}$ to extract product of free-variables from the context and then projection π_i to obtain the variable value. Abstraction (*abs*) takes the context *ctx* and function argument v and merges their additional contexts using $\mathsf{merge}_{r,s}$. Assuming the context Γ contains variables of types $\sigma_1, \ldots, \sigma_n$, this gives us a value $C^{r\wedge s}((\sigma_1 \times \ldots \times \sigma_n) \times \tau_1)$. Assuming that n-element tuples are associated to the left, the wrapped context is equivalent to $\sigma_1 \times \ldots \times \sigma_n \times \tau_1$, which can then be passed to the body of the function.

The semantics of application is more complex. It first duplicates the free-variable product inside the context (using map_r and duplication). Then it splits this context using $\mathsf{split}_{r,s\oplus t}$. The two contexts contain the same variables (as required by sub-expressions e_1 and e_2), but different coeffect annotations. The first context (with index r) is used to evaluate e_1, resulting in a function $C^t\tau_1 \to \tau_2$. To obtain the result, we compose this with a function created by applying $\mathsf{cobind}_{s,t}$ on the semantics of sub-expression e_2, which is of type $C^{s\circledast t}\sigma_1 \times \ldots \times \sigma_n \to C^t\tau_1$.

Finally, constants (*const*) are modelled by a global dictionary δ and subcoeffecting is interpreted by dropping additional context from the provided context *ctx* using $\mathsf{lift}_{r,r'}$ and providing it to the semantics of the assumption.

PROPERTIES. The categorical semantics can be used to embed context-dependent computations in functional programming languages, similarly to how monads provide a way of embedding effectful computations. More importantly, it also provides validation for the design of the type system developed in Section 1.2.4. As stated in the following theorem, the annotations in the type system match those of the semantic functions.

Remark 1 (Correspondence)**.** *In all of the typing rules of the flat coeffect system, the context annotations* r *of typing judgements* $\Gamma \mathbin{@} r \vdash e : \tau$ *and function types* $\tau_1 \xrightarrow{r} \tau_2$ *correspond to the indices of mappings* C^r *in the corresponding semantic function defined by* $[\![\Gamma \mathbin{@} r \vdash e : \tau]\!]$.

Proof. By analysis of the semantic rules in Figure 2. □

Thanks to the indexing, the statement of the remark is significantly stronger than for a non-indexed system, because it provides the justification for our choice of indices in the typing rules. In particular, we can see that the annotations follow from the annotations on primitive functions that define the semantics. Also, each function defining the semantics uses a distinct operation of the coeffect algebra and so the type system is the most general possible definition (within the comonadic framework we use).

1.4 EQUATIONAL THEORY

Each of the concrete coeffect calculi discussed in this chapter has a different notion of context, much like various effectful languages have different notions of effects (such as exceptions or mutable state). However, in all of the calculi, the context has a number of common properties that are captured by the *flat coeffect algebra*. This means that there are equational properties that hold for all of the systems we consider. Further properties hold for systems where the context satisfies additional properties.

In this section, we look at such shared syntactic properties. This accompanies the previous section, which provided a *semantic* justification for the axioms of coeffect algebra with a *syntactic* justification. Operationally, this section can also be viewed as providing a pathway to an operational semantics for two of our systems (implicit parameters and liveness), which can be based on syntactic substitution. As we discuss later, the notion of context for data-flow is more complex.

1.4.1 *Syntactic properties*

Before discussing the syntactic properties of general coeffect calculus formally, it should be clarified what is meant by providing "pathway to operational semantics" in this section. We do that by contrasting syntactic properties of coeffect systems with more familiar effect systems. Assuming $e_1[x \leftarrow e_2]$ is a standard capture-avoiding syntactic substitution, the following equations define four syntactic reductions on the terms:

$$
\begin{array}{rcll}
(\lambda x.e_1)\ e_2 & \longrightarrow_{\mathtt{cbn}} & e_1[x \leftarrow e_2] & (\textit{call-by-name}) \\
(\lambda x.e_1)\ v & \longrightarrow_{\mathtt{cbv}} & e_1[x \leftarrow v] & (\textit{call-by-value}) \\
(\lambda x.e_1)\ e_2 & \longrightarrow_{\mathtt{seq}} & \mathtt{glet}\ x = e_2\ \mathtt{in}\ e_1 & (\textit{internalized sequencing}) \\
e & \longrightarrow_{\eta} & \lambda x.e\ x & (\eta\textit{-expansion})
\end{array}
$$

The rules capture syntactic reductions that can be performed in a general calculus, without any knowledge of the specific notion of context. The `glet` notation models explicit sequencing of context-dependent computations an is inspired by Filinski [23]. In the rest of the section, we briefly outline the interpretation of the four rules and then we focus on call-by-value (Section 1.4.2) and call-by-name (Section 1.4.3) in more details.

The focus of this work is on the general coeffect system and so we do not discuss the operational semantics of the specific notions of context. However, some work in that area has been done by Brunel et al. [14].

CALL-BY-NAME. In call-by-name, the argument is syntactically substituted for all occurrences of a variable. It can be used as the basis for operational semantics of purely functional languages. However, using the rule in effectful languages breaks the *type preservation* property. For example, consider a language with effect system where functions are annotated with sets of effects such as $\{\mathsf{write}\}$. A function $\lambda x.y$ is a effect-free:

$$y{:}\tau_1 \vdash \lambda x.y : \tau_1 \xrightarrow{\emptyset} \tau_2 \;\&\; \emptyset$$

Substituting an expression e with effects $\{\mathsf{write}\}$ for y changes the type of the function by adding latent effects (without changing the immediate effects):

$$\vdash \lambda x.e : \tau_1 \xrightarrow{\{\mathsf{write}\}} \tau_2 \;\&\; \emptyset$$

Similarly to effect systems, substituting a context-dependent computation e for a variable y can add latent coeffects to the function type. However, this is not the case for *all* flat coeffect calculi. For example, call-by-name reduction preserves types and coeffects for the implicit parameters system. This makes the model suitable for languages such as Haskell.

CALL-BY-VALUE. The call-by-value evaluation strategy is often used by effectful languages. Here, an argument is first reduced to a *value* before performing the substitution. In effectful languages, value is defined syntactically. For example, in the *Effect* language [95], values are identifiers x or functions $(\lambda x.e)$.

The notion of *value* in coeffect systems differs from the usual syntactic understanding. A function $(\lambda x.e)$ does not delay all context requirements of the body e and may have immediate context requirements. Thus we say that e is a value if it is a value in the usual sense *and* has not immediate context requirements. We define this formally in Section 1.4.2.

The call-by-value evaluation strategy holds for a wide range of flat coeffect calculi, including all our three examples. However, it is rather weak – in order to use it, the concrete semantics needs to provide a way for reducing context-dependent term $\Gamma\,@\,\mathtt{r} \vdash e : \tau$ to a term $\Gamma\,@\,\mathtt{use} \vdash e' : \tau$ with no context requirements.

INTERNALIZED SEQUENCING. The (*internalized sequencing*) rule captures an operational semantics where the language provides a construct representing *sequential composition* and expressions can be reduced to a normal form, consisting of a sequenced context-dependent operations. We choose to write the sequencing operation as `glet` to emphasize that this is a separate primitive and not an ordinary syntactic `let`. The normal form looks as follows:

$$\mathtt{glet}\ x_1 = e_1\ \mathtt{in}\ \ldots\ \mathtt{glet}\ x_n = e_n\ \mathtt{in}\ e$$

Here, the expressions $e_1, \ldots, e_n, e$ do not contain further nested `glet` constructs. This evaluation strategy provides context-dependent counterpart to operational view of monads developed by Filinsky [23]. We discuss how expressions reduce to the normal form in Section 1.4.4

The (*internalized sequencing*) rule is useful when defining a concrete semantics for a language that provides constructs for explicitly providing the

context. For example, consider a language with implicit parameters where a parameter is defined by e_1 `with` $?p = e_2$. Semantics for such language would provide a reduction rule:

$$(\texttt{glet}\ x_1 = ?p\ \texttt{in}\ e)\ \texttt{with}\ ?p = e_1 \quad \rightsquigarrow \quad \texttt{let}\ x_1 = e_1\ \texttt{in}\ e$$

Here, the `glet` construct provides a way of sequentialising the context-requirements so that they can be discharged by matching constructs providing the required contexts. As discussed tfearlier, we focus on the general case and so we discuss when a flat coeffect calculus supports this form of evaluation (rather than looking at semantics for concrete systems).

LOCAL SOUNDNESS AND COMPLETENESS. Two desirable properties of calculi, coined by Pfenning and Davies [66], are *local soundness* and *local completeness*. They guarantee that the rules which introduce a function arrow (lambda abstraction) and eliminate it (application) are not too strong and sufficiently strong.

The local soundness property is witnessed by (call-by-name) β-reduction, which we discussed already. The local completeness is witnessed by the η-expansion rule. We discuss the flat coeffect algebra conditions under which the reduction holds in Section 1.4.3.

1.4.2 *Call-by-value evaluation*

As discussed in the previous section, call-by-value reduction can be used for most flat coeffect calculi, but it provides a very weak general model i. e. the hard work of reducing context-dependent term to a *value* has to be provided for each system. Syntactic category for values is defined as:

$$\begin{array}{lllll} v \in \mathit{SynVal} & v & ::= & x \mid c \mid (\lambda x.e) \\ n \in \mathit{NonVal} & n & ::= & e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ e \in \mathit{Expr} & e & ::= & v \mid n \end{array}$$

The category *SynVal* captures syntactic values, but a context-dependency-free value in coeffect calculus cannot be defined purely syntactically.

Definition 6. *An expression* e *is a* value, *written as* $\mathit{val}(e)$ *if it is a syntactic value, i. e.* $e \in \mathit{SynVal}$ *and it has no context-dependencies, i. e.* $\Gamma\ @\ \texttt{use} \vdash e : \tau$.

The call-by-value substitution substitutes a value, with context requirements `use`, for a variable, whose access is also annotated with `use`. Thus, it does not affect the type and context-requirements of the term:

Lemma 2 (Call-by-value substitution)**.** *In a flat coeffect calculus with a coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \texttt{use}, \texttt{ign}, \leqslant)$*, given a value* $\Gamma\ @\ \texttt{use} \vdash v : \sigma$ *and an expression* $\Gamma, x : \sigma\ @\ r \vdash e : \tau$*, then substituting* v *for* x *does not change the type and context requirements:* $\Gamma\ @\ r \vdash e[x \leftarrow v] : \tau$.

Proof. By induction over the type derivation, using the fact that x and v are annotated with `use` and that Γ is treated as a set in the flat calculus. □

The substitution lemma 2 holds for all flat coeffect systems. However, proving that call-by-value reduction preserves typing requires an additional constraint on the flat coeffect algebra, which relates the $\wedge$ and $\oplus$ operations:

$$r \wedge t \ \leqslant\ r \oplus t \qquad \text{(approximation)}$$

Intuitively, this specifies that the $\wedge$ operation (splitting of context requirements) under-approximates the actual context capabilities while the $\oplus$ operation (combining of context requirements) over-approximates the actual context requirements.

The property holds for all three systems we consider – for implicit parameters, this is an equality; for liveness and data-flow (which both use a lattice), the greatest lower bound is smaller than the least upper bound.

Assuming $\longrightarrow_{\mathsf{cbv}}$ is call-by-value reduction that reduces the term $(\lambda x.e)\,v$ to a term $e[x \leftarrow v]$, the type preservation theorem is stated as follows:

Theorem 3 (Call-by-value reduction). *In a flat coeffect system with the* (approximation) *property, if* $\Gamma \mathbin{@} r \vdash e : \tau$ *and* $e \longrightarrow_{\mathsf{cbv}} e'$ *then* $\Gamma \mathbin{@} r \vdash e' : \tau$.

Proof. Consider the typing derivation for the term $(\lambda x.e)\,v$ before reduction:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \tau_1 \mathbin{@} r \wedge t \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{t} \tau_2} \qquad \Gamma \mathbin{@} \mathsf{use} \vdash v : \tau_1}{\Gamma \mathbin{@} r \oplus (\mathsf{use} \circledast t) \vdash (\lambda x.e)\,v : \tau_2}}{\Gamma \mathbin{@} r \oplus t \vdash (\lambda x.e)\,v : \tau_2}$$

The last step simplifies the coeffect annotation using the fact that use is a unit of $\circledast$. From Lemma 2 $e[x \leftarrow v]$ has the same coeffect annotation as e. As $r \wedge t \leqslant r \oplus t$, we can apply sub-coeffecting:

$$(\mathit{sub})\ \dfrac{\Gamma \mathbin{@} r \wedge t \vdash e[x \leftarrow v] : \tau_2}{\Gamma \mathbin{@} r \oplus t \vdash e[x \leftarrow v] : \tau_2}$$

Thus, the reduction preserves type and coeffect annotation (although this may not be the *only* typing of the original term). □

1.4.3 *Call-by-name evaluation*

In the call-by-name reduction of $(\lambda x.e_1)\ e_2$, the expression e_2 is substituted for all occurrences of the variable v in an expression e_1. As discussed in Section 1.4.1, the call-by-name strategy does not *in general* preserve the type of a program, but it does preserve the typing in some interesting cases. The typing is preserved for different reasons in two of our systems, so we briefly review the concrete examples.

DATA-FLOW. The type preservation property does not hold for data-flow. This case is similar to the example shown earlier with effectful computations. As a minimal example, consider the substitution of $\mathtt{prev}\ z$ for a variable y in a function $\lambda x.y$:

$$\begin{aligned} y{:}\tau_1, z{:}\tau_1 \mathbin{@} 0 &\vdash \lambda x.y : \tau_1 \xrightarrow{0} \tau_2 && \text{(before)} \\ z{:}\tau_1 \mathbin{@} 1 &\vdash \lambda x.\mathtt{prev}\ z : \tau_1 \xrightarrow{1} \tau_2 && \text{(after)} \end{aligned}$$

After the substitution, the coeffect of the body is 1. The rule for lambda abstraction requires that $1 = \mathit{min}(r, s)$ and so the least solution is to set both r, s to 1. The substitution this affects the coeffects attached both to the function type and the overall context.

Semantically, the coeffect over-approximates the actual requirements – the code does not access previous value of the argument x. This cannot be caputred by a flat coeffect system, but can be captured using the structural system discussed in Chapter 2.

IMPLICIT PARAMETERS. In data-flow, there is no typing for the resulting expression that preserves the type of the function. However, this is not the case for all systems. Consider substituting an implicit parameter access $?p$ for a variable y:

$$y{:}\tau_1 \mathbin{@} \emptyset \vdash \lambda x.y : \tau_1 \xrightarrow{\emptyset} \tau_2 \qquad \text{(before)}$$
$$\emptyset \mathbin{@} \{?p\} \vdash \lambda x.?p : \tau_1 \xrightarrow{\emptyset} \tau_2 \qquad \text{(after)}$$

The above shows one possible typing of the body – one that does not change the coeffects of the function type and attaches all additional coeffects (implicit parameters) to the context. In case of implicit parameters (and, more generally, systems with set-like annotations) this is always possible.

LIVENESS. In liveness, the type preservation also holds, but for a different reason. Consider substituting any expression e of type τ_1 with coeffects r for a variable y:

$$y{:}\tau_1 \mathbin{@} L \vdash \lambda x.y : \tau_1 \xrightarrow{L} \tau_2 \qquad \text{(before)}$$
$$\emptyset \mathbin{@} L \vdash \lambda x.e : \tau_1 \xrightarrow{L} \tau_2 \qquad \text{(after)}$$

In the original expression, both the overall context and the function type are annotated with L, because the body contains a variable access. An expression e can always be treated as being annotated with L (because L is the top element of the lattice) and so substitution does not change any coeffects.

REDUCTION THEOREM. The above examples (implicit parameters and liveness) demonstrate two particular kinds of coeffect algebra for which typing preservation holds. Proving the type preservation separately provides more insight into how the systems work and so we choose to consider separately.

Definition 7. *We call a flat coeffect algebra* top-pointed *if* use *is the greatest (top) coeffect scalar* ($\forall r \in \mathcal{C} \,.\, r \leqslant \mathsf{use}$) *and* bottom-pointed *if it is the smallest (bottom) element* ($\forall r \in \mathcal{C} \,.\, r \geqslant \mathsf{use}$).

Liveness is an example of top-pointed coeffects as variables are annotated with L and $D \leqslant L$, while implicit parameters and data-flow are examples of bottom-pointed coeffects. For top-pointed flat coeffects, the substitution lemma holds without additional requirements:

Lemma 4 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with an aglebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$*, substituting an expression* e_s *with arbitrary coeffects* s *for a variable* x *in* e_r *does not change the coeffects of* e_r*:*

$$\Gamma \mathbin{@} s \vdash e_s : \tau_s \;\; \wedge \;\; \Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_r : \tau_r$$
$$\Rightarrow \;\; \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \vdash e_r[x \leftarrow e_s] : \tau_r$$

Proof. Using sub-coeffecting ($s \leqslant \mathsf{use}$) and a variation of Lemma 2. □

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

In a bottom pointed coeffect system, substituting e for x increases the context requirements. However, if the system satisfies the strong condition that $\wedge = \circledast = \oplus$ then the context requirements arising from the substitution can be associated with the context Γ, leaving the context requirements of a function value unchanged. As a result, substitution does not break soundness

as in effect systems. The requirement $\wedge = \circledast = \oplus$ holds for our implicit parameters example (all three operators are set union) and for other set-like coeffects. It allows the following substitution lemma:

Lemma 5 (Bottom-pointed substitution)**.** *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \circledast = \oplus$ *and the operation is also idempotent and commutative and* $r \leqslant r' \Rightarrow \forall s. r \circledast s \leqslant r' \circledast s$ *then:*

$$\begin{aligned} &\Gamma \mathbin{@} s \vdash e_s : \tau_s \quad \wedge \quad \Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_r : \tau_r \\ &\quad \Rightarrow \quad \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast s \vdash e_r[x \leftarrow e_s] : \tau_r \end{aligned}$$

Proof. By induction over $\vdash$, using the idempotent, commutative monoid structure to keep s with the free-variable context. See Appendix A.1. □

The flat system discussed here is *flexible enough* to let us always re-associate new context requirements (arising from the substitution) with the free-variable context. In contrast, the structural system discussed in Chapter 2 is *precise enough* to keep the coeffects associated with individual variables – thus preserving typing in a complementary way.

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

Theorem 6 (Call-by-name reduction)**.** *In a coeffect system that satisfies the conditions for Lemma 4 or Lemma 5, if* $\Gamma \mathbin{@} r \vdash e : \tau$ *and* $e \rightarrow_{\mathsf{cbn}} e'$ *then* $\Gamma \mathbin{@} r \vdash e' : \tau$.

Proof. For top-pointed coeffect algebra (using Lemma 4), the proof is similar to the one in Theorem 3, using the facts that $s \leqslant \mathsf{use}$ and $r \wedge t = r \oplus t$. For bottom-pointed coeffect algebra, consider the typing derivation for the term $(\lambda x.e_r)\, e_s$ before reduction:

$$\dfrac{\dfrac{\Gamma, x : \tau_s \mathbin{@} r \vdash e_r : \tau_r}{\Gamma \mathbin{@} r \vdash \lambda x.e_r : \tau_s \xrightarrow{r} \tau_r} \qquad \Gamma \mathbin{@} s \vdash e_s : \tau_s}{\Gamma \mathbin{@} r \oplus (s \circledast r) \vdash (\lambda x.e_r)\, e_s : \tau_r}$$

The derivation uses the idempotence of $\wedge$ in the first step, followed by the (*app*) rule. The type of the term after substitution, using Lemma 5 is:

$$\dfrac{\Gamma, x : \tau_s \mathbin{@} r \vdash e_r : \tau_r \qquad \Gamma \mathbin{@} s \vdash e_s : \tau_s}{\Gamma, x : \tau_r \mathbin{@} r \circledast s \vdash e_r[x \leftarrow e_s] : \tau_s}$$

From the assumptions of Lemma 5, we know that $\circledast = \oplus$ and the operation is idempotent, so trivially: $r \circledast s = r \oplus (s \circledast r)$ □

EXPANSION THEOREM. The η-expansion (local completeness) is similar to β-reduction (local soundness) in that it holds for some flat coeffect systems, but not for all. Out of the examples we discuss, it holds for implicit parameters, but does not hold for liveness and data-flow.

Recall that η-expansion turns e into $\lambda x.e\ x$. In the case of liveness, the expression e may require no variables (both immediate and latent coeffects are marked as D). However, the resulting expression $\lambda x.e\ x$ accesses a variable, marking the context and function argument as live. In case of data-flow, the coeffects are made larger by the lambda abstraction. We remedy this limitation in the next chapter.

However, the η-expansion preserves the type for implicit parameters and, more generally, for any flat coeffect algebra where $\oplus = \wedge$. Assuming $\rightarrow_\eta$ is the standard η-reduction:

Theorem 7 (η-expansion). *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \oplus$, *if* $\Gamma \mathbin{@} r \vdash e : \tau_1 \xrightarrow{s} \tau_2$ *and* $e \to_\eta e'$ *then* $\Gamma \mathbin{@} r \vdash e' : \tau_1 \xrightarrow{s} \tau_2$.

Proof. The following derivation shows that $\lambda x.f\ x$ has the same type as f:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \mathbin{@} r \vdash f : \tau_1 \xrightarrow{s} \tau_2 \quad x : \tau_1 \mathbin{@} \mathsf{use} \vdash x : \tau_1}{\Gamma, x : \tau_1 \mathbin{@} r \oplus (\mathsf{use} \circledast s) \vdash f\ x : \tau_2}}{\Gamma, x : \tau_1 \mathbin{@} r \oplus s \vdash f\ x : \tau_2}}{\Gamma, x : \tau_1 \mathbin{@} r \wedge s \vdash f\ x : \tau_2}}{\Gamma \mathbin{@} r \vdash \lambda x.f\ x : \tau_1 \xrightarrow{s} \tau_2}$$

□

The derivation starts with the expression e and derives the type for $\lambda x.e\ x$. The application yields context requirements $r \oplus s$. In order to recover the original typing, this must be equal to $r \wedge s$. Note that the derivation is showing just one possible typing – the expression $\lambda x.e\ x$ has other types – but this is sufficient for showing type preservation.

In summary, flat coeffect calculi do not *in general* permit call-by-name evaluation, but there are several cases where call-by-name evaluation can be used. Among the examples we discuss, these include liveness and implicit parameters. Moreover, for implicit parameters (and more generally, any set-like flat coeffect algebra), the η-expansion holds as well, giving us both local soundness and local completeness as coined by Pfenning and Davies [66].

1.4.4 *Internalized sequencing*

The call-by-value and call-by-name evaluation strategies discussed in the previous section are the key techniques for defining equational theory of flat coeffects. In this section, we briefly discuss another approach that follows the style of generic operational semantics designed by Filinski [23] for effectful computations.

The idea is to embed sequencing of context-dependent computations as an explicit construct into the language and define a *normal form* that consists of sequenced expressions. The reduction to the normal form is generic for all coeffect systems (satisfying certain conditions), while the reduction of the normal form is provided by each concrete coeffect system. The extended language with the `glet` construct and its typing is defined as follows:

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{glet}\ x = e_1\ \texttt{in}\ e_2 \\ n & ::= & e \mid \texttt{glet}\ x = e\ \texttt{in}\ n \\ \tau & ::= & T \mid \tau_1 \xrightarrow{r} \tau_2 \end{array}$$

$$(\mathit{glet})\ \dfrac{\Gamma \mathbin{@} s \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \mathbin{@} r \vdash e_2 : \tau_2}{\Gamma \mathbin{@} r \oplus (s \circledast r) \vdash \texttt{glet}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

The newly introduced syntactic form n represents a normal form. The construct $\texttt{glet}\ x = e\ \texttt{in}\ n$ models an explicit sequencing of an expression e, followed by an expression n. Note that `glet` can only contain further `glet` constructs in the body, but not in the argument. The typing of the `glet` is the same as the typing of ordinary `let` construct – as discussed in Section 1.2.4, for the examples we consider, this gives a *more precise* typing than the typing of the term $(\lambda x.n)\ e$.

The reduction rules that produce a normal form are shown in Figure 3. The (*eval*) rule introduces `glet` by reducing a redex $(\lambda x.e_2)\ e_1$ to an expres-

$$(\lambda x.e_2)\ e_1 \quad \rightsquigarrow \quad \texttt{glet}\ x = e_1\ \texttt{in}\ e_2 \qquad (\textit{eval})$$

$$(\texttt{glet}\ x = e_1\ \texttt{in}\ e_2)\ e_3 \quad \rightsquigarrow \quad \texttt{glet}\ x = e_1\ \texttt{in}\ (e_2\ e_3) \qquad x \notin fv(e_3) \qquad (\textit{glet-app})$$

$$\texttt{glet}\ x_2 = (\texttt{glet}\ x_1 = e_1\ \texttt{in}\ e_2)\ \texttt{in}\ e_3 \quad \rightsquigarrow \quad \texttt{glet}\ x_1 = e_1\ \texttt{in}\ (\texttt{glet}\ x_2 = e_2\ \texttt{in}\ e_3) \qquad x_1 \notin fv(e_3) \qquad (\textit{glet-glet})$$

Figure 3: Reduction to normal form.

sion representing explicit sequencing of e_1 and e_2. The remaining two rules specify how `glet` distributes with other constructs (`glet` and application). In (*glet-app*), the `glet` construct appearing inside an application is lifted to the top-level; similarly (*glet-glet*) lifts a `glet` construct nested in the argument of another `glet`.

As with *call-by-value* and *call-by-name* strategies, the *internalized sequencing* strategy can only be used with coeffect systems satisfying certain conditions. The general form of the conditions is summarized in Appendix ?. Here, we briefly consider our three examples.

CONDITIONS. The (*eval*) reduction can be safely applied for any flat coeffect calculus which satisfies the condition that the typing of the `glet` expression (shown above) is equivalent or more precise than typing of the expression $(\lambda x.e_2)\ e_1$. More formally:

Definition 8. *A flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \texttt{use}, \texttt{ign}, \leqslant)$ *is* oriented *if for all* $s, s_1, s_2, r \in \mathcal{C}$ *such that* $s = s_1 \wedge s_2$ *it is the case that* $s \circledast (s \oplus r) \leqslant s_1 \circledast (s_2 \oplus r)$.

As discussed in Section 1.2.6, this condition is satisfied for all our three examples (strictly for liveness and data-flow; and by equality for implicit parameters). For the (*glet-app*) and (*glet-glet*) rules, additional conditions arise from the typing of the original and reduced expression (showed in Appendix ?). The results are summarized in the following two tables.

Assuming $\Gamma\,@\,r_1 \vdash e_1 : \tau_1$ and $\Gamma\,@\,r_2 \vdash e_2 : \tau_3 \xrightarrow{s} \tau_2$ and $\Gamma\,@\,r_3 \vdash e_3 : \tau_3$, the typings of the original and reduced expressions in (*glet-app*) rule are:

	Before	After	Satisfied
Parameters	$max(r_1 + r_2, r_3 + s)$	$r_1 + max(r_2, r_3 + s)$	×
Liveness	$r_2 \sqcap (r_3 \sqcup s)$	$r_2 \sqcap (r_3 \sqcup s)$	✓
Data-flow	$(r_2 \cup r_1) \cup (r_3 \cup s)$	$(r_2 \cup (r_3 \cup s)) \cup r_1$	✓

Assuming $\Gamma\,@\,r_1 \vdash e_1 : \tau_1$ and $\Gamma\,@\,r_2 \vdash e_2 : \tau_2$ and $\Gamma\,@\,r_3 \vdash e_3 : \tau_3$, the typings of the original and reduced expressions in (*glet-glet*) rule are:

	Before	After	Satisfied
Parameters	$r_3 + (r_2 + r_1)$	$(r_3 + r_2) + r_1$	✓
Liveness	r_3	r_3	✓
Data-flow	$r_3 \cup (r_2 \cup r_1)$	$(r_3 \cup r_2) \cup r_1$	✓

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

$$(typ)\ \frac{\Gamma @ r \vdash e : \tau \qquad \tau <: \tau'}{\Gamma @ r \vdash e : \tau'}$$

$$(sub)\ \frac{\Gamma @ r' \vdash e : \tau \qquad r' \leqslant r}{\Gamma @ r \vdash e : \tau}$$

$\boxed{\tau <: \tau'}$

$$(sub\text{-}trans)\ \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$(sub\text{-}fun)\ \frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2' \qquad r' \geqslant r}{\tau_1 \xrightarrow{r} \tau_2 <: \tau_1' \xrightarrow{r'} \tau_2'}$$

$$(sub\text{-}refl)\ \frac{}{\tau <: \tau}$$

Figure 4: Subtyping rules for flat coeffect calculus

This means that *internalized sequencing* provides a basis for operational semantics of liveness and implicit parameters, but not for data-flow languages. For other coeffect systems, the conditions in Appendix ? have to be re-examined.

1.5 SYNTACTIC PROPERTIES AND EXTENSIONS

The flat coeffect algebra introduced in Section 1.2 requires a number of laws. The laws are required for three distinct reasons – to be able to define the categorical structure in Section 1.3, to prove equational properties in Section 1.4 and finally, to guarantee intuitive syntactic properties for constructs such as λ-abstraction and pairs in context-aware calculi.

In this section, we look at the last point. We discuss what syntactic equivalences are permitted by the properties of $\wedge$ and we extend the calculus with pairs and units and discuss their syntactic properties. In the following section, we further develop subtyping relation for the calculus.

1.5.1 *Subtyping for coeffects*

The typing rules discussed in Section 1.2.4 include sub-coeffecting rule which makes it possible to treat an expression with smaller context requirements as an expression with greater context requirements. In the corresponding categorical semantics, this means that we can *drop* some of the provided context.

Figure 4 adds sub-typing on function types, making it possible to treat a function with smaller context requirements as a function with greater context requirements. The definition uses the standard reflexive and transitive $<:$ operator. As the (*sub-fun*) shows, the function type is contra-variant in the input and co-variant in the output. The (*typ*) rule allows using sub-typing on an expression type in the coeffect calculus.

$$[\![\Gamma \mathbin{@} r \vdash e : \tau']\!] = [\![\tau <: \tau']\!] \circ [\![\Gamma \mathbin{@} r \vdash e : \tau]\!] \qquad (\textit{typ})$$

$$[\![\tau <: \tau]\!] = \mathsf{id} \qquad (\textit{sub-refl})$$

$$[\![\tau_1 <: \tau_3]\!] = [\![\tau_2 <: \tau_3]\!] \circ [\![\tau_1 <: \tau_2]\!] \qquad (\textit{sub-trans})$$

$$[\![\tau_1 \xrightarrow{r} \tau_2 <: \tau_1' \xrightarrow{r'} \tau_2']\!] = \lambda f. \quad [\![\tau_2 <: \tau_2']\!] \circ f \circ \mathsf{map}_r\ [\![\tau_1' <: \tau_1]\!] \circ \mathsf{lift}_{r',r} \qquad (\textit{sub-fun})$$

Figure 5: Semantics of subtyping for flat coeffects

SEMANTICS. We follow the same approach as with the rest of the calculus and use a categorical semantics to explain (and confirm) the design of the sub-typing rules. The semantics of a judgement $[\![\tau <: \tau']\!]$ is a function $[\![\tau]\!] \to [\![\tau']\!]$. As shown in Figure 5, the seamntics of the sub-typing rule (*typ*) then just composes the semantics of the original expression with the conversion produced by the semantics of the sub-typing judgement.

The rest of the Figure 5 shows the rules that define the semantics of $<:$. The reflexivity and transitivity are just the identity function and function composition, respectively. The (*sub-fun*) case is interesting – recall that the semantics of a function $\tau_1' \xrightarrow{r'} \tau_2'$ is $C^{r'}\tau_1' \to \tau_2'$. To build the required function, we first drop unnecessary context using $\mathsf{lift}_{r',r} : C^{r'}\tau_1' \to C^{r}\tau_1'$ and use the map_r function to transform the nested τ_1' to τ_1. Then we evaluate the original function f and turn the resulting τ_2 into the required result of type τ_2'.

1.5.2 *Alternative lambda abstraction*

In Section 1.2.1, we discussed how to reconcile two typings for lambda abstraction (for implicit parameters, the lambda function splits context requirements using $r \cup s$; for data-flow it suffices to duplicate the requirement r). We introduced the $\wedge$ operation as a way of providing the additional abstraction. Here, we identify coeffect calculi for which the simpler (duplicating) rule is sufficient.

IDEMPOTENCE. Recall that $(\mathcal{C}, \wedge)$ is a band, meaning that $\wedge$ is idempotent and associative. The idempotence means that the context requirements of the body can be required from both the declaration site and the call site. Thus, the following (*idabs*) typing is valid (for reference, it is shown side-by-side with the ordinary lambda abstraction rule):

$$(\textit{idabs})\ \frac{\Gamma, x : \tau_1 \mathbin{@} r \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{r} \tau_2} \qquad (\textit{abs})\ \frac{\Gamma, x : \tau_1 \mathbin{@} r \wedge r \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{r} \tau_2}$$

To derive (*idabs*), we use idempotence on the body annotation $r = r \wedge r$ and then use the standard (*abs*) rule. So, (*idabs*) follows from (*abs*), but the other direction is not necessarily the case. The condition below identifies coeffect calculi where (*abs*) follows from (*idabs*).

Definition 9. *A flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is* strictly oriented *if for all* $s, r \in \mathcal{C}$ *it is the case that* $r \wedge s \leqslant r$.

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

$$(proj)\ \frac{\Gamma @ r \vdash e : \tau_1 \times \tau_2}{\Gamma @ r \vdash \pi_i\ e : \tau_i}$$

$$(unit)\ \frac{}{\Gamma @ \mathsf{ign} \vdash () : \mathsf{unit}}$$

Figure 6: Typing rules for pairs and units

Remark 8. *For a flat coeffect calculus with a strictly oriented algebra, the standard (abs) rule can be derived from the (idfun) rule.*

Proof. The following derives the conclusion of (*abs*) using (*abs*), sub-coeffecting, sub-typing and the fact that the algebra is *stricly oriented*:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \tau_1 @ r \wedge s \vdash e : \tau_2}{\Gamma @ r \wedge s \vdash \lambda x.e : \tau_1 \xrightarrow{r \wedge s} \tau_2}\,(idabs)}{\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{r \wedge s} \tau_2}\,(sub)\ (r \leqslant r \wedge s)}{\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}\,(typ)\ (r \leqslant r \wedge s)$$

□

The practical consequence of the Remark 8 is that, for strictly oriented coeffect calculi (such as our liveness and data-flow computations), we can use the (*idabs*) rule and get an equivalent type system. This alternative formulation removes the non-determinism that arises from the splitting of context requirements in the original (*abs*) rule.

SYMMETRY. The $\wedge$ operation is idempotent and associative. In all of the three examples considered in this chapter, the operation is also *symmetric*. To make our definitions more general, we do not require this to be the case for *all* flat coeffect systems. However, systems with symmetric $\wedge$ have the following property.

Remark 9. *For a flat coeffect calculus such that* $r \wedge s = s \wedge r$*, assuming that* r', s', t' *is a permutation of* r, s, t*:*

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

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.

1.5.3 *Language with pairs and unit*

The calculus introduced in Section 1.2 consisted only of variables, abstraction, application and let binding to show the key aspects of flat coeffect systems. Here, we extend it with pairs and the unit value to sketch how it

can be turned into a full programming language. The syntax of the language is extended as follows:

$$\begin{array}{lcl} e & ::= & \ldots \mid () \mid e_1, e_2 \\ \tau & ::= & \ldots \mid \mathsf{unit} \mid \tau \times \tau \end{array}$$

The typing rules for pairs and the unit value are shown in Figure 6. The unit value (*unit*) is annotated with the ign coeffect (the same as other constants). Pairs, created using the (e_1, e_2) expression, are annotated with a coeffect that combines the coeffects of the two sub-expressions using the *point-wise* operator $\oplus$. The operator models the case when the (same) available context is split and passed to two independent sub-expressions. This matches the semantics of pairs discussed shortly. Finally, the (*proj*) rule is uninteresting, because π_i can be viewed as a pure function.

PROPERTIES. Pairs and the unit value in a lambda calculus should form a monoid – associativity means that the expression $(e_1, (e_2, e_3))$ should be isomorphic to $((e_1, e_2), e_3)$ and that $((), e) \simeq e \simeq (e, ())$, where the isomorphism appropriately transforms the values, without affecting other properties (here coeffects) of the expression.

In the following, we assume that assoc is a pure function transforming a pair $(x_1, (x_2, x_3))$ to a pair $((x_1, x_2), x_3)$. We write $e \equiv e'$ when for all Γ, τ and r, it is the case that $\Gamma \mathbin{@} r \vdash e : \tau$ if and only if $\Gamma \mathbin{@} r \vdash e' : \tau$.

Theorem 10. *For a flat coeffect calculus with pairs and units, the following holds:*

$$\begin{array}{rcll} \mathsf{assoc}\ (e_1, (e_2, e_3)) & \equiv & ((e_1, e_2), e_3) & \text{(associativity)} \\ \pi_1\ (e, ()) & \equiv & e & \text{(right unit)} \\ \pi_2\ ((), e) & \equiv & e & \text{(left unit)} \end{array}$$

Proof. Follows from the fact that $(\mathcal{C}, \oplus, \mathsf{ign})$ is a monoid and assoc, π_1 and π_2 are pure functions (treated as contstants in the langauge). □

The above properties follow from the laws of the flat coeffect algebra. In addition, if the $\oplus$ operation is symmetric (which is the case for all our examples in this chapter), it also holds that $\mathsf{swap}\ (e_1, e_2) \equiv (e_2, e_1)$.

1.6 RELATED WORK

Most of the related work leading to coeffects has already been discussed in Chapter **??** and we covered work related to individual concepts throughout the chapter. In this section, we do not repeat the discussion present elsewhere – we discuss one specific question that often arises when discussing coeffects and that is *when is coeffect (not) an effect?*

We start with a quick overview of the ways in which effects and coeffects differ and then we briefly look at one (but illustrative) example where the two concepts overlap. We focus mainly on the equivalence between the *categorical semantics*, which reveals the nature of the computations – rather than considering just the syntactic aspects of the type system.

1.6.1 *When is coeffect not a monad*

Coeffect systems differ from effect systems in three important ways:

- Semantically, coeffects capture very different notions of computation. As demonstrated in Chapter **??**, coeffects track additional contextual

properties required by a computation, many of which cannot be captured by a monad (e. g. liveness or data-flow).

- Syntactically, coeffect calculi use a richer algebraic structure with pointwise composition, sequential composition and context merging $(\oplus, \circledast, \bigwedge)$ while most effect systems only use a single operation for sequential composition (monadic bind).
- Syntactically, the second difference is in the lambda abstraction (*abs*). In coeffect systems, the context requirements of the body can be split between declaration-site and call-site, while monadic effect systems delay all effects.

Despite the differences, our implicit parameters example can be also represented by a monad. Semantically, the *reader* monad is equivalent to the *product* comonad. Syntactically, we use the $\cup$ operation for all three operations of the coeffect algebra. However, the last point requires us to extend monadic lambda abstraction.

1.6.2 *When is coeffect a monad*

As discussed in Section **??**, one of our examples, implicit parameters, can be also captured by a monad. However, *just* a monad is not enough because lambda abstraction in effect systems does not provide a way of splitting the context requirements between declaration-site and call-site (or combining the implicit parameters available in the scope where the function is deined and those specified by the caller).

CATEGORICAL RELATIONSHIP. Before looking at the necessary extensions, consider the two ways of modelling implicit parameters. We assume that the function $r \to \sigma$ is a lookup function for reading implicit parameter values that is defined on a set r. The two definitions are:

$$C^r\tau = \tau \times (r \to \sigma) \qquad (\textit{product comonad})$$
$$M^r\tau = (r \to \sigma) \to \tau \qquad (\textit{reader monad})$$

The *product comonad* simply pairs the value τ with the lookup function, while the *reader monad* is a function that, given a lookup function, produces a τ value. As noted by Orchard [57], when used to model computation semantics, the two representations are equivalent:

Remark 11. *Computations modelled as $C^r\tau_1 \to \tau_2$ using the product comonad are isomorphic to computations modelled as $\tau_1 \to M^r\tau_2$ using the currying/uncurrying isomoprhism.*

Proof. The isomorphism is demonstrated by the following equation:

$$\begin{aligned}
&C^r\tau_1 \to \tau_2 = \\
&(\tau_1 \times (r \to \sigma)) \to \tau_2 \\
&\tau_1 \to ((r \to \sigma) \to \tau_2) \\
&\tau_1 \to M^r\tau_2
\end{aligned}$$

□

The equivalence holds for monads and comonads (as well as *indexed* monads and comonads), but it does not extend to *flat* indexed comonads which also provide the $\mathsf{merge}_{r,s}$ operation to model context merging.

DELAYING EFFECTS IN MONADS. In the syntax of the language, the above difference is manifested in the (*abs*) rules for monadic effect systems and comonadic coeffect systems. The following listing shows the two rules side-by-side, using the coeffect notation for both of them:

$$(cabs)\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} r \cup s \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2} \qquad (mabs)\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} r \cup s \vdash e : \tau_2}{\Gamma \mathbin{@} \emptyset \vdash \lambda x.e : \tau_1 \xrightarrow{r \cup s} \tau_2}$$

In the comonadic (*cabs*) rule, the implicit parameters of the body are split. However, the monadic rule (*mabs*) places all requirements on the call-site. This follows from the fact that monadic semantics uses the unit operation in the interpretation of lambda abstraction:

$$[\![\lambda x.e]\!] \;=\; \mathsf{unit}\ (\lambda x.[\![e]\!])$$

The type of unit is $\alpha \to M^{\alpha}\emptyset$, but in this specific case, the α is instantiated to be $\tau_1 \to M^{r \cup s}\tau_2$ and so this use of unit has a type:

$$\mathsf{unit} \;:\; (\tau_1 \to M^{r \cup s}\tau_2) \to M^{\emptyset}(\tau_1 \to M^{r \cup s}\tau_2)$$

In order to split the implicit parameters of the body ($r \cup s$ on the left-hand side) between the declaration-site ($\emptyset$ on the outer M on the right-hand side) and the call-site ($r \cup s$ on the inner M on the right-hand side), we need an operation (which we call delay) with the following signature:

$$\mathsf{delay}_{r,s} \;:\; (\tau_1 \to M^{r \cup s}\tau_2) \to M^{r}(\tau_1 \to M^{s}\tau_2)$$

The operation reveals the difference between effects and coeffects – intuitively, given a function with effects $r \cup s$, it should execute the effects r when wrapping the function, *before* the function actually perforsm the effectful operation with the effects. The remaining effects s are delayed as usual, while effects r are removed from the effect annotation of the body.

Another important aspect of the signature is that the function needs to be indexed by the coeffect annotations r, s. The indices determine how the input context requirements $r \cup s$ are split – and thus guarantee determinism of the function.

The operation cannot be implemented in a useful way for most standard monads, but the reader monad is, indeed, an exception. It is not difficult to see how it can be implemented when we expand the definitions of $M^{r}\tau$:

$$\mathsf{delay}_{r,s} \;:\; (\tau_1 \to (r \cup s \to \sigma) \to \tau_2) \to ((r \to \sigma) \to \tau_1 \to (s \to \sigma) \to \tau_2)$$

RESTRICTING COEFFECTS IN COMONADS. As just demonstrated, we can extend monads so that the reader monad is capable of capturing the semantics of implicit parameters, including the splitting of implicit parameter requirements in lambda abstraction. Can we also go the other way round and *restrict* the comonadic semantics so that all requirements are delayed as in the (*mabs*) rule, thus modelling fully dynamically scoped parameters?

This is, indeed, also possible. Recall that the semantics of lambda abstraction in the flat coeffect calculus is modelled using $\mathsf{merge}_{r,s}$. The operation takes two contexts (wrapped in a comonad $C^{r}\alpha$), combines their carried values and additional contextual information (implicit parameters). To obtain the (*mabs*) rule, we can restrict the first parameter, which corresponds to the declaration-site context:

$$\mathsf{merge}_{r,s} \;:\; C^{r}\alpha \times C^{s}\beta \to C^{r \cup s}(\alpha \times \beta) \qquad (normal)$$
$$\mathsf{merge}_{r,s} \;:\; C^{\emptyset}\alpha \times C^{s}\beta \to C^{s}(\alpha \times \beta) \qquad (restricted)$$

In the (*restricted*) version of the operation, the declaration-site context requires no implicit parameters and so all implicit parameters have to be satis-

fied by the call-site. The semantics using the restricted version corresponds to the (*mabs*) rule shown above.

The idea of restricting the operations of the coeffect calculus semantics could be used more generally. We could allow any of the coeffect algebra operations $\circledast, \wedge, \oplus$ to be *partial* and thus the restricted (fully dynamically-scoped) version of implicit parameters could be obtained just by changing the definition of $\wedge$. Similarly, we could obtain e. g. a fully lexically-scoped version of the system. Similar idea has been used for the semantics of effectful computations by Tate [80].

1.7 CONCLUSIONS

This chapter presented the *flat coeffect calculus* – a unified system for tracking contextual properties that are *whole-context*, meaning that they are related to the execution environment or the enitre context in which computations are executed. This is the first of the two *coeffect calculi* developed in this thesis.

The flat coeffect calculus is parameterized by a *flat coeffect algebra* that captures the structure of the information tracked by the type system. We demonstrated how to instantiate the system to capture three specific systems discussed earlier in Chapter **??**, namely liveness, data-flow and implicit parameters.

Next, we introduced the notion of *flat indexed comonad*, which is a generalization of comonad, equipped with additional operations needed to provide categorical semantics of the flat coeffect calculus. The indices of the flat indexed comonad operations correspond to the coeffect annotations in the type system and provide a foundation for the design of the calculus.

Finally, we discussed the equational theory for flat coeffect calculus. Although each concrete instance of flat coeffect calculus models different notion of context, there are syntactic properties that hold for all flat coeffect systems satisfying certain additional conditions. In particular, two *subject reduction* theorems prove that the operational semantics for two classes of flat coeffect calculi (including liveness and implicit parameters) can be based on call-by-name reduction.

In the upcoming chapter, we move from *flat* coeffect calculi, tracking whole-context properties to *structural* coeffect calculi, tracking per-variable information, thus covering systems from the second half of Chapter **??**.

2 STRUCTURAL COEFFECT CALCULUS

As already discussed, the aim of this thesis is to identify abstractions for context-aware programming languages. We attempt to find abstractions that are general enough to capture a wide range of useful programming language features, but specific enough to let us identify interesting properties of the languages.

In Chapter **??**, we identified two notions of context. We generalized the class of flat calculi that capture whole-context properties in Chapter 1. In this chapter, we turn our attention to *structural* coeffect calculi that capture per-variable properties.

The flat coeffect system captures interesting use-cases (implicit parameters, liveness and data-flow), but provides relatively weak properties. We can define its categorical semantics, but the equational theory proofs had numerous additional requirements. For this reason, it is worthwhile to consider structural systems in a separate chapter. We will see that structural coeffects have a number of desirable properties that hold for all instances of the calculus.

2.1 INTRODUCTION

Two examples of flat systems from the previous chapter were liveness and data-flow. As discussed in **??**, these are interesting for theoretical reasons. However, tracking liveness of the whole context is not practically useful. Structural versions of liveness and data-flow let us track more fine-grained properties. Moreover, the equational theory of flat coeffect calculus did not reveal many useful properties for flat liveness and data-flow. As we show in this chapter, this is not the case with structural versions.

In this chapter, we focus on three example applications. We look at structural liveness and data-flow and we also consider calculus for bounded reuse, which checks how many times a variable is accessed and generalizes linear logics (that restrict variables to be used exactly once).

2.1.1 *Contributions*

Compared to the previous chapter, the structural coeffect calculi we consider are more homogeneous and so finding the common pattern is in some ways easier. However, the systems are somewhat more complicated as they need to keep annotations attached to individual variables. The contributions of this chapter are as follows:

- We present a *structural coeffect calculus* with a type system that is parameterized by a *structural coeffect algebra* and can be instantiated to obtain all of the three examples discussed (Section 2.2).
- We give the equational theory of the calculus. We prove the type-preservation property for all structural calculi for both call-by-name and call-by-value (Section 2.4).
- We show how to extend indexed comonads introduced in the previous section to *structural indexed comonads* and use them to give the seman-

tics of structural coeffect calculus (Section 2.3). As with the flat version, the categorical semantics provides a motivation for the design of the calculus.

2.1.2 *Related work*

In the previous chapter, we discussed the correspondence between coeffects and effects (and between comonads and monads). As noted earlier, the λ-calculus is assymetric in that an expression has multiple inputs (variables in the context), but just a single result (the resulting value) and so monads and effects have no notion directly corresponding to structural coeffect systems.

The work in this chapter is more closely related to sub-structural type systems [96]. While sub-structural systems remove some or all of *weakening*, *contraction* and *exchange* rules, our systems keep them, but use them to manipulate both the context and its annotations.

Our work follows the language semantics style in that we provide a structural semantics to the terms of ordinary λ-calculus. The most closely related work has been done in the meta-langauge style, which extends the terms and types with constructs working with the context explicitly. This includes Contextual Modal Type Theory (CMTT) [53], where variables may be of a type $A[\Psi]$ denoting a value of type A that requires context Ψ. In CMTT, $A[\Psi]$ is a first-class type, while structural coeffect systems do not expose coeffect annotations as stand-alone types.

Structural coeffect systems annotate the whole variable context with a *vector* of annotations. For example, a context with variables x and y annotated with s and t, respectively is written as $x : \tau_1, y : \tau_2 \mathbin{@} \langle s, t \rangle$. This means that the typing judgements have the same structure as those of the flat coeffect calculus. As discussed in Chapter 3, this makes it possible to unify the two systems and compose tracking of flat and structural properties.

2.2 STRUCTURAL COEFFECT CALCULUS

In the structural coeffect calculus, a vector of variables in the free-variable context is annotated with a vector of primitive (scalar) coeffect annotations. These annotations differ for different coeffect calculi and their properties are captured by the *structural coeffect scalar* definition below. The scalar annotations can be integers (how many past values we need) or annotations specifying whether a variable is live or not.

Scalar annotations are written as r, s, t (following the style used in the previous chapter). Functions always have exactly one input variable and so they are annotated with a coeffect scalar. Thus the expressions and types of structural coeffect calculi are the same as in the previous chapter (except that annotation on function type is now a structural coeffect scalar):

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

In the previous chapter, the free variable context Γ has been treated as a set. In the structural coeffect calculus, the order of variables matters. Thus we treat free variable context as a vector with a uniqueness condition. We also write $len(-)$ for the length of the vector:

$$\begin{array}{rl} \Gamma & = \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle \qquad \text{such that } \forall i, j\ .\ i \neq j \implies x_i \neq x_j \\ len(\Gamma) & = n \end{array}$$

For readability, we use the usual notation $x_1 : \tau_1, \ldots, x_1 : \tau_1 \vdash e : \tau$ for typing judgements, but the free variable context should be understood as a vector. Furthermore, the usual notation Γ_1, Γ_2 stands for the tensor product. Given $\Gamma_1 = \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle$ and $\Gamma_2 = \langle x_{n+1} : \tau_{n+1}, \ldots, x_m : \tau_m \rangle$ then $\Gamma_1, \Gamma_2 = \Gamma_1 \times \Gamma_2 = \langle x_1 : \tau_1, \ldots, x_m : \tau_m \rangle$.

The free variable contexts are annotated with vectors of structural coeffect scalars. In what follows, we write the vectors of coeffects as $\langle r_1, \ldots, r_n \rangle$. Meta-variables ranging over vectors are written as $\mathbf{r}, \mathbf{s}, \mathbf{t}$ (using bold face and colour to distinguish them from scalar meta-variables) and the length of a coeffect vector is written as $\mathit{len}(\mathbf{r})$. The structure for working with vectors of coeffects is provided by the *structural coeffect algebra* definition below.

2.2.1 *Structural coeffect algebra*

The structural coeffect scalar structure is similar to *flat coeffect algebra* with the exception that it drops the $\wedge$ operation. It only provides a monoid $(\mathcal{C}, \circledast, \mathsf{use})$ modelling sequential composition of computations and a monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ representing point-wise composition, as well as a relation $\leqslant$ that defines sub-coeffecting.

Definition 10. *A structural coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ is a set $\mathcal{C}$ together with elements $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$, relation $\leqslant$ and binary operations $\circledast, \oplus$ such that $(\mathcal{C}, \circledast, \mathsf{use})$ and $(\mathcal{C}, \oplus, \mathsf{ign})$ are monoids and $(\mathcal{C}, \leqslant)$ is a pre-order. That is, for all $r, s, t \in \mathcal{C}$:*

$$r \circledast (s \circledast t) = (r \circledast s) \circledast t \qquad \mathsf{use} \circledast r = r = r \circledast \mathsf{use} \qquad \text{(monoid)}$$

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

$$\text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t \qquad t \leqslant t \qquad \text{(pre-order)}$$

In addition, the following distributivity axioms hold:

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

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

In the flat coeffect calculus, we used the $\wedge$ operation to merge the annotations of contexts available from the declaration-site and the call-site or, in the syntactic reading, to split the context requirements.

In the structural coeffect calculus, we use a vector instead – combining and splitting of coeffects becomes just vector a concatenation or splitting, respectively, which is provided by the tensor product. The operations on vectors are indexed by integers representing the lengths of the vectors. The additional structure required by the type system for structural coeffect calculi is given by the following definition.

Definition 11. *A structural coeffect algebra is formed by a structural coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ equipped with the following additional structures:*

- *Coeffect vectors $\mathbf{r}, \mathbf{s}, \mathbf{t}$, ranging over structural coeffect scalars indexed by vector lengths $m, n \in \mathbb{N}$.*
- *An operation that constructs a vector from scalars indexed by the vector length $\langle - \rangle_n : \mathcal{C} \times \ldots \times \mathcal{C} \to \mathcal{C}^n$ and an operation that returns the vector length such that $\mathit{len}(\mathbf{r}) = n$ for $\mathbf{r} : \mathcal{C}^n$*
- *A point-wise extension of the $\circledast$ operator written as $t \circledast \mathbf{s}$ such that $t \circledast \langle r_1, \ldots, r_n \rangle = \langle t \circledast r_1, \ldots, t \circledast r_n \rangle$.*

- *An indexed tensor product* $\times_{n,m} : \mathcal{C}^n \times \mathcal{C}^m \to \mathcal{C}^{n+m}$ *that is used in both directions – for vector concatenation and for splitting – which is defined as* $\langle r_1, \ldots, r_n \rangle \times_{n,m} \langle s_1, \ldots, s_m \rangle = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$

The fact that the tensor product $\times_{n,m}$ is indexed by the lengths of the two vectors means that we can use it unambiguously for both concatenation of vectors and for splitting of vectors, provided that the lengths of the resulting vectors are known. In the following text, we usually omit the indices and write just $\mathbf{r} \times \mathbf{s}$, because the lengths of the coeffect vectors can be determined from the lengths of the matching free variable context vectors.

More generally, we could see the the coeffect annotations as a *container* [2] that supports certain operations. This approach is used in Chapter 3 as a way of unifying the flat and structural systems.

2.2.2 *Structural coeffect types*

The type system for structural coeffect calculus is similar to sub-structural type systems in how it handles free variable contexts. The *syntax-driven* rules do not implicitly allow weakening, exchange or contraction – this is done by checking the types of sub-expressions in disjoint parts of the free variable context. Unlike in sub-structural logics, our system allows weakening, exchange and contraction, but using explicit *structural* rules that perform corresponding transformation on the coeffect annoation.

SYNTAX-DRIVEN RULES. The variable access rule (*var*) annotates the corresponding variable as being used using `use`. Note that, as in sub-structural systems, the free variable context contains *only* the accessed variable. Other variables can be introduced using explicit weakening. Constants (*const*) are type checked in an empty variable context, which is annotated with an empty vector of coeffect annotations.

The (*abs*) rule assumes that the free variable context of the body can be split into a potentially empty *declaration site* and a singleton context containing the bound variable. The corresponding splitting is performed on the coeffect vector, uniquely associating the annotation s with the bound variable x. This means that the typing rule removes non-determinism present in flat coeffect systems.

In (*app*), the sub-expressions e_1 and e_2 use free variable contexts Γ_1, Γ_2 with coeffect vectors $\mathbf{r}, \mathbf{s}$, respectively. The function value is annotated with a coeffect scalar t. The coeffect annotation of the composed expression is obtained by combining the annotations associated with variables in Γ_1 and Γ_2. Variables in Γ_1 are only used to obtain the function value, resulting in coeffects $\mathbf{r}$. The variables in Γ_2 are used to obtain the argument value, which is then sequentially composed with the function, resulting in $t \circledast \mathbf{s}$.

STRUCTURAL RULES. The remaining rules, shown in Figure 7 (b), are not syntax-directed. They allow different transformation of the free variable context. We include sub-coeffecting (*sub*) as one of the rules, allowing sub-coeffecting on coeffect scalars belonging to individual variables. The remaining rules capture *weakening*, *exchange* and *contraction* known from sub-structural systems.

The (*weak*) allows adding a variable to the context, extending the coeffect vector with `ign` to mark it as unused, (*exch*) provides a way to rearrange variables in the context, performing the same reordering on the coeffect vector. Finally recall that variables in the free variable context are required to

a.) Syntax-driven typing rules:

$$(var)\ \frac{}{x{:}\tau \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau}$$

$$(const)\ \frac{c{:}\tau \in \Delta}{() \mathbin{@} \langle\rangle \vdash c : \tau}$$

$$(app)\ \frac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (t \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2}$$

$$(abs)\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \times \langle s \rangle \vdash e : \tau_2}{\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

$$(let)\ \frac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \qquad \Gamma_2, x{:}\tau_1 \mathbin{@} \mathbf{s} \times \langle t \rangle \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \mathbin{@} (t \circledast \mathbf{r}) \times \mathbf{s} \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

b.) Structural rules for context manipulation:

$$(sub)\ \frac{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s' \rangle \times \mathbf{q} \vdash e : \tau}{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s \rangle \times \mathbf{q} \vdash e : \tau} \quad (s' \leqslant s)$$

$$(weak)\ \frac{\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau}{\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \times \langle \mathsf{ign} \rangle \vdash e : \tau}$$

$$(exch)\ \frac{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle t, s \rangle \times \mathbf{q} \vdash e : \tau} \quad \begin{array}{l} len(\Gamma_1) = len(\mathbf{r}) \\ len(\Gamma_2) = len(\mathbf{s}) \end{array}$$

$$(contr)\ \frac{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau}{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s \oplus t \rangle \times \mathbf{q} \vdash e[z, y \leftarrow x] : \tau} \quad \begin{array}{l} len(\Gamma_1) = len(\mathbf{r}) \\ len(\Gamma_2) = len(\mathbf{s}) \end{array}$$

Figure 7: Type system for the structural coeffect calculus

be *unique*. The (*contr*) rule allows re-using a variable as we can type check sub-expressions using two separate varaibles and then unify them using substitution. The resulting variable is annotated with $\oplus$ and it is the only place in the structural coeffect system where context requiremens are combined, or semantically, where the same context is shared.

2.2.3 *Understanding structural coeffects*

The type system for structural coeffects appears more complicated when compared to the flat version, but it is in many ways simpler – it removes the ambiguity arising from the use of $\wedge$ in lambda abstraction and, as discussed in Section 2.4, has a cleaner equational theory.

FLAT AND STRUCTURAL CONTEXT. In flat systems, lambda abstraction splits context requirements using $\wedge$ and application combines them using $\oplus$. In the structural version, both of these are replaced with $\times$. The $\wedge$ operation is not needed, but $\oplus$ is still used in the (*contr*) rule.

This suggests that $\wedge$ and $\oplus$ serve two roles in flat coeffects. First, they are used as over- and under-approximations of $\times$. This is demonstrated by the (*approximation*) requirement introduced in Section 1.4.2, which requires that $r \wedge t \leqslant r \oplus t$. Semantically, flat abstraction combines available context, potentially discarding parts of it (under-approximation), while flat applica-

tion splits available context, potentially duplicating parts of it (over-approximation)[1].

Second, the operator $\oplus$ is used when the semantics passes the same context to multiple sub-expressions. In flat systems, this happens in (*app*) and (*pair*), because the sub-expressions may share variables. In structural systems, this is separated into an explicit contraction rule.

LET BINDING. The other aspect that makes structural systems simpler is that they remove the need for separate let binding. As discussed in Section 1.2.6, flat calculi include let binding that gives a *more precise* typing than combination of abstraction and application. This is not the case for structural coeffects.

Remark 12 (Let binding)**.** *In a structural coeffect calculus, the typing of* $(\lambda x.e_2)\ e_1$ *is equivalent to the typing of* $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

Proof. Consider the following typing derivation for $(\lambda x.e_2)\ e_1$. Note that in the last step, we apply (*exch*) repeatedly to swap Γ_1 and Γ_2.

$$\dfrac{\dfrac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \qquad \dfrac{\Gamma_2, x{:}\tau_1 \mathbin{@} \mathbf{s} \times \langle \mathbf{t} \rangle \vdash e_2 : \tau_2}{\Gamma_2 \mathbin{@} \mathbf{s} \vdash \lambda x.e_2 : \tau_1 \xrightarrow{\mathbf{t}} \tau_2}}{\Gamma_2, \Gamma_1 \mathbin{@} \mathbf{s} \times (\mathbf{t} \circledast \mathbf{r}) \vdash (\lambda x.e_2)\ e_1 : \tau_2}}{\Gamma_1, \Gamma_2 \mathbin{@} (\mathbf{t} \circledast \mathbf{r}) \times \mathbf{s} \vdash (\lambda x.e_2)\ e_1 : \tau_2}$$

The assumptions and conclusions match those of the (*let*) rule. □

2.2.4 *Examples of structural coeffects*

The structural coeffect calculus can be instantiated to obtain the structural coeffect calculi presented in Section **??**. Two of them – structural data-flow and structural liveness provide a more precise tracking of properties that can be tracked using flat systems. Formally, a flat coeffect algebra can be turned into a structural coeffect algebra (by dropping the $\wedge$ operator), but this does not always give us a meaningful system – for example, it is not clear why one would associate implicit parameters with individual variables.

On the other hand, some of the structural systems do not have a flat equivalent, typically because there is no appropriate $\wedge$ operator that could be added to form the flat coeffect algebra. This is the case, for example, for the bounded variable use.

Example 13 (Structural liveness)**.** *The structural coeffect algebra for liveness is formed by* $(\mathcal{L}, \sqcap, \sqcup, \mathsf{L}, \mathsf{D}, \sqsubseteq)$*, where* $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ *is the same two-point lattice as in the flat version, that is* $\mathsf{D} \sqsubseteq \mathsf{L}$ *with a join* $\sqcup$ *and meet* $\sqcap$.

Example 14 (Structural data-flow)**.** *In data-flow, context is annotated with natural numbers and the flat coeffect algebra is formed by* $(\mathbb{N}, +, max, 0, 0, \leqslant)$.

For the two examples that have both flat and structural version, obtaining the strucutral coeffect algebra is easy. As shown by the examples above, we simply omit the $\wedge$ operation. The laws required by a structural coeffect algebra are the same as those required by the flat version and so the above definitions are both valid. Similar construction can be used for the *optimized data-flow* example from Section 1.2.5.

It is important to note that this gives us a systems with *different* properties. The information are now tracked per-variable rather than for the enitre

1 Because of this duality, earlier version of coeffects in [63] used $\wedge$ and $\vee$.

context. For data-flow, we also need to adapt the typing rule for the `prev` construct. Here, we write $+$ for a point-wise extension of the $+$ operator, such that $\langle r_1, \ldots, r_n \rangle + k = \langle r_1 + k, \ldots, r_n + k \rangle$.

$$(\mathit{prev}) \quad \frac{\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau}{\Gamma \mathbin{@} \mathbf{r} + 1 \vdash \mathtt{prev}\ e : \tau}$$

The rule appears similar to the flat one, but there is an important difference. Because of the structural nature of the type system, it only increments the required number of values for variables that are used in the expression e. Annotations of other variables can be left unchanged.

Before looking at the semantics and equational properties of structural coeffect systems, we consider bounded variable use, which is an example of structural system that does not have a flat counterpart.

Example 15 (Bounded variable reuse)**.** *The structural coeffect algebra for tracking bounded variable use is given by* $(\mathbb{N}, *, +, 1, 0, \leqslant)$

Similarly to the structural calculus for data-flow, the calculus for bounded variable reuse annotates each variable with an integer. However, the integer denotes how many times is the variable *accessed* rather than how many *past values* are needed. The resulting type system is the one shown in Figure **??** in Chapter **??**.

2.3 CATEGORICAL MOTIVATION

When introducing structural coeffect systems in Section **??**, we included a concrete semantics of structural liveness and bounded variable reuse. In this section, we generalize the examples using the notion of *structural indexed comonad*, which is an extension of *indexed comonad* structure. As in the previous chapter, the main aim of this section is to motivate and explain the design of the structural coeffect calculus shown in Section 2.2. The semantics highlights the similarities and differences between the two systems.

Most of the differences between flat and structural systems arise from the fact that contexts in structural coeffect systems are treated as *vectors* rather than sets modelled using categorical products, so we start by discussing our treatment of vectors.

2.3.1 *Semantics of vectors*

In the flat coeffect calculus, the context is interpreted as a product and so a typing judgement $x_1 : \tau_1, \ldots, x_n : \tau_n \mathbin{@} r \vdash e : \tau$ is interpreted as a morphism $C^r(\tau_1 \times \ldots \times \tau_n) \to \tau$. In this model, we can freely transform the value contained in the context modelled using an indexed comonad C^r. For example, the function $\mathtt{map}_r\ \pi_i$ transforms a context $C^r(\tau_1 \times \ldots \times \tau_n)$ into a value $C^r\tau_i$. This changes the carried value without affecting the coeffect r.

The ability to freely transform the variable structure is not desirable in the model of structural coeffect systems. Our aim is to guarantee (by construction) that the structure of the coeffect annotations matches the structure of variables. To achieve this, we model vectors using a structure distinct from ordinary products which we denote $-\hat{\times}-$. For example, the judgement $x_1 : \tau_1, \ldots, x_n : \tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle \vdash e : \tau$ is modelled as a morphism $C^{\langle r_1, \ldots, r_n \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n) \to \tau$.

The operator is a bifunctor, but it is *not* a product in the categorical sense. In particular, there is no way to turn $\tau_1 \hat{\times} \ldots \hat{\times} \tau_n$ into τ_i (the structure does not have projections) and so there is also no way of turning

$C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ into $C^{\langle r_1,\ldots,r_n\rangle}\tau_i$, which would break the correspondence between coeffect annotations and variable structure.

The structure created using $-\hat{\times}-$ can be manipulated only using operations provided by the *strucutral indexed comonad*, which operate over variable contexts contained in an indexed comonad C^r.

In what follows, we model (finite) vectors of length n as $\tau_1 \hat{\times} \ldots \hat{\times} \tau_n$. We assume that the use of the operator can be freely re-associated. If an operation requires an input in the form $(\tau_1 \hat{\times} \ldots \hat{\times} \tau_i) \hat{\times} (\tau_{i+1} \hat{\times} \ldots \hat{\times} \tau_n)$, we call it with $(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ as an argument and assume that the appropriate transformation is inserted.

2.3.2 *Indexed comonads, revisited*

The semantics of structural coeffect calculus reuses the definition of *indexed comonad* almost without a change. The additional structure that is required for context manipulation (merging and splitting) is different and is provided by the *structural indexed comonad* structure that we introduce in this section.

Recall the definition from Section 1.3.3, which defines an indexed comonad over a monoid $(\mathcal{C}, \circledast, \mathsf{use})$ as a triple $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{r,s})$. The triple consists of a family of object mappings C^r, and two mappings that involve context-dependent morphisms of the form $C^r\tau \to \tau'$.

In the structural coeffect calculus, we work with morphisms of the form $C^r\tau \to \tau'$ representing function values (appearing in the language), but also of the form $C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n) \to \tau$, modelling expressions in a context. To capture this, we need to generalize some of the indices from *coeffect scalars* r, s, t to *coeffect vectors* $\mathbf{r}, \mathbf{s}, \mathbf{t}$.

Definition 12. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with a point-wise extension of the* $\circledast$ *operator to a vector (written as* $\mathbf{t} \circledast \mathbf{s}$*) and an operation lifting scalars to vectors* $\langle - \rangle$*, an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{\mathbf{s},\mathbf{r}})$*:*

- $C^{\mathbf{r}}$ *for all* $\mathbf{r} \in \bigcup_{m\in\mathbb{N}} \mathcal{C}^m$ *is a family of object mappings*
- $\mathsf{counit}_{\mathsf{use}}$ *is a mapping* $C^{\langle \mathsf{use} \rangle}\alpha \to \alpha$
- $\mathsf{cobind}_{\mathbf{s},\mathbf{r}}$ *is a mapping* $(C^{\mathbf{r}}\alpha \to \beta) \to (C^{\mathbf{s}\circledast\mathbf{r}}\alpha \to C^{\langle \mathbf{s} \rangle}\beta)$

The object mapping $C^{\mathbf{r}}$ is now indexed by a vector rather than by a scalar C^r as in the previous chapter. This new definition supersedes the old one, because a flat coeffect annotation can be seen as singleton vectors.

The operation $\mathsf{counit}_{\mathsf{use}}$ operates on a singleton-vector. This means that it will always return a single variable value rather than a vector created using $-\hat{\times}-$. The $\mathsf{cobind}_{\mathbf{s},\mathbf{r}}$ operation is, perhaps surprisingly, indexed by a coeffect vector and a coeffect scalar. This assymmetry is explained by the fact that the input function $(C^{\mathbf{r}}\alpha \to \beta)$ takes a vector of variables, but always produces just a single value. Thus the resulting function also takes a vector of variables, but always returns a context with singleton variable vector. In other words, α may contain $\hat{\times}$, but β may not, because the coeffect calculus has no way of constructing values containing $\hat{\times}$.

2.3.3 *Structural indexed comonads*

The flat indexed comonad structure extends indexed comonads with operations $\mathsf{merge}_{r,s}$ and $\mathsf{split}_{r,s}$ that combine or split the additional (flat) context and are annotated with the flat coeffect operations $\wedge$ and $\oplus$, respectively. In

the structural version, we use corresponding operations that operate on variable vectors represented using $\hat{\times}$ and are annotated with a tensor $\times$ which mirrors the variable structure.

The following definition also includes $\mathsf{lift}_{r',r}$, which is similar as before and models sub-coeffecting and also $\mathsf{dup}_{r,s}$ which models duplication of a variable in a context needed for the semantics of contraction:

Definition 13. *Given a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *with operations* $\langle - \rangle$ *and* $\circledast$*, a* structural indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$, $\mathsf{dup}_{r,s}$ *and* $\mathsf{lift}_{r',r}$ *where:*

- $\mathsf{merge}_{r,s}$ *is a family of mappings* $C^r\alpha \times C^s\beta \to C^{r\times s}(\alpha \hat{\times} \beta)$
- $\mathsf{split}_{r,s}$ *is a family of mappings* $C^{r\times s}(\alpha \hat{\times} \beta) \to C^r\alpha \times C^s\beta$
- $\mathsf{dup}_{r,s}$ *is a family of mappings* $C^{\langle r \oplus s\rangle}\alpha \to C^{\langle r,s\rangle}(\alpha \hat{\times} \alpha)$
- $\mathsf{lift}_{r',r}$ *is a family of mappings* $C^{\langle r'\rangle}\alpha \to C^{\langle r\rangle}\alpha$ *for all* r', r *such that* $r \leqslant r'$

Such that the following equalities hold:

$$\mathsf{merge}_{r,s} \circ \mathsf{split}_{r,s} \equiv \mathsf{id}$$
$$\mathsf{split}_{r,s} \circ \mathsf{merge}_{r,s} \equiv \mathsf{id}$$

The operations differ from those of the flat indexed comonad in that the merge and split operations are required to be inverse functions and to preserve the additional information about the context. This was not required for the flat system where the operations could under- or over-approximate. Note that the operations use $\hat{\times}$ to combine or split the contained values. This means that they operate on free-variable vectors rather than on ordinary products.

The dup mapping is a new operation that was not required for a flat calculus. It takes a variable context with a single variable annotated with $r \oplus s$, duplicates the value of the variable α and splits the additional context between the two new variables. In flat calculus, this operation has been expressed using ordinary tuple construction, which is not possible here – the returned context needs to contain a two-element vector $\alpha \hat{\times} \alpha$.

Finally, the lift mapping is almost the same as in the flat version. It operates on a singleton vector, which is equivalent to operating on a scalar as before. The operation could easily be extended to a vector in a point-wise way, but we keep it simple and perform sub-coeffecting separately on individual variables.

2.3.4 *Semantics of structural caluculus*

The concrete semantics for liveness and bounded variable use shown in Sections **??** and **??** suggests that semantics of structural coeffect calculi tend to be more complex than semantics of flat coeffect calculi. The complexity comes from the fact that we need a more expressive representation of the variable context – e. g. a vector of optional values – and that the structural system needs to pass separate variable contexts to the sub-expressions.

The latter aspect is fully captured by the semantics shown in this section. The earlier point is left to the concrete notion of structural coeffect. Our model still gives us the flexibility of defining the concrete representation of variable vectors. We explore a number of examples in Section 2.3.5 and start by looking at the unified categorical semantics defined in terms of *structural indexed comonads*.

$$[\![x{:}\tau \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau]\!]\ ctx = \mathsf{counit}_{\mathsf{use}}\ ctx \qquad (var)$$

$$[\![\Gamma \mathbin{@} \mathsf{ign} \vdash c_i : \tau]\!]\ ctx = \delta\ (c_i) \qquad (const)$$

$$\begin{aligned}&[\![\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (\mathbf{t} \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!]\ ctx = \\ &\quad \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{\mathbf{r},\mathbf{t}\circledast\mathbf{s}}\ ctx \\ &\quad \mathsf{in}\ [\![\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathbf{t}} \tau_2]\!]\ ctx_1\ (\mathsf{cobind}_{\mathbf{t},\mathbf{s}}\ [\![\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1]\!]\ ctx_2)\end{aligned} \qquad (app)$$

$$\begin{aligned}&[\![\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{\mathbf{s}} \tau_2]\!]\ ctx = \lambda v. \\ &\quad [\![\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \times \langle \mathbf{s} \rangle \vdash e : \tau_2]\!]\ (\mathsf{merge}_{\mathbf{r},\langle \mathbf{s} \rangle}\ (ctx, v))\end{aligned} \qquad (abs)$$

$$\begin{aligned}&[\![\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \times \langle \mathsf{ign} \rangle \vdash e : \tau]\!]\ ctx = \\ &\quad \mathsf{let}\ (ctx_1, \_) = \mathsf{split}_{\mathbf{r},\langle \mathsf{ign} \rangle}\ ctx\ \ \mathsf{in}\ \ [\![\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau]\!]\ ctx_1\end{aligned} \qquad (weak)$$

$$\begin{aligned}&[\![\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle \mathbf{s} \rangle \times \mathbf{q} \vdash e : \tau]\!]\ ctx = \\ &\quad [\![\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle \mathbf{s}' \rangle \times \mathbf{q} \vdash e : \tau]\!]\ (\mathsf{nest}_{\mathbf{r},\langle \mathbf{s} \rangle,\langle \mathbf{s}' \rangle,\mathbf{q}}\ \mathsf{lift}_{\mathbf{s},\mathbf{s}'}\ ctx)\end{aligned} \qquad (sub)$$

$$\begin{aligned}&[\![\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle \mathbf{t}, \mathbf{s} \rangle \times \mathbf{q} \vdash e : \tau]\!]\ ctx = \\ &\quad [\![\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle \mathbf{s}, \mathbf{t} \rangle \times \mathbf{q} \vdash e : \tau]\!] \\ &\qquad (\mathsf{nest}_{\mathbf{r},\langle \mathbf{t},\mathbf{s} \rangle,\langle \mathbf{s},\mathbf{t} \rangle,\mathbf{q}}\ \mathsf{swap}_{\mathbf{t},\mathbf{s}}\ ctx)\end{aligned} \qquad (exch)$$

$$\begin{aligned}&[\![\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle \mathbf{s} \oplus \mathbf{t} \rangle \times \mathbf{q} \vdash e[z, y \leftarrow x] : \tau]\!]\ ctx = \\ &\quad [\![\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle \mathbf{s}, \mathbf{t} \rangle \times \mathbf{q} \vdash e : \tau]\!] \\ &\qquad (\mathsf{nest}_{\mathbf{r},\langle \mathbf{s} \oplus \mathbf{t} \rangle,\langle \mathbf{s},\mathbf{t} \rangle,\mathbf{q}}\ \mathsf{dup}_{\mathbf{s},\mathbf{t}}\ ctx)\end{aligned} \qquad (contr)$$

Assuming the following auxiliary definitions:

$$\begin{aligned}&\mathsf{swap}_{\mathbf{t},\mathbf{s}} \ :\ C^{\langle \mathbf{t},\mathbf{s} \rangle}(\alpha \hat{\times} \beta) \to C^{\langle \mathbf{s},\mathbf{t} \rangle}(\beta \hat{\times} \alpha) \\ &\mathsf{swap}_{\mathbf{t},\mathbf{s}}\ ctx = \\ &\quad \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{\langle \mathbf{t} \rangle,\langle \mathbf{s} \rangle}\ ctx \\ &\quad \mathsf{in}\ \mathsf{merge}_{\langle \mathbf{s} \rangle,\langle \mathbf{t} \rangle}\ (ctx_2, ctx_1)\end{aligned}$$

$$\begin{aligned}&\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}} \ :\ (C^{\mathbf{s}}\beta \to C^{\mathbf{s}'}\beta') \to C^{\mathbf{r}\times\mathbf{s}\times\mathbf{t}}(\alpha \hat{\times} \beta \hat{\times} \gamma) \to C^{\mathbf{r}\times\mathbf{s}'\times\mathbf{t}}(\alpha \hat{\times} \beta' \hat{\times} \gamma) \\ &\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}}\ f\ ctx = \\ &\quad \mathsf{let}\ (ctx_1, ctx') = \mathsf{split}_{\mathbf{r},\mathbf{s}\times\mathbf{t}}\ ctx \\ &\quad \mathsf{let}\ (ctx_2, ctx_3) = \mathsf{split}_{\mathbf{s},\mathbf{t}}\ ctx' \\ &\quad \mathsf{in}\ \mathsf{merge}_{\mathbf{r},\mathbf{s}'\times\mathbf{t}}\ (ctx_1, \mathsf{merge}_{\mathbf{s}',\mathbf{t}}\ (f\ ctx_2, ctx_3))\end{aligned}$$

Figure 8: Categorical semantics of the structural coeffect calculus

CONTEXTS AND FUNCTIONS. In the structural coeffect calculus, expressions in context are interpreted as functions taking a vector (represented using $-\hat{\times}-$) wrapped in a structure indexed with a vector of annotations such as $C^{\mathbf{r}}$. Functions take only a single variable as an input and so the structure is annotated with a scalar, such as $C^{\mathbf{r}}$, which we treat as being equivalent to a singleton vector annotation $C^{\langle \mathbf{r} \rangle}$:

$$\begin{aligned}[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle \mathbf{r}_1, \ldots, \mathbf{r}_n \rangle \vdash e : \tau]\!] \ &:\ C^{\langle \mathbf{r}_1,\ldots,\mathbf{r}_n \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n) \to \tau \\ [\![\tau_1 \xrightarrow{\mathbf{r}} \tau_2]\!] \ &=\ C^{\langle \mathbf{r} \rangle}\tau_1 \to \tau_2\end{aligned}$$

Note that the instances of flat indexed comonad ignored the fact that the variable context wrapped in the data structure is a product. This is not generally the case for the structural indexed comonads – the definitions shown in Section 2.3.5 are given specifically for $C^{\langle \mathbf{r}_1,\ldots,\mathbf{r}_n \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ rather than

more generally for $C^{\mathbf{r}}\alpha$. The need for examining the structure of the variable context is another reason for using $-\hat{\times}-$ when interpreting expressions in contexts.

EXPRESSIONS. The semantics of strucutural coeffect calculi is shown in Figure 8. As in the previous chapter, the semantics is written in a programming language style using constructs such as let-binding rather than using a categorical (point-free) notation. As before, the semantics can be written using standard primitives (currying, uncurrying, function pairing etc.).

The following summarizes how the standard syntax-driven rules work, highlighting the differences from the flat version:

- When accessing a variable (*var*), the context now contains *only* the accessed variable and so the semantics is just $\mathsf{counit}_{\mathsf{use}}$ without a projection. Constants (*const*) are interpreted using a global dictionary δ as earlier.

- The semantics of flat function application first duplicated the context so that the same variables can be passed to both sub-expressions. This is no longer needed – the (*app*) rule splits the variables *including* the additional context into two parts. Passing the first context to the semantics of e_1 gives us a function $C^{\langle \mathbf{t} \rangle}\tau_1 \to \tau_2$.

  The argument for the function is obtained by applying $\mathsf{cobind}_{\mathbf{t},\mathbf{s}}$ to the semantics of e_2. The resulting function $C^{\mathbf{t} \circledast \mathbf{s}}(\ldots \hat{\times} \ldots \hat{\times} \ldots) \to C^{\langle \mathbf{t} \rangle}\tau_1$ is then called with the latter part of the context to obtain argument for the first function.

- The semantic of function abstraction (*abs*) is syntactically the same as in the flat version – the only difference is that we now merge a free-variable context with a singleton vector, both at the level of variable assignments and at the level of coeffect annotations.

The semantics for the non-syntax-driven rules performs transformations on the free-variable context. Weakening (*weak*) splits the context and ignores the part corresponding to the removed variable. If we were modelling the semantics in a language with a linear type system, this would require an additional operation for ignoring a context annotated with ign.

The remaining rules perform a transformation anywhere inside the free-variable vector. To simplify writing the semantics, we define a helper operation $\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}}$ that splits the variable vector into three parts, transforms the middle part and then merges them, using the newly transformed middle part.

The transformations on the middle part are quite simple. The (*sub*) rule uses $\mathsf{lift}_{\mathbf{s},\mathbf{s}'}$ to discard some of the available additional context; the (*exch*) rule swaps two single-variable contexts and the (*contr*) rule uses the $\mathsf{dup}_{\mathbf{s},\mathbf{t}}$ operation to duplicate a varaible while splitting its additional context.

PROPERTIES. As in the flat calculus, the main reason for defining the categorical semantics in this chapter is to provide validation for the design of the calculus. As we show in the next section, the discussed examples (liveness, data-flow, bounded variable reuse) form *structural indexed comonads* and so the calculus captures them correctly if the coeffect annotations in the typing rules match the indices in the semantics. More formally:

Remark 13 (Correspondence). *In all of the typing rules of the structural coeffect system, the context annotations* r *and* s *of typing judgements* $\Gamma \mathbin{@} r \vdash e : \tau$ *and function types* $\tau_1 \xrightarrow{s} \tau_2$ *correspond to the indices of mappings* C^r *and* $C^{\langle s \rangle}$ *in the corresponding semantic function defined by* $[\![\Gamma \mathbin{@} r \vdash e : \tau]\!]$.

Proof. By analysis of the semantic rules in Figure 8. □

As in the flat calculus, the primitive operations of the structural indexed comonad are all annotated with different operations provided by the coeffect annotations. This means that the semantics uniquely determines the structure of the typing rules of the strucutral coeffect calculus. Thanks to the correspondence between the product structure $\times$ of the annotations and the variable context $\hat{\times}$, the correspondence property also guarantees that variable values are split correctly, as required by the structural nature of the type system.

2.3.5 *Examples of structural indexed comonads*

The categorical semantics for structural coeffect calculus can be easily instantiated to give a semantics of a concrete calculus. In this section, we look at the three examples discussed throughout this chapter – structural liveness and data-flow and bounded variable reuse. Some aspects of the earlier two examples will be similar to flat versions discussed in Section 1.3 – they are based on the same data structures (option and a list, respectively), but the data structures are composed differently. Generally speaking – rather than having a data structure over a product of variables, we now have a vector of variables over a specific data structure.

The abstract semantics does not specify how vectors of variables should be represented, so this can vary in concrete instantiations. In all our examples, we represent a vector of variables as a product written using $\times$. To distinguish between products representing vectors and ordinary products (e. g. a product of contexts returned by split), we write vectors using $\langle a, \ldots, b \rangle$ rather than using parentheses.

DATA-FLOW. It is interesting to note that the semantics of data-flow and bounded variable both keep a product of multiple values for each variable, so they are both built around an *indexed list* data structure. However, their cobind and dup operations work differently. We start by looking at the structure modelling data-flow computations (variables written in bold face such as $\mathbf{a}_1$ range over vectors while a_1 ranges over individual values).

Example 16 (Indexed list for data-flow). *The indexed list model of data-flow computations is defined over a structural coeffect algebra* $(\mathbb{N}, +, max, 0, 0, \leqslant)$. *The data type* $C^{\langle n_1, \ldots, n_k \rangle}$ *is indexed by required number of past variables for each individual variable. It is defined over a vector of variables* $\alpha_1 \hat{\times} \ldots \hat{\times} \alpha_k$ *and it keeps a product containing a current value followed by* n_i *past values:*

$$C^{\langle n_1, \ldots, n_k \rangle}(\alpha_1 \hat{\times} \ldots \hat{\times} \alpha_k) = \underbrace{(\alpha_1 \times \ldots \times \alpha_1)}_{(n_1+1)-\text{times}} \times \ldots \times \underbrace{(\alpha_k \times \ldots \times \alpha_k)}_{(n_k+1)-\text{times}}$$

The mappings that define the structural indexed comonad include the split *and* merge *operations that are shared by the other two examples (discussed below):*

$$\mathsf{merge}_{\langle m_1,\ldots,m_k\rangle,\langle n_1,\ldots n_l\rangle}(\langle \mathbf{a_1},\ldots,\mathbf{a_k}\rangle,\langle \mathbf{b_1},\ldots,\mathbf{b_l}\rangle) = \langle \mathbf{a_1},\ldots,\mathbf{a_k},\mathbf{b_1},\ldots,\mathbf{b_l}\rangle$$

$$\mathsf{split}_{\langle m_1,\ldots,m_k\rangle,\langle n_1,\ldots n_l\rangle}\langle \mathbf{a_1},\ldots,\mathbf{a_k},\mathbf{b_1},\ldots,\mathbf{b_l}\rangle = (\langle \mathbf{a_1},\ldots,\mathbf{a_k}\rangle,\langle \mathbf{b_1},\ldots,\mathbf{b_l}\rangle)$$

The remaining mappings that are required by structural indexed comonad and capture the essence of data-flow computations are defined as:

$$\mathsf{counit}_0\ \langle\langle a_0\rangle\rangle = a_0$$

$$\mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ f\ \langle\langle a_{1,0},\ldots a_{1,m+n_1}\rangle,\ldots,\langle a_{k,0},\ldots a_{k,m+n_k}\rangle\rangle = \langle\langle\ f\langle\langle a_{1,0},\ldots a_{1,n_1}\rangle,\ldots,\langle a_{k,0},\ldots a_{k,n_k}\rangle\rangle,\ \ldots\ , f\langle\langle a_{1,m},\ldots a_{1,m+n_1}\rangle,\ldots,\langle a_{k,m},\ldots a_{k,m+n_k}\rangle\rangle\ \rangle\rangle$$

$$\mathsf{dup}_{m,n}\langle\langle a_1,\ldots,a_{max(m,n)}\rangle\rangle = \langle\langle a_1,\ldots,a_m\rangle,\langle a_1,\ldots,a_n\rangle\rangle$$

$$\mathsf{lift}_{k',k}\langle\langle a_0,\ldots,a_{k'}\rangle\rangle = \langle\langle a_0,\ldots,a_k\rangle\rangle \qquad (\text{when } k \leqslant k')$$

The definition of the indexed list data structure relies on the fact that the number of annotations corresponds to the number of variables combined using $-\hat{\times}-$. It then creates a vector of lists containing $n_i + 1$ values for i-*th* variable (the annotation represents the number of required *past* values so one more value is required).

The split and merge operations are defined separately, because they are not specific to the example. They operate on the top-level vectors of variables (without looking at the representation of the variable). This means that we can re-use the same definitions for the following two examples (with the only difference that $\mathbf{a_i}, \mathbf{b_i}$ will represent options rather than lists).

The mappings that explain how data-flow computations work are cobind (representing sequential composition) and dup (representing context sharing or parallel composition). In cobind, we get k vectors corresponding to k variables, each with $m + n_i$ values. The operation calls f m-times to obtain m past values required as the result of type $C^{\langle m\rangle}\beta$.

The dup operation needs to produce a two-varaible context containing m and n values, respectively, of the input variable. The input provides $max(m, n)$ values, so the definition is simply a matter of restriction. Finally, counit extracts the (only) value of the (only) variable and lift drops additional past values that are not required.

BOUNDED REUSE. As mentioned earlier, the semantics of calculus for bounded reuse is also based on the indexed list structure. Rather than representing possibly different past values that can be shared (*c.f.* dup), the list now represents multiple copies of the same value that cannot be shared.

Example 17 (Indexed list for bounded reuse)**.** *The indexed list model of bounded variable reuse is defined over a structural coeffect algebra* $(\mathbb{N}, *, +, 1, 0, \leqslant)$*. The data type* $C^{\langle n_1,\ldots,n_k\rangle}$ *is a vector containing* n_i *values of* i*-th variable:*

$$C^{\langle n_1,\ldots,n_k\rangle}(\alpha_1\,\hat{\times}\ldots\hat{\times}\,\alpha_k) = \underbrace{(\alpha_1\times\ldots\times\alpha_1)}_{n_1-\text{times}}\times\ldots\times\underbrace{(\alpha_k\times\ldots\times\alpha_k)}_{n_k-\text{times}}$$

The merge *and* split *operations are defined as in* indexed list for data-flow. *The operations that capture the behaviour of bounded reuse are defined as:*

$$\mathsf{counit}_1\ \langle\langle a_0\rangle\rangle = a_0$$

$$\mathsf{lift}_{k',k}\langle\langle a_0,\ldots,a_{k'}\rangle\rangle = \langle\langle a_0,\ldots,a_k\rangle\rangle \qquad (\text{when } k \leqslant k')$$

$$\mathsf{dup}_{m,n}\langle\langle a_1,\ldots,a_{m+n}\rangle\rangle = \quad \langle\langle a_1,\ldots,a_m\rangle,\langle a_{m+1},\ldots,a_{m+n}\rangle\rangle$$

$$\begin{array}{l}\mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ f\ \langle\langle a_{1,0},\ldots a_{1,m*n_1}\rangle,\ldots,\langle a_{k,0},\ldots a_{k,m*n_k}\rangle\rangle = \\ \quad \langle\ f\langle\langle a_{1,0},\ldots a_{1,n_1-1}\rangle,\ldots,\langle a_{k,0},\ldots a_{k,n_k-1}\rangle\rangle,\ \ldots\ , \\ \quad\ f\langle\langle a_{1,(m-1)*n_1},\ldots a_{1,(m-1)*n_1}\rangle,\ldots,\langle a_{k,m*n_k-1},\ldots a_{k,m*n_k-1}\rangle\rangle\ \rangle\end{array}$$

The counit and lift operations are defined as previously – variable access extracts the only value of the only variable and sub-coeffecting allows discarding multiple copies of a value that are not needed.

In the bounded variable reuse system, variable sharing is annotated with $+$ (in contrast with *max* used in data-flow). The dup operation thus splits the $m+n$ available values between two vectors of length m and n, without *sharing* a value. The cobind operation works similarly – it splits $m*n_i$ available values of each variable into m vectors containing n_i copies and then calls the f function m-times to obtain m resulting values without sharing any input value.

LIVENESS. In both data-flow and bounded reuse, the data type is defined as a vector of values obtained by applying some parameterized data type (indexed list) to types of individual variables. We can generalize this pattern and define $C^{\langle l_1,\ldots,l_n\rangle}$ in terms of D^l where D^l is a simpler indexed data type. For liveness, the definition lets us reuse the mapping used when defining the semantics of flat liveness. However, we cannot fully define the semantics of the structural version in terms of the flat version – the cobind operation is different and we need to provide the dup operation.

Example 18 (Structural indexed option)**.** *Given a structural coeffect algebra formed by* $(\{L,D\},\sqcap,\sqcup,L,D,\sqsubseteq)$ *and the indexed option data type* D^l*, such that* $D^D\alpha = 1$ *and* $D^L\alpha = \alpha$*, the data type for structural indexed option comonad is:*

$$C^{\langle n_1,\ldots,n_k\rangle}(\alpha_1\,\hat{\times}\ldots\hat{\times}\,\alpha_k) = D^{n_1}\alpha_1\times\ldots\times D^{n_k}\alpha_k$$

The merge *and* split *operations are defined as earlier. The remaining operations model variable liveness as follows:*

$$\begin{array}{l}\mathsf{cobind}_{L,\langle l_1,\ldots,l_n\rangle}\ f\ \langle a_1,\ldots,a_n\rangle = \langle f\ \langle a_1,\ldots,a_n\rangle\rangle \\ \mathsf{cobind}_{D,\langle D,\ldots,D\rangle}\ f\ \langle(),\ldots,()\rangle = \langle D\rangle\end{array}$$

$$\begin{array}{ll}\mathsf{dup}_{D,D}\langle()\rangle = \langle(),()\rangle & \mathsf{counit}_L\ \langle a\rangle = a \\ \mathsf{dup}_{L,D}\langle a\rangle = \langle a,()\rangle & \mathsf{lift}_{L,L}\langle a\rangle = \langle a\rangle \\ \mathsf{dup}_{D,L}\langle a\rangle = \langle(),a\rangle & \mathsf{lift}_{L,D}\langle a\rangle = \langle()\rangle \\ \mathsf{dup}_{L,L}\langle a\rangle = \langle a,a\rangle & \mathsf{lift}_{D,D}\langle()\rangle = \langle()\rangle\end{array}$$

When the expected result of the cobind operation is dead (second case), the operation can ignore all inputs and directly return the unit value $()$. Otherwise, it passes the vector of input variables to f as-is – no matter whether the individual values are live or dead. The L annotation is a unit with respect to $\cap$ and so the annotations expected by f are the same as those required by the result of cobind.

The dup operation resembles with the flat version of split – this is expected as duplication in the flat calculus is performed by first duplicating

the variable context (using map) and then applying split. Here, the duplication returns a pair which may or may not contain value, depending on the annotations.

Finally, counit extracts a value which is always present as guaranteed by the type $C^{\langle L \rangle}\alpha \to \alpha$. The lifting operation models sub-coeffecting which may drop an available value (second case) or behaves as identity.

2.4 EQUATIONAL THEORY

Similarly to the flat version, each concrete instance of the structural coeffect calculus has a different notion of context and thus a different operational interpretation. As before, the properties of the flat coeffect algebra guarantee that certain equational properties hold for all instances of the calculus. In this section, we look at these common properties that we get "for free" in all structural coeffect calculi.

We start the discussion by briefly considering the key aspects that make the equational theory of flat and structural coeffects different.

2.4.1 *From flat coeffects to structural coeffects*

When discussing equational theory for the flat calculus in Section 1.4, we noted that no single technique of specifying syntactic reduction works universally for all flat coeffect calculi. We considered multiple different options (call-by-name, call-by-value, internalized substitution) that can be used as the basis for operational semantics for different calculi that satisfy different additional properties.

The structural coeffect calculus has more desirable equational properties. In particular, we can prove both β-reduction and η-expansion using just the properties of strucutral coeffect algebra. For this reason, we focus on these two reductions in this section. Using the terminology of Pfenning and Davies [66], the structural coeffect calculus satisfies both the *local soundness* and the *local completeness* properties.

SUBSTITUTION FOR FLAT COEFFECTS. The less obvious (*top-pointed*) variant of the substitution lemma for flat coeffects (Lemma 5) required all operations of the flat coeffect algebra to coincide. This enables the substitution to preserve the type of expressions, because all additional requirements arising as the result of the substitution can be associated with the declaration context. For example, consider the following example where implicit parameter ?offset is substituted for the variable y:

$$\begin{aligned} y : \mathsf{int} \mathbin{@} \emptyset \vdash \lambda x.y + ?\mathsf{total} &\quad : \mathsf{int} \xrightarrow{\{?\mathsf{total}\}} \mathsf{int} \qquad \text{(before)} \\ () \mathbin{@} \{?\mathsf{offset}\} \vdash \lambda x.?\mathsf{offset} + ?\mathsf{total} &\quad : \mathsf{int} \xrightarrow{\{?\mathsf{total}\}} \mathsf{int} \qquad \text{(after)} \end{aligned}$$

The typing judgement obtained in (*after*) preserves the type of the expression (function value) from the original typing (*before*). This is possible thanks to the non-determinism involved in lambda abstraction – as all operators of the flat coeffect algebra used here are $\cup$, we can place the additional requirement on the outer context. Note that this is not the *only* possible typing, but it is *permissible* typing.

Here, the flat coeffect calculus gives us typing with limited *precision*, but enough *flexibility* to prove the substitution lemma.

SUBSTITUTION FOR STRUCTURAL COEFFECTS. In contrast, the substitution lemma for structural coeffects can be proven because structural coeffect systems provide enough *precision* to identify exactly with which variable should a context requirement be associated.

The following example shows a situation similar to the previous one – here, we use structural data-flow calculus (writing `prev` e to obtain previous value of the expression e) and we substitute $w + z$ for y:

$$\begin{aligned}
y : \mathsf{int} \mathbin{@} \langle 2 \rangle &\vdash \lambda x.\texttt{prev}\ (x + \texttt{prev}\ y) && : \mathsf{int} \xrightarrow{1} \mathsf{int} && \text{(before)} \\
w : \mathsf{int}, z : \mathsf{int} \mathbin{@} 2 * \langle 1, 1 \rangle &\vdash \lambda x.\texttt{prev}\ (x + \texttt{prev}\ (w + z)) && : \mathsf{int} \xrightarrow{1} \mathsf{int} && \text{(after)} \\
w : \mathsf{int}, z : \mathsf{int} \mathbin{@} \langle 2, 2 \rangle &\vdash \lambda x.\texttt{prev}\ (x + \texttt{prev}\ (w + z)) && : \mathsf{int} \xrightarrow{1} \mathsf{int} && \text{(final)}
\end{aligned}$$

The type of the function does not change, because the structural type system associates the annotation 1 with the bound variable x and the substitution does not affect how the variable x is used.

The other aspect demonstrated in the example is how the coeffect of the substituted variable affects the free-variable context of the substituted expression. Here, the original variable y is annotated with 2 and we substitute it for an expression $w + z$ with free variables w, z annotated with $\langle 1, 1 \rangle$. The substitution applies the operation $\circledast$ (modelling sequential composition) to the annotation of the new context – in the above example $2 * \langle 1, 1 \rangle = \langle 2, 2 \rangle$.

2.4.2 *Holes and substitution lemma*

As demonstrated in the previous section, reduction (and substitution) in the structural coeffect calculus may need to replace a *single* variable with a *vector* of variables. More importantly, because the system uses explicit contraction, we may also need to substitute for multiple variables in the variable context at the same time.

Consider the expression $\lambda x.x + x$. It is type-checked by type-checking $x_1 + x_2$, contracting x_1 and x_2 and then applying lambda abstraction. During the reduction of $(\lambda x.x + x)\ (y + z)$ we need to substitute $y_1 + z_1$ for x_1 and $y_2 + z_2$ for x_2. This is similar to substitution lemma in other structural variants of λ-calculus, such as the bunched typing system [54]. To express the substitution lemma, we define the notion of a *context with holes*:

Definition 14 (Context with holes)**.** *A* context with holes *is a context such as* $x_1 : \tau_1, \ldots, x_k : \tau_k \mathbin{@} \langle r_1, \ldots, r_k \rangle$*, where some of the variable typings* $x_i : \tau_i$ *and corresponding coeffects* r_i *are replaced by* holes *written as* $-$*.*

$$\Delta[- \mathbin{@} -]_n = \Delta[\underbrace{- \mathbin{@} - \mid \ldots \mid - \mathbin{@} -}_{n-\textit{times}}]$$

$$\begin{aligned}
\Delta[- \mathbin{@} -]_n &:= -, \Gamma \mathbin{@} \langle - \rangle \times \boldsymbol{s} && \text{where } \Gamma \mathbin{@} \boldsymbol{s} \in \Delta[- \mathbin{@} -]_{n-1} \\
\Delta[- \mathbin{@} -]_n &:= x : \tau, \Gamma \mathbin{@} \langle r \rangle \times \boldsymbol{s} && \text{where } \Gamma \mathbin{@} \boldsymbol{s} \in \Delta[- \mathbin{@} -]_n \\
\Delta[- \mathbin{@} -]_0 &:= () \mathbin{@} \langle \rangle
\end{aligned}$$

A context with n holes may either start with a hole, followed by a context with $n - 1$ holes, or it may start with a variable followed by a context with n holes. Note that the definition ensures that the locations of variable holes correspond to the locations of coeffect annotation holes. Given a context with holes, we can fill the holes with other contexts using the *hole filling* operation and obtain an ordinary coeffect-annotated context.

Definition 15 (Hole filling). *Given a context with n holes $\Delta @ \boldsymbol{s} \in \Delta[- @ -]_n$, the hole filling operation written as $\Delta @ \boldsymbol{s}[\Gamma_1 @ \mathbf{r_1} \mid \ldots \mid \Gamma_n @ \mathbf{r_n}]$, which replaces the holes by the specified variables and corresponding coeffect annotations, is defined as:*

$$
\begin{aligned}
-, \Delta @ \langle - \rangle \times \boldsymbol{s}\ [\Gamma_1 @ \boldsymbol{r_1} \mid \Gamma_2 @ \boldsymbol{r_2} \mid \ldots] &= \Gamma_1, \Gamma_2 @ \boldsymbol{r_1} \times \boldsymbol{r_2} \\
&\quad \text{where } \Gamma_2 @ \boldsymbol{r_2} = \Delta @ \boldsymbol{s}[\Gamma_2 @ \boldsymbol{r_2} \mid \ldots] \\
x_1 : \tau, \Delta @ \langle r_1 \rangle \times \boldsymbol{s}\ [\Gamma_1 @ \boldsymbol{r_1} \mid \Gamma_2 @ \boldsymbol{r_2} \mid \ldots] &= x_1 : \tau, \Gamma_2 @ \langle r_1 \rangle \times \boldsymbol{r_2} \\
&\quad \text{where } \Gamma_2 @ \boldsymbol{r_2} = \Delta @ \boldsymbol{s}[\Gamma_1 @ \boldsymbol{r_1} \mid \Gamma_2 @ \boldsymbol{r_2} \mid \ldots] \\
() @ \langle \rangle\ [\,] &= () @ \langle \rangle
\end{aligned}
$$

When we substitute an expression with coeffects $\mathbf{t}$ (associated with variables Γ) for a variable that has coeffects s, the resulting coeffects of Γ need to combine $\mathbf{t}$ and s. Unlike in the flat coeffect systems, the structural substitution does not require all coeffect algebra operations to coincide and so the combination is more interesting than in the bottom-pointed substitution for flat coeffects, where it used the only available operator (Lemma 5).

Substitution can be seen as sequential composition. Informally – we first need to obtain the value of the expression (requiring $\mathbf{t}$) and then use it in context with requirements s. Thus the free variables of the expression *after* substitution are annotated with $s \circledast \mathbf{t}$, using the (scalar-vector extension) of the sequential composition operator $\circledast$. This suggests that, to evaluate each of the variables annotated with coeffects in the vector $\mathbf{t}$, we first need to evaluate the substituted expression with coeffects s, followed by the rest of the expression with coeffects specified by $\mathbf{t}$.

Lemma 14 (Multi-nary substitution)**.** *Given an expression with multiple holes filled by variables $x_i : \tau_i$ with coeffects s_k:*

$$\Gamma @ \boldsymbol{r}\ [x_1 : \tau_1 @ \langle s_1 \rangle \mid \ldots \mid x_k : \tau_k @ \langle s_k \rangle] \vdash e_r : \tau_r$$

and a expressions e_i with free-variable contexts Γ_i annotated with $\mathbf{t_i}$:

$$\Gamma_1 @ \boldsymbol{t_1} \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma_k @ \boldsymbol{t_k} \vdash e_k : \tau_k$$

then substituting the expressions e_i for variables x_i results in an expression with a context where the original holes are filled by contexts Γ_i with coeffects $s_i \circledast \mathbf{t_i}$:

$$\Gamma @ \boldsymbol{r}\ [\Gamma_s @ s_1 \circledast \boldsymbol{t_1} \mid \ldots \mid \Gamma_s @ s_k \circledast \boldsymbol{t_k}] \vdash e_r[x_1 \leftarrow e_1] \ldots [x_k \leftarrow e_k] : \tau_r$$

Proof. By induction over $\vdash$, using the multi-nary aspect of the substitution in the proof of the contraction case. □

2.4.3 *Reduction and expansion*

In the previous chapter, we discussed call-by-value separately from call-by-name, because the proof of call-by-value substitution has fewer prerequisites. In this section, we consider full β-reduction (local soundness), which encompasses both types of evaluation and η-expansion (local completeness). Both of the properties hold for a system with any structural coeffect algebra.

REDUCTION THEOREM In a full β-reduction, written as $\rightarrow_\beta$, we replace the redex $(\lambda x.e_2)\ e_1$ by the expression $e_r[x \leftarrow e_s]$ anywhere inside a term. The subject reduction theorem guarantees that this does not change the type of the term.

Theorem 15 (β-reduction). *In a structural coeffect system with a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and operations* $\langle - \rangle$ *and* $\circledast$*, if* $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$ *and* $e \to_\beta e'$ *using the full* β*-reduction then* $\Gamma \mathbin{@} \mathbf{r} \vdash e' : \tau$.

Proof. Consider the typing derivation for the redex $(\lambda x.e_r)\, e_s$ before the reduction (note that this is similar to the typing of let binding discussed in Remark 12, but we do not swap the two parts of the free-variable context):

$$\frac{\dfrac{\Gamma_s \mathbin{@} \mathbf{s} \vdash e_s : \tau_s \qquad \dfrac{\Gamma_r, x{:}\tau_s \mathbin{@} \mathbf{r} \times \langle \mathbf{t} \rangle \vdash e_r : \tau_r}{\Gamma_r \mathbin{@} \mathbf{r} \vdash \lambda x.e_r : \tau_s \xrightarrow{\mathbf{t}} \tau_r}}{\Gamma_r, \Gamma_s \mathbin{@} \mathbf{r} \times (\mathbf{t} \circledast \mathbf{s}) \vdash (\lambda x.e_r)\ e_s : \tau_r}$$

For the substitution lemma, we first rewrite the typing judgement for e_r, i.e. $\Gamma_r, x : \tau_s \mathbin{@} \mathbf{r} \times \langle \mathbf{t} \rangle \vdash e_r : \tau_r$ as a context with a single hole filled by the x variable: $\Gamma_r, - \mathbin{@} \mathbf{r} \times - \ [\, x : \tau_s \mathbin{@} \langle \mathbf{t} \rangle \,] \vdash e_r : \tau_r$. Now we can perform the substitution using Lemma 14:

$$\frac{\Gamma_r, - \mathbin{@} \mathbf{r} \times - \ [\, x{:}\tau_s \mathbin{@} \langle \mathbf{t} \rangle \,] \vdash e_r : \tau_r \qquad \Gamma_s \mathbin{@} \mathbf{s} \vdash e_s : \tau_s}{\dfrac{\Gamma_r, - \mathbin{@} \mathbf{r} \times - \ [\, \Gamma_s \mathbin{@} \mathbf{t} \circledast \mathbf{s} \,] \vdash e_r[x \leftarrow e_s] : \tau_r}{\Gamma_r, \Gamma_s \mathbin{@} \mathbf{r} \times (\mathbf{t} \circledast \mathbf{s}) \vdash e_r[x \leftarrow e_s] : \tau_r}}$$

The last step applies the hole filling operation, showing that substitution preserves the type of the term. □

Because of the vector (free monoid) structure, coeffect annotations $\mathbf{r}$, $\mathbf{s}$, and $\langle \mathbf{t} \rangle$ are uniquely associated with Γ_r, Γ_s, and x respectively. Therefore, substituting e_s (which has coeffects $\mathbf{s}$) for x introduces the context dependencies specified by $\mathbf{s}$ which are composed with the existing requirements $\mathbf{t}$ on x.

EXPANSION THEOREM Structural coeffect systems also exhibit η-equality, therefore satisfying both *local soundness* and *local completeness*. Informally, this means that abstraction does not introduce too much, and application does not eliminate too much.

Theorem 16 (η-expansion). *In a structural coeffect system with a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and operations* $\langle - \rangle$ *and* $\circledast$*, if* $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$ *and* $e \to_\eta e'$ *using the full* η*-reduction then* $\Gamma \mathbin{@} \mathbf{r} \vdash e' : \tau$.

Proof. The following derivation shows that $\lambda x.f\ x$ has the same type as f:

$$\frac{\dfrac{\dfrac{\Gamma \mathbin{@} \mathbf{r} \vdash f : \tau_1 \xrightarrow{\mathbf{s}} \tau_2 \qquad x : \tau_1 \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau_1}{\Gamma, x : \tau_1 \mathbin{@} \mathbf{r} \times (\mathbf{s} \circledast \langle \mathsf{use} \rangle) \vdash f\ x : \tau_2}}{\Gamma, x : \tau_1 \mathbin{@} \mathbf{r} \times \langle \mathbf{s} \rangle \vdash f\ x : \tau_2}}{\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.f\ x : \tau_1 \xrightarrow{\mathbf{s}} \tau_2}$$

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

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

2.5 CONCLUSIONS

This chapter completes the key development of this thesis – the presentation of unified calculi for context-aware computations that capture the motivating examples introduced in Chapter **??**. In the previous chapter, we focused on the first category of contet-aware computations and we developed *flat coeffect calculus* that captures *whole-context* properties. This chapter develops *structural coeffect calculus*, capturing *per-variable* contextual properties. The system provides a precise analysis of liveness and data-flow and allows other interesting uses such as tracking of variable accesses based on bounded linear logic.

Following the structure of the previous chapter, the structural coeffect calculus is parameterized by a *structural coeffect algebra*. The two definitions are similar – both require operations $\circledast$ and $\oplus$ that model sequential and point-wise composition, respectively. For flat coeffects, we further require $\wedge$ to model context merging. For structural coeffects, we instead use a vector (free monoid) with the $\times$ operation – which serves a similar purpose as $\wedge$.

In order to keep track of separate annotations for each variable, we use a system with explicit structural rules (contraction, weakening and exchange) that manipulate the structure of variables and the structure of annotations at the same time.

The chapter presents two technical results. The first is the semantics of structural coeffect calculus in terms of *structural indexed comonads*. The semantics serves two purposes – it unifies our three example calculi (bounded reuse, data-flow and liveness) and it demonstrates suitability of the type system. The second technical result is equational theory for structural coeffect calculi. In particular, we show that β-reduction and η-expansion preserve the typing for any structural coeffect calculi. These two strong properties are desirable for programming languages, but are often not satisfied (e. g. by effectful langauges).

Two questions remain opened for the upcoming chapter. Firstly, is there a way to present flat and structural coeffect calculi in a uniform way as a single system? Secondly, can we simplify the definitions to make type inference easier and add other practical language aspects such as recursion?

3 UNIFIED COEFFECT LANGUAGE

3.1 INTRODUCTION

The main goal of this thesis is to provide a *unified* calculus for tracking context dependence. We have not achieved this goal yet. In Chapter **??**, we identified two kinds of contextual properties that we further covered separately – flat coeffects in Chapter 1 track whole-context properties and structural coeffects, covered in Chapter 2, track per-variable properties. In this chapter, we unify the two notions. We introduce a *unified coeffect* calculus that generalizes the two systems and can be instantiated to track both flat and structural properties (Section 3.2).

Although the results presented in this thesis are mainly of a theoretical nature, we indeed believe that coeffects should be integrated in main-stream programming languages. In the second part of this chapter, we outline one possible approach for practical implementations of coeffects (Section 3.4). Finally, we discuss an alternative approach to defining coeffect systems which highlights the relationship between our work and related work arising from modal logics (Section 3.5).

3.2 THE UNIFIED COEFFECT CALCULUS

The flat coeffect calculus (Figure 1) and the structural coeffect calculus differ in a number of ways (Figure 7). Understanding the differences is the key to reconciling the two systems:

- Structural coeffect calculus contains explicit rules for context manipulation (weakening, contraction, exchange). In flat coeffect calculus, these rules are not defined explicitly, but are admissible.
- In structural coeffect calculus, the variable context is treated as a vector and is annotated with a vector of (scalar) coeffects. In flat coeffect calculus, the variable context is a set and is annotated with a single (scalar) coeffect.
- In flat coeffect calculus, we distinguish between splitting of the context requirements and merging of context requirements ($\oplus$ and $\wedge$, respectively). In structural coeffect calculus, the operations (which model splitting and appending vectors) are invertible and so the structural coeffect algebra requires just $\times$.

In the unified calculus presented in this section, we address the three differences as follows. We use calculus with explicit rules for context manipulation. In systems that arise from flat calculi, the rules can be applied freely without changing the coeffects. We generalize the structure of coeffect annotations using the notion of a "container" which can be specialized to obtain a single annotation or a vector of annotations. Finally, we distinguish between splitting and merging of context requirements (using the notation $\between$ and $\barwedge$, respectively). For structural coeffect calculi, the two operators coincide, but for flat coeffect calculi, they provide the needed flexibility.

3.2.1 *Shapes and containers*

Our notion of a *coeffect container* is based on the idea of a container introduced by Abbott et al. [2]. Interestingly, the work on containers has later been linked to comonads by Ahman et al. [3]. Intuitively a container describes data types such as lists, trees or streams. A container is formed by shapes (e. g. lengths of lists). For every shape, we can obtain a set of positions in the container (e. g. offsets in a list of a specified length). More formally:

Definition 16. *A* container $S \triangleleft P$ *is given by a set* S *of shapes and a shape-indexed family* $P : S \to \mathsf{Set}$ *of positions.*

Well-known examples of containers include lists, non-empty lists, (unbounded) streams and singleton data type (which contains exactly one element). Two containers relevant to our work are lists and singleton data types:

- The container representing lists is given by $S \triangleleft P$ where shapes are integers $S = \mathsf{Nat}$ (lengths of a list). The set of positions for a given length n is a set of indices $Pn = \{1 \ldots n\}$.
- The contianer representing singleton data type is given by $S \triangleleft P$ where shapes are given by a singleton set $S = \{*\}$ and the set of positions for the shape $*$ contains exactly one position $P* = \{0\}$.

In the unified coeffect calculus, the structure of coeffect annotations is defined by a container with additional operations (discussed later) that links it with the free-variable context Γ.

3.2.2 *Structure of coeffects*

In structural coeffect calculus, the structure annotation was formed by a vector of coeffect sclars. The unified coeffect calculus is similar, but a *vector* is replaced with a *container*. The primitive coeffect annotations in the unified calculus are formed by a *coeffect scalar*, which remains the same as in structural coeffect calculus (Definition 10). In this section, we refer to it as *unified coeffect scalar* (and we repeat the definition below). Then we define *unified coeffect containers* which determines how coeffect scalar values are attached to the free-variable context. Finally, we define the *unified coeffect algebra* which consists of shape-indexed coeffect scalar values.

As in the structural coeffect calculus, the contexts in the unified calculus are annotated with shape-indexed coeffects, written as $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$; functions take just a single input parameter and so are annotated with scalar coeffect values $\sigma \xrightarrow{r} \tau$.

COEFFECT SCALAR. The following definition of the coeffect scalar structure repeats the Definition 10 from the previous chapter.

Definition 17. *A* unified coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, relation* $\leqslant$ *and binary operations* $\circledast, \oplus$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *and* $(\mathcal{C}, \oplus, \mathsf{ign})$ *are monoids and* $(\mathcal{C}, \leqslant)$ *is a pre-order. That is, for all* $r, s, t \in \mathcal{C}$*:*

$$r \circledast (s \circledast t) = (r \circledast s) \circledast t \qquad \mathsf{use} \circledast r = r = r \circledast \mathsf{use} \qquad \text{(monoid)}$$

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

$$\text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t \qquad t \leqslant t \qquad \text{(pre-order)}$$

As previously, the monoid $(\mathcal{C}, \circledast, \mathsf{use})$ models sequential composition; the laws guarantee an underlying category structre; use and ign represent an accessed and unused variable, respectively.

The $\oplus$ operation models combining of context requirements arising from multiple parts of a program. The meaning depends on the coeffect container. The operation can either combine requirements of individual variables (structural) or requirements attached to the whole context of multiple sub-expressions (flat).

COEFFECT CONTAINERS. The coeffect container is a container that determines how are scalar coeffect annotations attached to free-variable contexts. In addition to a container $S \triangleleft P$ with shapes and shape-indexed positions, the coeffect container also provides a mapping that returns the shape of a free-variable context. The mapping between the shape of the variable context and the shape of the coeffect annotation is not necessarily bijective. For example, coeffect annotations in flat systems have just a single shape $S = \{*\}$.

In the coeffect judgment $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$, the coeffect annotation $\mathbf{r}$ is drawn from the set of coeffect scalars $\mathcal{C}$ indexed by the shape of Γ. We write $s = \mathit{len}(\Gamma)$ for the shape corresponding to Γ. The operation Ps returns a *set* of positions and so we can write $\mathbf{r} \in Ps \to \mathcal{C}$ as a mapping from positions (defined by the shape) to scalar coeffects. We usually write this as the exponent $\mathbf{r} \in \mathcal{C}^{Ps}$.

The coeffect container is also equipped with an operation that appends shapes (when we concatenate variable contexts) and two special shapes in S representing empty context and singleton context.

Definition 18. *A* coeffect container *structure* $(S \triangleleft P, \diamond, \hat{0}, \hat{1}, \mathit{len}(-))$ *comprises a container* $S \triangleleft P$ *with a binary operation* $\diamond$ *on* S *for appending shapes, a mapping from free-variable contexts to shapes* $\mathit{len}(\Gamma) \in S$*, and elements* $\hat{0}, \hat{1} \in S$ *such that* $(S, \diamond, \hat{0})$ *is a monoid.*

The elements $\hat{0}$ *and* $\hat{1}$ *represent the shapes of empty and singleton free-variable contexts respectively. The* $\diamond$ *operation corresponds to concatentation of free-variable contexts. Given* Γ_1 *and* Γ_2 *such that* $s_1 = \mathit{len}(\Gamma_1), s_2 = \mathit{len}(\Gamma_2)$*, we require that* $s_1 \diamond s_2 = \mathit{len}(\Gamma_1, \Gamma_2)$*.*

As said earlier, we use two kinds of coeffect containers that describe the structure of vectors (for structural coeffects) and the shape of trivial singleton container (for flat coeffects):

Example 19. *Structural coeffect shape is defined as* $(S \triangleleft P, |\text{–}|, +, 0, 1)$ *where* $S = \mathbb{N}$ *and* $Pn = \{1 \ldots n\}$*. The shape mapping* $|\Gamma|$ *returns the number of variables in* Γ*. Empty and singleton contexts are annotated with* 0 *and* 1*, respectively, and shapes of combined contexts are added so that* $|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|$*.*

Therefore, a coeffect annotation is a vector $\boldsymbol{r} \in \mathcal{C}^{Pn}$ *and assigns a coeffect scalar* $\boldsymbol{r}(i) \in \mathcal{C}$ *for each position (corresponding to a variable* x_i *in the context).*

Example 20. *Flat coeffect shape is defined as* $(S \triangleleft P, |-|, \diamond, \star, \star)$*. The container is defined as a singleton data type* $S = \{\star\}$ *and* $P\star = \{0\}$ *with a constant function* $|\Gamma| = \star$ *and a trivial operation* $\star \diamond \star = \star$*.*

That is, there is a single shape $\star$ *with a single position and all free-variable contexts have the same singleton shape. Therefore, a coeffect annotation is drawn from* $\mathcal{C}^{\{\star\}}$ *which is isomorphic to* $\mathcal{C}$ *and so a coeffect scalar* $r \in \mathcal{C}$ *is associated with every free-variable context.*

Example 21. *Similarly to the previous example, we can also define a coeffect container with* no *positions, i. e.* $(S \triangleleft P, |-|, \diamond, \star, \star)$ *where* $S = \{\star\}$, $P\star = \emptyset$, $|\Gamma| = \star$ *and* $\star \diamond \star = \star$.

This reduces our system to the simply-typed λ*-calculus with no context annotations, because* $P\star = \emptyset$ *and so coeffect annotations would be from the set* $\mathcal{C}^{\emptyset}$.

UNIFIED COEFFECT ALGEBRA. The coeffect calculus annotates judgments with shape-indexed coeffect annotations. The *unified coeffect algebra* combines a coeffect scalar and coeffect container to define shape-indexed coeffects and operations for manipulating these.

The definition here reconciles the third point discussed in Section 3.2 – the fact that flat coeffects use separate operations for splitting and merging ($\oplus$ and $\wedge$) while structural coeffects use tensor $\times$. In the unified calculus, we use two operators that can, however, be coincide.

Definition 19. *Given a unified coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a coeffect container* $(S \triangleleft P, \mathit{len}(-), \diamond, \hat{0}, \hat{1})$ *a* unified coeffect algebra *extends the two structures with* $(\barwedge, \barvee, \bot)$ *where* $\bot$ *is a coeffect annotation for the empty context and* $\barwedge, \barvee$ *are families of operations that combine coeffect annotations indexed by shapes. That is* $\forall n, m \in S$*:*

$$\barvee_{m,n}, \barwedge_{m,n} \;:\; \mathcal{C}^{Pm} \times \mathcal{C}^{Pn} \to \mathcal{C}^{P(m \diamond n)}$$

A coeffect algebra induces the following three additional operations:

$$\begin{aligned} \langle - \rangle \;&:\; \mathcal{C} \to \mathcal{C}^{P\hat{1}} \\ \langle x \rangle \;&= \lambda \hat{1}.x \end{aligned}$$

$$\begin{aligned} \circledast_m \;&:\; \mathcal{C} \times \mathcal{C}^{Pm} \to \mathcal{C}^{Pm} \\ r \circledast \mathbf{s} \;&= \lambda s.r \circledast \mathbf{s}(s) \end{aligned}$$

$$\begin{aligned} \mathit{len}(-) \;&:\; \mathcal{C}^{Pm} \to m \\ \mathit{len}(\mathbf{r}) \;&= m \end{aligned}$$

The $\langle - \rangle$ operation lifts a scalar coeffect to a shape-indexed coeffect that is indexed by the singleton context shape. The $\circledast_m$ operation is a left multiplication of a vector by scalar. As we always use bold font for vectors and ordinary font for scalars (as well as distinct colour), using the same symbol is not ambiguous. We also tend to omit the subscript m and write $\circledast$.

Finally, we define $\mathit{len}()$ as an operation that returns the shape of a given shape-indexed coeffect. The only purpose is to simplify notation, as we often need to specify that shapes of variable context and coeffect match, e. g. $\mathit{len}(\mathbf{r}) = \mathit{len}(\Gamma)$.

SPLITTING AND MERGING COEFFECTS. The operators $\barvee$ and $\barwedge$ combine shape-indexed coeffects associated with two contexts. For example, assume we have Γ_1 and Γ_2 with coeffects $\mathbf{r} \in \mathcal{C}^{Pm}$ and $\mathbf{s} \in \mathcal{C}^{Pn}$. In the structural system, the context shapes m, n denote the number of variables in the two contexts. The combined context Γ_1, Γ_2 has a shape $m \diamond n$ and the combined coeffects $\mathbf{r} \barwedge \mathbf{s}, \mathbf{r} \barvee \mathbf{s} \in \mathcal{C}^{P(m \diamond n)}$ are indexed by that shape.

For structural coeffect systems such as bounded reuse, both $\barvee$ and $\barwedge$ are just the tensor product $\times$ of vectors. For flat coeffect systems, the operations can be defined independently, letting $\barwedge = \wedge$ and $\barvee = \oplus$.

The difference between $\barwedge$ and $\barvee$ is clarified by the semantics (Section 3.2.6), where $\mathbf{r} \barwedge \mathbf{s}$ is an annotation of the *codomain* of a morphism that merges

a.) Syntax-driven typing rules:

$$(var)\ \frac{}{x{:}\tau \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau}$$

$$(const)\ \frac{c{:}\tau \in \Delta}{() \mathbin{@} \bot \vdash c : \tau}$$

$$(app)\ \frac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \between (\mathbf{t} \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2}$$

$$(abs)\ \frac{\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \barwedge \langle \mathbf{s} \rangle \vdash e : \tau_2}{\Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

$$(let)\ \frac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \qquad \Gamma_2, x{:}\tau_1 \mathbin{@} \mathbf{s} \barwedge \langle \mathbf{t} \rangle \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \mathbin{@} (\mathbf{t} \circledast \mathbf{r}) \between \mathbf{s} \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

b.) Structural rules for context manipulation:

$$(sub)\ \frac{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \barwedge \langle \mathbf{s}' \rangle \barwedge \mathbf{q} \vdash e : \tau}{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \between \langle \mathbf{s} \rangle \between \mathbf{q} \vdash e : \tau} \quad (\mathbf{s}' \leqslant \mathbf{s})$$

$$(weak)\ \frac{\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau}{\Gamma, x{:}\tau_1 \mathbin{@} \mathbf{r} \between \langle \mathsf{ign} \rangle \vdash e : \tau}$$

$$(exch)\ \frac{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \mathbin{@} \mathbf{r} \barwedge \langle \mathbf{s} \rangle \barwedge \langle \mathbf{t} \rangle \barwedge \mathbf{q} \vdash e : \tau}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \between \langle \mathbf{t} \rangle \between \langle \mathbf{s} \rangle \between \mathbf{q} \vdash e : \tau} \quad \begin{array}{l} len(\Gamma_1) = len(\mathbf{r}) \\ len(\Gamma_2) = len(\mathbf{s}) \end{array}$$

$$(contr)\ \frac{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \barwedge \langle \mathbf{s} \rangle \barwedge \langle \mathbf{t} \rangle \barwedge \mathbf{q} \vdash e : \tau}{\Gamma_1, x{:}\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \between \langle \mathbf{s} \oplus \mathbf{t} \rangle \between \mathbf{q} \vdash e[z, y \leftarrow x] : \tau} \quad \begin{array}{l} len(\Gamma_1) = len(\mathbf{r}) \\ len(\Gamma_2) = len(\mathbf{s}) \end{array}$$

Figure 9: Type system for the unified coeffect calculus

the capabilities provided by two contexts (in the syntactic reading, splits the context requirements), while $\mathbf{r} \between \mathbf{s}$ is an annotation of the *domain* of a morphism that splits the capabilities of a single context into two parts (in the syntactic reading, merges their context requirements). Syntactically, this means that we always use $\barwedge$ in the rule *assumptions* and $\between$ in *conclusions*.

3.2.3 *Unified coeffect type system*

The unified coeffect system in Figure 9 resembles the structural type system shown in Figure 7. Rather than explaining the rules one-by-one, we focus on the key differences between the two.

The type system for the unified coeffect calculus is parameterized by a coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ together with a coeffect algebra $(\barwedge, \between, \bot)$ and the derived constructs $\langle - \rangle$, $len(-)$ and $\circledast$.

As in the structural system, free-variable contexts Γ are treated as vectors modulo duplicate use of variables – the associativity is built-int. The order of variables matters, but can be changed using the structural rules. The context annotations $\mathbf{r}, \mathbf{s}, \mathbf{t} \in \mathcal{C}^S$ are shape-indexed coeffects (rather than simple vectors as before). As before, functions are annotated with coeffects scalars.

DIFFERENCES FROM STRUCTURAL SYSTEM. Constants (*const*) and variable access (*var*) annotate the context with special values. Empty unused

context is annotated with $\bot \in \mathcal{C}^{\hat{0}}$, while singleton context is annotated with $\langle \mathsf{use} \rangle \in \mathcal{C}^{\hat{1}}$. Note that the shapes $\hat{0}, \hat{1}$ match the shape of the variable contexts.

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

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

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

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

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

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

3.2.4 *Structural coeffects*

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

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

Definition 20. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *a* structural coeffect system *has:*

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

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

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

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

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

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

3.2.5 *Flat coeffects*

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

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

Definition 21. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a binary* $\wedge : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *such that* $(r \wedge s) \leqslant (r \oplus s)$*, we define:*

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

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

EXAMPLES. Implicit parameters are the prime example of a flat coeffect system, but other examples include rebindable resources [71] and Haskell type classes [**?**].

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

Remark 17. *We previously described flat systems for* liveness *and* dataflow *[**?**]. Turning a structural system to flat requires finding $\wedge$ that underapproximates the capabilities of combined contexts. For dataflow, this is given by the min function as* $\mathsf{min}(r, s) \leqslant \mathsf{max}(r, s)$.

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

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

Both the declaration-site and the call-site need to provide at least the number of past values required by the body. The overapproximation means that both r *and* s *can be greater than actually required. For dataflow, we could annotate both contexts with the same coeffect, but that would require treating* $\times$ *as a partial function.*

3.2.6 *Semanitcs*

Not really - but sketch.

3.3 SEMI-LATTICE FORMULATION

3.3.1 *Semi-lattice formulation*

$$\begin{aligned}
\mathsf{counit}_{\mathsf{use}} &: C^{\mathsf{use}}\alpha \to \alpha \\
\mathsf{cobind}_{r,s} &: (C^{r}\alpha \to \beta) \to (C^{r \circledast s}\alpha \to C^{s}\beta) \\
\mathsf{merge}_{r,s} &: C^{r}\alpha \times C^{s}\beta \to C^{r \wedge s}(\alpha \times \beta) \\
\mathsf{split}_{r,s} &: C^{r \oplus s}(\alpha \times \beta) \to C^{r}\alpha \times C^{s}\beta \\
\mathsf{lift}_{r,s} &: C^{r}\alpha \to C^{s}\alpha \qquad (s \leqslant r)
\end{aligned}$$

Change to

$$\begin{aligned}
\mathsf{counit}_{\mathsf{use}} &: C^{\mathsf{use}}\alpha \to \alpha \\
\mathsf{cobind}_{r,s} &: (C^{r}\alpha \to \beta) \to (C^{r \circledast s}\alpha \to C^{s}\beta) \\
\mathsf{merge}_{r,s,t} &: C^{r}\alpha \times C^{s}\beta \to C^{t}(\alpha \times \beta) \qquad (t \leqslant r, t \leqslant s) \\
\mathsf{split}_{r,s,t} &: C^{t}(\alpha \times \beta) \to C^{r}\alpha \times C^{s}\beta \qquad (r \leqslant t, s \leqslant t) \\
\mathsf{lift}_{r,s} &: C^{r}\alpha \to C^{s}\alpha \qquad (s \leqslant r)
\end{aligned}$$

Gives us..

q

$$(\textit{var})\ \frac{x : \tau \in \Gamma}{\Gamma \mathbin{@} \mathsf{use} \vdash x : \tau}$$

$$(\textit{const})\ \frac{c : \tau \in \Delta}{\Gamma \mathbin{@} \mathsf{ign} \vdash c : \tau}$$

$$(\textit{sub})\ \frac{\Gamma \mathbin{@} r' \vdash e : \tau}{\Gamma \mathbin{@} r \vdash e : \tau} \qquad (r' \leqslant r)$$

$$(\textit{app})\ \frac{\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma \mathbin{@} s \vdash e_2 : \tau_1}{\Gamma \mathbin{@} r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2}$$

$$(\textit{abs})\ \frac{\Gamma, x : \tau_1 \mathbin{@} r \wedge s \vdash e : \tau_2}{\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}$$

$$(\textit{let})\ \frac{\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \mathbin{@} s \vdash e_2 : \tau_2}{\Gamma \mathbin{@} s \oplus (s \circledast r) \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Figure 10: Type system for the flat coeffect calculus

a

a

3.3.2 *Type inference algorithm*

3.4 TOWARDS PRACTICAL COEFFECTS

As discussed earlier, the main focus of this thesis is the development of the much needed *theory of context-aware computations* and so discussing the details of a practical implementation of coeffect tracking is beyond the scope of the thesis. However, this section briefly outlines one possible pathway towards this goal.

Many of the examples of contextual computation that we discussed earlier have been implemented as a single-purpose programming language feature (e. g. implicit parameters [46] or distributed computations [51, 17]). However, the main contribution of this thesis is that it captures *multiple* different notions of context-aware computations using just a *single* common abstraction. For this reason, we advocate that future practical implementations of coeffects should not be single-purpose language features, but instead reusable abstractions that can be instantiated with a concrete *coeffect algebra* specified by the user.

In order to do this, programming languages need to provide two features; one that allows embedding of context-aware computations themselves in programs (akin to the "do" notation in Haskell) and one that allows tracking of the contextual information in the type system.

3.4.1 *Embedding contextual computations*

The embedding of *contextual* computations in programming languages can learn from better explored emedding of *effectful* computations. In purely functional programming languages such as Haskell, effectful computations are embedded by *implementing* them and inserting the necessary (monadic) plumbing. This is made easier by the "do" notation [38] that inserts the monadic operations automatically.

The recently proposed "codo" notation [58] provides a similar plumbing for context-aware computations based on comonads. The notation is close to the semantics of our flat coeffect calculus (Chapter 1). Extending the "codo" notation to support calculi based on the structural coeffect calculus (Chapter 2) is an interesting futrue work – this requires explicitly manipulating individual context variables and application of structural rules, which is not needed in flat coeffects.

In ML-like languages, effects (and many coeffects) are built-in into the language semantics, but they can still use a special notation for explicitly marking effectful (coeffectful) blocks of code. In F#, this is done using *computation expressions* [65] that differ from the "do" notation in two ways. First, they support wider range of constructs, making it possible to wrap existing F# code in a block without other changes. Second, they support other abstractions including monads, applicative functors and monad transformers. It would be interesting to see if computation expressions can be extended to handle computations based on flat/structural indexed comonads.

More lightweight syntax for effectful computation can be obtained using techniques that automatically insert the necessary monadic plumbing (wihtout a special syntax). This has been done in the context of effecful computations [74] and similar approach would be worth exploring for coeffects.

3.4.2 *Coeffect annotations as types*

The other aspect of practical implementation of coeffects is tracking the context requirements (coeffect annotations) in the type system. To achieve this (without resorting to a single-purpose language feature) the type system needs to be able to capture various kinds of coeffect algebras. The structures used in this thesis include sets (with union or interesection), natural numbers (with addition, maximum, minimum and multiplication), two-point lattice (for liveness) and free monoids (vectors of annotations).

The work on embedding effect systems in Haskell [59] shows that the recent additions to the Haskell type system provide enough poer to implement structures such as sets at the type level. Using these to embed coeffect systems in Haskell is one fruitful future direction for applied coeffects.

In dependently-typed programming languages such as Agda [11] or Idris [12], the embedding of coeffects can be implemented more directly. However, we believe that coeffect tracking does not require full dependent types and can be made accessible in more main-stream languages. Dependent ML [97] provides an interesting example of a language with some dependent typing which is still close to its non-dependently-typed predecessor ML.

Another approach for embedding computations into the type system has been pioneered by F# *type providers* [76]. Technically, type providers are compiler extensions that generate types based on external data sources or other information in order to provide easy access to data or services. A similar approach could be used for embedding *algberas* such as coeffect algebras into the type system. A *algebra provider* would be a library that specifies the objects of the algebra, equational laws and generalization rules for type inference. This could provide an easy to use way of embedding coeffect tracking in pragmatic languages such as F#. It is worth noting that the mechanism could also subsume F# units of measure [42], which could be provided via one such *algebra provider*.

3.5 COEFFECT META-LANGUAGE

In Section **??**, we discussed two ways of using monads in programming language semantics introduced by Moggi [50]. The first approach is to use monads in the *semantics* of an effectful language. The second approach is to extend the language with (additional) monadic constructs that can then be used for writing effectful monads explicitly.

In this thesis, we focused on the first approach. In both flat and structural coeffect calculi, the term language is that of simply-typed λ-calculus, and we used (flat or structural) indexed comonads to give the semantics for the language and to derive type system for it.

In this section, we briefly discuss the alternative approach. That is, we embed indexed comonads into a λ-calculus as additional constructs. To do that, we introduce the type constructor $C^r\tau$ which represents a value τ wrapped in additional context (semantically, this corresponds to an indexed comonad) and we add language constructs that correspond to the operations of indexed comonads.

This section provides a brief sketch of coeffect meta-language to highlight the relationship between coeffects and important related work on contextual modal type theory (CMTT) [53]. Developing the system further is an interesting future research direction.

3.5.1 *Coeffects and contextual modal type theory*

As discussed in Section **??**, context-aware computations are related to modal logics – comonads have been used to model the $\Box$ modality and as a basis for meta-languages that include $\Box$ as a type constructor [10, 66, 10, 53]. Nanevski et al. [53] extend an S4 term language to a contextual modal type theory (CMTT). From the perspective of this thesis, CMTT can be viewed as a *meta-language* version of our coeffect calculus.

CONTEXT IN CMTT AND COEFFECTS. Aside from the fact that coeffect calculi use comonads for *semantics* and CMTT embeds comonads (the $\Box$ modality) into the meta-language, there are two important differences.

Firstly, the *context* in CMTT is a set of variables required by a computation, which makes CMTT useful for meta-programming and staged computations. In coeffect calculi, the context requirements are formed by a coeffect algebra, which is more general and can capture variable contexts, but also integers, two-point lattices, *etc.*

Secondly, CMTT uses different intuitive understanding of the comonad (type constructor) and the associated operations. In categorical semantics of coeffect calculi, the $C^r\tau$ constructor refers to a value of type τ *together* with additional context specified by r (e. g. list of past values or additional resources). In contrast in CMTT[1] the type $C^\Psi\tau$ models a value that *requires* the context Ψ in order to produce value τ. This also changes the interpretation of the two operations of a comonad:

$$\begin{aligned}\mathsf{counit} &: C^{\mathsf{use}}\alpha \to \alpha \\ \mathsf{cobind} &: (C^r\alpha \to \beta) \to C^{r \circledast s}\alpha \to C^s\beta\end{aligned}$$

Both readings are possible, but they give quite different meanings to the operations:

- *Coeffect interpretation.* The counit operation extracts a value and does not require any additional context; the cobind operation requires context $r \circledast s$, uses the part specified by r to evaluate the function, ending with a value β together with remaining context s.
- *CMTT interpretation.* The counit operation evaluates a computation that requires no additional context to obtain a α value; given a function that turns a computation requiring context r into a value β, the cobind operation can turn a computation that requires context $r \circledast s$ into a computation that requires just s and contains β (a part of the context requirements is eliminated by the function).

Although the different reading does not affect formal properties of the systems, it is important to understand the difference when discussing the two systems.

The sketch of a coeffect meta-language in the following section attempts to bridge the gap between coeffects and CMTT. Just like CMTT, it embeds comonads as language constructs, but it annotates them with a (flat) coeffect algebra, thus it generalizes CMTT which tracks only sets of variables. Future work on the coeffect meta-language would thus be an interesting development for both coeffect systems and CMTT.

1 To avoid using different notations, we write $C^\Psi\tau$ instead of the original $[\Psi]\tau$

3.5.2 *Coeffect meta-language*

The coeffect meta-langauge could be designed using both flat and structural indexed comonads. For simplicity, this section only discusses the flat variant. The syntax of types and terms of the langauge includes the type constructor $C^r\tau$ and four expression forms:

$$\begin{aligned}
\tau &::= \alpha \mid \tau_1 \to \tau_2 \mid C^r\tau \\
e &::= v \mid \lambda x.e \mid e_1\ e_2 \mid !e \\
&\mid \texttt{let box}\ x = e_1\ \texttt{in}\ e_2 \\
&\mid \texttt{split}\ e_1\ \texttt{into}\ x, y\ \texttt{in}\ e_2 \\
&\mid \texttt{merge}\ e_1, e_2\ \texttt{into}\ x\ \texttt{in}\ e_2
\end{aligned}$$

The $!e$ and `let box` constructs correspond to the counit and cobind operation of the comonad. To define meta-langauge for flat indexed comonads, we also include `split` and `merge` that embed the corresponding operations.

TYPES FOR COEFFECT META-LANGUAGE. The type system for the language is shown in Figure 11. The first part shows the usual typing rules for simply-typed λ-calculus. For simplicity, we omit typing rules for pairs, but those need to be present as the `merge` operation works on tuples.

The second part of the typing rules is more interesting. The (*counit*) operation extracts a value from a comonadic context and corresponds to variable access in coeffect calculi. The `let box` construct (*cobind*) takes an input e_1 with context $r \circledast s$ and a computation that turns a variable x with a context r into a value τ_2. The result is a computation that produces a τ_2 value with the remaining context specified by s. Note that the expression e_2 and e_1 corresponds to the first and second arguments of the cobind operation. The keyword `let box` is chosen following CMTT[2].

The `split` and `merge` constructs follow a similar pattern. They both apply some transformation on one or two values in a context and then add the new value as a fresh variable to the variable context. For simplicity, we omit sub-coeffecting, but it could be easily added following the pattern used elsewhere.

3.5.3 *Embedding flat coeffect calculus*

The *meta-language* approach of embedding comonads in a language is more general than the *semantics* approach. This thesis focuses on a narrower use that better guides the design of a type system for context-aware programming languages.

However, it is worth noting that (flat) coeffect calculus can be embedded in the meta-language described above. This may be desirable, e. g. when using the meta-language for reasoning about context-aware computations. We briefly consider the embedding as it illuminates the relationship between coeffects and CMTT (although it is not possible to embed coeffect calculi in CMTT because of the more general annotations structure).

Given a typing judgement $\Gamma \mathbin{@} r \vdash e : \tau$ in a flat coeffect calculus, we define $[\![\Gamma \mathbin{@} r \vdash e : \tau]\!]_v$ as its emebdding in the coeffect meta-language. Note that the translation is indexed by v, which is a name of variable used to represent the entire variable context of the source language. The translation is defined in

2 The rule is similar to the letbox rule for ICML [53, p. 14], although it differs because of our generalization of comonads where bind composes coeffect annotations rather than requiring the same annotation everywhere.

a.) Typing rules for the simply typed λ-calculus

$$(var)\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$(app)\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$(abs)\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

b.) Additional typing rules arising from *flat indexed comonads*

$$(cobind)\ \frac{\Gamma \vdash e_1 : C^{r \circledast s}\tau_1 \qquad \Gamma, x : C^{r}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let box}\ x = e_1\ \texttt{in}\ e_2 : C^{s}\tau_2}$$

$$(counit)\ \frac{\Gamma \vdash e : C^{\mathsf{use}}\tau}{\Gamma \vdash !e : \tau}$$

$$(split)\ \frac{\Gamma \vdash e_1 : C^{r \oplus s}\tau_1 \qquad \Gamma, x : C^{r}\tau_1, y : C^{s}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{split}\ e_1\ \texttt{into}\ x, y\ \texttt{in}\ e_2 : \tau_2}$$

$$(merge)\ \frac{\begin{array}{c}\Gamma \vdash e_1 : C^{r}\tau_1 \qquad \Gamma \vdash e_2 : C^{s}\tau_2 \\ \Gamma, x : C^{r \wedge s}(\tau_1 \times \tau_2) \vdash e_2 : \tau_3\end{array}}{\Gamma \vdash \texttt{merge}\ e_1, e_2\ \texttt{into}\ x\ \texttt{in}\ e_2 : \tau_3}$$

Figure 11: Type system for the (flat) coeffect meta-language

$$[\![\Gamma \mathbin{@} \mathsf{use} \vdash x_i : \tau_i]\!]_\nu = \pi_i(!\nu)$$

$$[\![\Gamma \mathbin{@} r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2]\!]_\nu =$$
$$\quad \texttt{split}\ \nu\ \texttt{into}\ \nu_s, \nu_{rt}\ \texttt{in}$$
$$\quad [\![\Gamma \mathbin{@} s \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]_{\nu_s}\ (\texttt{let box}\ \nu_r = \nu_{rt}\ \texttt{in}\ [\![\Gamma \mathbin{@} r \vdash e_2 : \tau_1]\!]_{\nu_r})$$

$$[\![\Gamma \mathbin{@} r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!]_\nu = \lambda x.$$
$$\quad \texttt{merge}\ \nu, x\ \texttt{into}\ \nu_{rs}\ \texttt{in}\ [\![(\Gamma, x : \tau_1) \mathbin{@} r \wedge s \vdash e : \tau_2]\!]_{\nu_{rs}}$$

Figure 12: Embedding flat coeffect calculus in coeffect meta-language

Figure 12. The embedding resembles the semantics discussed in Section 1.3. This is not surprising as the meta-language directly mirrors the operations of a monad.

3.6 CONCLUSIONS

4 CONCLUSIONS

Some of the most fundamental academic work is not the one solving hard research problems, but the one that changes how we understand the world. Some philosophers argue that *language* is the key for understanding how we think, while in science the dominant thinking is determined by *paradigms* [44] or *research programmes* [45]. In a way, programming languages play a similar role for computer science and software development.

This thesis aims to change how developers and programming language designers think about *context* or *execution environments* of programs. Such context or execution environment has numerous forms – mobile applications access network, GPS location or user's personal data; journalists obtain information published on the web or through open government data initiatives. This thesis aims to change our understanding of *context* so that the above examples are viewed uniformly through a single programming language abstraction, which we call *coeffects*, rather than as disjoint cases. This is done through the following three contributions.

4.1 UNIFIED PRESENTATION.

In λ-calculus, the term *context* is usually used for the free-variable context. However, there are many other programming language features that are related to context or program's execution environment. In Chapter **??**, we revisit many of such features – including resource tracking in distributed computing, cross-platform development, data-flow programming and liveness analysis, but also Haskell's type classes and implicit parameters.

The main contribution of the chapter is that it presents the disjoint language features in a unified way. We show type systems and semantics for many of the languages, illuminating the fact that they are closely related.

Considering applications is one way of approaching the theory of coeffects introduced in this thesis. Other pathways to coeffects are discussed in Chapter **??**, which looks at theoretical developments leading to coeffects, including the work on effect systems, comonadic semantics and linear logics.

4.2 FLAT COEFFECT CALCULUS.

The applications discussed in Chapter **??** fall into two categories. In the first category (Section **??**), the additional contextual information are *whole-context* properties. They either capture properties of the execution environment or affect the whole free-variable context.

In Chapter 1, we develop a *flat coeffect calculus* which gives us an unified way of tracking *whole-context* properties. The calculus is parameterized by a *flat coeffect algebra* that captures the algebraic properties that contextual information satisfy. Concrete instances of flat coeffects include Haskell's implicit parameters, whole-context liveness and whole-context data-flow.

Our focus is on the syntactic properties of the calculus – we give a type system (Section 1.2) and discuss equational theory of the calculus (Section 1.4). In the flat coeffect calculus, β-reduction and η-expansion do not generally preserve type of expressions, but we identify two conditions when

this is the case – this gives us a basis for operational semantics of our calculi for liveness and implicit parameters. The design of the type system is further validated by categorical semantics (Section 1.3) which models flat coeffect calculus in terms of *flat indexed comonad* – a structure based on categorical dual of monads.

4.3 STRUCTURAL COEFFECT CALCULUS.

In the second category of context-aware systems discussed in Section **??**, contextual properties are *per-variable*. The systems discussed here resemble sub-structural logics, but rather than *restricting* variable use, they *track* additional information about how variables are used.

We unify systems with *per-variable* contextual properties in Chapter 2, which describes our *structural coeffect calculus* (Section 2.2). Similarly to the flat variant, the calculus is parameterized by a *structural coeffect algebra*. Concrete instances of the calculus track bounded variable use (i. e. how many times is variable accessed), data-flow properties (how many past values are needed) and liveness (i. e. can variable be accessed).

The structural coeffect calculus has desirable equational properties (Section 2.4). In particular, type preservation for β-reduction and η-expansion holds for all structural coeffect calculi. This follows from the fact that structural coeffect associates contextual requirements with individual variables and preserves the connection by including explicit structural rules (weakening, exchange and contraction).

The Chapters 1 and 2 present the main novel technical contributions of this thesis. In Chapter 3, we develop the two calculi further by introducing a unified presentation of the two (by parameterizing the calculus by the *shape* of context) and we present an alternative presentation that makes type inference easier.

4.4 SUMMARY

BIBLIOGRAPHY

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

[2] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[3] D. Ahman, J. Chapman, and T. Uustalu. When is a container a comonad? In *Proceedings of the 15th international conference on Foundations of Software Science and Computational Structures*, FOSSACS'12, pages 74–88, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] A. W. Appel. *Modern compiler implementation in ML*. Cambridge University Press, 1998.

[5] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19, 2009.

[6] J. E. Bardram. The java context awareness framework (jcaf)–a service infrastructure and programming framework for context-aware applications. In *Pervasive Computing*, pages 98–115. Springer, 2005.

[7] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 361–365. IEEE, 2004.

[9] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 99–110, New York, NY, USA, 2003. ACM.

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

[11] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[12] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[13] S. Brookes and S. Geva. Computational comonads and intensional semantics. Applications of Categories in Computer Science. London Mathematical Society Lecture Note Series, Cambridge University Press, 1992.

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

[15] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. In *Proceedings of the 11th international conference on Database programming languages*, DBPL'07, pages 138–152, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] J. Clarke. *SQL Injection Attacks and Defense*. Syngress, 2009.

[17] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. FMCO '00, 2006.

[18] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[19] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999.

[20] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, May 2001.

[21] Developers (Android). Creating multiple APKs for different API levels. http://developer.android.com/training/multiple-apks/api.html, 2013.

[22] W. Du and L. Wang. Context-aware application programming for mobile devices. In *Proceedings of the 2008 C3S2E conference*, C3S2E '08, pages 215–227, New York, NY, USA, 2008. ACM.

[23] A. Filinski. Monads in action. POPL, pages 483–494, 2010.

[24] A. Filinski. Towards a comprehensive theory of monadic effects. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 1–1, 2011.

[25] C. Flanagan and M. Abadi. Types for Safe Locking. ESOP '99, 1999.

[26] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of Conference on Programming Language Design and Implementation*, PLDI '03.

[27] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, 1991.

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

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

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

[31] Google. What is API level. Retrieved from http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels.

[32] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[33] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Eng.*, 34(1):65–81, Jan. 2008.

[34] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

[35] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[36] M. Hicks, J. T. Moore, and S. Nettles. *Dynamic software updating*, volume 36. ACM, 2001.

[37] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[38] S. L. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[39] P. Jouvelot and D. K. Gifford. Communication Effects for Message-Based Concurrency. Technical report, Massachusetts Institute of Technology, 1989.

[40] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645, New York, NY, USA, 2014. ACM.

[41] A. Kennedy. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*, pages 268–305. Springer, 2010.

[42] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 442–455. ACM, 1997.

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

[44] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago Press, 1970.

[45] I. Lakatos. *Methodology of Scientific Research Programmes: Philosophical Papers: v. 1*. Cambridge University Press.

[46] 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.

[47] F. Loitsch and M. Serrano. Hop client-side compilation. *Trends in Functional Programming, TFP*, pages 141–158, 2007.

[48] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[49] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[50] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, July 1991.

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

[52] T. Murphy VII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. LICS '04, pages 286–295, 2004.

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

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

[55] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer-Verlag.

[56] D. Orchard. Programming contextual computations.

[57] D. Orchard. Should I use a Monad or a Comonad? Unpublished draft, 2012.

[58] D. Orchard and A. Mycroft. A notation for comonads. In *Implementation and Application of Functional Languages*, pages 1–17. Springer, 2013.

[59] D. Orchard and T. Petricek. Embedding effect systems in haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24, 2014.

[60] T. Petricek. Client-side scripting using meta-programming.

[61] T. Petricek. Evaluations strategies for monadic computations. In *Proceedings of Mathematically Structured Functional Programming*, MSFP 2012.

[62] T. Petricek. Understanding the world with f#. Available at `http://channel9.msdn.com/posts/Understanding-the-World-with-F`.

[63] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: unified static analysis of context-dependence. In *Proceedings of International Conference on Automata, Languages, and Programming - Volume Part II*, ICALP 2013.

[64] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 123–135, 2014.

[65] T. Petricek and D. Syme. The f# computation expression zoo. In *Proceedings of Practical Aspects of Declarative Languages*, PADL 2014.

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

[67] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 13–24, 2008.

[68] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.

[69] T. Sans and I. Cervesato. QWeSST for Type-Safe Web Programming. In *Third International Workshop on Logics, Agents, and Mobility*, LAM'10, 2010.

[70] M. Serrano. Hop, a fast server for the diffuse web. In *Coordination Models and Languages*, pages 1–26. Springer, 2009.

[71] 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.

[72] V. Simonet. Flow caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

[73] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *ACM SIGPLAN Notices*, volume 40, pages 183–194. ACM, 2005.

[74] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ml. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 15–27, New York, NY, USA, 2011. ACM.

[75] D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.

[76] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming*, DDFP '13, pages 1–4, 2013.

[77] D. Syme, A. Granicz, and A. Cisternino. Building mobile web applications. In *Expert F# 3.0*, pages 391–426. Springer, 2012.

[78] D. Syme, T. Petricek, and D. Lomov. The f# asynchronous programming model. In *Practical Aspects of Declarative Languages*, pages 175–189. Springer, 2011.

[79] J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS'92.*, pages 162–173, 1994.

[80] R. Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

[81] P. Thiemann. A unified framework for binding-time analysis. In *TAPSOFT'97: Theory and Practice of Software Development*, pages 742–756. Springer, 1997.

[82] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[83] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[84] T. Uustalu and V. Vene. The essence of dataflow programming. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 2–18, Berlin, Heidelberg, 2005. Springer-Verlag.

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

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

[87] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

[88] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[89] J. Vouillon and V. Balat. From bytecode to javassript: the js_of_ocaml compiler. *Software: Practice and Experience*, 2013.

[90] B. Wadge. Monads and intensionality. In *International Symposium on Lucid and Intensional Programming*, volume 95, 1995.

[91] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[92] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM, 1988.

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

[94] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

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

[96] D. Walker. *Substructural Type Systems*, pages 3–43. MIT Press.

[97] H. Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(02):215–286, 2007.

APPENDIX A

This appendinx provides additional details for some of the proofs for equational theory of flat coeffect calculus from Chapter 1 and structural coeffect calculus from Chapter 2.

A.1 SUBSTITUTION FOR FLAT COEFFECTS

In Section 1.4.3, we stated that, for a bottom-pointed flat coeffect algebra (i.e. $\forall r \in \mathcal{C} \,.\, r \geqslant \mathsf{use}$), the call-by-name substitution preserves type if all operators of the flat coeffect algebra coincide (Lemma 5). This section provides the corresponding proof.

Lemma (Bottom-pointed substitution)**.** *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \circledast = \oplus$ *and the operation is also idempotent and commutative and* $r \leqslant r' \Rightarrow \forall s. r \circledast s \leqslant r' \circledast s$ *then:*

$$\Gamma \mathbin{@} S \vdash e_s : \tau_s \quad \wedge \quad \Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_r : \tau_r$$
$$\Rightarrow \quad \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e_r[x \leftarrow e_s] : \tau_r$$

Proof. Assume that $\Gamma \mathbin{@} S \vdash e_s : \tau_s$ and we are substituting a term e_s for a variable x. Note that we use upper-case S to distinguish the coeffect of the expression that is being substituted into an expression. Using structural induction over $\vdash$:

(VAR) Given the following derivation using (*var*):

$$\frac{}{\Gamma_1, y : \tau, \Gamma_2 \mathbin{@} \mathsf{use} \vdash y : \tau}$$

There are two cases depending on whether y is the variable x or not:

- If $y = x$ then also $\tau = \tau_s$ and thus $y[x \leftarrow e_s] = e_s$. Using the assumption, implicit weakening and the fact that use is a unit of $\circledast$:

$$\frac{\dfrac{\Gamma \mathbin{@} S \vdash y[x \leftarrow e_s] : \tau_s}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} S \vdash y[x \leftarrow e_s] : \tau}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} \mathsf{use} \circledast S \vdash y[x \leftarrow e_s] : \tau}$$

- If $y \neq x$ then $y[x \leftarrow e_s] = y$. Using the fact that use is the bottom element and sub-coeffecting:

$$\frac{\Gamma_1, y : \tau, \Gamma_2 \mathbin{@} \mathsf{use} \vdash y : \tau}{\Gamma_1, y : \tau, \Gamma_2 \mathbin{@} \mathsf{use} \circledast S \vdash y : \tau}$$

(CONST) Similar to the (*var*) case when the variable is not substituted.

(SUB) Given the following typing derivation using (*sub*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r' \vdash e : \tau}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e : \tau} \qquad (r' \leqslant r)$$

From the induction hypothesis, we have that $\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r' \circledast S \vdash e[x \leftarrow e_s] : \tau$. The condition on $\leqslant$ means that $r' \circledast S \leqslant r \circledast S$ and so we can apply the (*sub*) rule to obtain $\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e[x \leftarrow e_s] : \tau$.

(ABS) Given the following typing derivation using (*abs*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2, y : \tau_1 \mathbin{@} r \wedge s \vdash e : \tau_2}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash \lambda y.e : \tau_1 \xrightarrow{s} \tau_2}$$

Assume w.l.o.g. that $x \neq y$. From the induction hypothesis, we have that:

$$\Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} r \circledast s \vdash e[x \leftarrow e_s] : \tau_2$$

Now using the fact that $\wedge = \circledast$, associativity and commutativity and (*abs*):

$$\frac{\dfrac{\dfrac{\Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} (r \wedge s) \circledast S \vdash e[x \leftarrow e_s] : \tau_2}{\Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} (r \circledast S) \wedge s \vdash e[x \leftarrow e_s] : \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash \lambda y.(e[x \leftarrow e_s]) : \tau_1 \xrightarrow{s} \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash (\lambda y.e)[x \leftarrow e_s] : \tau_1 \xrightarrow{s} \tau_2}$$

(APP) Given the following typing derivation using (*app*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} s \vdash e_2 : \tau_1}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2}$$

From the induction hypothesis, we have that:

$$\begin{array}{l} \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e_1[x \leftarrow e_s] : \tau_1 \xrightarrow{t} \tau_2 \\ \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} s \circledast S \vdash e_2[x \leftarrow e_s] : \tau_1 \end{array} \quad (*)$$

Now using (*app*) rule and the fact that $\oplus = \circledast$, associativity, commutativity and idempotence (note that all three properties are needed):

$$\frac{\dfrac{(*)}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (r \circledast S) \oplus ((s \circledast S) \circledast t) \vdash e_1[x \leftarrow e_s]\ e_2[x \leftarrow e_s] : \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (r \oplus (s \circledast t)) \circledast S \vdash (e_1\ e_2)[x \leftarrow e_s] : \tau_2}$$

(LET) Given the following typing derivation using (*let*):

$$\frac{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} r \vdash e_1 : \tau_1 \qquad \Gamma_1, x : \tau_s, \Gamma_2, y : \tau_1 \mathbin{@} s \vdash e_2 : \tau_2}{\Gamma_1, x : \tau_s, \Gamma_2 \mathbin{@} s \oplus (s \circledast r) \vdash \texttt{let}\ y = e_1\ \texttt{in}\ e_2 : \tau_2}$$

From the induction hypothesis, we have that:

$$\begin{array}{l} \Gamma_1, \Gamma, \Gamma_2 \mathbin{@} r \circledast S \vdash e_1[x \leftarrow e_s] : \tau_1 \\ \Gamma_1, \Gamma, \Gamma_2, y : \tau_1 \mathbin{@} s \circledast S \vdash e_2[x \leftarrow e_s] : \tau_2 \end{array} \quad (\dagger)$$

Now using (*let*) rule and similarly to the (*app*) case:

$$\frac{\dfrac{(\dagger)}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (s \circledast S) \oplus ((s \circledast S) \circledast (r \circledast S)) \vdash \texttt{let}\ y = e_1[x \leftarrow e_s]\ \texttt{in}\ e_2[x \leftarrow e_s] : \tau_2}}{\Gamma_1, \Gamma, \Gamma_2 \mathbin{@} (s \oplus (s \circledast r)) \circledast S \vdash (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)[x \leftarrow e_s] : \tau_2}$$

□