CONTENTS

Part II

COEFFECT CALCULI

This part presents the key novel contributions of the thesis. We develop *flat* and *structural* coeffect calculi that capture a wide range of context-aware programming languages. We discuss their type systems and semantics together with safety properties.

4 TYPES FOR FLAT COEFFECTS

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 systems* capturing whole-context properties and *structural systems* capturing per-variable properties. As we show in Section X, the systems can be further unified using a single abstraction, 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, we discuss *flat coeffects* and *structural coeffects* separately.

In this and the next chapter, we discuss the type system and the semantics of flat coeffect systems, respectively. In this chapter, we develop a parameterized type system for flat coeffect systems and we study its general syntactic properties. We also consider variations of the type system that resolve the ambiguity of coeffectful lambda abstraction for concrete instances of the system. In the next chapter, we give operational meaning of concrete coeffect languages using the unified system and we discuss their safety.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We present a *flat coeffect calculus* as a type system that is parameterized by a *flat coeffect algebra* (Section 4.2). We show that the system can be instantiated to obtain three of the systems discussed in Section **??**, namely implicit parameters, liveness and dataflow.
- The type system permits multiple typing derivation for certain programs due to the ambiguity inherent in conextual lambda abstraction rule. In Section 4.3, we discuss variations of the type system that resolve the ambiguity and give a unique typing derivation for the three coeffect systems covered in this chapter.
- We discuss syntactic properties of the calculus, covering type-preservation for call-by-name and call-by-value reduction (Section 4.4). We also extend the calculus with subtyping and pairs (Section 4.5). These two sections motivate the laws of the flat coeffect algebra.

4.1 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. First, their lambda abstraction duplicates the requirements. Given a body with context requirements r, the declaration site context *as well as* the function arrow are annotated with r. Second, their application arises from the combination of *sequential* and *pointwise* composition.

The system for tracking of implicit parameters and similar (Section **??**) 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 $\cup$.

Despite the differences, the systems fit the same unified framework. This becomes apparent when we consider the categorical structure (Section **??**). However, rather than starting from the semantics, we first explain how the systems can be unified syntactically (Section 4.2.1) and then provide the semantics as a justification.

The development in this chapter can be seen as a counterpart to the well-known development of *effect systems* [3]. The Chapter 5 then links *coeffects* with *comonads* in the same way in which effect systems can be linked with monads [8]. The syntax and type system of the flat coeffect calculus follows a similar style as effect systems [6, 15], but differs in the structure, as explained in the previous chapter, most importantly in lambda abstraction (the relationship with monads is further discussed in Section **??**).

4.2 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, versions represented as integers or other values. The expressions of the calculus are those of the λ-calculus with *let* binding. We also include a type `num` as an example of a concrete base type with numerical constants written as n:

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

Note that the lambda abstrction in the syntax is written in the Church-style and requires a type annotation. This will be used in Section 4.3 where we discuss how to find a unique typing derivation for context-aware computations. Using Church-style lambda abstraction, we can directy focus on the more interesting problem of finding unique *coeffect annotations* rather than solving the problem of type reconstruction.

We discuss subtyping and pairs in Section 4.5. 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 categorically-inspired translation in the next chapter, the function $\tau_1 \xrightarrow{r} \tau_2$ is translated into a function $C^r\tau_1 \to \tau_2$. However, the type constructor C^r does not itself exist as a syntactical value in the coeffect calculus. 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 has been discussed in Section **??**). The annotations r are formed by an algebraic structure discussed next.

4.2.1 *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):

$$(\textit{param})\ \frac{\Gamma, x{:}\tau_1\ @\ r \cup s \vdash e : \tau_2}{\Gamma\ @\ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2} \qquad (\textit{df1})\ \frac{\Gamma, x{:}\tau_1\ @\ n \vdash e : \tau_2}{\Gamma\ @\ n \vdash \lambda x.e : \tau_1 \xrightarrow{n} \tau_2}$$

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:

$$(df2)\quad \frac{\Gamma, x{:}\tau_1 \mathbin{@} \min(n, m) \vdash e : \tau_2}{\Gamma \mathbin{@} n \vdash \lambda x.e : \tau_1 \xrightarrow{m} \tau_2}$$

The rule (*df1*) is admissible in a system that includes the (*df2*) rule. Furthermore, 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 $min(n, m) \leqslant n$. In other words (*df1*) is more precise, but (*df2*) gives a sound over-approximation with a structure that can be unified with (*param*).

Using a rule such as (*df2*) allows us to give a unified formulation of the flat coeffect calculus in Section 4.2.4, however the coeffect-specific handling of lambda abstraction remains important in practical implementation in order to obtain a unique typing derivation for each coeffect program as discussed in Section 4.3 (and in the implementation in Chatper **??**).

4.2.2 *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 $\Gamma \mathbin{@} r \vdash e : \tau$ 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 function application, the context requirements of multiple assumptions (arising from two sub-expressions) are *merged*; in lambda abstraction, the requirements of a single expression (the body) 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 **??**). It means that we need to be careful about using the words *split* and *merge*, because they can be read as meaning exactly the opposite things. To disambiguate, we always use the term *context requirements* when using the syntactic view, especially in the rest of Chapter **??**, and *context capabilities* or just *available context* when using the semantic view, especially in Chapter 5.

4.2.3 *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 *pointwise* composition, which are mainly used in function application. The third operator, $\wedge$ is used in lambda abstraction and represents *splitting* of context requirements (or, semantically, *merging* of available context capabilities).

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* $(\mathcal{C}, \circledast, \mathsf{use})$ *is a monoid,* $(\mathcal{C}, \oplus, \mathsf{ign})$ *is an idempotent monoid,* $(\mathcal{C}, \leqslant)$ *is a pre-order and* $(\mathcal{C}, \wedge)$ *is a band (idempotent semigroup). That is, for all* $r, s, t \in \mathcal{C}$*:*

$$r \circledast (s \circledast t) = (r \circledast s) \circledast t \qquad \mathsf{use} \circledast r = r = r \circledast \mathsf{use}$$
$$r \oplus (s \oplus t) = (r \oplus s) \oplus t \qquad r \oplus r = r \qquad \mathsf{ign} \oplus r = r = r \oplus \mathsf{ign}$$
$$r \wedge (s \wedge t) = (r \wedge s) \wedge t \qquad r \wedge r = r$$
$$\text{if } r \leqslant s \text{ and } s \leqslant t \text{ then } r \leqslant t \qquad t \leqslant t$$

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 in the data-flow system all three are distinct. Similarly, the two special elements 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 semantics (Section **??**).
- The idempotent monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ represents *pointwise* 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 4.5) preserve the coeffect. Idempotence holds for all our examples and allows us to unify the flat and structural systems in Chapter **??**.
- 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. Using the example from Section 4.2.1, this guarantees that the rule (*df1*) is not a special case, but can always be derived from (*df2*). In some cases, the operator forms a monoid with the unit being the greatest element of the set $\mathcal{C}$.

It is worth noting that, in some of the systems, the operators $\oplus$ and $\wedge$ are the least upper bound and the greatest lower bounds of a lattice. For example, in data-flow computations, they are *max* and *min* respectively. However, this duality does not hold for implicit parameters (we discuss lattice-based formulation of coeffects in Section **??**.

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

$$(\textit{const})\ \frac{}{\Gamma\,@\,\mathsf{ign} \vdash n : \mathsf{num}}$$

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

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

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

Figure 1: Type system for the flat coeffect calculus

Using the syntactic reading, the two operators 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 $\leqslant$. This will be used to introduce sub-coeffecting and subtyping in Section 4.5.1, but we make it a part of the flat coeffect algebra, as it will be useful for characterization of different kinds of coeffect calculi. When the idempotent monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ also has the commutative property (i. e. forms a semi-lattice), the $\leqslant$ relation can be defined as the ordering of the semi-lattice:

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

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 or the bottom element of the semi-lattice. As discussed in Section 4.4, when this is the case, we are able to prove certain syntactic properties of the calculus.

4.2.4 *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.

The (*abs*) rule is defined as discussed in Section 4.2.1. 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 typing derivation for $(\lambda x.e_2)\ e_1$, but it corresponds to *one* possible

typing derivation. As we show in 4.5.2, the typing used in (*let*) is more precise than the general rule that can be derived from $(\lambda x.e_2)\ e_1$. Additional constructs such as pairs, sub-coeffecting and sub-typing are covered in Section 4.5.

4.2.5 *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, the general calculus has a lambda abstraction that is compatible with those discussed in Chapter 4, but includes implicit sub-coeffecting.

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 `num` and so the annotations only track sets of names. The definition uses a 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 language has additional syntax for defining an implicit parameter and for accessing it, together with associated typing rules:

$$e ::= \ldots \mid \texttt{?p} \mid \texttt{let ?p} = e_1 \texttt{ in } e_2$$

$$(\textit{param})\ \frac{}{\Gamma\,@\,\{\texttt{?p}\} \vdash \texttt{?p} : \texttt{num}}$$

$$(\textit{letpar})\ \frac{\Gamma\,@\,r \vdash e_1 : \tau_1 \qquad \Gamma\,@\,s \vdash e_2 : \tau_2}{\Gamma\,@\,r \cup (s \setminus \{\texttt{?p}\}) \vdash \texttt{let ?p} = e_1 \texttt{ in } e_2 : \tau_2}$$

The (*param*) rule specifies that the accessed parameter `?p` needs to be in the set of required parameters r. As discussed earlier, we use the same type `num` for all parameters, but it is possible to define a coeffect calculus that uses mappings from names to types (care is needed to avoid assigning multiple types to a parameter of the same type).

The (*letpar*) rule is the same as the one discussed in Section **??**. As both of the rules are specific to implicit parameters, we write the operations on coeffects directly using set operations – coeffect-specific operations such as set subtraction are not a part of the unified coeffect algebra.

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

The liveness example is interesting because it does not require any additional syntactic extensions to the langauge. It annotates constants and variables with D and L, respectively and it captures how those annotation propagate through the remaining language constructs.

As in Section **??**, sequential composition $\circledast$ is modelled by the meet operation $\sqcap$ and pointwise composition $\oplus$ is modelled by join $\sqcup$. The two-point lattice is a commutative, idempotent monoid. Distributivity $(r \sqcup s) \sqcap t = (r \sqcap t) \sqcup (s \sqcap t)$ does not hold for *every* lattice, but it trivially holds for the 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. The latter is less precise, but permissible over-approximation, which could otherwise be obtained via sub-typing.

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

As discussed earlier, sequential composition $\circledast$ is represented by $+$ and pointwise composition $\oplus$ uses *max*. For data-flow, we need a third separate operator for lambda abstraction. Annotating the body with $min(r, s)$ 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 $(\mathbb{N}, max, 0)$ form monoids and $(\mathbb{N}, min)$ forms a band. Note that data-flow is our first example where $\circledast$ is not idempotent. The distributivity laws require the following to be the case: $max(r, s) + t = max(r + t, s + t)$, which is easy to see.

A simple dataflow langauge includes an additional construct `prev` for accessing the previous value in a stream with an additional typing rule that look as follows:

$$e ::= \ldots \mid \texttt{next}\ e$$

$$(prev)\ \frac{\Gamma\ @\ n \vdash e : \tau}{\Gamma\ @\ n+1 \vdash \texttt{prev}\ e : \tau}$$

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 the 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* $\mathbb{N}_\bot = \{\bot, 0, 1, 2, 3, \ldots\}$ *such that* $\forall n \in \mathbb{N}. \bot \leqslant n$. *The flat coeffect algebra is* $(\mathbb{N}_\bot, +, max, min, 0, \bot, \leqslant)$ *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 $+$; for the bottom element $0 + \bot = \bot$ and for natural numbers $0 + n = n$. The structure $(\mathbb{N}, max, \bot)$ is also a monoid, because $\bot$ is the least element and so $max(n, \bot) = n$. Finally, $(\mathbb{N}, min)$ is a band (the extended *min* is still idempotent and associative) and the distributivity law also holds for $\mathbb{N}_\bot$.

4.3 CHOOSING UNIQUE TYPING

As discussed in Chapter **??**, the lambda abstraction rule for coeffect systems differs from the rule for effect systems in that it does not delay all context requirements. In case of implicit parameters (Section **??**), the requirements can be satisfied either by the call-site or by the declaration-site. In case of dataflow and liveness, the rule discussed in Section 4.2 reintroduces similar ambiguity because it allows multiple valid typing derivations.

Furthermore, the semantics of context-aware languages in Chapter **??** and also in Chapter 5 is defined over *typing derivation* and so the same program could have a different meaning, depending on the typing derivation chosen. In this section, we specify how to choose *unique* typing derivation in each of the coeffect systems we consider.

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

$$(\textit{const})\ \frac{}{\Gamma; \Delta \mathbin{@} \mathsf{ign} \vdash n : \mathsf{num}}$$

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

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

$$(\textit{param})\ \frac{}{\Gamma; \Delta \mathbin{@} \{?p\} \vdash ?p : \mathsf{num}}$$

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

$$(\textit{letpar})\ \frac{\Gamma; \Delta \mathbin{@} r \vdash e_1 : \mathsf{num} \qquad \Gamma; \Delta \cup \{?p\} \mathbin{@} s \vdash e_2 : \tau}{\Gamma; \Delta \mathbin{@} r \cup (s \setminus \{?p\}) \vdash \mathtt{let}\ ?p = e_1\ \mathtt{in}\ e_2 : \tau}$$

Figure 2: Choosing unique typing for implicit parameters

The most interesting case is that of implicit parameters. For example, consider the following program written using the coeffect calculus with implicit parameter extensions:

```
let f = (let ?x = 1 in (λy. ?x)) in
let ?x = 2 in f 0
```

There are two possible typings allowed by the typing rules discussed in Section 4.2.4 that lead to two possible meanings of the program – evaluating to 1 and 2, respectively:

- $f : \mathsf{num} \xrightarrow{\emptyset} \mathsf{num}$ – in this case, the value of ?x is captured from the declaration-site and the program produces 1.
- $f : \mathsf{num} \xrightarrow{\{?x\}} \mathsf{num}$ – in this case, the parameter ?x is required from the call-site and the program produces 2.

The coeffect calculus intentionally allows both of the options, acknowledging the fact that the choice needs to be made for each individual concrete context-aware programming language. In this section, we discuss the choices for implicit parameters, dataflow and liveness.

In this section, we use the fact that the coeffect calculus uses Church-style syntax for lambda abstraction and has a type annotation for the type of the variable. This does not affect the handling of coeffects (those are not defined by the type annotation), but it lets us prove uniqueness typing property of the specialized coeffect type systems. This shows that we define a *unique* way of assigning coeffects to otherwise well-typed programs.

4.3.1 *Implicit parameters*

For implicit parameters we follow the behaviour implemented by Haskell [5] where function abstraction captures all parameters that are available at the declaration-site and places all other requirements on the call-site. For the

example in the introduction, this means that the body of f captures the value of ?p available from the declaration-site and f will be typed as a function requiring no parameters (coeffect $\emptyset$). The program thus evaluates to 1.

To express this behaviour formally, we extend the coeffect type system to additionally track implicit parameters that are currently in scope. The typing judgement becomes:

$$\Gamma; \Delta \mathbin{@} r \vdash e : \tau$$

Here, Δ is a set of implicit parameters that are in scope at the declaration-site. The modified typing rules are shown in Figure 2. The rules (*var*), (*const*), (*app*) and (*let*) are modified to use the new typing judgement, but they simply propagate the information tracked by Δ to all assumptions. The (*param*) rule also remains unchanged – the implicit parameter access is still tracked by the coeffect r meaning that we still allow a form of dynamic binding (the parameter does not have to be in static scope).

The most interesting rule is (*abs*). The body of a function requires implicit parameters tracked by r and the parameters currently in (static) scope are Δ. The coeffect on the declaration site becomes Δ (capture all available parameters) and the latent coeffect attached to the function becomes $r \setminus \Delta$ (require all remaining parameters from the call-site). Finally, in the (*letpar*) rule, we add the newly bound implicit parameter ?p to the static scope in the sub-expression e_2.

PROPERTIES. If a program written in a coeffect language with implicit parameters is well-typed according to the type system presented in Figure 2, then the type system gives a unique derivation. We use this unique typing derivation to give the semantics of coeffect language with implicit parameters in Chapter 5 and we also implement this algorithm as discussed in Chatper **??**.

The type system is more restrictive than the fully general one and it reject certain programs that could be typed using the more general system. This is expected – we are restricting the fully general coeffect calculus to match the typing and semantics of implicit parameters as known from Haskell.

In order to prove the uniqueness of typing theorem (Theorem 2), we first need the inversion lemma (Lemma 1).

Lemma 1 (Inversion lemma for implicit parameters)**.** *For the type system defined in Figure 2:*

1. *If* $\Gamma; \Delta \mathbin{@} c \vdash x : \tau$ *then* $x : \tau \in \Gamma$ *and* $c = \emptyset$.
2. *If* $\Gamma; \Delta \mathbin{@} c \vdash n : \tau$ *then* $\tau = \mathsf{num}$ *and* $c = \emptyset$.
3. *If* $\Gamma; \Delta \mathbin{@} c \vdash e_1\ e_2 : \tau_2$ *then there is some* τ_1, r, s *and* t *such that* $\Gamma; \Delta \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma; \Delta \mathbin{@} s \vdash e_2 : \tau_1$ *and also* $c = r \cup s \cup t$.
4. *If* $\Gamma; \Delta \mathbin{@} c \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2$ *then there is some* τ_1, s *and* r *such that* $\Gamma; \Delta \mathbin{@} r \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1; \Delta \mathbin{@} s \vdash e_2 : \tau_2$ *and also* $c = s \cup r$.
5. *If* $\Gamma; \Delta \mathbin{@} c \vdash ?p : \mathsf{num}$ *then* $?p \in c$ *and* $c = \{?p\}$.
6. *If* $\Gamma; \Delta \mathbin{@} c \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* τ_2 *such that* $\tau = \tau_1 \xrightarrow{s} \tau_2$ $\Gamma, x{:}\tau_1; \Delta \mathbin{@} r \vdash e : \tau_2$ *and* $c = \Delta$ *and also* $s = r \setminus \Delta$.
7. *If* $\Gamma; \Delta \mathbin{@} c \vdash \mathtt{let}\ ?p = e_1\ \mathtt{in}\ e_2 : \tau$ *then there is some* r, s *such that* $\Gamma; \Delta \mathbin{@} r \vdash e_1 : \mathsf{num}$ *and* $\Gamma; \Delta \cup \{?p\} \mathbin{@} s \vdash e_2 : \tau$ *and also* $c = r \cup (s \setminus \{?p\})$.

Proof. Follows from the individual rules given in Figure 2. □

Theorem 2 (Uniqueness of coeffect typing for implicit parameters)**.** *In the type system for implicit parameters defined in Figure 2, when* $\Gamma; \Delta \mathbin{@} r \vdash e : \tau$ *and* $\Gamma; \Delta \mathbin{@} r' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $r = r'$.

Proof. Suppose that (A) $\Gamma; \Delta \mathbin{@} c \vdash e : \tau$ and (B) $\Gamma; \Delta \mathbin{@} c' \vdash e : \tau'$. We show by induction over the typing derivation of $\Gamma; \Delta \mathbin{@} c \vdash e : \tau$ that $\tau = \tau'$ and $c = c'$.

Case (*abs*): $e = \lambda x : \tau_1.e_1$ and $c = \Delta$. $\tau = \tau_1 \xrightarrow{r \backslash \Delta} \tau_2$ for some r, τ_2 and also $\Gamma, x : \tau_1; \Delta \mathbin{@} r \vdash e : \tau_2$. By case (6) of Lemma 1, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma, x{:}\tau_1; \Delta \mathbin{@} r \vdash e : \tau_2'$. By the induction hypothesis $\tau_2 = \tau_2'$ and $c = c'$ and therefore $\tau = \tau'$.

Case (*param*): $e = ?p$, from Lemma 1, $\tau = \tau' = \mathsf{int}$ and $c = c' = \{?p\}$.

Cases (*var*), (*const*) are direct consequence of Lemma 1.

Cases (*var*), (*const*), (*app*), (*let*), (*param*) and (*letpar*) similarly to (*abs*). □

IMPLEMENTATION. From the presentation in this section, it might appear that resolving the ambiguity related to lambda abstraction for implicit parameters requires a type system that is quite different from the core flat coeffect type system shown earlier in Figure 1. This is not the case. As discussed in Chapter **??**, the required changes in the implementation are simpler.

Briefly, the implementation collects constraints on the coeffects and then finds the smallest sets of implicit parameters to satisfy the constraints. We still need to track implicit parameters in scope Δ, but the rest of the (*abs*) rule from the implementation is close to the one from Figure 1:

$$(abs)\ \frac{\Gamma, x{:}\tau_1; \Delta \mathbin{@} t \vdash e : \tau_2 \mid C}{\Gamma; \Delta \mathbin{@} r \vdash \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{s} \tau_2 \mid C \cup \{t = r \wedge s, r = \Delta\}}$$

Given a typing derivation for the body that produced constraints C, we generate an additional constraint that restricts r (declaration-site requirements) to those available in the current static scope Δ. The constraint satisfaction algorithm then finds the minimal set s which is $t \backslash \Delta$.

4.3.2 *Dataflow and liveness*

Resolving the ambiguity for liveness and dataflow computations is easier than for implicit parameters. It suffices to use a lambda abstraction rule that duplicates the coeffects of the body:

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

This is the rule that we originally used for liveness and dataflow computations in Chapter **??**. This rule cannot be used with implicit parameters and so the additional flexibility provided by the $\wedge$ operator is needed in the general flat coeffect calculus.

For livenes and dataflow, the (*idabs*) rule provides the most precise coeffect. Assume we have lambda abstraction with body that has coeffects r. The ordinary (*abs*) rule requires us to find s, t such that $r = s \wedge t$.

- For dataflow, this is $r = min(s, t)$. The smallest s, t such that the equality holds are $s = t = r$.

– For livenes, this is $r = s \sqcup t$. When $r = \mathsf{L}$, the only solution is $s = t = \mathsf{L}$; when $r = \mathsf{D}$, the most precise solution is $s = t = \mathsf{D}$ because $\mathsf{D} \sqsubseteq \mathsf{L}$.

The notion of "more precise" solution can be defined in terms of subcoeffecting and subtyping. We return to this topic in Section 4.5.3 and we also precisely characterise for which coeffect system is the (*idabs*) rule preferable over the (*abs*) rule.

PROPERTIES. If a program written in a coeffect language for liveness or dataflow is well-typed according to the type system presented in Figure 1 with the (*abs*) rule replaced by (*idabs*), then the type system gives a unique derivation. As for implicit parameters, this defines the semantics of coeffect program (Chapter 5) and it is used in the implementation (Chatper **??**).

In order to prove the uniqueness of typing theorem (Theorem 4), we first need the inversion lemma (Lemma 3).

Lemma 3 (Inversion lemma for liveness and dataflow)**.** *For the type system defined in Figure 1 with the (abs) rule replaced by (idabs):*

1. *If* $\Gamma \mathbin{@} c \vdash x : \tau$ *then* $x : \tau \in \Gamma$ *and* $c = \mathsf{use}$.
2. *If* $\Gamma \mathbin{@} c \vdash n : \tau$ *then* $\tau = \mathsf{num}$ *and* $c = \mathsf{ign}$.
3. *If* $\Gamma \mathbin{@} c \vdash e_1\ e_2 : \tau_2$ *then there is some* τ_1, r, s *and* t *such that* $\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma \mathbin{@} s \vdash e_2 : \tau_1$ *and also* $c = r \oplus (s \circledast t)$.
4. *If* $\Gamma \mathbin{@} c \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2$ *then there is some* τ_1, s *and* r *such that* $\Gamma \mathbin{@} r \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1 \mathbin{@} s \vdash e_2 : \tau_2$ *and also* $c = s \oplus (s \circledast r)$.
5. *If* $\Gamma \mathbin{@} c \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* τ_2 *such that* $\tau = \tau_1 \xrightarrow{c} \tau_2$ *and* $\Gamma, x{:}\tau_1 \mathbin{@} c \vdash e : \tau_2$.

Proof. Follows from the individual rules given in Figure 2. □

Theorem 4 (Uniqueness of coeffect typing for liveness and dataflow)**.** *In the type system for implicit parameters defined in Figure 1 with the (abs) rule replaced by (idabs), when* $\Gamma \mathbin{@} r \vdash e : \tau$ *and* $\Gamma \mathbin{@} r' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $r = r'$.

Proof. Suppose that (A) $\Gamma \mathbin{@} c \vdash e : \tau$ and (B) $\Gamma \mathbin{@} c' \vdash e : \tau'$. We show by induction over the typing derivation of $\Gamma \mathbin{@} c \vdash e : \tau$ that $\tau = \tau'$ and $c = c'$.

Case (*abs*): $e = \lambda x{:}\tau_1.e_1$. $\tau = \tau_1 \xrightarrow{c} \tau_2$ for some τ_2 and $\Gamma, x{:}\tau_1 \mathbin{@} c \vdash e : \tau_2$. By case (5) of Lemma 3, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma, x{:}\tau_1 \mathbin{@} c' \vdash e : \tau_2'$. By the induction hypothesis $\tau_2 = \tau_2'$ and $c = c'$ and therefore also so $\tau = \tau'$.

Cases (*var*), (*const*) are direct consequence of Lemma 1.

Cases (*var*), (*const*), (*app*), (*let*), (*param*) and (*letpar*) similarly to (*abs*). □

IMPLEMENTATION. As with implicit parameters, the implementation (discussed in Chapter **??**) does not require changing the typing (*abs*) rule of the flat coeffect system. The typing specified by (*idabs*) can be easily obtained by generating additional constraints:

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

Here, the two additional constraints restrict both r and s to be equal to the coeffect of the body t and so the only possible resolution is the one specified by (*idabs*).

4.4 SYNTACTIC 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 coeffect systems. 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.

4.4.1 *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:

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

The rules capture syntactic reductions that can be performed in a general calculus, without any knowledge of the specific notion of context. If the reductions preserve the type of the expression (type preservation), then operational semantics can be defined as a repeated application of the rules, until a specified normal form (i. e. a value) is reached.

In the rest of the section, we briefly outline the interpretation of the three rules and then we focus on call-by-value (Section 4.4.2) and call-by-name (Section 4.4.3) in more details.

The focus of this chapter 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. [2]. We discuss concrete semantics of implicit parameters and dataflow in Chapter 5.

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 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 means that certain coeffect systems support call-by-name evaluation strategy and could be embedded in purely functional language (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 [18], 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 defer 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 4.4.2.

The call-by-value evaluation strategy preserves typing 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 $\Gamma \mathbin{@} \mathtt{r} \vdash e : \tau$ to a value, i. e. a term $\Gamma \mathbin{@} \mathsf{use} \vdash e' : \tau$ with no context requirements.

LOCAL SOUNDNESS AND COMPLETENESS. Two desirable properties of calculi, coined by Pfenning and Davies [13], are *local soundness* and *local completeness*. They guarantee that the rules which introduce a function arrow (lambda abstraction) and eliminate it (application) are sufficiently strong, but not too 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 4.4.3.

4.4.2 *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 values are defined in the usual way:

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

The syntactic form *SynVal* captures syntactic values, but a context-dependency-free value in coeffect calculus cannot be defined purely syntactically, because a function $(\lambda x.e)$ does not automatically defer all context requirements.

Definition 2. *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. $\Gamma \mathbin{@} \mathsf{use} \vdash e : \tau$.*

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

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 5 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. This is captured by the (*approximation*) property:

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

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 the 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 6 (Type preservation for call-by-value)**.** *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 final step simplifies the coeffect annotation using the fact that use is a unit of $\circledast$. From Lemma 5, $e[x \leftarrow v]$ has the same coeffect annotation as e. As $r \wedge t \leqslant r \oplus t$, we can apply sub-coeffecting:

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

Comparing the final conclusions of the above two typing derivations shows that the reduction preserves type and coeffect annotation. □

4.4.3 *Call-by-name evaluation*

When reducing the expression $(\lambda x.e_1)\ e_2$ using the call-by-name strategy, the sub-expression e_2 is substituted for all occurrences of the variable v in an expression e_1. As discussed in Section 4.4.1, the call-by-name strategy does not *in general* preserve the type of a terms in coeffect calculi, but it does preserve the typing in two interesting cases.

The typing is preserved for different reasons in two of our systems, so we briefly review the concrete examples. Then, we prove the substitution lemma for two special cases of flat coeffects (Lemma 7 and Lemma 8) and

finally, we state the conditions under which typing preservation hold for flat coeffect calculi (Theorem 9).

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 a context-dependent expression `prev z` for a variable y in a function $\lambda x.y$:

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

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 – at run-time, the code does not actually access a previous value of the argument x. This cannot be captured by a flat coeffect system, but it can be captured using the structural system discussed in Chapter **??**.

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 free variable y under a lambda:

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

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 an arbitrary expression e of type τ_1 with coeffects r for a variable y:

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

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 we can also treat e as being annotated with coeffects L. As a result, substitution does not change the coeffect.

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. We consider the two cases separately, but find a more general formulation for both of them.

Definition 3. *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 $\mathsf{D} \leqslant \mathsf{L}$, 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 7 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$*, substituting an expression* e_s *with arbitrary coeffects* s *for a variable* x *in* e_r *does not change the coeffects of* e_r*:*

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

Proof. Using sub-coeffecting ($s \leqslant \mathsf{use}$) and a variation of Lemma 5. □

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 a set union) and for other set-like coeffects. It allows the following substitution lemma:

Lemma 8 (Bottom-pointed substitution)**.** *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \circledast = \oplus$ *is an idempotent and commutative operation' ' 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 **??**. □

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 **??** is *precise enough* to keep the coeffects associated with individual variables, thus preserving typing in a complementary way.

The two substitution lemmas discussed above 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 theorem holds:

Theorem 9 (Type preservation for call-by-name)**.** *In a coeffect system that satisfies the conditions for Lemma 7 or Lemma 8, if* $\Gamma \mathbin{@} r \vdash e : \tau$ *and* $e \rightarrow_{\mathsf{cbn}} e'$ *then it is also the case that* $\Gamma \mathbin{@} r \vdash e' : \tau$*.*

Proof. For top-pointed coeffect algebra (using Lemma 7), the proof is similar to the one in Theorem 6, 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 8 is:

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

From the assumptions of Lemma 8, 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 immediate coeffects are made larger by the lambda abstraction – the context requirements of the function value are imposed on the declaration site of the new lambda abstraction. We remedy this limitation in the next chapter.

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

Theorem 10 (Type preservation of 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:

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

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 [13].

4.5 SYNTACTIC PROPERTIES AND EXTENSIONS

The flat coeffect algebra introduced in Section 4.2 requires a number of laws. The laws are required for three reasons – to be able to define the categorical structure in Section **??**, to prove equational properties in Section 4.4 and finally, to guarantee intuitive syntactic properties for constructs such as λ-abstraction and pairs in context-aware calculi.

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

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

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

Figure 3: Subtyping rules for flat coeffect calculus

In this section, we look at the last point. We consider sub-coeffecting and subtyping (Section 4.5.1), discuss what syntactic equivalences are permitted by the properties of $\wedge$ (Section 4.5.3) and we extend the calculus with pairs and units and discuss their syntactic properties (Section 4.5.4).

4.5.1 *Subcoeffecting and subtyping*

The *flat coeffect algebra* includes the $\leqslant$ relation which captures the ordering of coeffects and can be used to define sub-coeffecting. Syntactically, an expression with context requirements r can be treated as an expression with greater context requirements:

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

Semantically, this means that we can *drop* some of the provided context. For example, if an expression requires implicit parameters $\{?p\}$ it can be treated as requiring $\{?p, ?q\}$. The semantic function will then be provided with a dictionary containing both assignments and it can drop the value for the unused parameter $?q$.

Sub-coeffecting only affects the immediate coeffects attached to the free-variable context. In Figure 3, we add sub-typing on function types, making it possible to treat a function with smaller context requirements as a function with greater context requirements:

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

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 expressions in the coeffect calculus.

4.5.2 *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 an expression $(\lambda x.e_2)\ e_1$, using the idempotence of $\wedge$ as follows:

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

This design decision is similar to ML value restriction, but it works the other way round. Our *let* binding is more restrictive than the typing of abstraction-application, rather than being more general. The choice is motivated by the

	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 instances

fact that the typing obtained using the special rule for let-binding is more precise for all the examples consider in this chapter. Table 1 shows how the coeffect annotations are simplified for our examples.

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 more precise, 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 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):

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

In other words, the inequality states that using idempotence, we get a more precise typing. Using the $\geqslant$ operator this property can be expressed using the abstract operators of the flat coeffect algebra as:

$$s_1 \oplus (s_2 \circledast r) \geqslant (s_1 \wedge s_2) \oplus ((s_1 \wedge s_2) \circledast r)$$

This property 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 $(\lambda x.e_2)\ e_1$.

4.5.3 *Properties of lambda abstraction*

In Section 4.2.1, we discussed how to reconcile two typings for lambda abstraction – for implicit parameters, the lambda function needs to split context requirements using $r \cup s$, but for data-flow and liveness it suffices to duplicate the requirement r of the body. We introduced the $\wedge$ operation as a way of providing the additional abstraction.

In this section, we first identify coeffect calculi for which the simpler (*id-abs*) rule introduced in Section **??** is sufficient. Then we look at syntactic transformations corresponding to other common properties of $\wedge$.

SIMPLIFIED ABSTRACTION. Recall that $(\mathcal{C}, \wedge)$ is a band, that is, $\wedge$ 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 (*idabs*) typing is valid. For reference, we repeat the earlier definitions here:

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

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 following condition identifies coeffect calculi where (*abs*) can be derived from (*idabs*).

Definition 4. *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* $r \wedge s \leqslant r$.

Remark 11. *For a flat coeffect calculus with a strictly oriented algebra, equipped with sub-coeffecting and subtyping, the standard (abs) rule can be derived from the (idabs) rule.*

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

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

□

The practical consequence of the Remark 11 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 of type checking 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 12. *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*:*

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

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.

4.5.4 *Language with pairs and unit*

To show the key aspects of flat coeffect systems, the calculus introduced in Section 4.2 consists only of variables, abstraction, application and let binding. Here, we extend it with pairs and the unit value to sketch how it can be turned into a more complete programming language and to motivate the laws required about $\oplus$. The syntax of the language is extended as follows:

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

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

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

Figure 4: Typing rules for pairs and units

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

The typing rules for pairs and the unit value are shown in Figure 4. 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 *pointwise* operator $\oplus$. The operator models the case when the (same) available context is split and passed to two independent sub-expressions. 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 typically form a monoid. Assuming $\simeq$ is an isomorphism that performs appropriate transformation on values, without affecting other properties (here, coeffects) of the expressions. The monoid laws then correspond to the requirement that $(e_1, (e_2, e_3)) \simeq ((e_1, e_2), e_3)$ (associativity) and the requirement that $((), e) \simeq e \simeq (e, ())$ (unit).

Thanks to the properties of $\oplus$, the flat coeffect calculus obeys the monoid laws for pairs. 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 $\Gamma @ r \vdash e : \tau$ if and only if $\Gamma @ r \vdash e' : \tau$.

Remark 13. *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 constants in the language). □

The Remark 13 motivates the requirement of the monoid structure $(\mathcal{C}, \oplus, \mathsf{ign})$ of the flat coeffect algebra. We require only unit and associativity laws. In our three examples, the $\oplus$ operator is also symmetric, which additionally gives us the property that $(e_1, e_2) \simeq (e_2, e_1)$.

4.6 CONCLUSIONS

This chapter presented the *flat coeffect calculus* – a unified system for tracking *whole-context* properties of computations, that is properties related to the execution environment or the entire context in which programs 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

instantiated the system to capture three specific systems, namely liveness, data-flow and implicit parameters. However, the system is more general and an capture numerous other applications outlined in Section **??**.

An inherent property of flat coeffect systems is the ambiguity of the typing for lambda abstraction. The body of a function requires certain context, but the context can be often provided by either the declaration-site or the call-site. Resolving this ambiguity has to be done differently for each concrete coeffect system, depending on its specific notion of context. We discussed this for implicit parameters, dataflow and liveness in Section **??**.

Finally, we introduced 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 *typing preservation* theorems prove that the operational semantics for two classes of flat coeffect calculi (including liveness and implicit parameters) can be based on the standard call-by-name reduction.

In the next section, we move from abstract treatment of the flat coeffect calculus to a more concrete discussion. We explain its category-theoretical motivation, we use it to define translational semantics (akin to Haskell's do notation) and we prove that well-typed programs in flat coeffect calcului for implicit parameters and dataflow do not get stuck.

5 SEMANTICS OF FLAT COEFFECTS

The *flat coeffect calculus* introduced in the previous chapter uniformly captures a number of context-aware systems discussed in Chapter **??**. The coeffect calculus can be seen as a *language framework* that simplifies the construction of concrete *domain-specific* coeffect lnguages. In the previous chapter, we discussed how it provides a type system that tracks the required context. In this chapter, we show that the langauge framework also provides a way for defining the semantics of concrete domain-specific coeffect languages, guides their implementation and simplifies safety proofs.

This is done using a *comonadically-inspired translation*. We translate a program written using the coeffect calculus into a simple functional language with additional coeffect-specific comonadically-inspired primitives that implement the concrete notion of context-awareness.

We use comonads in a syntactic way, following the example of Haskell's use of monads. The translation is the same for all coeffect languages, but the safety depends on the concrete coeffect-specific comonadically-inspired primitives. We prove the soundness of two concrete coeffect calculi (dataflow and implicit parameters). We note that the proof crucially relies on a relationship between coeffect annotations (provided by the type system) and the comonadically-inspired primitives (defining the semantics), which makes it easy to extend it to other concrete context-aware langauges.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We introduce *indexed comonads*, a generalization of comonads, a category-theoretical dual of monads (Section 5.2) and we discuss how they provide semantics for coeffect calculus. This provides an insight into how (and why) the calculus works and shows an intriguing link with effects and monads.
- We use indexed comonads to guide our *translational semantics* of coeffect calculus (Section 5.3). We define a simple functional programming language (with type system and operational semantics) and prove that well-typed programs in the language do not get stuck. We extend it with uninterpreted comonadically-inspired primitives and define a translation that turns well-typed context-aware coeffect programs into programs of our functional language.
- For two of the coeffect calculi discussed earlier (dataflow and implicit parameters), we give reduction rules for the comonadically-inspired primitives and we extend the progress and preservation proofs, showing that well-typed programs produced by translation from two coeffect languages do not get stuck (Section 5.4)
- We leave the details to future work, but we note that the proof for concrete coeffect language (dataflow and implicit parameters) can be generalized – rather than reconsidering progress and preservation of the whole target language, we rely just on the correctness of the coeffect-specific comonadically-inspired primitives and abstraction mechanism provided by languages such as ML and Haskell (Section 5.6).

5.1 INTRODUCTION AND SAFETY

This chapter links together all the different technical developments presented in this thesis. We take the flat coeffect calculus introduced in Chatper 4, define its *abstract comonadic semantics* and use it to define a translation that gives a *concrete operational semantics* to a number of concrete context-aware languages. The type system is used to guarantee that the resulting programs are correct. Finally, the development in this chapter is closely mirrored by the implementation presented in Chapter **??**, which implements the translation together with an interpreter for the target language.

The key claim of this thesis is that writing context-aware programs using coeffects is easier and safer. In this chapter, we substantiate the claim by showing that programs written in the coeffect calculus and evaluated using the translation provided here do not "go wrong".

To provide an intuition, consider two context-aware programs. The first calls a function that adds two implicit parameters in a context where one of them is defined. The second calculates the difference between the current and the previous value in a dataflow computation. For comparison, we show the code written in a coeffect dataflow language (on the left) and using standard ML-like libraries (on the right):

```
let add = fun x →                 let add = fun x params →
  ?one + ?two in                    lookup "one" params +
let ?one = 10 in                      lookup "two" params in
add 0                             add 0 (add "one" 10 params)

let diff = fun x →                let diff = fun x →
  x − prev x                        List.head x − List.head (List.tail x)
```

The add function (on the left) has a type $\mathsf{int} \xrightarrow{\{?one,?two\}} \mathsf{int}$. We call it in a context containing ?one and so the coeffect of the program is {?two}. The safety property for implicit parameters (Theorem **??**) guarantees that, when executed in a context that provides a value for the implicit parameter ?one, the program reduces to a value of an correct type (or never terminates).

If we wrote the code without coeffects (on the right), we could use a dynamic map to pass around a dictionary of parameters (the lookup function obtains a value and add adds a new assignment to the map). In that case, the type of add is just $\mathsf{int} \to \mathsf{int}$ and so the user does not know which implicit parameters it will need.

Similarly, the diff function can be implemented in terms of lists (on the right) as a function of type $\mathsf{num\ list} \to \mathsf{num}$. The function fails for input lists containing only zero or one elements and this is not reflected in the type and it is not enforced by the type checker.

Using coeffects (on the left), the function has a type $\mathsf{num} \xrightarrow{1} \mathsf{num}$ meaning that it requires one past value (in addition to the current value). The safety property for dataflow (Theorem **??**) shows that, when called with a context that contains the required number of past values as captured by the coeffect type system, the function does not get stuck.

In summary, coeffect type system, captures certain runtime requirements of context-aware programs and (as we show in this chapter), eliminates common errors related to working with context.

5.2 CATEGORICAL MOTIVATION

The type system of flat coeffect calculus arises syntactically, 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 translation discussed in Section 5.3.

5.2.1 *Comonads are to coeffects what monads are to effects*

The development in this chapter closely follows the example of effectful computations. Effect systems provide a type system for tracking effects and monadic translation can be used as a basis for implementing effectful domain-specific languages (e.g. through the do-notation in Haskell).

The correspondence between effect system and monads has been pointed out by Wadler and Thiemann [18] and further explored by Atkey [1] and Vazou and Leijen [9]). This line of work relates 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, defining the semantics in terms of comonadic computations is not a simple mechanical dualisation of the work on effect systems and monads.

Our approach is inspired by the work of Uustalu and Vene [17] who present the semantics of contextual computations (mainly for dataflow) 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. $C^{r}\tau_1 \to \tau_2$. This is similar to the recent development on monads and effects by Katsumata [4] who parameterizes monads in a similar way to our parameterization of comonads.

5.2.2 *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 [8] showed that the semantics of various effectful computations can be captured uniformly using (*strong*) *monads*. 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 side-effects can be modelled using $M\alpha = S \to (\alpha \times S)$ (state monad). Here, the structure of a strong monad provides necessary "plumbing" for composing monadic computations – sequential composition and strength for lifting free variables into the body of computation under a lambda abstraction.

Following similar approach to Moggi, Uustalu and Vene [17] showed that (*monoidal*) *comonads* uniformly capture the semantics of various kinds of context-dependent computations [17]. For example, data-flow computations over non-empty lists are modelled using the non-empty list comonad $\mathsf{NEList}\,\alpha = \alpha + (\alpha \times \mathsf{NEList}\,\alpha)$.

The monadic and comonadic model outlined here represents at most a binary analysis of effects or context-dependence. A function $\tau_1 \to \tau_2$ per-

forms *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[1].

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.

5.2.3 *Introducing comonads*

In category theory, *comonad* is a dual of *monad*. As already outlined in Chapter **??**, we obtain a definition of a comonad by taking a definition of a monad and "reversing the arrows". More formally, one of the equivalent definitions of comonad looks as follows (repeated from Section **??**):

Definition 5. *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 [17] 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 operation 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

1 This is an over-simplification as we can use e. g. stacks of monad transformers and model functions with two different effects using $\tau_1 \to M_1(M_2\ \tau_2)$. However, monad transformers require defining complex system of lifting to be composable. Consequently, they are usually used for capturing different kinds of impurities (exceptions, non-determinism, state), but not for capturing fine-grained properties (e. g. a set of memory regions that may be accessed by a stateful computation).

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` $()$. 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 functions $\mathsf{Maybe}\ \tau_1 \to \tau_2$. To use such a model, we need to generalise comonads to *indexed comonads*.

5.2.4 *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 6. *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* $g : C^s\beta \to \gamma$*:f*

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

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

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

Rather than defining a single mapping C, we are now defining a family of mappings C^r indexed by a monoid structure. Similarly, the $\mathsf{cobind}_{r,s}$ operation is now 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 omit the superscripts and treat cobind just as a family of mappings (rather than natural transformations). When this does not introduce ambiguity, we also occasionally omit the subscripts.

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 and as a useful reminder to the reader. Crucially, this means that the operation is defined only for 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.

COMPOSITION. The co-Kleisli category that models sequential composition is formed by the unit arrow (provided by counit) together with the (asso-

ciative) 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 $1 * 1 = 1$. The mapping C^1 is the mapping C of the underlying 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 Section **??**, 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. As in the direct model discussed in Chapter **??**, the cobind operation 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 live contexts are distinguished by the (type-level) coeffect annotation and counit only needs to be defined on live contexts.

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 $C^{r}\alpha = \alpha \times (r \to \rho)$ 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:*

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

In the definition, we use the notation (a, g) for a pair containing the value of type α together with g, which is a function $r \to \rho$. The counit operation

takes a value and a function (with empty set as a domain), ignores the function and extracts the value. The cobind operation uses the restriction operation $g|_r$ to restrict the domain of g to implicit parameters r and s in oder to get implicit parameters required by the argument of f and by the resulting computation, respectively (i. e. semantically, it *splits* the available context capabilities). The function g passed to cobind is initially defined on $r \cup s$ and so the restriction is valid in both cases.

The structure of *indexed comonads* is sufficient to model sequential composition of computations that use a single variable (as discussed in Section **??**). To model full λ-calculus with lambda abstraction and multiple-variable contexts, we need additional operations introduced in the next section.

5.2.5 *Flat indexed comonads*

Because of the asymmetry of λ-calculus (discussed in Section **??**), the duality between monads and comonads can no longer help us with defining the additional structure required to model full λ-calculus. In comonadic computations, additional information is attached to the context. In application and lambda abstraction, the context is propagated differently than in effectful computations.

To model the effectful λ-calculus, Moggi [8] requires a *strong* monad which has an additional operation $\mathsf{strength} : \alpha \times M\beta \to M(\alpha \times \beta)$. This allows lifting of free variables into an effectful computation. In Haskell, strength can be expressed in the host language and so it is implicit.

To model λ-calculus with contextual properties, Uustalu and Vene [17] require *lax semi-monoidal* comonad. This structure requires an additional monoidal operation:

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

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.

Definition 7. *Given a flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$*, a* flat indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$ *where:*

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

The $\mathsf{merge}_{r,s}$ operation is the most interesting one. Given two comonadic values with additional contexts specified by r and s, it combines them into a single value with additional context $r \wedge s$. The $\wedge$ operation often represents *greatest lower bound*[2], 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,

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

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].

SEMANTICS OF SUB-COEFFECTING. Although we do not include sub-coeffecting in the core flat coeffect calculus, it is an interesting extension to consider. Semantically, sub-coeffecting drops some of the available contextual capabilities (drops some of the implicit parameters or some of the past values). This can be modelled by adding a (family of) lifting operation(s):

- $\mathsf{lift}_{r',r}$ is a family of mappings $C^{r'}\alpha \to C^{r}\alpha$ for all r', r such that $r \leqslant r'$

Although we do not demand 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 special case allows a simpler definition of *indexed flat comonad* by expressing the split operation in terms of lifting (sub-coeffecting) as follows:

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

The map_r operation is the mapping on arrows 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 use the argument c twice, 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 three reasons. First, it requires making sub-coeffecting a part of the core definition. Second, this would be the only place where our semantics uses a variable *twice* (in this case c). Thus using an explicit split means that the structure required by our semantics does not need to provide variable duplication and our model could be embedded in linear or affine category. Finally, explicit split is similar to the definition that is needed for structural coeffects in Chapter **??** and it makes the connection between the two easier to see.

EXAMPLES. All the examples of *indexed comonads* discussed in Section 5.2.4 can be extended into *flat indexed comonads*. Note that this is not true *in general*, because each example requires us to define additional operations, specific for the example.

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 [17]. 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.*

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:*

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

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 $D = r \sqcap s$), but when the input is *live*, the operation can perform implicit sub-coeffecting and drop one of the values.

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 $\alpha \times (r \to \rho)$ where ρ is the type of implicit parameter values. The additional operations are defined as:*

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

The split operation splits the tuple and restricts the function (representing available implicit parameters) to the required sub-sets. The merge operation is more interesting. It uses the $\uplus$ 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 number of available 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:*

$$\begin{aligned} &\mathsf{counit}_0\langle a_0\rangle = a_0 \qquad\qquad C^n\alpha = \underbrace{\alpha \times \ldots \times \alpha}_{(n+1)-\text{times}} \\ &\mathsf{cobind}_{m,n}\ f\langle a_0, \ldots a_{m+n}\rangle = \\ &\quad \langle f\langle a_0, \ldots, a_m\rangle, \ldots, f\langle a_n, \ldots, a_{m+n}\rangle\rangle \\ &\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) = \\ &\quad \langle (a_0, b_0), \ldots, (a_{min(m,n)}, b_{min(m,n)})\rangle \\ &\mathsf{split}_{m,n}\langle (a_0, b_0), \ldots, (a_{max(m,n)}, b_{max(m,n)})\rangle = \\ &\quad (\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) \end{aligned}$$

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.

5.2.6 *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 se-

The semantics is defined over a typing derivation:

$$\frac{}{\Gamma\,@\,\mathsf{use} \vdash x_i : \tau_i} \;=\; \frac{}{\pi_i \circ \mathsf{counit}_{\mathsf{use}}} \qquad (\textit{var})$$

$$\frac{}{\Gamma\,@\,\mathsf{ign} \vdash n : \mathsf{num}} \;=\; \frac{}{\mathsf{const}\ n} \qquad (\textit{num})$$

$$\frac{\Gamma, x : \tau_1\,@\,r \wedge s \vdash e : \tau_2 \;=\; f}{\Gamma\,@\,r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2 \;=\; f \circ \mathsf{curry}\ \mathsf{merge}_{r,s}} \qquad (\textit{abs})$$

$$\frac{\begin{array}{rcl}\Gamma\,@\,r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 &=& f \\ \Gamma\,@\,s \vdash e_2 : \tau_1 &=& g\end{array}}{\Gamma\,@\,r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2 \;=\; \begin{array}{l}\mathsf{app} \circ f \times (\mathsf{cobind}_{s,t}\ g) \circ \mathsf{split}_{r,s \circledast t} \\ \quad \circ\, \mathsf{map}_{r \oplus (s \circledast t)}\ \mathsf{dup}\end{array}} \qquad (\textit{app})$$

Assuming the following auxiliary operations:

$$\begin{array}{rcl}\mathsf{map}_r\ f & = & \mathsf{cobind}_{\mathsf{use},r}\ (f \circ \mathsf{counit}_{\mathsf{use}}) \\ \mathsf{const}\ v & = & \lambda x.v \\ \mathsf{curry}\ f\ x\ y & = & \lambda f.\lambda x.\lambda y.f\ (x, y) \\ \mathsf{dup}\ x & = & (x, x) \\ f \times g & = & \lambda(x, y).(f\ x, g\ y) \\ \mathsf{app}\ (f, x) & = & f\ x\end{array}$$

Figure 5: Categorical semantics of the flat coeffect calculus

mantics that is capable of capturing all our examples, as well as other computations that can be modelled by the structure.

As discussed in Section 4.3, different typing derivations of coeffect programs may have different meaning (e. g. when working with implicit parameters) and so the semantics is defined over a *typing derivation* rather than over an *term*. To assign a semantics to a term, we choose its unique coeffect-specific typing derivation as defined in Section 4.3.

CONTEXTS AND TYPES. 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:

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

EXPRESSIONS. The definition of the semantics is shown in Figure 5. For consistency with earlier work [17, 10], the definitions use a point-free categorical notation. The semantics uses a number of auxiliary definitions that can be expressed in a Cartesian-closed category such as currying curry, value duplication dup, function pairing (given $f : A \to B$ and $g : C \to D$ then $f \times g : A \times C \to B \times D$) and application app. We will embed the definitions in a simple programming language later (Section 5.3).

The semantics of variable access and abstraction are the same as in the semantics of Uustalu and Vene [17], modulo the indexing. The semantics of variable access (*var*) uses $\mathsf{counit}_{\mathsf{use}}$ to extract a product of free variables and projection π_i to obtain the variable value. Abstraction (*abs*) is interpreted

as a curried function that takes the declaration-site context and a function argument, merges them using $\mathsf{merge}_{r,s}$ and passes the result to the semantics of the body f. 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 (*app*) 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 using the semantic function f. The result is a function $C^t \tau_1 \to \tau_2$. The second context (with index $s \circledast t$) is used to evaluate e_2 and using the semantic function g and wrap it with context required by the function e_1 by applying $\mathsf{cobind}_{s,t}$. The app operation than applies the function (first element) on the argument (second element). Finally, numbers (*num*) become constant functions that ignore the context.

PROPERTIES. We the categorical semantics in Section 5.3 to define a translation that embeds context-dependent computations in a functional programming langauge, similarly to how monads and the do notation provide a way of embedding effectful computations in Haskell.

An important property of the translation is that it respects the coeffect annotations provided by the type system. The annotations of the semantic functions match the annotations in the typing judgement and so the semantics is well-defined. This provides a further validation for the design of the type system developed in Section 4.2.4 – if the coeffect annotations for (*app*) and (*abs*) were different, we would not be able to provide a well-defined semantics using flat indexed comonads.

Informally, the following states that if we see the semantics as a translation, the resulting code is well-typed. We revisit the property in Lemma 21 once we define the target langauge and its typing.

Lemma 14 (Correspondence)**.** *In the semantics defined in Figure 5, the context annotations r of typing judgements $\Gamma \mathbin{@} r \vdash e : \tau$ and function types $\tau_1 \xrightarrow{r} \tau_2$ on the left-hand side correspond to the indices of mappings C^r in the corresponding semantic function on the right-hand side.*

Proof. By analysis of the semantic rules in Figure 5. □

Thanks to indexing, the correspondence provides more guarantees than for a non-indexed system. In the semantics, we not only know which values are comonadic, but we also know what contextual information they are required to provide. In Section 5.5, we note that this lets us generalize the proofs about concrete languages discussed in this chapter to a more general setting.

The semantics is also a generalization of the concrete semantics given when introducing context-aware programming languages in Chapter **??**.

Theorem 15 (Generalization)**.** *Consider a typing derivation obtained according to the rules for finding unique typing derivations as specified in Section 4.3 for a coeffect language with liveness, dataflow or implicit parameters.*

The semantics obtained by instantiating the rules in Figure 5 with the concrete operations defined in Example 10, Example 11 or Example 12 is the same as the one defined in Figure **??**, *Figure* **??** *and Figure* **??**, *respectively.*

Proof. Simple expansion of the definitions for the unique typing derivation. □

LANGUAGE SYNTAX

$$
\begin{array}{rcl}
v & = & n \mid \lambda x.e \mid (v_1, \ldots, v_n) \\
e & = & x \mid n \mid \pi_i\ e \mid (e_1, \ldots, e_n) \mid e_1\ e_2 \mid \lambda x.e \\
\tau & = & \mathsf{num} \mid \tau_1 \times \ldots \times \tau_n \mid \tau_1 \to \tau_2 \\
C & = & (v_1, \ldots, v_{i-1}, \_, e_{i+1}, \ldots e_n) \mid v\ \_ \mid \_\ e \mid \pi_i\ \_
\end{array}
$$

REDUCTION RULES

$$
\begin{array}{ll}
(fn) & (\lambda x.e)\ v \to e[x \leftarrow v] \\
(prj) & \pi_i(v_1, \ldots, v_n) \to v_i \\
(ctx) & C[e] \to C[e'] \qquad\qquad (\text{when } e \to e')
\end{array}
$$

TYPING RULES

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

$$
(num)\ \frac{}{\Gamma \vdash n : \mathsf{num}}
$$

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

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

$$
(proj)\ \frac{\Gamma \vdash e : \tau_1 \times \ldots \tau_i \times \ldots \times \tau_n}{\Gamma \vdash \pi_i\ e : \tau_i}
$$

$$
(tup)\ \frac{\forall i \in \{1 \ldots n\}.\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}
$$

Figure 6: Common syntax and reduction rules of the target langauge

5.3 EMBEDDING COEFFECT LANGUAGES

Although the notion of indexed comonads presented in the previous section is novel and interesting in its own, the main reason for introducing it is that we can view it as a translation that provides embedding of context-aware domain-specific languages in a simple target functional language. In this section, we follow the example of effects and monads and we use the semantics to define a translation akin to the do notation in Haskell.

A context-aware *source* program written using a concrete context-aware domain-specific language (capturing dataflow, implicit parameters or other kinds of context awareness) with domain-specific language extensions (the `prev` keyword, or the `?impl` syntax) is translated to a *target* langauge that is not context-aware. The target language is a small functional language consisting of:

- Simple functional subset formed by lambda calculus with support for tuples and numbers.
- Comonadically-inspired primitives corresponding to *counit*, *cobind* and other operations of flat indexed comonads.
- Additional primitives that model contextual operations of each concrete coeffect language (*prev* for the `prev` keyword, *lookup* for the `?p` syntax and *letimpl* for the `let ?p = ...` notation).

The syntx, typing and reduction rules of the first part (simple functional language) are used by all concrete coeffect domain-specific langauges. The syntax and typing rules of the second part (comonadically-inspired) primitives are also shared by all coeffect DSLs, however the *reduction rules* for the comonadically-inspired primitives differ – they capture the concrete notion of context. Finally, the third part (domain-specific primitives) will differ for each coeffect DSL.

5.3.1 *Functional target langauge*

The target langauge for the translation is a simply typed lambda calculus with integers and tuples. We include integers as an example of a concrete type. Tuples are needed by the translation, which keeps a tuple of variable assignments. Encoding those without tuples would be possible, but cumbersome. In this section, we define the common parts of the language without the comonadically-inspired primitives.

The syntax of the target programming langauge is shown in Figure 6. The values include numbers n, tuples and function values. The expressions include variables x, values, lambda abstraction and application and operations on tuples. We do not need recursion or other data types (although a realistic programming language would include them). In what follows, we also use the following syntactic sugar for let binding:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \quad = \quad (\lambda x.e_2)\ e_1$$

Finally, $C[e]$ defines the context in which sub-expressions are evaluated. Together with the evaluation rules shown in Figure 6, this captures the standard call-by-name semantics of the common parts of the target language. The (standard) typing rules for the common expressions of the target langauge are also shown in Figure 6.

5.3.2 *Safety of functional target language*

The functional subset of the langauge described so far models a simple ML-like langauge. We choose call-by-value over call-by-name for no particular reason and Haskell-like language would work equally well.

The subset of the language introduced so far is type-safe in the standard sense that "well-typed programs do not get stuck". Although standard, we outline the important parts of the proof for the functional subset here, before we extend it to concrete context-aware languages in Section 5.4.

We use the standard syntactic approach to type safety introduced by Milner [7]. Following Wright, Felleisen and Pierce [14, 19], we prove the type preservation property (reduction does not change the type of an expression) and the progress property (a well-typed expression is either a value or can be further reduced).

Lemma 16 (Canonical forms)**.** *For all* e, τ*, if* $\vdash e : \tau$ *and* e *is a value then:*

1. *If* $\tau = \mathsf{num}$ *then* $e = n$ *for some* $n \in \mathbb{Z}$
2. *If* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'
3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i

Proof. For (1), the last typing rule must have been (*num*); for (2), it must have been (*abs*) and for (3), the last typing rule must have been (*tup*) □

Lemma 17 (Preservation under substitution). *For all* $\Gamma, e, e'\tau, \tau'$*, if* $\Gamma, x : \tau \vdash e : \tau'$ *and* $\Gamma \vdash e' : \tau$ *then* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash e : \tau'$. □

Theorem 18 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\to$.

Case (*fn*): $e = (\lambda x.e_0)\ v$, from Lemma 17 it follows that $\Gamma \vdash e_0[x \leftarrow v] : \tau$.

Case (*prj*): $e = \pi_i(v_1, \ldots, v_n)$ and so the last applied typing rule must have been (*tup*) and $\Gamma \vdash (v_1, \ldots, v_n) : \tau_1 \times \ldots \times \tau_n$ and $\tau = \tau_i$. After applyng (*prj*) reduction, $e' = v_i$ and so $\Gamma \vdash e' : \tau_i$.

Case (*ctx*): By induction hypothesis, the type of the reduced sub-expression does not change and the last used rule in the derivation of $\Gamma \vdash e : \tau$ also applies on e' giving $\Gamma \vdash e' : \tau$. □

Theorem 19 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exits* e' *such that* $e \to e'$

Proof. By rule induction over $\vdash$.

Case (*num*): $e = n$ for some n and so e is a value.

Case (*abs*): $e = \lambda x.e'$ for some x, e', which is a value.

Case (*var*): This case cannot occur, because e is a closed expression.

Case (*app*): $e = e_1\ e_2$ which is not a value. By induction, e_1 is either a value or it can reduce. If it can reduce, apply (*ctx*) reduction with context $\_\ e$. Otherwise consider e_2. If it can reduce, apply (*ctx*) with context $v\ \_$. If both are values, Lemma 16 guarantees that $e_1 = \lambda x.e_1'$ and so we can apply reduction (*fn*).

Case (*proj*): $e = \pi_i e_0$ and $\tau = \tau_1 \times \ldots \tau_n$. If e_0 can be reduced, apply (*ctx*) with context $\pi_i\ \_$. Otherwise from Lemma 16, we have that $e_0 = (v_1, \ldots, v_n)$ and we can apply reduction (*prj*).

Case (*tup*): $e = (e_1, \ldots, e_n)$. If all sub-expressions are values, then e is also a value. Otherwise, we can apply reduction using (*ctx*) with a context $(v_1, \ldots, v_{i-1}, \_, e_{i+1}, \ldots, e_n)$. □

Theorem 20 (Safety). *If* $\Gamma \vdash e : \tau$ *and* $e \to^{*e'}$ *then either* e' *is a value or there exists* e'' *such that* $e' \to e''$

Proof. Rule induction over $\to^*$ using Theorem 18 and Theorem 19. □

5.3.3 *Comonadically-inspired translation*

In Section 5.2, we presented the semantics of the flat coeffect calculus in terms of indexed comonads. We treated the semantics as denotational – interpreting the meaning of a given typing derivation of a program in terms of category theory.

In this chapter, we use the same structure in a different way. Rather than treating the rules as *denotation* in categorical sense, we treat them as *translation* from a source domain-specific coeffect language into a target language with comonadically-inspired primitives described in the previous section.

LANGUAGE SYNTAX Given $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, extend the syntax:

$$\begin{array}{rcl} e & = & \ldots \mid \mathsf{cobind}_{s,r}\ e_1\ e_2 \mid \mathsf{counit}_{\mathsf{use}}\ e \mid \mathsf{merge}_{r,s}\ e \mid \mathsf{split}_{r,s}\ e \mid \mathsf{lift}_{r,r'}\ e \\ \tau & = & \ldots \mid C^r \tau \\ C & = & \ldots \mid \mathsf{cobind}_{s,r}\ \_\ e \mid \mathsf{cobind}_{s,r}\ v\ \_ \mid \mathsf{counit}_{\mathsf{use}}\ \_ \\ & & \mid \mathsf{merge}_{r,s}\ \_ \mid \mathsf{split}_{r,s}\ \_ \mid \mathsf{lift}_{r,r'}\ \_ \end{array}$$

TYPING RULES Given $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, add the typing rules:

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

$$(\textit{cobind})\ \frac{\Gamma \vdash e_1 : C^r \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : C^{r \circledast s} \tau_1}{\Gamma \vdash \mathsf{cobind}_{r,s}\ e_1\ e_2 : C^s \tau_2}$$

$$(\textit{merge})\ \frac{\Gamma \vdash e : C^r \tau_1 \times C^s \tau_2}{\Gamma \vdash \mathsf{merge}_{r,s}\ e : C^{r \wedge s} (\tau_1 \times \tau_2)}$$

$$(\textit{split})\ \frac{\Gamma \vdash e : C^{r \oplus s} (\tau_1 \times \tau_2)}{\Gamma \vdash \mathsf{split}_{r,s}\ e : C^r \tau_1 \times C^s \tau_2}$$

Figure 7: Comonadically-inspired extensions for the target langauge

LANGUAGE EXTENSION. Given a coeffect language with a flat coeffect algebra $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, we first extend the language syntax and typing rules with terms that correspond to the comonadically-inspired operations. This is done in the same way for all concrete coeffect domain-specific languages and so we give the additional syntax, evaluation context and typing rules in Figure 7 for all languages we consider later in Section 5.4.

The new type C^r represents an indexed comonad, which is left abstract for now. The additional expressions such as $\mathsf{counit}_{\mathsf{use}}$ and $\mathsf{cobind}_{r,s}$ correspond to the operations of indexed comonads. Note that the we embed the coeffect annotations into the target language – these are known when translationg a typed term from a source language and they will be useful when proving that sufficient context (as specified by the coeffect annotations) is available.

The figure defines the syntax and the typing rules, but it does not define the reduction rules. Those – together with the values for a concrete notion of context – will be defined separately for each individual coeffect language.

CONTEXTS AND TYPES. The interpretation of contexts and types in the category now becomes a translation from types and contexts in the source language into the types of the target language:

$$\begin{array}{rcl} [\![x_1 : \tau_1, \ldots, x_n : \tau_n \mathbin{@} r]\!] & = & C^r([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \\ [\![\tau_1 \xrightarrow{r} \tau_2]\!] & = & C^r [\![\tau_1]\!] \to [\![\tau_2]\!] \\ [\![\mathsf{num}]\!] & = & \mathsf{num} \end{array}$$

Here, a context becomes a comonadically-inspired data type wrapping a tuple of variable values and a coeffectful function is translated into an ordinary function in the target language with a comonadically-inspired data type wrapping the input type.

The translation is defined over a typing derivation:

$$\frac{}{\Gamma\,@\,\mathsf{use} \vdash x_i : \tau_i} \;=\; \frac{}{\lambda ctx.\pi_i\ (\mathsf{counit}_{\mathsf{use}}\ ctx)} \qquad (var)$$

$$\frac{}{\Gamma\,@\,\mathsf{ign} \vdash n : \mathsf{num}} \;=\; \frac{}{\lambda ctx.n} \qquad (num)$$

$$\frac{\Gamma, x_i : \tau_1\,@\,r \wedge s \vdash e : \tau_2}{\Gamma\,@\,r \vdash \lambda x_i.e : \tau_1 \xrightarrow{s} \tau_2} \;=\; \frac{\mathsf{f}}{\begin{array}{l}\lambda ctx.\lambda v.\\ \quad \mathsf{let}\ \mathsf{assoc} = \lambda x.\\ \qquad (\pi_1(\pi_1\ x), \ldots, \pi_{i-1}(\pi_1\ x), \pi_2\ x)\\ \quad \mathsf{f}\ (\mathsf{map}_{r\wedge s}\ \mathsf{assoc}\ (\mathsf{merge}_{r,s}\ (ctx, v)))\end{array}} \qquad (abs)$$

$$\frac{\begin{array}{rcl}\Gamma\,@\,r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 &=& \mathsf{f}\\ \Gamma\,@\,s \vdash e_2 : \tau_1 &=& \mathsf{g}\end{array}}{\Gamma r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2} \;=\; \frac{}{\begin{array}{l}\lambda ctx.\\ \quad \mathsf{let}\ ctx_0 = \mathsf{map}_{r\,\oplus\,(s\,\circledast\,t)}\ \mathsf{dup}\ ctx\\ \quad \mathsf{let}\ (ctx_1, ctx_2) = \mathsf{split}_{r,s\,\circledast\,t}\ ctx_0\\ \quad \mathsf{f}\ ctx_1\ (\mathsf{cobind}_{s,t}\ \mathsf{g}\ ctx_2)\end{array}} \qquad (app)$$

Assuming the following auxiliary operations:

$$\begin{array}{rcl}\mathsf{map}_r\ \mathsf{f} &=& \mathsf{cobind}_{\mathsf{use},r}\ (\lambda x.\mathsf{f}\ (\mathsf{counit}_{\mathsf{use}}\ x))\\ \mathsf{dup} &=& \lambda x.(x, x)\end{array}$$

Figure 8: Translation from a flat DSL to a comonadically-inspired target language

EXPRESSIONS. The rules shown in Figure 8 define how expressions of the source language are translated into the target language. The rules are very similar to those shown earlier in Figure 5. The result is now written as source code in the target programming language rather than as composition of morphisms in a category. However, thanks to the equivalence between λ-calculus and category theory, both interpretations are equivalent.

One change from Figure 5 is that we are now more explicit about the tuple that contains variable assignments. Previously, we assumed that the tuple is appropriately reassociated. Now, we perform the reassociation explicitly. We keep a flat tuple of variables, so given $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, the tuple has a type $\tau_1 \times \ldots \times \tau_n$. In (*var*), we access a variable using π, but in (*abs*), the merge operation produces a tuple $(\tau_1 \times \ldots \times \tau_{i-1}) \times \tau_i$ that we turn into a flat tuple $\tau_1 \times \ldots \times \tau_{i-1} \times \tau_i$ using the assoc function.

PROPERTIES. The most important property of the translation is that it produces well-typed programs in the target language. This is akin to the correspondence property of the semantics discussed earlier (Theorem 14), but now it has more obvious practical consequences.

In Section 5.4, we will prove safety properties of well-typed programs in the target language. Thanks to the fact that the translation produces a well-typed program (as complex as it might be) means that we are also proving safety of well-typed programs in the source context aware languages.

Theorem 21 (Well-typedness of the translation)**.** *Given a typing derivation for a well-typed closed expression* $@\,r \vdash e : \tau$ *written in a context-aware programming langugae that is translated to the target langauge as:*

$$\frac{(\ldots)}{\Gamma\,@\,r \vdash e : \tau} \;=\; \frac{(\ldots)}{f}$$

Then f *is well-typed, i. e. in the target langauge:* $\vdash f : [\![\Gamma\,@\,r]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation. Given a judgement $x_1 : \tau_1 \ldots x_n : \tau_n\,@\,c \vdash e : \tau$, the translation constructs a function of type $C^c([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$.

Case (*var*): $c = \mathsf{use}$ and $\tau = \tau_i$ and so $\pi_i(\mathsf{counit}_{\mathsf{use}}\ ctx)$ is well-typed.

Case (*num*): $\tau = \mathsf{num}$ and so the body n is well-typed.

Case (*abs*): The type of ctx is $C^r(\ldots)$ and the type of v is $C^s\tau_1$, calling $\mathsf{merge}_{r,s}$ and reassociating produces $C^{r \wedge s}(\ldots)$ as expected by f.

Case (*num*): After applying $\mathsf{split}_{r,s \circledast t}$, the types of ctx_1, ctx_2 are $C^r(\ldots)$ and $C^{s \circledast t}(\ldots)$, respectively. g requires $C^s(\ldots)$ and so the result of $\mathsf{cobind}_{s,t}$ is $C^t\tau_1$ as required by f. □

5.4 SAFETY OF CONTEXT-AWARE LANGUAGES

The previous two languages provide a general structrue that we now use to prove the safety of concrete context-aware programming languages based on the coeffect language framework. We consider a langauge for dataflow computations (Section 5.4.1) and for implicit parameters (Section **??**). In both cases, we extend the progress and preservation theorems of the functional subset of the target language, but the approach can be generalised as discussed in Section 5.5.

5.4.1 *Coeffect langauge for dataflow*

The types of the comonadically-inspired operations are the same for each concrete coeffect DSL, but each DSL introduces its own *values* of type $C^r\tau$ and also its own reduction rules that define how comonadically-inspired operations evaluate.

We first consider dataflow computations. As discussed earlier in the semantics of dataflow, the indexed comonad for a context with n past values carries $n + 1$ values. When reducing translated programs, the comonadic values will not be directly manipulated by the user code. In a programming language, it could be seen as an *abstract data type* whose only operations are the comonadically-inspired ones defined earlier.

The Figure 9 shows extensions to the target language for modelling dataflow computations. We introduce a new kind of values written as $\mathsf{Df}\langle v_0, \ldots, v_n\rangle$ and a matching kind of expressions. We add a language primitive modelling the semantics of the `prev` keyword together with its typing rule (*prev*) and we also add a typing rule (*df*) that checks the types of the elements of the stream and also guarantees that the number of elements in the stream matches the number in the coeffect.

The additional reduction rules mirror the semantics that we discussed earlier when talking about indexed dataflow comonad.

LANGUAGE SYNTAX

$$\begin{array}{rcl} v & = & \ldots \mid \mathsf{Df}\langle v_0, \ldots, v_n \rangle \\ e & = & \ldots \mid \mathsf{Df}\langle e_0, \ldots, e_n \rangle \mid \mathsf{prev}_n\ e \\ C & = & \ldots \mid \mathsf{prev}_n\ \_ \end{array}$$

TYPING RULES

$$(df)\ \frac{\forall i \in \{0 \ldots n\}.\ \Gamma \vdash e_i : \tau}{\Gamma \vdash \mathsf{Df}\langle e_0, \ldots, e_n \rangle : C^n \tau}$$

$$(prev)\ \frac{\Gamma \vdash e : C^{n+1}\tau}{\Gamma \vdash \mathsf{prev}_n\ e : C^n \tau}$$

TRANSLATION

$$\frac{\Gamma @ n \vdash e : \tau}{\Gamma @ n+1 \vdash \mathsf{prev}\ e : \tau} = \frac{f}{\lambda ctx.\mathsf{prev}_n\ ctx}$$

REDUCTION RULES

$$\begin{array}{ll} (counit) & \mathsf{counit}_0(\mathsf{Df}\langle v_0 \rangle) \to v_0 \\ (cobind) & \mathsf{cobind}_{m,n}\ f\ (\mathsf{Df}\langle v_0, \ldots v_{m+n} \rangle) \to \\ & \quad (\mathsf{Df}\langle f(\mathsf{Df}\langle v_0, \ldots, v_m \rangle), \ldots, f(\mathsf{Df}\langle v_n, \ldots, v_{m+n} \rangle) \rangle) \\ (merge) & \mathsf{merge}_{m,n}((\mathsf{Df}\langle v_0, \ldots, v_m \rangle), (\mathsf{Df}\langle v'_0, \ldots, v'_n \rangle)) \to \\ & \quad (\mathsf{Df}\langle (v_0, b_0), \ldots, (v_{min(m,n)}, v'_{min(m,n)}) \rangle) \\ (split) & \mathsf{split}_{m,n}(\mathsf{Df}\langle (v_0, b_0), \ldots, (v_{max(m,n)}, b_{max(m,n)}) \rangle) \to \\ & \quad \mathsf{Df}\langle v_0, \ldots, v_m \rangle, (\mathsf{Df}\langle b_0, \ldots, b_n \rangle \\ (prev) & \mathsf{prev}_n(\mathsf{Df}\langle v_0, \ldots, v_n, v_{n+1} \rangle) \to \\ & \quad \mathsf{Df}\langle v_0, \ldots, v_n \rangle \end{array}$$

Figure 9: Additional constructs for modelling dataflow in the target langauge

PROPERTIES. Now consider a target langauge consisting of the core (ML-subset) defined by the syntax, reduction rules and typing rules given in Figure 6 and comonadically-inspired primtives defined in Figure 7 and also concrete notion of comonadically-inspired value and reduction rules for dataflow as defined in Figure 9.

In order to prove type safety, we first extend the canonical forms lemma (Lemma 16) and the preservation under substitution lemma (Lemma 17). Those need to consider the new (*df*) and (*prev*) typing rules and substitution under the newly introduced expression forms $\mathsf{Df}\langle \ldots \rangle$ and prev_n. Then we extend the type preservation (Theorem 18) and progress (Theorem 19).

Lemma 22 (Canonical forms). *For all* e, τ, *if* $\vdash e : \tau$ *and* e *is a value then:*

1. *If* $\tau = \mathsf{num}$ *then* $e = n$ *for some* $n \in \mathbb{Z}$
2. *If* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'
3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i
4. *If* $\tau = C^n \tau_1$ *then* $e = \mathsf{Df}\langle v_0, \ldots v_n \rangle$ *for some* v_i

Proof. (1,2,3) as before; for (4) the last typing rule must have been (*df*). □

Lemma 23 (Preservation under substitution). *For all* $\Gamma, e, e'\tau, \tau'$, *if* $\Gamma, x : \tau \vdash e : \tau'$ *and* $\Gamma \vdash e' : \tau$ *then* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

Proof. By induction over the derivation of $\Gamma, x : \tau \vdash e : \tau'$ as before, with new cases for $\mathsf{Df}\langle\ldots\rangle$ and prev_n. □

Theorem 24 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\to$.

Case (*fn*, *prj*, *ctx*): As before, using Lemma 23 for (*fn*).

Case (*counit*): $e = \mathsf{counit}_0(\mathsf{Df}\langle v_0\rangle)$. The last rule in the type derivation of e must have been (*counit*) with $\Gamma \vdash \mathsf{Df}\langle v_0\rangle : C^0\tau$ and therefore $\Gamma \vdash v_0 : \tau$.

Case (*cobind*): $e = \mathsf{cobind}_{m,n}\ f\ (\mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle)$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^n\tau_2$ and assumptions $\Gamma \vdash f : C^m\tau_1 \to \tau_2$ and $\Gamma \vdash \mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle : C^{m+n}\tau$. The reduced expression has a type $C^n\tau_2$:

$$\frac{\dfrac{\Gamma \vdash f : C^m\tau_1 \to \tau_2 \qquad \forall i \in 0\ldots n.\ \Gamma \vdash \mathsf{Df}\langle v_i, \ldots, v_{i+m}\rangle : C^m\tau_1}{\forall i \in 0\ldots n.\ \Gamma \vdash f(\mathsf{Df}\langle v_i, \ldots, v_{i+m}\rangle) : \tau_2}}{\Gamma \vdash \mathsf{Df}\langle f(\mathsf{Df}\langle v_0, \ldots, v_m\rangle), \ldots, f(\mathsf{Df}\langle v_n, \ldots, v_{m+n}\rangle)\rangle : C^n\tau_2}$$

Case (*merge*, *split*, *next*): Similar. In all three cases, the last typing rule in the derivation of e guarantees that the stream contains a sufficient number of elements of correct type. □

Theorem 25 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exits* e' *such that* $e \to e'$

Proof. By rule induction over $\vdash$.

Case (*num,abs,var,app,proj,tup*): As before, using the adapted canonical forms lemma (Lemma 22) for (*app*) and (*proj*).

Case (*counit*): $e = \mathsf{counit}_{\mathsf{use}}\ e_1$. If e_1 is not a value, it can be reduced using (*ctx*) with context $\mathsf{counit}_{\mathsf{use}}\ \_$, otherwise it is a value. From Lemma 22, $e_1 = \mathsf{Df}\langle v\rangle$ and so we can apply (*counit*) reduction rule.

Case (*cobind*): $e = \mathsf{cobind}_{m,n}\ e_1\ e_2$. If e_1 is not a value, reduce using (*ctx*) with context $\mathsf{cobind}_{m,n}\ \_\ e$. If e_2 is not a value reduce using (*ctx*) with context $\mathsf{cobind}_{m,n}\ v\ \_$. If both are values, then from Lemma 22, we have that $e_2 = \mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle$ and so we can apply the (*cobind*) reduction.

Case (*merge*): $e = \mathsf{merge}_{m,n}e_1$. If e_1 is not a value, reduce using (*ctx*) with context $e = \mathsf{merge}_{m,n}\ \_$. If e_1 is a value, it must be a pair of streams $(\mathsf{Df}\langle v_0, \ldots, v_m\rangle, \mathsf{Df}\langle v'_0, \ldots, v'_n\rangle)$ using Lemma 22 and it can reduce using (*merge*) reduction.

Case (*split*, *prev*): Similar. Either sub-expression is not a value, or the type guarantees that it is a stream with correct number of elements to enable the (*split*) or (*prev*) reduction, respectively. □

Theorem 26 (Safety). *If* $\Gamma \vdash e : \tau$ *and* $e \to^{*e'}$ *then either* e' *is a value or there exists* e'' *such that* $e' \to e''$

Proof. Rule induction over $\to^*$ using Theorem 24 and Theorem 25. □

5.5 GENERALISING

... works for any good comonad...

5.6 RELATED AND FUTRUE 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. Instead, we discuss one specific question that often arises when discussing coeffects and that is *when is a 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.

5.6.1 *Properties and related notions*

The flat indexed comonad structure is all we need to give the semantics of the flat coeffect calculus. Before doing so in Section **??**, we briefly consider additional properties and other categorical structures that have been proposed mainly in the context of monads and effects and we look how they relate to indexed comonads.

SHAPE PRESERVATION. Ordinary comonads have the *shape preservation* property [12]. Intuitively, this means that the core comonad structure does not provide a way of modeling computations where the additional context changes 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 comonad, in the computation $\mathsf{cobind}_{r,s}\,f$, the shape of the context changes from providing implicit parameters $r \cup s$ to providing just implicit parameters s.

FAMILIES OF MONADS. When linking effect systems and monads, Wadler and Thiemann [8] propose a *family of monads* as the categorical structure. The dual structure, *family of comonads*, is defined as follows.

Definition 8. *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.*

A *family of comonads* is more restrictive than an *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 [4] (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 the unit operation only on the unit of a monoid and the bind operation composes effect annotations using the provided monoidal structure.

5.6.2 *When is coeffect not a monad*

Coeffect systems differ from effect systems in three important ways:

- Semantically, coeffects capture 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 ($\oplus$, $\circledast$, and $\wedge$) while most effect systems only use a single operation for sequential composition (used by 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 (or duplicated at) declaration site and call site, while monadic effect systems always defer 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, to enable splitting of implicit parameter requirements using the reader monad, we need to extend the monad structure and change the translation of monadic lambda abstraction.

5.6.3 *When is coeffect a monad*

Implicit parameters can be captured by a monad, but *just* a monad is not enough. Lambda abstraction in effect systems does not provide a way of splitting the context requirements between declaration site and call site (or, semantically, combining the implicit parameters available in the scope where the function is defined 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 [11], when used to model computation semantics, the two representations are equivalent:

Remark 27. *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 reader monad via currying/uncurrying isomorphism.*

Proof. The isomorphism is demonstrated by the following equation:

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

□

This 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 by the (*abs*) rules for monadic effect systems and comonadic coeffect systems. The following listing shows the two rules side-by-side, using the effect system notation for both of them:

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

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 performs 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 at run-time.

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, 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 an indexed comonad $C^{r}\alpha$), combines their carried values and additional contextual information (implicit parameters). To ob-

tain the (*mabs*) rule, we can restrict the first parameter, which corresponds to the declaration site context:

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

In the (*restricted*) version of the operation, the declaration site context requires no implicit parameters and so all implicit parameters have to be satisfied 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. The ability to restrict operations to partial functions has been used in the semantics of effectful computations by Tate [16].

5.7 SUMMARY

Next, we introduced the notion of *flat indexed comonad*, which generalizes of comonads and adds 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.

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 **??**.

BIBLIOGRAPHY

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] R. Milner. *The Definition of Standard ML: Revised*. MIT press, 1997.

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

[9] D. L. Niki Vazou. Remarrying effects and monads. *Proceedings of MSFP (to appear)*, 2014.

[10] D. Orchard. Programming contextual computations.

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

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

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

[14] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

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

[16] 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.

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

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

[19] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1):38–94, 1994.