CONTENTS

Part II

COEFFECT CALCULI

In this part, we capture the similarities between the concrete context-aware langauges presented in the previous chapter. We also develop the key novel technical contributions of the thesis. We define a *flat coeffect type system* (Chapter 1) that is parameterized by a *coeffect algebra* and a mechanism for choosing unique typing derivation. We instantiate a coeffect type system with a concrete coeffect algebra and procedure for choosing unique typing derivation for three languages to capture dataflow, implicit parameters and liveness.

The type system is complemented with a translational semantics for coeffect-based context-aware programming languages (Chapter 2). The semantics is inspired by a categorical model based on *indexed comonads* and it translates source context-aware program into a target program in a simple functional language with comonadically-inspired primitives. We give concrete definition of the primitives for dataflow, implicit parameters and liveness and present a syntactic safety proof for these three languages.

The following page provides a detailed overview of the content of Chapters 1 and Chapters 2, highlighting the split between general definitions and properties (about the coeffect calculus) and concrete definitions and properties (about concrete context-awre language). The Chapter **??** mirrors the same development for *structural coeffect systems*.

CHAPTER 4		
	Coeffect calculus	Language-specific
Syntax	Coeffect λ-calculus (Section 1.2)	Extensions such as ?param and prev (Section 1.2.4)
Type system	Abstract coeffect algebra (Section 1.2.1)	Concrete instances of the coeffect algebra (Section 1.2.4)
	Coeffect type system parameterized by the coeffect algebra (Section 1.2.2)	Typing for language-specific extensions (Section 1.2.4)
		Procedure for determining a unique typing derivation (Section 1.3)
Properties	Syntactic properties of coeffect λ-calculus (Section 1.4)	Uniqueness of the above (Section 1.3)

CHAPTER 5		
	Coeffect calculus	Language-specific
Categorical	Indexed comonads (Sectiuon 2.2.4)	Examples including indexed product, list and maybe comonads (Section 2.2.5)
	Categorical semantics of coeffect λ-calculus (Section 2.2.6)	
Translational	Functional target language (Section 2.3.1)	
	Translation from coeffect λ-calculus to target language (Section 2.3.3)	Translation for language-specific extensions (prev, ?p) (Sections 2.4.1 and 2.4.2)
Operational	Abstract comonadically-inspired primitives (Section 2.3.3)	Concrete reduction rules for comonadically-inspired primitives (Sections 2.4.1 and 2.4.2)
		Reduction rules for language-specific extensions (prev, ?p) (Sections 2.4.1 and 2.4.2)
	Sketch of generalized syntactic soundness (Section 2.5)	Syntactic soundness (Sections 2.4.1 and 2.4.2)

1 TYPES FOR FLAT COEFFECTS

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 **??**, 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* (Chapter 1 and Chapter 2) and *structural coeffects* (Chapter **??**) separately.

In this chapter, we develop a *flat coeffect calculus* that provides a type system for tracking per-context properties of context-aware programming languages. The *coeffect calculus* captures the shared properties of such languages. It is parameterized by a *flat coeffect algebra* and can be instantiated to track implicit parameters, liveness and number of required past values in dataflow languages. To capture contextual properties in full generality, the flat coeffect calculus permits multiple valid typing derivations for a given term. To resolve the ambiguity arising from such generality, each concrete context-aware language is also equipped with an algorithm for choosing a unique typing derivation. This allows us to explore the language design landscape, while still follow the usual scoping rules for languages with established approaches (e.g. implicit parameters in Haskell).

In the next chapter, we give operational meaning for concrete coeffect languages based on the flat coeffect calculus 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 1.2). We show that the system can be instantiated to obtain three of the systems discussed in Section **??**, namely implicit parameters, liveness and dataflow.
- The coeffect calculus permits multiple typing derivations due to the ambiguity inherent in contextual lambda abstraction. Each concrete context-aware language based on the coeffect calculus must specify how such ambiguities are to be resolved. We give the procedure for choosing unique typing derivation for our three examples (Section 1.3).
- We discuss equational properties of the calculus, covering type-preservation for call-by-name and call-by-value reduction (Section 1.4). We also extend the calculus with subtyping and pairs (Section 1.5).

1.1 INTRODUCTION

In the previous chapter, we looked at three examples of systems that track whole-context properties. The type systems for whole-context liveness (Section **??**) and whole-context dataflow (Section **??**) have a similar structure in two ways. First, lambda abstraction duplicates their *context demands*. Given a body with context demands r, the declaration site context *as well as* the function arrow are annotated with r. Second, the context demands in the type systems are combined using two different operators (representing sequential and pointwise operations).

The system for tracking implicit parameters (Section **??**) differs. In lambda abstraction, it partitions the context demands between the declaration site and the call site. Furthermore, the operator that combines context demands is $\cup$ for both sequential and pointwise composition.

Despite the differences, the systems fit the same framework. This becomes apparent when we consider the categorical structure (Section **??**). Rather than starting from the categorical semantics, we first explain how the systems can be unified syntactically (Section 1.1.1) and then provide the semantics as an additional justification.

The development in this chapter can be seen as a counterpart to the well-known development of *effect systems* [11]. Chapter 2 then links *coeffects* with *comonads* in the same way in which effect systems have been linked with monads [21]. The syntax and type system of the flat coeffect calculus follows a similar style as effect systems [17, 36], but differs in the structure of lambda abstraction as discussed briefly here and in Section **??** (the relationship with monads is further discussed in Section 2.6).

1.1.1 *A unified treatment of lambda abstraction*

Recall the lambda abstraction rules for the implicit parameters coeffect system (annotating contexts with sets of required parameters) and the dataflow system (annotating contexts 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 rewriting the (*df1*) rule such that the annotation in the assumption is in the form nm for some operation . For the dataflow system, this can be achieved by using the *min* function:

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

The rule (*df1*) is admissible in a system that includes the (*df2*) rule. That is, a typing derivation using (*df1*) is also valid when using (*df2*). Furthermore, if we include sub-typing rule (on annotations of functions) and subcoeffecting rule (on annotations of contexts), then the reverse is also true – because $\mathit{min}(n, m) \leqslant m$ and $\mathit{min}(n, m) \leqslant n$. In other words (*df2*) permits an implicit subcoeffecting (and sub-typing) that is not possible when using the (*df1*) rule, but it has a structure that can be unified with (*param*).

1.2 FLAT COEFFECT CALCULUS

This section describes the *flat coeffect calculus*. A small programming language based on the λ-calculus with a type system that statically tracks context demands. The calculus can capture different notions of context. The structure of context demands is provided by a *flat coeffect algebra* (defined in the next section) which is an abstract algebraic structure that can be instantiated to model concrete context demands (sets of implicit parameters, number of past values as integers or other information). Annotations that specify context demands are written as r, s, t.

We enrich types and typing judgements with coeffect annotations r, s, t; typing judgements are written as $\Gamma @ r \vdash e : \tau$. 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:

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

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 1.3 where we discuss how to find a unique typing derivation for context-aware computations. Using Church-style lambda abstraction, we can directly 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 1.5. The type $\tau_1 \xrightarrow{r} \tau_2$ represents a function from τ_1 to τ_2 that requires additional context r. It can be viewed as a pure function that takes τ_1 *with* or *wrapped in* a context r.

In the 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 syntactic 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.

1.2.1 *Flat coeffect algebra*

To make the flat coeffect system general enough, the algebra consists of three operations. Two of them, $\circledast$ and $\oplus$, represent *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 demands.

In addition to the three operations, the algebra also requires 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 $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$, binary 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, $(\mathcal{C}, \wedge)$ is a band (idempotent semigroup) and $(\mathcal{C}, \leqslant)$ is a pre-order. That is, for all $r, s, t \in \mathcal{C}$:*

$$\begin{array}{rcc} r \circledast (s \circledast t) = (r \circledast s) \circledast t & & \mathsf{use} \circledast r = r = r \circledast \mathsf{use} \\ r \oplus (s \oplus t) = (r \oplus s) \oplus t & r \oplus r = r & \mathsf{ign} \oplus r = r = r \oplus \mathsf{ign} \\ r \wedge (s \wedge t) = (r \wedge s) \wedge t & & r \wedge r = r \\ \text{if}\ \ r \leqslant s\ \ \text{and}\ \ s \leqslant t\ \ \text{then}\ \ r \leqslant t & & t \leqslant t \end{array}$$

In addition, the following distributivity axioms hold:

$$\begin{aligned} (r \oplus s) \circledast t &= (r \circledast t) \oplus (s \circledast t) \\ t \circledast (r \oplus s) &= (t \circledast r) \oplus (t \circledast s) \end{aligned}$$

In two of the three systems, some of the operators of the flat coeffect algebra coincide, but in the dataflow system all three are distinct. Similarly, the two special elements coincide in some, but not all systems. The required axioms 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 monoid axioms 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 axioms guarantee that usual syntactic transformations on tuples and the unit value (Section 1.5) 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 demand makes it possible to duplicate the given coeffects and place the same demand on both call site and declaration site. Using the example from Section 1.1.1, 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 dataflow computations, they are *max* and *min* respectively. However, this duality does not hold for implicit parameters (we discuss the lattice-based formulation of coeffects in Section **??**.

ORDERING. The flat coeffect algebra includes a pre-order relation $\leqslant$. This will be used to introduce subcoeffecting and subtyping in Section 1.5.1, 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 $(\mathcal{C}, \oplus, \mathsf{ign})$ is also commutative (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 (Chapter **??**), where it fails for the bounded reuse calculus. For this reason, we choose not to use it for flat coeffect calculus either.

Furthermore, the use coeffect is often the top or the bottom element of the semi-lattice. As discussed in Section 1.4, when this is the case, we are able to prove certain syntactic properties of the calculus.

1.2.2 *Type system*

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 1.1.1. The body is annotated with context demands $r \wedge s$, which are then split between the context-demands on the declaration site r and context-demands on the call site s.

In function application (*app*), context demands of both expressions and the function are combined. As discussed in Chapter **??**, sequential composition is used to combine the context-demands of the argument s with the context-demands of the function t. The result $s \circledast t$ is then composed using pointwise composition with the context demands of the expression that represents the function r, giving the coeffect $r \oplus (s \circledast t)$.

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

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

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

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

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

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

Figure 1: Type system for the flat coeffect calculus

The type system also includes a rule for let-binding. The rule is *not* equivalent to the derived rule for $(\lambda x.e_2)\ e_1$, but it corresponds to *one* possible typing derivation. As we show in 1.5.2, the typing used in (*let*) is more precise than the general rule that can be derived from $(\lambda x.e_2)\ e_1$.

To guide understanding of the system, we also show non-syntax-directed (*sub*) rule for subcoeffecting. The rule states that an expression with context demands r' can be treated as an expression with greater context demands r. We return to subcoeffecting, subtyping and additional constructs such as pairs in Section 1.5. When discussing procedure for choocing unique typing in Section 1.3, we consider only the syntax-directed part of the system.

1.2.3 *Understanding flat coeffects*

Before proceeding, let us clarify how the typing judgements should be understood. The coeffect calculus can be understood in two ways discussed in this and the next chapter. As a type system (Chapter 1), it provides analysis of context dependence. As a semantics (Chapter 2), it specifies how context is propagated. These two readings provide different ways of interpreting the judgements $\Gamma @ r \vdash e : \tau$ and the typing rules used to define it.

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

  Syntactically, the typing rules are best read top-down (from assumptions to the consequent). In function application, the context demands of multiple assumptions (arising from two sub-expressions) are *merged*; in lambda abstraction, the demands 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 (from the consequent to assumptions). 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.

For example, using the syntactic reading, the operators $\wedge$ and $\oplus$ represent *merging* and *splitting* of context demands – in the (*abs*) rule, $\wedge$ appears in the assumption and the combined context demands of the body are split between two positions in the conclusions; in the (*app*) rule, $\oplus$ appears in the conclusion and combines two context demands from the assumptions.

The reason for this asymmetry follows from the fact that the context appears in a *negative position* in the semantic model (Section 2.2). 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 demands* when using the syntactic view, especially in the rest of Chapter 1, and *context capabilities* or just *available context* when using the semantic view, especially in Chapter 2.

1.2.4 *Examples of flat coeffects*

The flat coeffect calculus generalizes the three flat systems discussed in Section **??** of the previous chapter. We can instantiate it to a specific use just by providing a flat coeffect algebra.

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

For simplicity, 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 axioms 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:

$$e ::= \ \dots \mid ?\mathsf{p} \mid \mathtt{let}\ ?\mathsf{p} = e_1\ \mathtt{in}\ e_2$$

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

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

The (*param*) rule specifies that the accessed parameter $?\mathsf{p}$ 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 also 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 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 it is a valid derivation that could also be obtained via sub-typing.

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

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

As required by the axioms, $(\mathbb{N}, +, 0)$ and $(\mathbb{N}, \mathit{max}, 0)$ form monoids and $(\mathbb{N}, \mathit{min})$ forms a band. Note that dataflow is our first example where $\circledast$ is not idempotent. The distributivity axioms require the following to be the case: $\mathit{max}(r, s) + t = \mathit{max}(r + t, s + t)$, 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:

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

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

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

Example 4 (Optimized dataflow)**.** *In optimized dataflow, context is annotated with natural numbers extended with the* $\perp$ *element, that is* $\mathbb{N}_\perp = \{\perp, 0, 1, 2, 3, \ldots\}$ *such that* $\forall n \in \mathbb{N}. \perp \leqslant n$*. The flat coeffect algebra is* $(\mathbb{N}_\perp, +, \mathit{max}, \mathit{min}, 0, \perp, \leqslant)$ *where* $m + n$ *is* $\perp$ *whenever* $m = \perp$ *or* $n = \perp$ *and* min*,* max *treat* $\perp$ *as the least element.*

Note that $(\mathbb{N}_\perp, +, 0)$ is a monoid for the extended definition of $+$; for the bottom element $0 + \perp = \perp$ and for natural numbers $0 + n = n$. The structure $(\mathbb{N}, \mathit{max}, \perp)$ is also a monoid, because $\perp$ is the least element and so $\mathit{max}(n, \perp) = n$. Finally, $(\mathbb{N}, \mathit{min})$ is a band (the extended *min* is still idempotent and associative) and the distributivity axioms also hold for $\mathbb{N}_\perp$.

1.3 CHOOSING A 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 demands. In case of implicit parameters (Section **??**), the demands can be satisfied either by the call-site or by the declaration-site. In case of dataflow and liveness, the rule discussed in Section 1.2 reintroduces similar ambiguity because it allows multiple valid typing derivations.

Furthermore, the semantics of context-aware languages in Chapter **??** and also in Chapter 2 is defined over *typing derivation* and so the meaning of a program depends on the typing derivation chosen. In this section, we specify how to choose the desired *unique* typing derivation in each of the coeffect systems we consider.

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 = 42 in (λy. ?x)) in
let ?x = 666 in f 0
```

There are two possible typings allowed by the typing rules discussed in Section 1.2.2 that lead to two possible meanings of the program – evaluating to 1 and 2, respectively:

- $\mathsf{f} : \mathsf{num} \xrightarrow{\emptyset} \mathsf{num}$ – in this case, the value of ?x is captured from the declaration-site and the program produces 1.
- $\mathsf{f} : \mathsf{num} \xrightarrow{\{?\mathsf{x}\}} \mathsf{num}$ – 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 the above case, one typing derivation represents dynamic binding and the other static binding, but more subtleties arise when the nested expression uses multiple implicit parameters.

In this section, we discuss the specific choices of typing derivation for implicit parameters, dataflow and liveness. We use the fact that the coeffect calculus uses Church-style syntax for lambda abstraction giving a type annotation for the bound variable. This does not affect the handling of coeffects (those are not defined by the type annotation), but it lets us prove *uniqueness of typing*; a theorem showing that we define a *unique* way of assigning coeffects to otherwise well-typed programs.

1.3.1 *Implicit parameters*

For implicit parameters we choose to follow the behaviour implemented by Haskell [16] where function abstraction captures all parameters that are statically available at the declaration-site and places all other demands on the call-site. For the example above, 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 a numerical value 42.

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

$$\Gamma; \Delta @ r \vdash e : \tau$$

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*)

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

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

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

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

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

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

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

Figure 2: Choosing unique typing for implicit parameters

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 any 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 in a type system presented in Figure 2 then this identifies the unique prefered derivation for the program. We use this unique typing derivation to give the semantics of coeffect language with implicit parameters in Chapter 2 and we also implement this algorithm as discussed in Chatper 3.

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 follow the standard approach [30] and first give the inversion lemma (Lemma 1).

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

1. *If* $\Gamma ; \Delta @ c \vdash x : \tau$ *then* $x : \tau \in \Gamma$ *and* $c = \emptyset$.
2. *If* $\Gamma ; \Delta @ c \vdash n : \tau$ *then* $\tau = \mathsf{num}$ *and* $c = \emptyset$.
3. *If* $\Gamma ; \Delta @ c \vdash e_1\ e_2 : \tau_2$ *then there is some* τ_1, r, s *and* t *such that* $\Gamma ; \Delta @ r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma ; \Delta @ s \vdash e_2 : \tau_1$ *and also* $c = r \cup s \cup t$.

4. *If* $\Gamma;\Delta @ c \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2$ *then there is some* τ_1, s *and* r *such that* $\Gamma;\Delta @ r \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1;\Delta @ s \vdash e_2 : \tau_2$ *and also* $c = s \cup r$.

5. *If* $\Gamma;\Delta @ c \vdash {?p} : \mathsf{num}$ *then* ${?p} \in c$ *and* $c = \{?p\}$.

6. *If* $\Gamma;\Delta @ c \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* τ_2 *such that* $\tau = \tau_1 \xrightarrow{s} \tau_2$ $\Gamma, x{:}\tau_1;\Delta @ r \vdash e : \tau_2$ *and* $c = \Delta$ *and also* $s = r \setminus \Delta$.

7. *If* $\Gamma;\Delta @ c \vdash \texttt{let}\ {?p} = e_1\ \texttt{in}\ e_2 : \tau$ *then there is some* r, s *such that* $\Gamma;\Delta @ r \vdash e_1 : \mathsf{num}$ *and* $\Gamma;\Delta \cup \{?p\} @ s \vdash e_2 : \tau$ *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* $\Gamma;\Delta @ r \vdash e : \tau$ *and* $\Gamma;\Delta @ r' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $r = r'$.

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

Case (*abs*): $e = \lambda x : \tau_1.e_1$ and $c = \Delta$. $\tau = \tau_1 \xrightarrow{r\setminus\Delta} \tau_2$ for some r, τ_2 and also $\Gamma, x : \tau_1;\Delta @ r \vdash e : \tau_2$. 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 $\Gamma, x{:}\tau_1;\Delta @ r \vdash e : \tau_2'$. 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*). □

Finally, we note that unique typing derivations obtained using the type system given in Figure 2 are valid typing derivation under the original flat coeffect type system in Figure 1.

Theorem 3 (Admisibility of unique typing for implicit parameters)**.** *If* $\Gamma;\Delta @ r \vdash e : \tau$ *(using the rules in Figure 2) then also* $\Gamma @ r \vdash e : \tau$ *(using the rules in Figure 1 and Example 1).*

Proof. Each typing rule in the unique type derivation is a special case of the corresponding typing rule in the flat coeffect calculus (ignoring the additional context Δ); the splitting of coeffects in (*abs*) in Figure 2 is a special case of splitting two sets using $\cup$. □

1.3.2 *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:

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

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 liveness and dataflow, the (*idabs*) rule provides the most precise coeffect. Assume we have a lambda abstraction with a 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 $r = \mathit{min}(s, t)$. The smallest s, t such that the equality holds are $s = t = r$.

– For liveness, 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 1.5.3 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 2) and it is used in the implementation (Chatper 3).

We note that the unique typing derivation is admissible in the original coeffect type system. For dataflow and liveness, this follows directly from the fact that (*idabs*) is a special case of the (*abs*) rule and so we do not state this explicitly as in Theorem 3 for implicit parameters.

In order to prove the uniqueness of typing theorem (Theorem 5), we first need the inversion lemma (Lemma 4).

Lemma 4 (Inversion lemma for liveness and dataflow)**.** *For the type system defined in Figure 1 with the (abs) rule replaced by (idabs):*

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

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

Theorem 5 (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* $\Gamma @ r \vdash e : \tau$ *and* $\Gamma @ r' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $r = r'$.

Proof. Suppose that (A) $\Gamma @ c \vdash e : \tau$ and (B) $\Gamma @ c' \vdash e : \tau'$. We show by induction over the typing derivation of $\Gamma @ c \vdash e : \tau$ 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 $\Gamma, x{:}\tau_1 @ c \vdash e : \tau_2$. By case (5) of Lemma 4, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma, x{:}\tau_1 @ c' \vdash e : \tau_2'$. 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*). □

1.4 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 dataflow is more complex.

1.4.1 *Syntactic properties*

Before discussing the syntactic properties of general coeffect calculus formally, it should be clarified what is meant by providing a "pathway to operational semantics" in this section. We do that by contrasting syntactic properties of coeffect systems with more familiar effect systems. Writing $e_1[x \leftarrow e_2]$ for 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, together with additional domain-specific rules for each context-aware language, 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 1.4.2) and call-by-name (Section 1.4.3) in more details.

The focus of this chapter is on the general coeffect system and so we do not discuss the domain-specific reduction rules for individual context-aware language. Some work on operational semantics of general coeffect systems has been done by Brunel et al. [6]. We give formal semantics of implicit parameters and dataflow in Chapter 2 by translation to a simple functional programming language instead.)

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, the notion of value is defined syntactically. For example, in the *Effect* language [42], 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 demands of the body e and may have immediate context demands. Thus we say that e is a value if it is a value in the usual sense *and* has no immediate context demands. We define this formally in Section 1.4.2.

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 domain-specific semantics needs to provide a way for reducing a context-dependent term $\Gamma @ r \vdash e : \tau$ to a value, i. e. a term $\Gamma @ \mathsf{use} \vdash e' : \tau$ with no context demands.

1.4.2 *Call-by-value evaluation*

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

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

The syntactic form *SynVal* captures syntactic values, but a context-dependency-free value in coeffect calculus cannot be defined purely syntactically, because a function $(\lambda x.e)$ may still have context demands – for example a function $(\lambda x.\mathtt{prev}\ x)$ has an immediate context demand 1 (requiring 1 past value of all variables in the context).

Definition 2. *An expression e is a* value, *written as $val(e)$ if it is a syntactic value, i. e. $e \in \mathit{SynVal}$ and it has no context-dependencies, i. e. $\Gamma @ \mathsf{use} \vdash e : \tau$.*

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

Lemma 6 (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 $\Gamma @ \mathsf{use} \vdash v : \sigma$ and an expression $\Gamma, x : \sigma @ r \vdash e : \tau$, then substituting v for x does not change the type and context demands, that is $\Gamma @ r \vdash e[x \leftarrow v] : \tau$.*

Proof. By induction over the type derivation, using the fact that x and v are annotated with use and that variables are never removed from the set Γ in the flat coeffect calculus. □

The substitution lemma 6 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:

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

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

The property holds for the three systems we consider – for implicit parameters, this is an equality; for liveness and dataflow (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 7 (Type preservation for call-by-value). *In a flat coeffect system satisfying the approximation property, that is* $r \wedge t \leqslant r \oplus t$*, if* $\Gamma @ r \vdash e : \tau$ *and* $e \longrightarrow_{\mathsf{cbv}} e'$ *then* $\Gamma @ r \vdash e' : \tau$.

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

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

The final step simplifies the coeffect annotation using the fact that use is a unit of $\circledast$. From Lemma 6, $e[x \leftarrow v]$ has the same coeffect annotation as e. As $r \wedge t \leqslant r \oplus t$, we can apply subcoeffecting:

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

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

1.4.3 *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 1.4.1, 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.

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 8 and Lemma 9) and finally, we state the conditions under which typing preservation holds for flat coeffect calculi (Theorem 10).

DATAFLOW. Reducing an expression $(\lambda x.e_1)\ e_2$ to $e_1[x \leftarrow e_2]$ does not always preserve the type of the expression in dataflow languages. 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 $\lambda x.y$:

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

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

Semantically, the coeffect over-approximates the actual demands – at runtime, the code does not actually access a previous value of the argument x. This fact cannot be captured by a flat coeffect type system, but it can be captured using the structural system discussed in Chapter **??**.

IMPLICIT PARAMETERS. In dataflow, substituting `prev` x for a variable y in an expression $\lambda z.y$ changes the context demands attached to the type of the function. This is the case not just for the preferred unique typeing derivation, but for all possible typings that can be obtained using the (*abs*) rule. However, this is not the case for all systems. Consider a substitution $\lambda x.y[y \leftarrow ?p]$ that substitutes an implicit parameter access $?p$ for a free variable y under a lambda:

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

The (*after*) judgement 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 a substitution $\lambda x.y[y \leftarrow e]$ that substitutes an arbitrary expression e of type τ_1 with coeffects r for a variable y:

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

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.

A GRAND CBN REDUCTION THEOREM. The above examples (implicit parameters and liveness) demonstrate two particular kinds of coeffect algebra for which call-by-name reduction preserves typing. 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 dataflow are examples of bottom-pointed coeffects. For top-pointed flat coeffects, the substitution lemma holds without additional demands:

Lemma 8 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with an algebra $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, when we substitute an expression e_s with arbitrary coeffects s for a variable x in e_r, the resulting expression is still typeable in a context with the original coeffect of e_r:*

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

Proof. Using subcoeffecting ($s \leqslant \mathsf{use}$) and a variation of Lemma 6. □

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

In a bottom pointed coeffect system, substituting e for x increases the context demands. However, if the system satisfies the strong condition that $\wedge = \circledast = \oplus$ then the context demands arising from the substitution can be associated with the context Γ, leaving the context demands 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 coeffect systems that track sets of context demands discussed in Section **??**. It allows the following substitution lemma:

Lemma 9 (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:*

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

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 demands (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 10 (Type preservation for call-by-name)**.** *In a coeffect system that satisfies the conditions for Lemma 8 or Lemma 9, if $\Gamma @ r \vdash e : \tau$ and $e \rightarrow_{\mathsf{cbn}} e'$ then it is also the case that $\Gamma @ r \vdash e' : \tau$.*

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

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

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 9 is:

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

From the assumptions of Lemma 9, 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 dataflow.

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 dataflow, the immediate coeffects are made larger by the lambda abstraction – the context demands 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 $\rightarrow_\eta$ performs an expansion that turns a function-typed term e to a syntactic function $\lambda x.e\ x$, the following theorem holds:

Theorem 11 (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* $\Gamma @ r \vdash e : \tau_1 \xrightarrow{s} \tau_2$ *and* $e \rightarrow_\eta e'$ *then* $\Gamma @ r \vdash e' : \tau_1 \xrightarrow{s} \tau_2$.

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

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

The derivation starts with the expression e and derives the type for $\lambda x.e\ x$. The application yields context demands $r \oplus s$. In order to recover the original typing, this must be equal to $r \wedge s$. Note that the derivation shows just one possible typing – the expression $\lambda x.e\ x$ has other types – but this is sufficient for 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 the η-expansion holds as well, giving us both local soundness and local completeness as coined by Pfenning and Davies [29].

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

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

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

Figure 3: Subtyping rules for flat coeffect calculus

1.5 SYNTACTIC PROPERTIES AND EXTENSIONS

The flat coeffect algebra introduced in Section 1.2 requires a number of axioms. The axioms are required for three reasons – to be able to define the categorical structure in Section 2.2, to prove equational properties in Section 1.4 and finally, to guarantee intuitive syntactic properties for constructs such as λ-abstraction and pairs in context-aware calculi.

In this section, we turn to the last point. We consider subcoeffecting and subtyping (Section 1.5.1), discuss what syntactic equivalences are permitted by the properties of $\wedge$ (Section 1.5.3) and we extend the calculus with pairs and units and discuss their syntactic properties (Section 1.5.4).

1.5.1 *Subcoeffecting and subtyping*

The *flat coeffect algebra* includes the $\leqslant$ relation which captures the ordering of coeffects and can be used to define subcoeffecting. Syntactically, an expression with context demands r' can be treated as an expression with a greater context. This is captured by the (*sub*) rule shown in Figure 1 (recall that for implicit parameters $\leqslant = \subseteq$):

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

Semantically, when read from the consequent to the assumption, 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 ignore (or even actively drop) the value for the unused parameter $?q$.

Subcoeffecting 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 demands as a function with greater context demands:

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

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 (*sub-typ*) rule allows using sub-typing on expressions in the coeffect calculus.

	Derived	Definition	Simplified
Implicit parameters	$s_1 \cup (s_2 \cup r)$	$s \cup (s \cup r)$	$s \cup r$
Liveness	$s_1 \sqcap (s_2 \sqcup r)$	$s \sqcap (s \sqcup r)$	s
Dataflow	$max(s_1, s_2 + r)$	$max(s, s + r)$	$s + r$

Table 1: Simplified coeffect annotation for let binding in three flat calculi instances

1.5.2 *Typing of let binding*

Recall the (*let*) rule in Figure 1. It annotates the expression `let` $x = e_1$ `in` e_2 with context demands $s \oplus (s \circledast r)$. This rule can be derived from the typing derivation for an expression $(\lambda x.e_2)\ e_1$ as a special case. We use the idempotence of $\wedge$ as follows:

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

This is one possible derivation, but other derivations may be valid for concrete coeffect system. The design decision of using this particular derivation for the typing of `let` is motivated by the fact that the typing obtained using the special rule is more precise for all the examples consider in this chapter. To see this, assume an arbitrary splitting $s = s_1 \wedge s_2$. Table 1 shows the coeffect annotation derived from $(\lambda x.e_2)\ e_1$, the coeffect annotation obtained by the (*let*) rule and the simplified coeffect annotation using the particular flat coeffect algebras.

It is perhaps somewhat unexpected that the annotation can be simplified in different ways for different examples. However, for all our systems, the simplfied annotation (right column in Table 1) is more precise than the original (left column). Recall that $s = s_1 \wedge s_2$. The following inequalities hold:

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

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

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

This property does not follow from the axioms 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 a reasonable basis for using the specialized (*let*) rule in the flat coeffect system.

1.5.3 *Properties of lambda abstraction*

In Section 1.1.1, we discussed how to reconcile two typings for lambda abstraction – for implicit parameters, the lambda function needs to split context demands using $r \cup s$, but for dataflow and liveness it suffices to duplicate the demand r of the body. We consider coeffect calculi for which the simpler duplication of coeffects is sufficient.

SIMPLIFIED ABSTRACTION. Recall that $(\mathcal{C}, \wedge)$ is a band, that is, $\wedge$ is idempotent and associative. The idempotence means that the context demands of the body can be required from both the declaration site and the call site. In Section 1.3.2, we introduced the (*idabs*) rule (repeated below for reference), which uses the idempotence and duplicates coeffect annotations:

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

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 12. *For a flat coeffect calculus with a strictly oriented algebra, equipped with subcoeffecting and subtyping, the standard (abs) rule can be derived from the (idabs) rule.*

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

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

□

The practical consequence of the Remark 12 is that, for strictly oriented coeffect calculi (such as our liveness and dataflow computations), the (*idabs*) rule not only determines a unique typing derivation (as discussed in Section 1.3.2), but it gives (together with subtyping and subcoeffecting) an equivalent type system.

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

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

Intuitively, this means that the context demands of a function with multiple arguments can be split arbitrarily between the declaration site and (multiple) call sites.

1.5.4 *Language with pairs and unit*

To focus on the key aspects of flat coeffect systems, the calculus introduced in Section 1.2 consists only of variables, abstraction, application and let bind-

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

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

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

Figure 4: Typing rules for pairs and units

ing. 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 further motivate the axioms for $\oplus$. The syntax of the language is extended as follows:

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

The typing rules for pairs and the unit value are shown in Figure 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 unary 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 axioms then correspond to the requirement that $(e_1, (e_2, e_3)) \simeq ((e_1, e_2), e_3)$ (associativity) and the demand that $((), e) \simeq e \simeq (e, ())$ (unit).

Thanks to the properties of $\oplus$, the flat coeffect calculus obeys the monoid axioms 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 14. *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 14 motivates the demand of the monoid structure $(\mathcal{C}, \oplus, \mathsf{ign})$ of the flat coeffect algebra. We require only unit and associativity axioms. In our three examples, the $\oplus$ operator is also symmetric, which additionally guarantees that $(e_1, e_2) \simeq (e_2, e_1)$, which is a property that is expected to hold for λ-calculus.

1.6 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, dataflow and implicit parameters. However, the system is more general and an capture various 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 1.3 and noted that the result for dataflow and liveness generalizes for any coeffect calculus with strictly oriented coeffect algebra (Remark 12).

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 *type preservation* theorems prove that the operational semantics for two classes of flat coeffect calculi (including liveness and implicit parameters) can be based on 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 a soundness result that well-typed programs in flat coeffect calcului for implicit parameters and dataflow do not get stuck in the translated version.

2 SEMANTICS OF FLAT COEFFECTS

The *flat coeffect calculus* introduced in the previous chapter uniformly captures a number of context-aware systems outlined 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 language 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 Wadler and Thiemann [42] and 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 languages.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We introduce *indexed comonads*, a generalization of comonads, a category-theoretical dual of monads (Section 2.2) and we discuss how they provide semantics for coeffect calculus. This provides an insight into how (and why) the coeffect calculus works and shows an intriguing link with effects and monads.

- We use indexed comonads to guide our *translational semantics* of coeffect calculus (Section 2.3). We define a simple sound functional programming language (with type system and operational semantics). 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 sample 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 2.4)

- 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 2.6).

2.1 INTRODUCTION AND SAFETY

This chapter links together a number of different technical developments presented in this thesis. We take the flat coeffect calculus introduced in Chapter 1, 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 3, 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 less error-prone. 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 (cons "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 int $\xrightarrow{\{?one,?two\}}$ 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 the 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 int → 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 num list → num. The function fails for input lists containing only zero or one elements and this is not reflected in the type and is not enforced by the type checker.

Using coeffects (on the left), the function has a type num $\xrightarrow{1}$ 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, a coeffect type system, captures certain runtime demands of context-aware programs and (as we show in this chapter), eliminates common errors related to working with context.

2.2 CATEGORICAL MOTIVATION

The type system of the 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 2.3.

2.2.1 *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 [42] and further explored by Atkey [2] and Vazou and Leijen [22]). This line of work relates effectful functions $\tau_1 \xrightarrow{\sigma} \tau_2$ to monadic computations $\tau_1 \rightarrow M^{\sigma}\tau_2$. 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 [39] who present the semantics of contextual computations (mainly for dataflow) in terms of comonadic functions $C\tau_1 \rightarrow \tau_2$. We introduce *indexed comonads* that annotate the structure with information about the required context, i.e. $C^r\tau_1 \rightarrow \tau_2$. This is similar to the recent development on monads and effects by Katsumata [15] who parameterizes monads in a similar way to our indexed comonads.

2.2.2 *Categorical semantics*

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

For a well-defined semantics, we need to ensure that a well-typed term is assigned exactly one meaning. This can be achieved in a number of ways. First, we can prove the *coherence* [10] and show the morhphisms assigned to multiple typing derivations are equivalent. Second, the typing judgement can have a unique typing derivation. We follow the latter approach, using the uniqye typing derivation specified in Section 1.3.

As a best known example, Moggi [21] showed that the semantics of various effectful computations can be captured uniformly using (*strong*) *monads*. In that approach, computations are interpreted as $\tau_1 \times \ldots \times \tau_n \rightarrow M\tau$, for some monad M. For example, $M\alpha = \alpha \cup \{\bot\}$ models failures (the Maybe monad), $M\alpha = \mathcal{P}(\alpha)$ models non-determinism (list monad) and side-effects can be modelled using $M\alpha = S \rightarrow (\alpha \times S)$ (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 a similar approach to Moggi, Uustalu and Vene [39] showed that (*monoidal*) *comonads* uniformly capture the semantics of various kinds of context-dependent computations [39]. 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 aboe represents at most a binary analysis of effects or context-dependence. A function $\tau_1 \rightarrow \tau_2$ performs *no* effects (requires no context) whereas $\tau_1 \rightarrow M\tau_2$ performs *some* effects and $C\tau_1 \rightarrow \tau_2$ 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 demands r as functions $C^r\tau_1 \rightarrow \tau_2$ using an *indexed comonad* C^r.

2.2.3 *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* $(\mathsf{C}, \mathsf{counit}, \mathsf{cobind})$ *where:*

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

such that, for all $f : \mathsf{C}\,\alpha \rightarrow \beta$ *and* $g : \mathsf{C}\,\beta \rightarrow \gamma$*:*

$$\begin{aligned} \mathsf{cobind}\ \mathsf{counit} &= \mathsf{id} && (\textit{left identity}) \\ \mathsf{counit} \circ \mathsf{cobind}\ f &= f && (\textit{right identity}) \\ \mathsf{cobind}\ (g \circ \mathsf{cobind}\ f) &= (\mathsf{cobind}\ g) \circ (\mathsf{cobind}\ f) && (\textit{associativity}) \end{aligned}$$

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 $\mathsf{C}\,\alpha$. The cobind operation turns a context-dependent function $\mathsf{C}\,\alpha \rightarrow \beta$ 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 [39] 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*

1 This is an over-simplification as we can use e. g. stacks of monad transformers and model functions with two different effects using $\tau_1 \rightarrow M_1(M_2\ \tau_2)$. However, monad transformers require the user to define complex systems 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).

left and right cases, respectively. The type NEList *forms a comonad together with the following* counit *and* cobind *mappings:*

$$\begin{array}{lll} \mathsf{counit}\ l &= h & \text{when } l = \mathtt{inl}\ h \\ \mathsf{counit}\ l &= h & \text{when } l = \mathtt{inr}\ (h, t) \\[1ex] \mathsf{cobind}\, f\ l &= \mathtt{inl}\ (f\, l) & \text{when } l = \mathtt{inl}\ h \\ \mathsf{cobind}\, f\ l &= \mathtt{inr}\ (f\, l,\ \mathsf{cobind}\ f\ t) & \text{when } l = \mathtt{inr}\ (h, t) \end{array}$$

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 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 $\mathtt{inl}\ ()$. 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 generalize comonads to *indexed comonads*.

2.2.4 *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 element use represents a variable access. An indexed comonad is formed by a data type (object mapping) $C^r\alpha$ where the r (also called *annotation*) is a member of the set $\mathcal{C}$ and 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*

$$\begin{array}{rlr} \mathsf{cobind}_{\mathsf{use},s}\ \mathsf{counit}_{\mathsf{use}} &= \mathsf{id} & (\textit{left identity}) \\ \mathsf{counit}_{\mathsf{use}} \circ \mathsf{cobind}_{r,\mathsf{use}}\ f &= f & (\textit{right identity}) \\ \mathsf{cobind}_{r \circledast s,t}\ (g \circ \mathsf{cobind}_{r,s}\ f) &= (\mathsf{cobind}_{s,t}\ g) \circ (\mathsf{cobind}_{r,s \circledast t}\ f) & (\textit{associativity}) \end{array}$$

Rather than defining a single mapping C, we are now defining a family of mappings C^r indexed by elements of the monoid structure $\mathcal{C}$. 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 objects α, β (modeling types) 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. Nevertheless we continue to write $\mathsf{counit}_{\mathsf{use}}$, 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 comonad axioms, 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 (associative) composition operation that composes computations with contextual demands as follows:

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

The composition $\hat{\circ}$ best expresses the intention of indexed comonads. Given two functions with contextual demands r and s, their composition is a function that requires $r \circledast s$. The contextual demands 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 Maybe)**.** *The* indexed Maybe 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:*

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

The *indexed Maybe 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 $\mathsf{counit}\ ()$ 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. We assume that, all implicit parameters have the type* num*. The data type* $C^r\alpha = \alpha \times (r \to \mathsf{num})$ *represents a value* α *together with a function that associates a parameter value* num *with every implicit parameter name in* $r \subseteq \mathsf{Id}$*. The cobind and counit operations are defined as:*

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

In the definition, we use the notation (a, g) for a pair containing a value of type α together with g, which is a function of type $r \to \mathsf{num}$. 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 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.

2.2.5 *Flat indexed comonads*

Because of the asymmetry of λ-calculus (discussed in Section **??**), the duality between monads and comonads does not lead us towards 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 [21] 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 is implicit.

To model λ-calculus with contextual properties, Uustalu and Vene [39] require *lax semi-monoidal* comonad. This structure requires an additional monoidal operation:

$$m : C\alpha \times C\beta \to C(\alpha \times \beta)$$

The m operation is needed in the semantics of lambda abstraction. Semantically, it represents merging of contextual capabilities attached to the variable contexts 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 not only 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*. We look at examples of this operation in the next section.

The $\mathsf{split}_{r,s}$ operation splits a single comonadic value (containing a tuple) into two separate values. Note that this does not simply duplicate the value, because the additional context is also split. To obtain coeffects r and s, the input needs to provide *at least* r and s, so the tags are combined using the $\oplus$, which is often the *least upper-bound*[2].

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 $r \leqslant r'$

The axioms of flat coeffect algebra do not, in general, require that $r \leqslant r \oplus s$ and $s \leqslant r \oplus s$, but the property holds for the three sample coeffect systems we consier. For systems with the above property, the split operation can be expressed 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 definition 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). Note therefore that our use of 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 2.2.4 can be extended into *flat indexed comonads*. Note however that this cannot be done mechanically, because each example requires us to define additional operations, specific for the example.

Example 9 (Monoidal comonads)**.** *Just like indexed comonads generalize comonads, the additional structure of flat indexed comonads generalizes the symmetric semimonoidal comonads of Uustalu and Vene [39]. 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.*

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.

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

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

Without the indexing, the merge operations implements *zip* on Maybe values, returning a value only when both values are present. The behaviour of the split operation is partly determined by the indices. When the input is *dead*, both values have to be dead (this is also the only solution of $\mathsf{D} = r \sqcap s$), 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 comonadic data type $\mathsf{C}^r\alpha$ is $\alpha \times (r \to \mathsf{num})$ where num is the type of implicit parameter values. The additional operations are defined as:*

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

The split operation splits the tuple and restricts the function (representing available implicit parameters) to the required subsets. 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:*

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

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

$$\begin{aligned}
&\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) =\\
&\quad \langle (a_0, b_0), \ldots, (a_{\mathit{min}(m,n)}, b_{\mathit{min}(m,n)})\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{split}_{m,n}\langle (a_0, b_0), \ldots, (a_{\mathit{max}(m,n)}, b_{\mathit{max}(m,n)})\rangle =\\
&\quad (\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle)
\end{aligned}$$

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 semantics is defined over a typing derivation:

$$\frac{}{[\![\Gamma @ \mathsf{use} \vdash x_i : \tau_i]\!]} \quad = \quad \frac{}{\pi_i \circ \mathsf{counit}_{\mathsf{use}}} \qquad (var)$$

$$\frac{}{[\![\Gamma @ \mathsf{ign} \vdash n : \mathsf{num}]\!]} \quad = \quad \frac{}{\mathsf{const}\ n} \qquad (num)$$

$$\frac{[\![\Gamma, x : \tau_1 @ r \wedge s \vdash e : \tau_2]\!]}{[\![\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!]} \quad \begin{array}{c} = \\ = \end{array} \quad \frac{f}{f \circ \mathsf{curry}\ \mathsf{merge}_{r,s}} \qquad (abs)$$

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

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

the length of the shorter argument. When splitting a list, $\mathsf{split}_{m,n}$ needs the maximum of the required lengths.

2.2.6 *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 semantics that is capable of capturing all our examples, as well as various other computations.

As discussed in Section 1.3, 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 need to choose a particular typing derivation. The algorithm for choosing a unique typing derivation for our three systems has been defined in Section 1.3.

CONTEXTS AND TYPES. The modelling of contexts and functions generalizes the concrete examples discussed in Chapter **??**. 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 [39, 23], 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 2.3).

The semantics of variable access and abstraction are the same as in the semantics of Uustalu and Vene [39], modulo the indexing. The semantics of variable access (*var*) uses $\mathsf{counit}_{\mathsf{use}}$ to extract a product of free variables, followed by 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. The categorical semantics in Section 2.3 defines a translation that embeds context-dependent computations in a functional programming language, 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 1.2.2 – 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 22 once we define the target language and its typing.

Lemma 15 (Correspondence)**.** *In the semantics defined in Figure 5, the context annotations r of typing judgements $\Gamma @ r \vdash e : \tau$ 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. We need to check that the domains and codomains of the morphisms in the semantics (right-hand side) match. □

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

LANGUAGE SYNTAX

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

REDUCTION RULES

$$\begin{array}{lll} (\mathit{fn}) & (\lambda x.e)\ v \rightsquigarrow e[x \leftarrow v] & \\ (\mathit{prj}) & \pi_i(v_1, \ldots, v_n) \rightsquigarrow v_i & \\ (\mathit{ctx}) & K[e] \rightsquigarrow K[e'] & (\text{when } e \rightsquigarrow e') \end{array}$$

TYPING RULES

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

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

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

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

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

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

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

to provide. In Section 2.5, 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 16 (Generalization). *Consider a typing derivation obtained according to the rules for finding unique typing derivations as specified in Section 1.3 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. □

2.3 TRANSLATIONAL SEMANTICS

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* language 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 $= \ldots$ notation).

The syntax, typing and reduction rules of the first part (simple functional language) are common to all concrete coeffect domain-specific languages. 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 notions of context. Finally, the third part (domain-specific primitives) will differ for each coeffect domain-specific language.

2.3.1 *Functional target language*

The target language 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 language 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, $K[e]$ defines the syntactic evaluation 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 language are also shown in Figure 6.

2.3.2 *Safety of functional target language*

The functional subset of the language described so far models a simple ML-like language. 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 2.4.

We use the standard syntactic approach to type safety introduced by Milner [20]. Following Wright, Felleisen and Pierce [30, 43], 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 17 (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* $\tau = \tau_1 \rightarrow \tau_2$ *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 18 (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 19 (Type preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \rightarrow e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn*): $e = (\lambda x.e_0)\ v$, from Lemma 18 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 20 (Progress)**.** *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$

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 17 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 17, 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 21 (Safety of functional target language)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow^{*} e'$ *then either* e' *is a value of type* τ *or there exists* e'' *such that* $e' \rightsquigarrow e''$ *and* $\Gamma \vdash e'' : \tau$.

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

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

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

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

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

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

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

Figure 7: Comonadically-inspired extensions for the target language

Proof. Rule induction over $\rightsquigarrow^*$ using Theorem 19 and Theorem 20. □

2.3.3 *Comonadically-inspired translation*

In Section 2.2, 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 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 common additional syntax, evaluation context and typing rules once in Figure 7. We consider examples later in Section 2.4.

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 translating a term with a chosen typing derivation from a source language and they will

The translation is defined over a typing derivation:

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

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

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

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

Assuming the following auxiliary operations:

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

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

be useful when proving that sufficient context (as specified by the coeffect annotations) is available.

Figure 7 defines the syntax and the typing rules, but it does not define the reduction rules. These – 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:

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

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.

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 consequent is now written as source code in the target programming language rather than as composition of morphisms in a category. However, thanks to the relationship between λ-calculus and cartesian closed categories, 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. For programming language translation and the implementation (discussed in Chapter 3), 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 15), but now it has more obvious practical consequences.

In Section 2.4, 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 means that we are also proving safety of well-typed programs in the source context-aware languages.

Theorem 22 (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 language as (we write* $\ldots$ *for the omitted part of the translation tree):*

$$\frac{[\![\ (\ldots)\ ]\!]}{[\![@\, r \vdash e : \tau]\!]} \quad \begin{matrix} = \\ = \end{matrix} \quad \frac{(\ldots)}{f}$$

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

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 $\mathsf{C}^c([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \rightarrow [\![\tau]\!]$.

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

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

2.4 SAFETY OF CONTEXT-AWARE LANGUAGES

The language defined in Figure 6 and Figure 7 provide a general structure that we now use to prove the safety of various context-aware programming languages based on the coeffect language framework. As examples, we consider a language for dataflow computations (Section 2.4.1) and for implicit parameters (Section 2.4.2). In both cases, we extend the progress and preservation theorems of the functional subset of the target language, but the approach can be generalized as discussed in Section 2.5.

As outlined in the table at the beginning of Part ii, we now covered the parts of the semantics that are shared by all context-aware languages. This includes the functional target language with comonadically-inspired uninterpreted type $\mathsf{C}^r\tau$ and the syntax for comonadically-inspired uninterpreted primitives such as $\mathsf{cobind}_{s,r}$ and $\mathsf{counit}_{\mathsf{use}}$, together with their typing.

Using dataflow and implicit parameters as two examples, we now add the domain-specific extensions needed for a concrete context-aware programming language. This includes syntax for values and expressions of the comonad-inspired type $C^{r}\tau$ and reduction rules for the comonadically-inspired operations ($\mathsf{cobind}_{s,r}$, $\mathsf{counit}_{\mathsf{use}}$, etc.).

2.4.1 *Coeffect language 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, together with an additional *domain-specific* operation that models the `prev` construct.

The Figure 9 extensionds the target language with syntax, typing rules, additional translation rule and reductions 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 specify how the `prev` keyword is translated into a prev_r operation of the target language nd 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 in Example 12 when discussing the indexed list comonad.

PROPERTIES. Now consider a target language 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 17) and the *preservation under substitution lemma* (Lemma 18). 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. We show that the translation rule for `prev` produces well-typed expressions. Finally, we extend the type preservation (Theorem 19) and progress (Theorem 20) theorems.

Theorem 23 (Well-typedness of the `prev` translation)**.** *Given a typing derivation for a well-typed closed expression* $@\,r \vdash e : \tau$*, the translated program* f *obtained using the rules in Figure 8 and Figure 9 is well-typed, i. e. in the target language:* $\vdash f : [\![\Gamma\,@\,r]\!] \rightarrow [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation.

Case (*var, num, abs, app*): As before.

Case (*prev*): Type of *ctx* is $C^{n+1}\tau$ and so we can apply the (*prev*) rule. □

Lemma 24 (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* $\tau = \tau_1 \rightarrow \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'

LANGUAGE SYNTAX

$$\begin{array}{rcl} v & = & \ldots \mid \mathsf{Df}\langle v_0, \ldots, v_n \rangle \\ e & = & \ldots \mid \mathsf{Df}\langle e_0, \ldots, e_n \rangle \mid \mathsf{prev}_n\ e \\ K & = & \ldots \mid \mathsf{prev}_n\ \_ \mid \mathsf{Df}\langle v_0, \ldots, v_{i-1}, \_, e_{i+1} \ldots, e_n \rangle \end{array}$$

TYPING RULES

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

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

TRANSLATION

$$\frac{[\![\Gamma @ n \vdash e : \tau]\!]}{[\![\Gamma @ n+1 \vdash \mathsf{prev}\ e : \tau]\!]} \begin{array}{c} = \\ = \end{array} \frac{f}{\lambda ctx.\mathsf{prev}_n\ \mathit{ctx}}$$

REDUCTION RULES

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

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

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 25 (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 26 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn*, *prj*, *ctx*): As before, using Lemma 25 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 $\Gamma \vdash f : C^{m}\tau_1 \rightarrow \tau_2$ and $\Gamma \vdash \mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle : C^{m+n}\tau$. The reduced expression has a type $C^{n}\tau_2$:

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

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 27 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$

Proof. By rule induction over $\vdash$.

Case (*num,abs,var,app,proj,tup*): As before, using the adapted canonical forms lemma (Lemma 24) 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 24, $e_1 = \mathsf{Df}\langle v\rangle$ 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 24, 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 24 and it can reduce using (*merge*) reduction.

Case (*df*): $e = \mathsf{Df}\langle e_0, \ldots, e_n\rangle$. If e_i is not a value then reduce using (*ctx*) with context $\mathsf{Df}\langle v_0, \ldots, v_{i-1}, \_, e_{i+1} \ldots, e_n\rangle$. Otherwise, $e_0, \ldots, e_n$ are values and so $\mathsf{Df}\langle e_0, \ldots, e_n\rangle$ is also a value.

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 28 (Safety of context-aware dataflow language). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow^{*} e'$ *then either* e' *is a value of type* τ *or there exists* e'' *such that* $e' \rightsquigarrow e''$ *and* $\Gamma \vdash e'' : \tau$.

Proof. Rule induction over $\rightsquigarrow^{*}$ using Theorem 26 and Theorem 27. □

2.4.2 *Coeffect language for implicit parameters*

We now turn to our second example. As discussed earlier (Example 11), implicit parameters can be modelled by an indexed product comonad, which annotates a value with additional context – in our case, a mapping from implicit parameter names to their values. In this section, we embed this model into the target language.

As with dataflow computations, we take the core functional subset (Figure 6) with comonadically-inspired extensions (Figure 7) and we specify a new kind of values of type $C^{r}\tau$ and domain-specific reduction rules that specify how the operations propagate and access the context containing implicit parameter bindings. Again, the $C^{r}\tau$ values can be seen as an *abstract data type*, which are never manipulated directly, except by the comonadically-inspired operations ($\mathsf{cobind}_{s,r}$, $\mathsf{counit}_{\mathsf{use}}$, etc.).

DOMAIN-SPECIFIC EXTENSIONS. The Figure 10 shows extensions to the target language for modelling implicit parameters. A comonadic value with a coeffect $\{?p_1, \ldots, ?p_n\}$ is modelled by a new kind of value written as $\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots ?p_n \mapsto v_n\})$ which contains a value v together with implicit parameter assignments for all the parameters specified in the coeffect. We add a corresponding kind of expression with its typing rule (*impl*).

There are also two domain-specific operations for working with implicit parameters. The $\mathsf{lookup}_{?p}$ operation reads a value of an implicit parameter and the $\mathsf{letimpl}_{?p,r}$ operation adds a mapping assigning a value to an implicit parameter $?p$. The typing rule (*lookup*) specifies that the accessed parameter need to be a part of the context and the rule (*letimpl*) specifies that the $\mathsf{letimpl}$ operation extends the context with a new implicit parameter binding.

The new translation rules specify how implicit parameter access, written as $?p$, and implicit parameter binding, written as $\mathtt{let}\ ?p\ = e_1\ \mathtt{in}\ e_2$ are translated to the target language. The first one is straightforward. The binding is similar to the translation for function application – we split the context, evaluate e_1 using the first part of the context ctx_1 and then add the new binding to the remaining context ctx_2.

Finally, Figure 10 also defines the reduction rules. The (*lookup*) rule accesses an implicit parameter and (*letimpl*) adds a new binding. The reduction rules closely model the product comonad discussed in Example 11. Reductions for (*cobind*) and (*split*) restrict the set of available implicit parameters according to the annotations and (*merge*) combines them, preferring the values from the call-site.

For the semantics of implicit parameter programs that we consider, the preference of call-site bindings over declaration-site bindings in (*merge*) does not matter. The unique typing derivations for implicit parameter coeffects obtained in Section 1.3 always split implicit parameters into *disjoint sets*, so preferences do not come into play.

LANGUAGE SYNTAX

$$\begin{array}{lcl} v &=& \ldots \mid \mathsf{Impl}(v,\{?p_1 \mapsto v_1,\ldots,?p_n \mapsto v_n\}) \\ e &=& \ldots \mid \mathsf{Impl}(e,\{?p_1 \mapsto e_1,\ldots,?p_n \mapsto e_n\}) \\ && \mid \mathsf{lookup}_{?p}\ e \mid \mathsf{letimpl}_{?p,r}\ e_1\ e_2 \\ K &=& \ldots \mid \mathsf{lookup}_{?p}\ \_ \mid \mathsf{letimpl}_{?p,r}\ \_\ e \mid \mathsf{letimpl}_{?p,r}\ v\ \_ \\ && \mid \mathsf{Impl}(\_,\{?p_1 \mapsto e_1,..,?p_n \mapsto e_n\}) \\ && \mid \mathsf{Impl}(v,\{?p_1 \mapsto v_1,..,?p_{i-1} \mapsto v_{i-1},?p_i \mapsto \_,?p_{i+1} \mapsto v_{i+1},..?p_n \mapsto e_n\}) \end{array}$$

TYPING RULES

$$(\mathit{impl})\quad \frac{\Gamma \vdash e : \tau \qquad \forall i \in \{1\ldots n\}.\ \Gamma \vdash e_i : \mathsf{num}}{\Gamma \vdash \mathsf{Impl}(e,\{?p_1 \mapsto e_1,\ldots,?p_n \mapsto e_n\}) : C^{\{?p_1,\ldots,?p_n\}}\tau}$$

$$(\mathit{lookup})\quad \frac{\Gamma \vdash e : C^{\{?p\}}\tau}{\Gamma \vdash \mathsf{lookup}_{?p}\ e : \mathsf{num}}$$

$$(\mathit{letimpl})\quad \frac{\Gamma \vdash e_1 : \mathsf{num} \qquad \Gamma \vdash e_2 : C^{\{?p_1,\ldots,?p_n\}}\tau}{\Gamma \vdash \mathsf{letimpl}_{?p,\{?p_1,\ldots,?p_n\}}\ e_1\ e_2 : C^{\{?p_1,\ldots,?p_n,?p\}}\tau}$$

TRANSLATION

$$(\mathit{lookup})\quad \frac{}{[\![\Gamma @ \{?p\} \vdash ?p : \mathsf{num}]\!]} \;=\; \frac{}{\lambda ctx.\mathsf{lookup}_{?p}\ \mathit{ctx}}$$

$$(\mathit{letimpl})\quad \frac{\begin{array}{c}[\![\Gamma @ r \vdash e_1 : \tau_1]\!] \\ [\![\Gamma @ s \vdash e_2 : \tau_2]\!]\end{array}}{\left[\!\!\left[\begin{array}{l}\Gamma @ r \cup (s \setminus \{?p\}) \\ \vdash \mathtt{let}\ ?p = e_1 \\ \quad\ \mathtt{in}\ e_2 : \tau_2\end{array}\right]\!\!\right]} \;\begin{array}{c} = \\ = \\ \\ = \end{array}\; \frac{\begin{array}{l} f \\ g\end{array}}{\begin{array}{l}\lambda \mathit{ctx}. \\ \quad \mathtt{let}\ \mathit{ctx}_0 = \mathsf{map}_{r \cup (s\setminus\{?p\})}\ \mathsf{dup}\ \mathit{ctx} \\ \quad \mathtt{let}\ (\mathit{ctx}_1,\mathit{ctx}_2) = \mathsf{split}_{r,(s\setminus\{?p\})}\ \mathit{ctx}_0 \\ \quad g\ (\mathsf{letimpl}_{?p,(s\setminus\{?p\})}\ (f\ \mathit{ctx}_1)\ \mathit{ctx}_2)\end{array}}$$

REDUCTION RULES

$$\begin{array}{rl}
(\mathit{counit}) & \mathsf{counit}_{\emptyset}\ (\mathsf{Impl}(v,\ldots)) \rightsquigarrow v \\[1em]
(\mathit{cobind}) & \begin{array}{l}\mathsf{cobind}_{r,s}\ f\ (\mathsf{Impl}(v,\{?p_1 \mapsto v_1,\ldots,?p_n \mapsto v_n\})) \rightsquigarrow \\ \quad \mathsf{Impl}(f\ (\mathsf{Impl}(v,\{?p_i \mapsto v_i \mid p_i \in r\})),\{?p_i \mapsto v_i \mid p_i \in s\})\end{array} \\[1.5em]
(\mathit{merge}) & \begin{array}{l}\mathsf{merge}_{r,s}(\ \mathsf{Impl}(v,\{?p_1 \mapsto v_1,\ldots,?p_n \mapsto v_n\}) \\ \qquad\qquad \mathsf{Impl}(v',\{?p'_1 \mapsto v'_1,\ldots,?p'_n \mapsto v'_n\})\ ) \rightsquigarrow \\ \quad \mathsf{Impl}((v,v'),\{?p_i \mapsto v_i \mid ?p \in r \setminus s\} \cup \{?p'_i \mapsto v'_i \mid ?p' \in s\})\end{array} \\[1.5em]
(\mathit{split}) & \begin{array}{l}\mathsf{split}_{r,s}(\mathsf{Impl}((v,v'),\{?p_1 \mapsto v_1,\ldots,?p_n \mapsto v_n\})) \rightsquigarrow \\ \quad \mathsf{Impl}(v,\{?p_i \mapsto v_i \mid p_i \in r\}),\mathsf{Impl}(v',\{?p_i \mapsto v_i \mid p_i \in s\})\end{array} \\[1.5em]
(\mathit{letimpl}) & \begin{array}{l}\mathsf{letimpl}_{?p,r}\ v'\ (\mathsf{Impl}(v,\{?p_1 \mapsto v_1,\ldots,?p_n \mapsto v_n\})) \rightsquigarrow \\ \quad \mathsf{Impl}(v,\{?p_i \mapsto v_i \mid ?p_i \in r, ?p_i \neq ?p\} \cup \{?p \mapsto v'\})\end{array} \\[1.5em]
(\mathit{lookup}) & \mathsf{lookup}_{?p_i}(\mathsf{Impl}(v,\{?p_i \mapsto v_i\}) \rightsquigarrow v_i
\end{array}$$

Figure 10: Additional constructs embedding implicit parameters into the language

PROPERTIES. We now prove the type safety of of a context-aware programming language with implicit parameters. To do this, we prove safety of the target functional language with specific extensions for implicit parameters and we show that the translation from context-aware programming language with implicit parameters produces well-typed programs in the target language.

The target language consists of the core functional language subset (Figure 6) with the comonadically-inspired extensions (Figure 7) and the domain-specific extensions for implicit parameters defined in Figure 10. The well-typedness of the translation has been discussed earlier (Theorem 22) and we extend it to cover operations specific for implicit parameters below (Theorem 29).

As for dataflow computations, we prove the type safety by extending the preservation (Theorem 19) and progress (Theorem 20) for the core functional subset of the language, but it is worth noting that the key parts of the proofs are centered around the new reduction rules for comonadically-inspired primitives and newly defined Impl values. These do not interact with the rest of the language in any unexpected ways.

Theorem 29 (Well-typedness of the implicit parameters translation). *Given a typing derivation for a well-typed closed expression* $@\, r \vdash e : \tau$*, the translated program* f *obtained using the rules in Figure 8 and Figure 10 is well-typed, i. e. in the target language:* $\vdash f : [\![\Gamma \,@\, r]\!] \to [\![\tau]\!]$.

Proof. By rule induction over the derivation of the translation.

Case (*var, num, abs, app*): As before.

Case (*lookup*): The type of *ctx* has a coeffect $\{?p\}$ which includes the parameter $?p$ as required in order to use the (*lookup*) typing rule.

Case (*letimpl*): The type of *ctx* matches with the input type of $\mathsf{map}_{r \cup (s \setminus \{?p\})}$. After duplication and splitting the context, ctx_1 and ctx_2 have types $C^{r}(\ldots)$ and $C^{s \setminus \{?p\}}(\ldots)$, respectively. This matches with the expected types of f and $\mathsf{letimpl}$. The context returned by $\mathsf{letimpl}$ then matches the one required by g. □

Lemma 30 (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* $\tau = C^{\{?p_1,\ldots,?p_n\}}\tau_1$ *then* $e = \mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_n \mapsto v_n\})$

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

Lemma 31 (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 $\mathsf{Impl}(e, \{\ldots\})$, $\mathsf{lookup}_{?p}$ and $\mathsf{letimpl}_{?p,r}$. □

Theorem 32 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\rightsquigarrow$.

Case (*fn, prj, ctx*): As before, using Lemma 31 for (*fn*).

Case (*counit*): $e = \mathsf{counit}_{0}(\mathsf{Impl}(v, \{\ \}))$. The last rule in the type derivation of e must have been (*counit*) with $\Gamma \vdash \mathsf{Impl}(v, \{\ \}) : C^{\emptyset}\tau$ which, in turn, required that $\Gamma \vdash v : \tau$.

Case (*cobind*): $e = \mathsf{cobind}_{r,s}\ f\ (\mathsf{Impl}(\nu, \{?p_1 \mapsto \nu_1, \ldots, ?p_n \mapsto \nu_n\}))$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^s\tau_2$ and assumptions $\Gamma \vdash \mathsf{Impl}(\nu, \{?p_1 \mapsto \nu_1, \ldots, ?p_n \mapsto \nu_n\}) : C^r\tau$ and $\Gamma \vdash f : C^r\tau_1 \to \tau_2$. The reduced expression has a type $C^s\tau_2$:

$$\dfrac{\dfrac{\Gamma \vdash f : C^r\tau_1 \to \tau_2 \qquad \Gamma \vdash \mathsf{Impl}(\nu, \{?p_i \mapsto \nu_i \mid p_i \in r\}) : C^r\tau_1}{\Gamma \vdash f\ (\mathsf{Impl}(\nu, \{?p_i \mapsto \nu_i \mid p_i \in r\})) : \tau_2}}{\Gamma \vdash \mathsf{Impl}(f\ (\mathsf{Impl}(\nu, \{?p_i \mapsto \nu_i \mid p_i \in r\})), \{?p_i \mapsto \nu_i \mid p_i \in s\}) : C^s\tau_2}$$

Case (*lookup*): $e = \mathsf{lookup}_{?p_i}(\mathsf{Impl}(\nu, \{\ \ldots, ?p_i \mapsto \nu_i \ldots\}))$. The last rule in the type derivation must have been (*lookup*) with $\tau = \mathsf{num}$ and an assumption $\Gamma \vdash \mathsf{Impl}(\nu, \{\ \ldots, ?p_i \mapsto \nu_i \ldots\}) : C^{\{\ldots, ?p_i, \ldots\}}\tau$, which requires $\Gamma \vdash \nu_i : \mathsf{num}$.

Case (*merge*, *split*, *letimpl*): Similar. In all three cases, the last typing rule in the derivation of e guarantees that all values of all implicit parameters that are required for the reduction are available. □

Theorem 33 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$

Proof. By rule induction over $\vdash$.

Case (*num,abs,var,app,proj,tup*): As before, using the adapted canonical forms lemma (Lemma 30) 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 24, $e_1 = \mathsf{Impl}(\nu, \{\ \})$ and so we can apply (*counit*) reduction rule.

Case (*cobind*): $e = \mathsf{cobind}_{r,s}\ e_1\ e_2$. If e_1 is not a value, reduce using (*ctx*) with context $\mathsf{cobind}_{r,s}\ \_\ e$. If e_2 is not a value reduce using (*ctx*) with context $\mathsf{cobind}_{r,s}\ \nu\ \_$. If both are values, then from Lemma 30, we have $e_2 = \mathsf{Impl}(\nu, \{?p_i \mapsto \nu_i \mid ?p_i \in r \cup s\})$ and we apply the (*cobind*) reduction.

Case (*merge*): $e = \mathsf{merge}_{r,s} e_1$. If e_1 is not a value, reduce using (*ctx*) with context $e = \mathsf{merge}_{r,s}\ \_$. If e_1 is a value, it must be a pair of values $(\mathsf{Impl}(\nu, \{?p_i \mapsto \nu_i \mid ?p_i \in r\}), \mathsf{Impl}(\nu', \{?p_i \mapsto \nu_i \mid ?p_i \in s\}))$ using Lemma 24 and it can reduce using (*merge*) reduction.

Case (*impl*): $e = \mathsf{Impl}(e', \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\})$. If e is not a value, reduce using (*ctx*) with context $\mathsf{Impl}(\_, \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\})$. If e_i is not a value, reduce using (*ctx*) with context $\mathsf{Impl}(\nu, \{?p_1 \mapsto \nu_1, \ldots, ?p_{i-1} \mapsto \nu_{i-1}, ?p_i \mapsto \_, ?p_{i+1} \mapsto \nu_{i+1}, \ldots ?p_n \mapsto e_n\})$. Otherwise, $e, e_0, \ldots, e_n$ are values and so $\mathsf{Impl}(e', \{?p_1 \mapsto e_1, \ldots, ?p_n \mapsto e_n\})$ is also a value.

Case (*split*, *letimpl*): Similar. Either sub-expression is not a value, or the type guarantees that it is a comonadic value with implicit parameter bindings that enable the (*split*) or (*letimpl*) reduction, respectively. □

Theorem 34 (Safety of context-aware langauge with implicit parameters). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow^{*e'}$ *then either* e' *is a value of type* τ *or there exists* e'' *such that* $e' \rightsquigarrow e''$ *and* $\Gamma \vdash e'' : \tau$.

Proof. Rule induction over $\rightsquigarrow^*$ using Theorem 32 and Theorem 33. □

2.5 GENERALIZED SAFETY OF COMONADIC EMBEDDING

In Section **??** and Section 2.4.2, we proved the safety property of two concrete context-aware programming languages based on the coeffect language framework. The proofs for the two systems were very similar and relied on the same key principle.

The principle is that the coeffect annotation r on the type modelling the indexed comonad structure $C^r\tau$ in the target language guarantees that the comonadic value will provide the necessary context. As a result the reductions for operations accessing the context do not get stuck. In case of dataflow, prev_n can always access the tail of the stream and and $\mathsf{counit}_{\mathsf{use}}$ can always access the head (because the stream has a sufficient number of elements). In case of implicit parameters, the context passed to $\mathsf{lookup}_{?p}$ will always contain a binding for $?p$.

Our core functional target language is not expressive enough to capture the relationship between the coeffect annotation and the structure of the Df or Impl value and so we resorted to adding those as ad-hoc extensions. However, given a target language with a sufficiently expressive type system, the properties proved in Section 2.4 would be guaranteed directly by the target language. This includes dependently-typed languages such as Idris or Agda [4, 3], but type-level numerals and sets can also be encoded in the Haskell type system [26].

In other words, the flat coeffect type system, together with the translation for introduced in this chapter, can be embedded into a Haskell-like languages and it can provide a succinct and safe way of implementing context-aware domain specific languages.

COEFFECTS FOR LIVENESS. As an example, we consider the third instance of coeffect calculus that was discussed in Chapter 1. If we wanted to follow the development in the previous section for liveness, we would extend the target language with two kinds of expressions, Dead representing a dead context with no value and Live representing a context with a value:

$$e \quad = \quad \ldots \mid \mathsf{Dead} \mid \mathsf{Live}\ e$$

The typing rules promote the information about whether a value is available into the type-level and so a context carrying a live value is marked as $C^{\mathsf{L}}\tau$ while a dead context has a type $C^{\mathsf{D}}\tau$.

$$(\textit{live})\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Live}\ e : C^{\mathsf{L}}\tau} \qquad\qquad (\textit{dead})\ \frac{}{\Gamma \vdash \mathsf{Dead} : C^{\mathsf{D}}\tau}$$

Finally, we need to add reduction rules that define the meaning of the comonadically-inspired operations for liveness. Those follow the definitions given in Example 10 when discussing the categorical semantics:

$$
\begin{array}{ll}
(\textit{counit}) & \mathsf{counit}_{\mathsf{L}}\ (\mathsf{Just}\ v) \leadsto v \\[1em]
(\textit{cobind-1}) & \mathsf{cobind}_{\mathsf{L,L}}\ f\ (\mathsf{Live}\ v) \leadsto \mathsf{Live}\ (f\ v) \\
(\textit{cobind-2}) & \mathsf{cobind}_{\mathsf{D,L}}\ f\ (\mathsf{Live}\ v) \leadsto \mathsf{Live}\ (f\ \mathsf{Dead}) \\
(\textit{cobind-3}) & \mathsf{cobind}_{\mathsf{L,D}}\ f\ v \leadsto \mathsf{Dead} \\
(\textit{cobind-4}) & \mathsf{cobind}_{\mathsf{D,D}}\ f\ v \leadsto \mathsf{Dead}
\end{array}
$$

This language extension is safe because the reductions respect the typing of the comonadically-inspired operations. The $\mathsf{counit}_{\mathsf{use}}$ reduction does not get stuck for well-typed terms because $\mathsf{use} = \mathsf{L}$ in the coeffect algebra and thus its argument is of type $C^{\mathsf{L}}\tau$ and will always be a value $\mathsf{Live}\ v$.

Similarly, the when reducing $\mathsf{cobind}_{r,s}$, the typing ensures that the value passed as the second argument is of type $C^{r \circledast s}\tau_1$. In case of liveness, $r \circledast s = \mathsf{L}$ if either $r = \mathsf{L}$ or $s = \mathsf{L}$. This means that reductions (*cobind-1*) and (*cobind-2*) will not get stuck because the value will be Live v and not Dead. The reduction rules also preserve typing – the resulting value is of type $C^{s}\tau_2$, that is Live v for (*cobind-1*), (*cobind-2*) and Dead for (*cobind-3*) and (*cobind-4*).

ENCODING LIVENESS IN HASKELL. The liveness example can be encoded in Haskell using type-level features such as generalized algebraic data types (GADTs) and type families [18, 44, 8], which encode some of the features known from dependently-typed languages such as Agda [3]. We do not aim to give a complete implementation, but to show that such encoding is possible and would provide the necessary safety guarantees.

We first define types D and L to capture the coeffect annotations. Then we define a comonadically-inspired type C r a as a GADT with cases for Live and Dead contexts. The type parameter r represents a coeffect annotation:

```
data L
data D

data C r a where
  Live  :: a → C L a
  Dead :: C D a
```

The definition matches with the typing rules (*live*) and (*dead*). The coeffect annotation for a live value is L and the annotation for a dead value is D. To give the type of cobind, we need a type-level function that encodes the operations of the flat coeffect algebra. We model $\circledast$ as Seq a b. The operation is defined on types L and D and returns a type D if and only if both its arguments are D:

```
type family Seq r s :: *
  type instance Seq D D = D
  type instance Seq L s = L
  type instance Seq r L = L
```

The counit and cobind operations can then be defined as Haskell functions that have types corresponding to the typing rules (*counit*) and (*cobind*) given in Figure 7:

```
counit  ::  C L a → a
cobind  ::  (C r a → b) → C (Seq r s) a → C s b
```

Here, the additional type parameter is used as a phantom type [?] and ensures that counit can only be called on a context that contains a value and so calling the operation is not going to fail. Similarly, the type of the cobind operation now guarantees that if a function (used as the first argument) or the result require a live context, it will be called with a value C L a that is guaranteed to contain a value.

COEFFECTS IN DEPENDENTLY-TYPED LANGUAGES. If we use the above encoding, type preservation is guaranteed by the type system of the target language. The equivalent of the progress property is guaranteed by the fact that the the implementation of the operations is well-defined.

It is worth noting that this is where coeffects need a target language with a more expressive type system than monads. For monadic computations, it is sufficient to use a type $\mathsf{M}\ \alpha$ which represents that *some* effect may happen. The the type does not specify which effects and, indeed, this means that *all possible effects* may happen.

With coeffects, we need to use indexed comonads $\mathsf{C}\ r\ \alpha$ where the annotation r specifies what context may be required. Without the annotation, a type $\mathsf{C}\ \alpha$ would represent a comonadic context that has *all possible context* available, which is rarely useful in practice.

2.6 RELATED CATEGORICAL STRUCTURES

Related work leading to coeffects has already been discussed in Chapter **??** and we covered work related to individual concepts throughout the thesis. However, there is a number of related categorical structures that are related to our *indexed comonads* (Section 2.2.4) that deserve additional discussion.

In Section 2.6.1, we discuss related approaches to adding indices to categorical structures (mostly monads). In Section 2.6.2, we discuss a question that often arises when discussing coeffects and that is *when is a coeffect (not) an effect?*

2.6.1 *Indexed categorical structures*

Ordinary comonads have the *shape preservation* property [25]. 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. Thus *indexed comonads* are a generalization of *comonads* that captures structures that do not form a (non-indexed) comonad. However, indexing has been also discussed in the case of *monads.*

FAMILIES OF MONADS. When linking effect systems and monads, Wadler and Thiemann [21] 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* $(\mathsf{C}^r, \mathsf{cobind}_r, \mathsf{counit}_r)$ *for all* r *such that each triple forms a comonad. Given* r, r' *such that* $r \leqslant r'$*, there is also a mapping* $\iota_{r',r} : \mathsf{C}^{r'} \to \mathsf{C}^r$ *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 Maybe does not form a family of comonads (again, because counit is not defined on $\mathsf{C}^{\mathsf{D}}\alpha = 1$). 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 [15] (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.

2.6.2 *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 demands 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 demands using the reader monad, we need to extend the monad structure and change the translation of monadic lambda abstraction.

2.6.3 *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 demands 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:

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

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 [24], when used to model computation semantics, the two representations are equivalent:

Remark 35. *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 demands 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 demands $r \cup s$ 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 demands in lambda abstraction. Can we also go the other way round and *restrict* the comonadic semantics so that all demands 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 obtain the (*mabs*) rule, we can restrict the first parameter, which corresponds to the declaration site context:

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

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 [37].

2.7 SUMMARY

In the previous chapter, we defined *type system* for flat coeffect calculi that uniformly captures the shared structure of context-aware computations. In this chapter, we completed the unification by providing *semantics* for flat coeffect calculi and proving the *safety* of coeffect languages for dataflow and implicit parameters. The semantics shown here also guides the implementation that is discussed later in Chapter 3.

The development presented in this chapter follows the well-known example of effects and monads. We introduced the notion of *indexed comonad*, which generalizes comonads and adds additional operations needed to provide categorical semantics of the flat coeffect calculus and we demonstrated how implicit parameters, liveness and dataflow computations form indexed comonads.

We then used the comonadic semantics to define a *comonadically-inspired translation* that turns programs written in a domain-specific coeffect language into a functional target language. This is akin to the Haskell do notation for monads. Finally, we extended the target language with concrete implementation of comonadic operations for dataflow and implicit parameters and we presented syntactic safety proof. In summary, the proof states that well-typed context-aware programs written in a coeffect language *do not go wrong* (when translated to a simple functional language and evaluated).

The proof relies on the fact that coeffect annotations (provided by the coeffect type system) guarantee that the required context is available in the comonadic value that represents the context and we also discussed how this would guarantee safety in languages with sufficiently expressive type system such as Haskell.

In the following 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 **??**.

3 IMPLEMENTATION

In the previous three chapters, we presented the theory of coeffects consisting of type system and comonadically inspired semantics of two parameterized coeffect calculi. The theory provides a framework that simplifies the implementation of safe context-aware programming languages. To support this claim, this chapter presents a prototype implementation of three coeffect languages – language with implicit parameters and both flat and structural versions of a data-flow language.

The implementation directly follows the thoery presented in the previous three chapters. It consists of a common framework that provides type checking and translation to a simple functional target language with comonadically-inspired primitives. Each concrete context-aware language then adds a domain-specific rule for choosing unique typing derivation (as discussed in Section 1.3) together with a domain-specific definition of the comonadically-inspired primitives that define the runtime semantics (see Section 2.4).

The main goal of the implementation is to show that the theory is practically useful and to present it in a more practical way. However, we do not intend to build a complete real-world programming language. For this reason, the implementation is available primarilly as an interactive web-based essay, though it can be also downloaded and run locally.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We discuss how the implementation follows from the theory presented earlier (Section 3.1). This applies to the implementation of the *type checker* and the implementation of the *translation* to a simple target langauge that is then iterpereted. We also discuss how the common framework makes it easy to implement additional context-aware languages (Section 3.1.3).
- We consider a number of case studies (Section 3.2) that illustrate interesting aspects of the theories discussed earlier. This includes the typing of lambda abstraction and the difference between flat and structural systems (Section 3.2.1) and the comonadically-inspired translation (Section 3.2.2).
- The implementation is available not just as downloadable code, but also in the format of interactive web-based essay (Section 3.3), which aims to make coeffects accessible to a broader audience. We discuss the most interesting aspects of the essay format and briefly discuss some of the interesting implementation detail (Section 3.3.2).

3.1 FROM THEORY TO IMPLEMENTATION

The theory discussed so far provides the two key components of the implementation. In Chapter 1, we discussed the type checking of context-aware programs and Chapter 2 models the execution of context-aware programs (in terms of translation and operational semantics). For structural coeffects, the same components are discussed in Chapter **??**. In this section, we discuss how those provide foundation for the implementation.

3.1.1 *Type checking and inference*

To simplify the writing of context-aware programs, the implementation provides a limited form of type inference (Section **??**). This is available just for convenience and so we do not claim any completeness or complexity result about the algorithm and we do not present full formalization. However, it is worth noting how the domain-specific procedures for choosing a unique type derivation (Section 1.3) are adapted.

The type inference works in the standard way [32, 9] by generating type constraints and solving them. Solving of type constraints is done in the standard way, but we additionally collect and solve *coeffect constraints*. In order to obtain unique type derivation, we generate additional coeffect constraints in lambda abstraction of each flat coeffect language.

FLAT IMPLICIT PARAMETERS. As discussed in Section 1.3, when choosing unique typing derivation for implicit parameters, we keep track of the implicit parameters available in lexical scope (written as Δ). In lambda abstraction rule, the implicit parameters required by the body (tracked by r) are split so that all parameters available in lexical scope are captured and only the remaining parameters ($r \setminus \Delta$) are required from the caller of the function.

From the presentation in Section 1.3, it might appear that resolving the ambiguity related to lambda abstraction for implicit parameters requires a type system different from the core flat coeffect type system shown earlier in Section 1.2.2. This is not the case.

We track implicit parameters in scope Δ, but the rest of the (*abs*) rule from the implementation only generates an additional coeffect constraint. Writing $\Gamma @ t \vdash e : \tau \mid C$ for a judgement that generates coeffect constraints C, the (*abs*) rule used for implicit parameters looks as follows:

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

Given a typing derivation for the body that produced constraints C, we generate an additional constraint that restricts r (declaration-site demands) to those available in the current static scope Δ. It is not necessary to generate a constraint for the coeffect s, because our constraint satisfaction algorithm finds the minimal set s which is $t \setminus \Delta$.

FLAT DATA-FLOW. In context-aware language for data-flow (and in language with liveness tracking), the inherent ambiguity of the (*abs*) rule is resolved by duplicating the context requirements of the body. In Section 1.3, this was defined by replacing the standard coeffect (*abs*) rule with a rule (*idabs*) that uses an annotation r for the body of the function, declaration-site coeffect and call-site coeffect.

As with implicit parameters, the implementation does not require changing the core (*abs*) typing rule of the flat coeffect system. Instead, the unique resulution is obtained by generating additional coeffect constraints:

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

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*).

3.1.2 *Execution of context-aware programs*

Context-aware programs are executed by translating the source program into a simple functional target language. For simplicity, programs in the simple target language are then interpreted, but they could equally be compiled using standard techniques for compiling functional code. The translation follows the rules defined in Section 2.3 (for flat coeffect languages) and Section X (for structural coeffect languages). The result of the translation is a program that consists of the following:

- Functional constructs. Those include binary operations, tuples, let binding, constants, variables, function abstraction and application. The interpreter keeps a map of assignments for variables in scope and recursively evaluates the expression.
- Comonadic operations. Those are the comonadic primitives provided by indexed comonads – cobind, counit together with merge and split for flat coeffects or merge and choose for structural coeffects. The translation that produces these is shared by all context-aware languages, but their definition in the interpreter is domain-specific.
- Domain-specific operations. Each context-aware language may additionally include operations that model domain-specific operations. For data-flow, this is prev (accessing past values) and for implicit parameters, this is letimpl and lookup for implicit parameter binding and access, respectively.

The fact that the prototype implementation is based on the theoretical framework provided by coeffect calculi means that it has the desirable properties proved in Section 2.4 and Section X. In particular, evaluating a well-typed context-aware program in a context that provides sufficient contextual capabilities will not cause an error.

In the interactive essay (Section 3.3), we further use the coeffects to automatically generate a user interface that requires the user to provide the required contextual capabilities (past values for individual variables, or values for implicit parameters).

Another benefit of using the common framework is that the implementation can be easily extended to support additional context-aware languages.

3.1.3 *Supporting additional context-aware languages*

The prototype implementation supports two of the context-aware languages discussed in this thesis: implicit parameters and data-flow. The remaining examples are calculus for tracking variable liveness (of ordinary variables) and the structural language based on bounded reuse (counting number of times values are used). However, the prototype implementation is based on the common coeffect framework and makes it easy to add support for these and also for other context aware languages based on coeffects.

In order to extend the implementation with support for liveness or bounded reuse tracking (or other context-aware language), the following 4 additions are required:

1. A domain-specific function abstraction rule that resolves the ambiguity in the general (*abs*) rule of the flat coeffect calculus. For liveness, the handling would be the same as for data-flow, but for other flat coeffect systems, another resolution mechanism might be used instead.

2. A domain-specific instance of coeffect algebra needs to be provided. In order to support the type inference in the prototype implementation, the constraint solver needs to be extended to solve constraint using the coeffect algebra. For liveness, this would be solving simple two-point lattice constraints.

3. For evaluation, a new kind of comonadic values needs to be added. For liveness, this would be an option value that may or may not contain a value. The semantics of comonadic operations on the values needs to be defined.

4. For context-aware languages that have additional primitives (such as `prev` or `?param`), the parser and AST needs to be extended, custom type-checking and translation rules added and domain-specific primitive operations (with their semantics) provided. Liveness and bounded reuse do not have additional custom syntax and so supporting these would not require this step.

The list mirrors a list of steps that need to be done when supporting a new effectful computation in a language that supports monadic do-notation. The step (3) corresponds to implementing a new monad and (4) corresponds to adding monad-specific effectful operations. The step (2) applies when using indexing to track effects more precisely [26]. The only step that does not have effectful/monadic counterpart is (1).

When adding a new context-aware programming language support, much of the existing infrastructure can be reused. This includes the implementation of the core coeffect and type checking rules and also the translation for standard language constructs as well as the interpreter for the target language.

3.2 CASE STUDIES

The prototype implementation illustrates a number of interesting aspects of coeffect systems. Those appear as examples in the interactive essay (discussed in Section 3.3), but we briefly review them in this section.

3.2.1 *Typing context-aware programs*

We first consider two case studies of how coeffect type checking works. The first one exposes the resolution of the ambiguity in typing for implicit parameters and the second one exposes the difference between flat and structural system for data-flow.

ABSTRACTION FOR IMPLICIT PARAMETERS. As discussed in Section 3.1.1, the implementation of the language with implicit parameters resolves the ambiguity in the lambda abstraction by generating a coeffect constraint that restricts the set of parameters required from the declaration-site to those that are lexically available. Remaining parameters are required from the call-site. This is illustrated by the following example:

```
let both =
  let ?fst = 100 in
  fun trd → ?fst + ?snd + trd in
let ?fst = 200 in
both 1
```

In this expression, the lambda function on line 3 requires implicit parameters ?fst and ?snd. Since ?fst is available in scope, the type of both is a function that requires only ?snd. In the text-based notation used in the prototype, the type of the function both is: `num -{?snd:num}-> num`.

FLAT AND STRUCTURAL DATA-FLOW. In flat data-flow, the context requirements of the body is required from both the declaration-site and from the call-site. In structural data-flow, the context requirements are tracked separately for each variable, which provides a more precise type. Consider the following two examples (the `let` keyword is used to define a curried function of two arguments):

```
let oldy x y = x + prev y in
oldy
```

When type checking the expression using the flat system, the type of oldy is inferred as `num -{1}-> num -{1}-> num`, but when using the structural system, the type becomes `num -{0}-> num -{1}-> num`.

This illustrates the difference between the two - the flat system keeps only one annotation for the whole body (which requires 1 past value). In lambda abstraction (or function declaration written using `let`), this requirement is duplicated. The structural system keeps information per-variable and so the resulting type reflects the fact that only the variable y appears inside `prev`.

3.2.2 *Comonadically-inspired translation*

In addition to running coeffect programs, the implementation can also print the result of the translation to the simple functional target language with comonadically-inspired primitives. The following two case studies illustrate important aspects of the translation for flat coeffect systems (Section 2.3) and structural coeffect systems (Section ?).

MERGING IMPLICIT PARAMETER CONTEXTS. The following example illustrates the lambda abstraction for implicit parameters. It defines a parameter ?param and then returns a function value that captures it, but also requires an implicit parameter ?other:

```
let ?param = 10 in
fun x → ?param + ?other
```

Translating the code to the target language produces the code below. The reader is encouraged to view the translation in the interactive essay (Section 3.3), which displays the types and coeffect annotations of the individual values and primitives. As in the theory, the comonadically-inspired primitives are families of operations indexed by the coeffects (we omit the annotations here):

```
let (ctx2, ctx3) = split (duplicate finput) in
let ctx1 = letimpl_?param (ctx2, 10) in
fun x →
  let ctx4 = merge (x, ctx1) in
  let (ctx5, ctx6) = split (duplicate ctx4) in
  lookup_?param ctx5 + lookup_?other ctx6
```

The `finput` value on the first line represents an empty context in which the expression is evaluated and is of type C^{ign}unit. The context is duplicated; ctx3 is not needed, because 10 is a constant; ctx2 is passed to letimpl, which assigns an implicit parameter value in the newly returned context ctx1 of type $C^{\{?param\}}$unit (this now carries the implicit parameter value, but it does not represent any ordinary variables, thus the unit type representing an empty tuple).

In the body of the function, the context ctx1 is merged with the context provided by the variable x. The type of ctx4 is $C^{\{?param,?other\}}(\mathsf{num} \times \mathsf{unit})$. This is then split into two parts that contain just one of the implicit parameters and those are then accessed using lookup.

COMPOSITION IN STRUCTURAL DATA-FLOW. In structural coeffect systems, the translation works differently in that the context passed to a sub-expression contains only assignments for the variables used in the sub-expression (in the flat version, we always duplicated the variable context before using split). To illustrate this, consider the following simple function:

```
fun x → fun y → prev x
```

In structural coeffect systems, the comonadic value is annotated with a vector of coeffect annotations that correspond to individual variables. The initial structural input sinput is a value of type $C^{[]}()$ containing no variables (we write () rather than unit to make that more explicit). The translated code then looks as follows:

```
fun x →
  let ctx1 = merge (x, sinput) in
  (fun y →
    let ctx2 = merge (y, ctx1) in
    counit (prev (choose⟨0,1⟩ ctx2))
```

The two variables are merged with the initial context, obtaining a value ctx2 of type $C^{[0,1]}\mathsf{num} \times \mathsf{num}$ that contains two values with 0 and 1 past values, respectively.

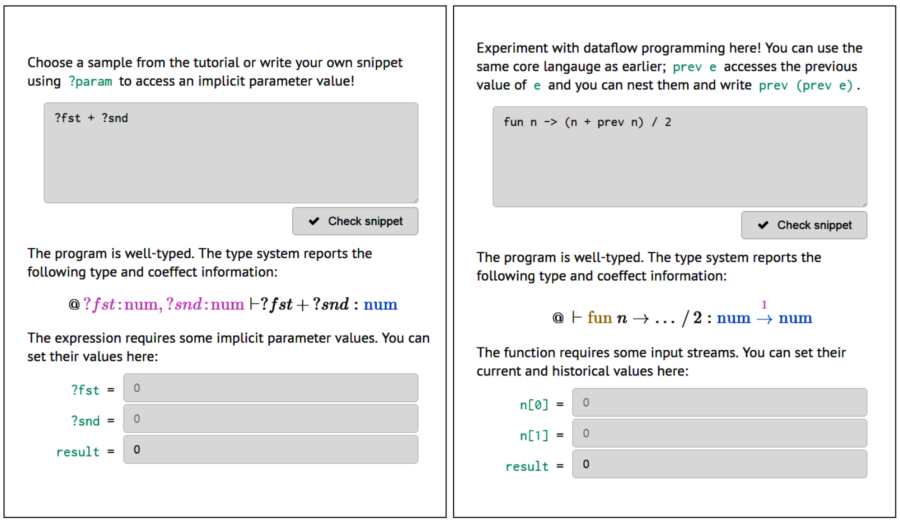
For simplicity, the implementation does not use the split/merge pair of operations of the structural coeffects to obtain the correct subset of variables. This can be done, but it would make the translated code longer and more cumbersome. Instead, we use a higher-level operation choose (which can be expressed in terms of split/merge) that projects the variable subset as specified by the index. Here, $\langle 0, 1\rangle$ means that the first variable should be dropped an the second one should be kept.

The resulting single-variable context is then passed to prev (to shift the stream by one) and then to counit to obtain the current value.

3.3 INTERACTIVE ESSAY

As explained in the introduction of this chapter, the purpose of the implementation presented in this thesis is not to provide a real-world programming language, but to support the theory discussed in the rest of the thesis. The goal is to explain the theory and inspire authors of real-world programming langauges to include support for context-aware programming, ideally using coeffects as a sound foundation. For this reason, the implementation needs to be:

- ACCESSIBLE. Anyone interested should be able to use the implemented langauges without downloading the source code and compiling it and without installing specialized software.


Choose a sample from the tutorial or write your own snippet using `?param` to access an implicit parameter value!

```
?fst + ?snd
```

The program is well-typed. The type system reports the following type and coeffect information:

$$@\,?fst : \text{num}, ?snd : \text{num} \vdash ?fst + ?snd : \text{num}$$

The expression requires some implicit parameter values. You can set their values here:

```
?fst   = 0
?snd   = 0
result = 0
```

Experiment with dataflow programming here! You can use the same core langauge as earlier; `prev e` accesses the previous value of `e` and you can nest them and write `prev (prev e)`.

```
fun n -> (n + prev n) / 2
```

The program is well-typed. The type system reports the following type and coeffect information:

$$@ \vdash \text{fun } n \to \ldots / 2 : \text{num} \xrightarrow{1} \text{num}$$

The function requires some input streams. You can set their current and historical values here:

```
n[0]   = 0
n[1]   = 0
result = 0
```

Figure 11: Interactive evaluation of implicit parameters (left) and data-flow (right)

- EXPLORABLE. It should be possible to explore the inner workings – how is the typing derived, how is the source code translated to the target language and how is it evaluated.

To make the work *accessible*, we implement sample context-aware languages in a way that makes it possible to use them in any standard web browser with JavaScript support (Section 3.3.2) without requiring any server-side component. Following the idea that "the medium is the message" [19], we choose medium that encourages *exploration* and make the implementation available not just as source code that can be compiled and run locally, but also in the format of interactive essay (Section 3.3.1). The live version of the essay can be found at: `http://tomasp.net/coeffects`.

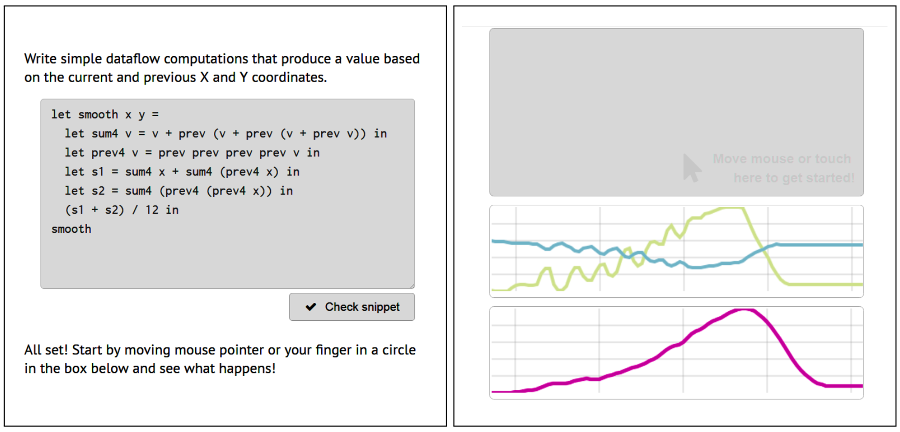
3.3.1 *Explorable language implementation*

The interactive essay format used of the implementation is inspired by Bret Victor's work on *explorable explanations* [40]:

> *Do our reading environments encourage active reading? Or do they utterly oppose it? A typical reading tool, such as a book or website, displays the author's argument, and nothing else. The reader's line of thought remains internal and invisible, vague and speculative. We form questions, but can't answer them. We consider alternatives, but can't explore them. We question assumptions, but can't verify them. And so, in the end, we blindly trust, or blindly don't, and we miss the deep understanding that comes from dialogue and exploration.*

The interactive essay we present encourages active reading in the sense summarized in Victor's quote. We show the reader an example (program, typing derivation or translation), but the reader is encouraged to modify it and see how the explanation in the essay adapts.

The idea of active reading is older and has been encouraged in the context of art Josef Albers' classic 1963 work on color [1] (which has been turned into an interactive essay 60 years later [31]). More recently, similar formats have been used to explain topics in areas such as signal processing [33] (explaining Fourrier transformations) and sociology [12] (visualizing and



Write simple dataflow computations that produce a value based on the current and previous X and Y coordinates.

```
let smooth x y =
  let sum4 v = v + prev (v + prev (v + prev v)) in
  let prev4 v = prev prev prev prev v in
  let s1 = sum4 x + sum4 (prev4 x) in
  let s2 = sum4 (prev4 (prev4 x)) in
  (s1 + s2) / 12 in
smooth
```

All set! Start by moving mouse pointer or your finger in a circle in the box below and see what happens!

Figure 12: Function smoothing the X coordinate (left) with a sample run (right).

explaining game theoretical model of segregation in the society [34]). To our best knowledge, no such work exists in the area of programming language theory and so we briefly outline some of the interesting features that our essay provides in order to encourage active reading.

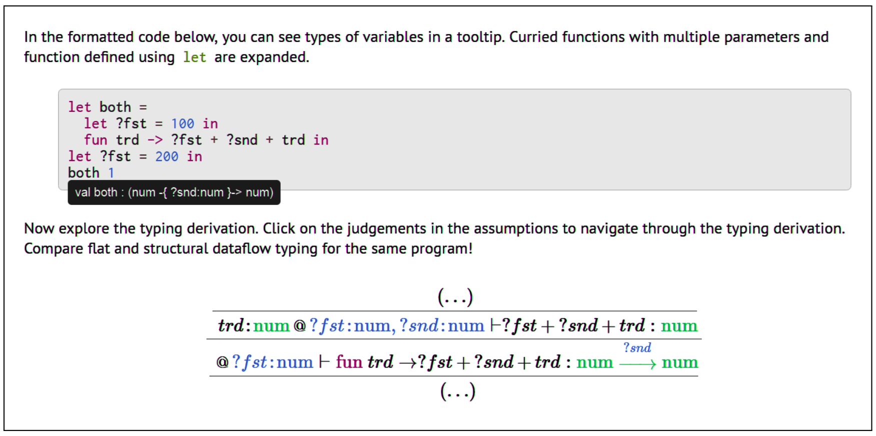
INTERACTIVE PROGRAM EXCUTION. After providing the practical motivation for coeffects (based on Chapter **??**), the interactive essay shows the reader the two sample context-aware programming languages. Readers can write code in a panel that type checks the input, generates user interface for entering the required context and runs the sample code.

The panels for implicit parameters and for data-flow computations are shown in Figure 11. The sample code on the left adds two implicit parameters and the generated UI lets the user enter implicit parameter values as required by the context in the typing judgement. The sample on the right calculates the average of the current and past value in a data-flow; the UI lets the user enter past values required by the type of the function.

The essay guides the readers through a number of interesting programs (shown in Section 3.2.1) and encourages them to run and try modifying them. For implicit parameters, this includes the case where an implicit parameter is available at both declaration-site and call-site (showing that the declaration-site value is captured). For data-flow, the examples include the comparison between the inferred type when using flat and structural type systems.

REACTIVE DATA-FLOW. The interactive program execution lets the reader run sample programs, but not in a realistic context. To show a more real-world scenario, the essay includes a widget shown in Figure 12. This lets the user write a function taking a stream of X and Y coordinates and calculate value based on the current and past values of the mouse pointer. The X and Y values, together with the result are plotted using a live chart.

In the example run shown above, the sample program calculates the average of the last 12 values of the X coordinate (green line in Figure 12). The example also illustrates one practical use of the coeffect type system – when running, the widget keeps the coordinates in a pre-allocated fixed-size array, because the coeffect type system guarantees that at most 12 past values will be accessed.



In the formatted code below, you can see types of variables in a tooltip. Curried functions with multiple parameters and function defined using `let` are expanded.

```
let both =
  let ?fst = 100 in
  fun trd -> ?fst + ?snd + trd in
let ?fst = 200 in
both 1
```

```
val both : (num -{ ?snd:num }-> num)
```

Now explore the typing derivation. Click on the judgements in the assumptions to navigate through the typing derivation. Compare flat and structural dataflow typing for the same program!

$$\dfrac{\dfrac{(\ldots)}{\mathit{trd}:\mathrm{num} \,@\, ?\mathit{fst}:\mathrm{num}, ?\mathit{snd}:\mathrm{num} \vdash ?\mathit{fst} + ?\mathit{snd} + \mathit{trd} : \mathrm{num}}}{@\, ?\mathit{fst}:\mathrm{num} \vdash \mathsf{fun}\ \mathit{trd} \to ?\mathit{fst} + ?\mathit{snd} + \mathit{trd} : \mathrm{num} \xrightarrow{?\mathit{snd}} \mathrm{num}}$$

$$(\ldots)$$

Figure 13: Source code with type information and explorable typing derivation.

EXPLORABLE TYPING DERIVATIONS. Perhaps the most important aspect of the implemented context aware programming languages is their coeffect and type system. In a conventional implementation, the functioning of the type system would remain mostly hidden – we would get an "OK" message for a well-typed program and a type error for invalid program (likely not very informative, given that reporting good error messages for type errors is a notoriously hard problem even for mature language implementations).

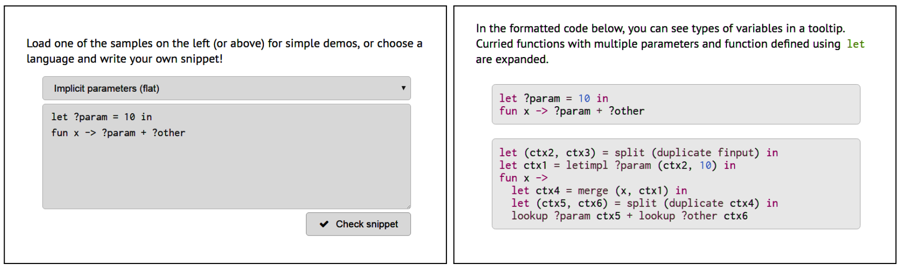
The presented interactive essay provides two features to help the reader understand and explore typing derivations. First, when reading code samples in the text, tooltips show the typing of individual identifiers, most interestingly functions (Figure 13, above).

Second, a later part of the essay provides a type checker that lets the user enter a source code in a context aware programming language and produces an explorable typing derivation for the program. The output (shown in Figure 13, below) displays a typing judgement with assumptions and conclusions and lets the reader navigate through the typing derivation by clicking on the assumptions or conclusions. This way, the reader can see how is the final typing derivation obtained, exploring interesting aspects, such as the abstraction rule (shown in Figure 13).

COMONADICALLY-INSPIRED TRANSLATION. The last interactive element of the essay lets the reader explore the translation of source context aware language to a target simple functional language (Section 3.1.2). Compared with the monadic do-notation [14], the comonadic translation is more complex for two reasons. First, it merges all variables into a single (comonadic) value representing the context. Second, there is a flat and structural variation of the system. For these two reasons, understanding the translation based just on the rules is harder than for monads. The essay lets readers see translation for carefuly chosen case studies that illustrate the key aspects of the translation (some discussed in Section 3.2.2), but also for their own programs encouraging *active reading*. Sample input and output are shown in Figure 14.

3.3.2 *Implementation overview*

The core part of our implementation mostly follows standard techniques for implementing type checkers and interpreters for statically-typed func-



Load one of the samples on the left (or above) for simple demos, or choose a language and write your own snippet!

```
let ?param = 10 in
fun x -> ?param + ?other
```

In the formatted code below, you can see types of variables in a tooltip. Curried functions with multiple parameters and function defined using `let` are expanded.

```
let ?param = 10 in
fun x -> ?param + ?other
```

```
let (ctx2, ctx3) = split (duplicate finput) in
let ctx1 = letimpl ?param (ctx2, 10) in
fun x ->
  let ctx4 = merge (x, ctx1) in
  let (ctx5, ctx6) = split (duplicate ctx4) in
  lookup ?param ctx5 + lookup ?other ctx6
```

Figure 14: TBD

tional programming langauges. Two interesting aspects that are worth discussing is the JavaScript targetting (for running the langauge implementation in a web browser) and integration with client-side (JavaScript) libraries for building the user interface. The full source code can be obtained from `https://github.com/coeffects/coeffects-playground` and is structured as follows:

- Parsing is implemented using smple parser combinator library; `ast.fs` defines the abstract syntax tree for the languages; `parsec.fs` implements a small parser combinator library and `lexer.fs` with `parser.fs` parse the source code by first tokenizing the input and then parsing the stream of tokens.

- The type checking is implemented by `typechecker.fs`, which annotates an untyped AST with types and generates set of type and coeffect constraints. The constraints are later solved (using domain-specific algorithms for each of the languages) in `solver.fs`.

- Type-checked programs in context-aware languages are translated to simple target functional subset of the language in `translation.fs`; `evaluation.fs` then interprets programs in the target language. The interpretation does not handle the source language and so programs containing context-aware contstructs cannot be interpreted directly.

- The user interface of the interactive essay discussed in Section 3.3.1 is implemented partly in F# and partly in JavaScript. The two most important components include a pretty-printer `pretty.fs` which formats source code (with type tooltips) and typing derivations and `gui.fs` that implements user interaction (e.g. navigation in explorable typing derivations).

As discussed in Section 3.1.3, the implementation can be easily extended to support additional context-aware programming languages. This is due to the fact that it is based on the unified theory of coeffects. In practice, adding support for liveness tracking would require adding a domain-specific constraint solver in `solver.fs` and extending the interpreter in `evaluation.fs` with a new kind of comonadically-inspired data type (indexed maybe comonad) with its associated operations.

3.4 RELATED WORK

To our best knowledge, combining Bret Victor's idea of explorable explanations with programming language theory (as discussed in Section 3.3) is a novel contribution of our work. On the technical side, we build on a number of standard methods.

PARSING AND TYPE CHECKING. The implementation of the parser, type checker and interpreter follows standard techniques for implementing functional programming languages. In order to be able to compile the implementation to JavaScript (see below), we built a small parser combinator library [13] rather than using one of the already available libraries [38].

TARGETTING JAVASCRIPT. In order to make the implementation accessible to a broad audience, it can be executed in a web browser. This is achieved by automatically translating the implementation from F# to JavaScript. We use an F# library called FunScript [5] (which is a more recent incarnation of the idea developed by the author [27]). We choose F#, but similar tools exist for other functional languages such as OCaml [41]. It is worth noting that FunScript is implemented as a *library* rather than as a compiler extension. This is done using the meta-programming capabilities of F# [35].

CLIENT-SIDE LIBRARY INTEGRATION. An interesting aspect of the interactive essay user interface is the integration with third-party JavaScript libraries. We use a number of libraries including JQuery (for web browser DOM manipulation) and MathJax [7] (for rendering of typing derivation). In order to call those from F# code, we use a number of mapping methods described in [28]. For example, the following F# declaration is used to invoke the Queue function in MathJax:

```
[⟨JSEmitInline("MathJax.Hub.Queue({0});")⟩]
let queueAction (f : unit → unit) : unit = failwith "JS only!"
```

The JSEmitInline syntax on the first line is an attribute that instructs FunScript to compile all invocations of the queueAction function into the JavaScript literal specified in the attribute (with `{0}` replaced by the argument).

3.5 SUMMARY

This chapter supplements the theory of coeffects presented in the previous three chapters with a prototype implementation. We implemented three simple context-aware programming langauges that track implicit parameters and past values in dataflow computations (in flat and structural way).

The implementation discussed in this chapter provides an evidence for our claim that the theory of coeffects can be used as a basis for a wide range of sound context-aware programming languages. Our implementation consists of a shared coeffect framework (handling type checking and translation). Each context-aware language then adds a domain-specific rule for choosing a unique typing derivation, an interpretation of comonadically-inspired primitives and (optionally) domain-specific primitives such as the `prev` construct for dataflow.

We make the implementation available in the usual form (as source code that can be downloaded, compiled and executed), but we also present it in the form of interactive essay. This encourages *active reading* and lets the reader explore a number of aspects of the implementation including the type checking (through explorable typing derivations) and the translation. The key contribution of this thesis is that it provides a unified way for *thinking* about context in programming langauges and the interactive presentation of the implementation is aligned with this goal. Programming languages of the future will need a mechanism akin to coeffects and we aim to provide a convincing argument supported by a prototype implementation.

BIBLIOGRAPHY

[1] J. Albers. *Interaction of color*. Yale University Press, 2013.

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

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

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

[5] Z. Bray. Funscript: F# to javascript with type providers. Available at `http://funscript.info/`, 2016.

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

[7] D. Cervone. Mathjax: a platform for mathematics on the web. *Notices of the AMS*, 59(2):312–316, 2012.

[8] M. M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ACM SIGPLAN Notices*, volume 40, pages 241–253. ACM, 2005.

[9] L. Damas. Type assignment in programming languages. 1984.

[10] D. R. Ghica and A. I. Smith. Bounded linear types in a resource semiring. In *Programming Languages and Systems*, pages 331–350. Springer, 2014.

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

[12] V. Hart and N. Case. Prable of the polygons: A playable post on the shape of society. Available at `http://ncase.me/polygons/`, 2014.

[13] G. Hutton and E. Meijer. Monadic parser combinators. 1996.

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

[15] 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.

[16] 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.

[17] 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.

[18] C. McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

[19] M. McLuhan and Q. Fiore. The medium is the message. *New York*, 123:126–128, 1967.

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

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

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

[23] D. Orchard. Programming contextual computations.

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

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

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

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

[28] T. Petricek, D. Syme, and Z. Bray. In the age of web: Typed functional-first programming revisited. In *Post-proceedings of ML Workshop*, ML 2014.

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

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

[31] Potion Design Studio, based on the work of Josef Albers. Interaction of color: App for iPad. Available at http://yupnet.org/interactionofcolor/, 2013.

[32] F. Pottier and D. Rémy. The essence of ml type inference, 2005.

[33] J. Schaedler. Seeing circles, sines, and signals: A compact primer on digital signal processing. Available at https://github.com/jackschaedler/circles-sines-signals, 2015.

[34] T. Schelling. Dynamic models of segregation. *Journal of mathematical sociology*, 1(2):143–186, 1971.

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

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

[37] 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.

[38] S. Tolksdorf. Fparsec-a parser combinator library for f#. Available at http://www.quanttec.com/fparsec, 2013.

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

[40] B. Victor. Explorable explanations. Available at http://worrydream.com/ExplorableExplanations/, 2011.

[41] J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

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

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

[44] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.