CONTENTS

1 CONTEXT-AWARE SYSTEMS

Software developers as well as programming language researchers choose abstractions based not just on how appropriate they are. Other factors include social aspects – how well is the abstraction known, how well is it documented and whether it is a standard tool of the *research programme*[1] that the researcher unconsciously subscribes to.

For tracking of effects, such *standard tools* are well known. When faced with an effectful computation, programming language designers immediately pick monads. For context-aware computations, there are no standard tools. Thus contextual properties may, at first, appear as a set of disconnected examples. Existing systems that capture contextual properties use a wide range of methods including special-purpose type systems, approaches arising from modal logic S4, as well as techniques based on abstractions designed for other purpose, most frequently monads.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We start with a characterization of contextual properties. The Section 1.1 explains what is a *coeffect* and contrasts it with a better known notion of *effect*. It explains what is the nature of properties that can be tracked using coeffect systems presented in this thesis.

- We describe a number of simple calculi for tracking a wide range of contextual properties. The systems are adapted from diverse sources (type systems, static analyses, logics) and apply to various domains (cross-compilation, liveness, distributed computing, data-flow, security), but share a common structure.

- The uniform presentation of the systems is the key contribution of this chapter. We distinguish between *flat coeffect* systems (Section 1.2) and *structural coeffect* systems (Section 1.3). The fact that we find a common structure in all systems presented here lets us develop unified coeffect calculi in the upcoming three chapters.

- In addition, the coeffect systems for tracking the number of accessed past values in data-flow languages (Sections 1.2.4 and 1.3.3) presents novel results and can be used to optimize data-flow programs.

As mentioned, this chapter may appear as a collection of disconnected examples[2]. But at the end, we will see that they share a common pattern.

1.1 STRUCTURE OF COEFFECT SYSTEMS

When introducing coeffect systems in Section **??**, we related coeffect systems with effect systems. Effect systems track how a program affects the environment, or, in other words capture some *output impurity*. In contrast, coeffect systems track what a program requires from the execution envionment, or *input impurity*.

[1] A research programme, as introduced by Lakatos [59], is a network of scientists sharing the same basic assumptions and techniques.

[2] The different properties captured by monads may appear similarly disconnected at first!

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

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

Figure 1: Lambda abstraction for pure and effectful computations

Effect systems generally use judgements of the form $\Gamma \vdash e : \tau \,\&\, \sigma$, associating effects σ with the output type. We write coeffect systems using judgements of the form $\Gamma @ \sigma \vdash e : \tau$, associating the context requirements with Γ. Thus, we extend the traditional notion of free-variable context Γ with richer notions of context. This notation emphasizes the right intuition, but there are more important differences between effects and coeffects.

1.1.1 *Effectful lambda abstraction*

The difference between effects and coeffects becomes apparent when we consider lambda abstraction. The typical lambda abstraction rule for effect systems looks as (*effect*) in Figure 1. Wadler and Thiemann [127] explain how the effect analysis works as follows:

> *In the rule for abstraction, the effect is empty because evaluation immediately returns the function, with no side effects. The effect on the function arrow is the same as the effect for the function body, because applying the function will have the same side effects as evaluating the body.*

This is the key property of *output impurity*. The effects are only produced when the function is evaluated and so the effects of the body are attached to the function. A recent work by Tate [108] uses the term *producer* effect systems for such standard systems and characterises them as follows:

> *Indeed, we will define an effect as a producer effect if all computations with that effect can be thunked as "pure" computations for a domain-specific notion of purity.*

The thunking is typically performed by a lambda abstraction – given an effectful expression e, the function $\lambda x.e$ is an effect free value (thunk) that delays all effects. As shown in the next section, contextual properties do not follow this pattern.

1.1.2 *Notions of context*

We look at three notions of context. The first is the standard free-variable context in λ-calculus. This is well understood and we use it to demonstrate how contextual properties behave. Then we consider two notions of context introduced in this thesis – *flat coeffects* refer to overall properties of the environment and *structural coeffects* refer to properties attached to individual variables. We could track properties associated with values in data structures (e.g. fields of a tuple), but this is left as future work.

VARIABLE COEFFECTS. In standard λ-calculus, variable access can be seen as a primitive operation that accesses the context. The variable access expression introduces a context requirement – the expression x is typeable only in a context that contains $x : \tau$ for some type τ.

The standard lambda abstraction (*pure*), shown in Figure 1, splits the free-variable context of an expression into two parts. At runtime, the value of the parameter has to be provided by the *call site* (dynamic scope) and the remaining values are provided by the *declaration site* (lexical scope). In the type checking, the splitting is determined syntactically – the notation $\lambda x.e$ names the variable whose value comes from the call site.

Flat and structural coeffects also split context-requirements between the declaration site and the call site. The flat and structural coeffects capture two different ways of doing this.

FLAT COEFFECTS. In Section **??**, we used *resources* in a distributed system as an example of flat coeffects. These could be, for example, a database, GPS sensor or access to the current time. We also outlined that such context requirements can be tracked as part of the typing assumption, for example, say we have an expression e that requires GPS coordinates and the current time. The variable context of such expression will be annotated with a set of required resources, i. e. $\Gamma @ \{\, \mathsf{gps},\ \mathsf{time}\, \}$.

The interesting case is when we construct a lambda function $\lambda x.e$, marshall it and send it to another node. In systems such as Acute [99], the context requirements can be satisfied in a number of ways. When the same resource is available at the target machine (e. g. current time), we can transfer the function with a context requirement and *rebind* the resource. However, if the resource is not available (e. g.. GPS on the server), we need to a capture *remote reference*.

In the example discussed here, $\lambda x.e$ would require GPS sensor from the declaration site (lexical scope) where the function is declared, which is attached to the current context as $\Gamma @ \{\, \mathsf{gps}\, \}$. The current time is required from the caller of the function. So, the context requirement on the call site (dynamic scope) will be $r = \{\, \mathsf{time}\, \}$. In coeffect systems, we attach this information to the function, writing $\tau_1 \xrightarrow{r} \tau_2$.

We look at resources in distributed programming in more details in Section 1.2.2. The important point here is that in flat coeffect systems, contextual requirements are *split* between the call site and declaration site. Furthermore, there is no syntactic structure that determines how the requirements are split. In the case of distributed programming, the resources can be freely associated with either of the two sites.

STRUCTURAL COEFFECTS. On the one hand, variable context provides a *fine-grained tracking* mechanism of how context (variables) are used. On the other hand, flat coeffects let us track *additional information* about the context. The purpose of *structural coeffects* is to reconcile the two and to provide a way for fine-grained tracking of additional information linked to variables in programs. Structural coeffects follow the lexical scoping structure determined by the typing rules.

In Section **??**, we used an example of tracking array access patterns. For every variable, the additional coeffect annotation keeps a range of indices that may be accessed relatively to the current cursor. For example, consider an expression $x[\mathtt{cursor}] = y[\mathtt{cursor} - 1] + y[\mathtt{cursor} + 1]$.

Here, the variable context Γ contains two variables, both of type Arr. This means $\Gamma = x{:}\mathsf{Arr},\, y{:}\mathsf{Arr}$. For simplicity, we treat $\mathtt{cursor}$ as a language primitive. The coeffect annotations will be $(0, 0)$ for x and $(-1, 1)$ for y, denoting that we access only the current value in x, but we need access to both left and right neighbours in the y array. In order to unify the flat and structural notions, we attach this information as a *vector* of annotations associated with a *vector* of variable and write: $x : \mathsf{Arr},\ y : \mathsf{Arr} @ \langle (0,0), (-1,1) \rangle$. The unification is discussed in Chapter **??**.

In structural systems, the splitting of context is determined by the name (variable) binding. For example, consider a function that takes y and contains the above body: $\lambda y.x[\mathtt{cursor}] = y[\mathtt{cursor} - 1] + y[\mathtt{cursor} + 1]$. Here, the declaration site contains x and needs to provide access at least within a range $(0, 0)$. The call site provides a value for y, which needs to be accessible at least within $(-1, 1)$. In this way, structural coeffects remove the non-determinism arising from the splitting of requirements in flat coeffect systems.

Before looking at concrete flat and structural systems, we briefly overview some notation used in this thesis. Structural coeffects keep annotations as *vectors* and use a number of operations related to scalars and vectors.

1.1.3 *Scalars and vectors*

The λ-calculus is asymmetric. It maps a context with *multiple* variables to a *single* result. An expression with n free variables of types τ_i can be modelled by a function $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ with a product on the left, but a single value on the right. In both effect systems and coeffect systems, we *write* the annotation as part of the function arrow. However, in the underlying categorical model, effects are attached to the result τ, while coeffects are attached to the context $\tau_1 \times \ldots \times \tau_n$.

Structural coeffects have one coeffect annotation per each variable. Thus, the annotation consists of multiple values – one belonging to each variable. To distinguish between the overall annotation and individual (per-variable) annotations, we call the overall coeffect a *vector* consisting of *scalar* coeffects. This asymmetry also explains why coeffect systems are not trivially dual to effect systems.

It is useful to clarify how vectors are used in this thesis. Suppose we have a set $\mathcal{C}$ of *scalars* ranged over by r, s, t. A vector R over $\mathcal{C}$ is a tuple $\langle r_1, \ldots, r_n \rangle$ of scalars. We use bold face letters like $\mathbf{r}, \mathbf{s}, \mathbf{t}$ for vectors and normal face r, s, t for scalars[3]. We also say that a *shape* of a vector $\mathit{len}(\mathbf{r})$ (or more generally any container) determines the set of *positions* in a vector. So, a vector of a shape (length) n has positions $\{1, 2, \ldots, n\}$. We discuss containers and shapes further in Chapter **??** and also discuss how our use relates to containers of Abbott, Altenkirch and Ghani [3].

Just as in the usual pointwise multiplication of a vector by a scalar, we lift any binary operation on scalars into a scalar-vector one. For a binary operation on scalars $\circ : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$, we define $s \circ \mathbf{r} = \langle s \circ r_1, \ldots, s \circ r_n \rangle$. Relations on scalars can be also lifted to vectors. Given two vectors $\mathbf{r}, \mathbf{s}$ of the same shape with positions $\{1, \ldots, n\}$ and a relation $\propto \subseteq \mathcal{C} \times \mathcal{C}$ we define $\mathbf{r} \propto \mathbf{s} \Leftrightarrow (r_1 \propto s_1) \wedge \ldots \wedge (r_n \propto s_n)$ Finally, we often concatenate vectors – for example, when joining two variable contexts. Given vectors $\mathbf{r}, \mathbf{s}$ with (possibly different) shapes $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, the associative operation for concatenation $\times$ is defined as $\mathbf{r} \times \mathbf{s} = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$.

We note that an environment Γ containing n uniquely named, typed variables is also a vector, but we continue to write ',' for the product, so $\Gamma_1, x{:}\tau, \Gamma_2$ should be seen as $\Gamma_1 \times \langle x{:}\tau \rangle \times \Gamma_2$.

[3] For better readability, the thesis also distinguishes different structures using colours. However ignoring the colour does not introduce any ambiguity.

1.2 FLAT COEFFECT SYSTEMS

In flat coeffect systems, the additional contextual information are independent of lexically scoped variables. As such, flat coeffects capture properties where the execution environment provides some additional data, resources or information about the execution context.

As mentioned in the introduction, coeffect systems in this chapter may appear as a disconnected set of examples at first. Indeed, this section covers a diverse set of calculi including Haskell's implicit parameters (Section 1.2.1), distributed computing and cross-compilation (Section 1.2.2), liveness analysis (Section 1.2.3) and data-flow (Section 1.2.4).

For three of the examples, we present a type system and a simple semantics. Although the examples are not new, our novel presentation of the systems (and the fact that they appear side-by-side) makes it possible to see that they share a common structure. The structure is captured by a unified *flat coeffect calculus* in Chapter 2.

1.2.1 *Implicit parameters and type classes*

Haskell provides two examples of flat coeffects – type class constraints and implicit parameter constraints [126, 61]. Both of the features introduce additional *constraints* on the context requiring that the environment provides certain operations for a type (type classes) or that it provides values for named implicit parameters. In the Haskell type system, constraints C are attached to the types of top-level declarations, such as let-bound functions. The Haskell notation $\Gamma \vdash e : C \Rightarrow \tau$ corresponds to our notation $\Gamma @ C \vdash e : \tau$.

In this section, we present a type system for implicit parameters in terms of the coeffect typing judgement. We briefly consider type classes, but do not give a full type system.

IMPLICIT PARAMETERS. Implicit parameters are a special kind of variables that support dynamic scoping. They make it possible to parameterise a computation (involving a long chain of function calls) without passing parameters explicitly as additional arguments of all involved functions.

The dynamic scoping means that if a function uses a parameter ?param then the caller of the function must set a value of ?param before calling the function. However, implicit parameters also support lexical scoping. If the parameter ?param is available in the lexical scope where a function is defined, then the function will not require a value from the caller.

A simple language with implicit parameters has an expression ?param to read a parameter and an expression[4] `letdyn` ?param $= e_1$ `in` e_2 that sets a parameter ?param to the value of e_1 and evaluates e_2 in a context containing ?param.

The fact that implicit parameters support both lexical and dynamic scoping becomes interesting when we consider nested functions. The following function does some pre-processing and then returns a function that builds a formatted string based on two implicit parameters ?width and ?size:

```
let format = λstr →
  let lines = formatLines str ?width in
  (λrest → append lines rest ?width ?size)
```

[4] Haskell uses `let` ?p $= e_1$ `in` e_2, but we use a different keyword to avoid confusion.

$$(\textit{var})\quad \frac{x : \tau \in \Gamma}{\Gamma @ \emptyset \vdash x : \tau}$$

$$(\textit{param})\quad \frac{}{\Gamma @ \{?\mathsf{param} : \tau\} \vdash ?\mathsf{param} : \tau}$$

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

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

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

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

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

Figure 2: Coeffect rules for tracking implicit parameters

The body of the outer function accesses the parameter ?width, so it certainly requires a context {?width : int}. The nested function (returned as a result) uses the parameter ?width, but in addition also uses ?size. Where should the parameters used by the nested function come from?

To keep examples in this chapter uniform, we do not use the Haskell notation and instead write $\tau_1 \xrightarrow{r} \tau_2$ for a function that requires implicit parameters specified by r. In a purely dynamically scoped system, they would have to be defined when the user invokes the nested function. However, implicit parameters behave as a combination of lexical and dynamic scoping. This means that the nested function can capture the value of ?width and require just ?size. The following shows the two options:

$$\mathsf{string} \xrightarrow{\{?\mathsf{width}:\mathsf{int}\}} (\mathsf{string} \xrightarrow{\{?\mathsf{width}:\mathsf{int}, ?\mathsf{size}:\mathsf{int}\}} \mathsf{string}) \qquad (\textit{dynamic})$$

$$\mathsf{string} \xrightarrow{\{?\mathsf{width}:\mathsf{int}\}} (\mathsf{string} \xrightarrow{\{?\mathsf{size}:\mathsf{int}\}} \mathsf{string}) \qquad (\textit{mixed})$$

This is not a complete list of possible typings, but it demonstrates the options. The (*dynamic*) case requires the parameter ?width twice (this may be confusing when the caller provides two different values). In the (*mixed*) case, the nested function captures the ?width parameter available from the declaration site. Using the latter typing, the function can be called as follows:

```
let formatHello = ( letdyn ?width = 5 in format "Hello")
in ( letdyn ?size = 10 in formatHello "world" )
```

For different typings of format, different ways of calling it are valid. This illustrates the point made in Section 1.1.1 – flat coeffect systems may introduce certain non-determinism in the typing. The following section shows how this looks in the type system for implicit parameters.

TYPE SYSTEM. Figure 2 shows a type system that tracks the set of expression's implicit parameters. The type system uses judgements of the form $\Gamma @ r \vdash e : \tau$ meaning that an expression e has a type τ in a free-variable context Γ with a set of implicit parameters specified by r. The annotations

r, s, t are finite partial functions mapping implicit parameter names to types, i. e. $r, s, t \subseteq \mathsf{Names} \mapsto \mathsf{Types}$. The expressions include ?param to read implicit parameter and letdyn to bind an implicit parameter. The types are standard, but functions are annotated with the set of implicit parameters that must be available on the call site, i. e. $\tau_1 \xrightarrow{s} \tau_2$.

Accessing an ordinary variable (*var*) does not require any implicit parameters. The rule that introduces primitive context requirements is (*param*). Accessing a parameter ?param of type τ requires it to be available in the context. The context may provide more (unused) implicit parameters thanks to the (*sub*) rule.

When we read the rules from the top to the bottom, application (*app*) and let binding (*let*) simply union the context requirements of the sub-expressions. However, lambda abstraction (*abs*) is where the example differs from effect systems. The implicit parameters required by the body $r \cup s$ can be freely split between the declaration site ($\Gamma @ r$) and the call site ($\tau_1 \xrightarrow{s} \tau_2$). Finally, (*letdyn*) defines an implicit parameter and removes it from the set of requirements.

The union operation $\cup$ is not a disjoint union, which means that the values for implicit parameters can also be provided by both sites. For example, consider a function with a body $?a + ?b$. Assuming that the function takes and returns int, the following list shows 4 out of 9 possible valid typing. Full typing derivations can be found in Appendix **??**:

$$\begin{array}{rcccll}
\Gamma @ \{?a : \mathsf{int}\} & \vdash & \lambda x.?a + ?b & : & \mathsf{int} \xrightarrow{\{?b:\mathsf{int}\}} \mathsf{int} & (1)\\
\Gamma @ \{?b : \mathsf{int}\} & \vdash & \lambda x.?a + ?b & : & \mathsf{int} \xrightarrow{\{?a:\mathsf{int}\}} \mathsf{int} & (2)\\
\Gamma @ \{?a : \mathsf{int}\} & \vdash & \lambda x.?a + ?b & : & \mathsf{int} \xrightarrow{\{?a:\mathsf{int},?b:\mathsf{int}\}} \mathsf{int} & (3)\\
\Gamma @ \emptyset & \vdash & \lambda x.?a + ?b & : & \mathsf{int} \xrightarrow{\{?a:\mathsf{int},?b:\mathsf{int}\}} \mathsf{int} & (4)
\end{array}$$

The first two examples demonstrate that the system does not have the principal typing property. Both (1) and (2) are valid typings and they may both be desirable in certain contexts where the function is used.

The next typing derivation (3) requires the parameter ?a from both the declaration site and the call site. This means that, at runtime, two values will be available. Our semantics for the system describes *dynamic rebinding*, meaning that when the caller provides a value for a parameter that is already specified by the declaration site, the new value hides the old one. This means that only the value from the call site is actually used. This (4) gives a more precise typing for this situation.

SEMANTICS. Implicit parameters can be implemented by passing around a hidden dictionary that provides values to the implicit parameters. Accessing a parameter then becomes a lookup in the dictionary and the new letdyn construct extends the dictionary. To elucidate how such hidden dictionaries are propagated through the program when using lambda abstractions and applications, we present a simple semantics for implicit parameters. The goal here is not to prove properties of the language, but simply to provide a better explanation. A detailed semantics in terms of indexed comonads is shown in Chapter 2.

For simplicity, we assume that all implicit parameters have a type σ. In that setting, coeffect annotations r are just sets of names, i. e. $r, s, t \subseteq \mathsf{Names}$. Given an expression e of type τ that requires free variables Γ and implicit parameters r, the semantics is a function that takes a product of variables from Γ together with a dictionary of implicit parameters and returns τ:

The semantics is defined inductively over the typing derivation:

$$[\![\Gamma\,@\,r \vdash x_i : \tau_i]\!] = \lambda((x_1,\ldots,x_n),\_) \to x_i \qquad (\textit{var})$$

$$[\![\Gamma\,@\,r \vdash ?p : \sigma]\!] = \lambda(\_,f) \to f\ ?p \qquad (\textit{param})$$

$$[\![\Gamma\,@\,r \vdash e : \tau]\!] = \lambda(x,f) \to [\![\Gamma\,@\,r' \vdash e : \tau]\!]\ (x, f|_{r'}) \qquad (\textit{sub})$$

$$\begin{aligned}&[\![\Gamma\,@\,r \vdash \lambda y.e : \tau_1 \xrightarrow{s} \tau_2]\!] = \lambda((x_1,\ldots,x_n),f) \to \\ &\qquad \lambda(y,g) \to [\![\Gamma, y:\tau_1\,@\,r \cup s \vdash e : \tau_2]\!]\ ((x_1,\ldots,x_n,y), f \uplus g)\end{aligned} \qquad (\textit{abs})$$

$$\begin{aligned}&[\![\Gamma\,@\,r \cup s \cup t \vdash e_1\ e_2 : \tau_2]\!] = \lambda(x,f) \to \\ &\qquad \texttt{let}\ g = [\![\Gamma\,@\,r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]\ (x, f|_r) \\ &\qquad \texttt{in}\ g\ ([\![\Gamma\,@\,s \vdash e_2 : \tau_1]\!]\ (x, f|_s), f|_t)\end{aligned} \qquad (\textit{app})$$

$$\begin{aligned}&[\![\Gamma\,@\,r \cup (s \setminus \{?p:\tau_1\}) \vdash \texttt{letdyn}\ ?p = e_1\ \texttt{in}\ e_2 : \tau_2]\!] = \lambda(x,f) \to \\ &\qquad \texttt{let}\ v = [\![\Gamma\,@\,r \vdash e_1 : \tau_1]\!]\ (x, f|_r) \\ &\qquad \texttt{in}\ [\![\Gamma\,@\,s \vdash e_2 : \tau_2]\!]\ (x, f|_{s\setminus\{?p:\tau_1\}} \uplus \{?p \mapsto v\})\end{aligned} \qquad (\textit{letdyn})$$

Here $\uplus$ and $f|_r$ are auxiliary definitions:

$$\begin{aligned} f|_r &= \{(p,v) \mid (p,v) \in f,\ p \in r\} \\ f \uplus g &= f|_{\mathit{dom}(f)\setminus\mathit{dom}(g)} \cup g \end{aligned}$$

Figure 3: Semantics of a language with implicit parameters

$$[\![x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n\,@\,r \vdash e : \tau]\!]\ :\ (\tau_1 \times \ldots \times \tau_n) \times (r \to \sigma) \to \tau$$

The dictionary is represented as a function from r to σ. This means that it provides a σ value for all implicit parameters that are required according to the typing. Note that the domain of the function is not the set of all possible implicit parameter names, but only the finite subset of names that are required according to the typing.

The dictionary is also attached to the inputs of all functions. That is, a function $\tau_1 \xrightarrow{s} \tau_2$ is interpreted by a function that takes τ_1 together with a dictionary that defines values for implicit parameters in s:

$$[\![\tau_1 \xrightarrow{s} \tau_2]\!] = \tau_1 \times (s \to \sigma) \to \tau_2$$

The definition of the semantics is shown in Figure 3. The (*var*) and (*param*) rules are simple – they project the appropriate variable and implicit parameter, respectively.

When an expression requires implicit parameters r, the semantics always provides a dictionary defined *exactly* on r. To achieve this, the (*sub*) rule restricts the function to r' (which is valid because $r' \subseteq r$).

The most interesting rules are (*abs*) and (*app*). In abstraction, we get two dictionaries f and g (from the declaration site and call site, respectively), which are combined and passed to the body of the function. The semantics prefers values from the call site, which is captured by the $\uplus$ operation. In application, we first evaluate the expression e_1, then e_2 and finally call the returned function. The three calls use (possibly overlapping) restrictions of the dictionary as required by the static types.

Finally, the (*letdyn*) rule specifies the semantics of the `letdyn` construct, which assigns a value to an implicit parameter. This is similar to (*app*), because it needs to evaluate the sub-expression first. After evaluating e_1, the result is added to the dictionary using $\uplus$. The semantics of ordinary let binding is omitted, because let binding can be treated as a syntactic sugar for $(\lambda x.e_2)\ e_1$.

Without providing a proof here, we note that the semantics is sounds with respect to the type system – when evaluating an expression, it provides it with a dictionary that is guaranteed to contain values for all implicit parameters that may be accessed. This can be easily checked by examining the semantic rules (and noting that the restriction and union always provide the expected set of parameters).

MONADIC SEMANTICS. Implicit parameters are related to the *reader monad*. The type $\tau_1 \times (r \to \sigma) \to \tau_2$ is equivalent to $\tau_1 \to ((r \to \sigma) \to \tau_2)$ through currying. Thus, we can express the function as $\tau_1 \to M\tau_2$ for $M\tau = (r \to \sigma) \to \tau$. Indeed, the reader monad can be used to model dynamic scoping. However, there is an important distinction from implicit parameters. The usual monadic semantics models fully dynamic scoping, while implicit parameters combine lexical and dynamic scoping.

When using the usual monadic semantics based on the reader monad, the semantics of the (*abs*) rule would be modified as follows:

$$\begin{aligned}&[\![\Gamma @ \emptyset \vdash \lambda y.e : \tau_1 \xrightarrow{s} \tau_2]\!] = \lambda((x_1, \ldots, x_n), \_) \to \\ &\qquad \lambda(y, g) \to [\![\Gamma, y : \tau_1 @ s \vdash e : \tau_2]\!]\ ((x_1, \ldots, x_n, y), g)\end{aligned}$$

Note that the declaration site dictionary is ignored and the body is called with only the dictionary provided by the call site. This is a consequence of the fact that monadic functions are always pure values created using monadic *unit*, which turns a function $\tau_1 \to M^r\tau_2$ into a monadic computation with no side-effects $M^{\emptyset}\tau_1 \to M^r\tau_2$.

As we discuss later in Section **??**, the reader monad can be extended to model rebinding. However, later examples in this chapter, such as liveness in Section 1.2.3 show that other context-aware computations cannot be captured by *any* monad.

TYPE CLASSES. Another type of constraints in Haskell that is closely related to implicit parameters are *type class* constraints [126]. They provide a principled form of ad-hoc polymorphism (overloading). When code uses an overloaded operation (e. g. comparison or numeric operators) a constraint is placed on the context in which the operation is used. For example:

$$\begin{aligned}&\mathsf{twoTimes}\ ::\ \mathsf{Num}\ \alpha \Rightarrow \alpha \to \alpha \\ &\mathsf{twoTimes}\ x = x + x\end{aligned}$$

The constraint $\mathsf{Num}\ \alpha$ on the function type arises from the use of the $+$ operator. Similarly to implicit parameters, type classes can be implemented using a hidden dictionary. In the above case, the function twoTimes takes an additional dictionary that provides an operation $+$ of type $\alpha \times \alpha \to \alpha$.

Type classes could be modelled as a coeffect system. The type system would annotate the context with a set of required type classes. The typing of the body of twoTimes would look as follows:

$$x : \alpha @ \{\mathsf{Num}_\alpha\} \vdash x + x : \alpha$$

Similarly, the semantics of a language with type class constraints can be defined in a way similar to implicit parameters. The interpretation of the body is a function that takes α together with a hidden dictionary of operations: $\alpha \times \mathsf{Num}_\alpha \to \alpha$.

Type classes and implicit parameters show two important points about flat coeffect systems. First, the context requirements are associated with some *scope*, such as the body of a function. Second, they are associated with the input. To call a function that takes an implicit parameter or has a type-class constraint, the caller needs to pass a (hidden) parameter together with the function inputs.

SUMMARY. Implicit parameters are the simplest example of a system where function abstraction does not “delay” all impurities of the body. Here, the term “delay” refers to the fact that some implicit parameters may be captured (from the declaration site) at the time when the function is defined, but before it is executed. As discussed in Section 1.1.1, this is the defining feature of *coeffect* systems.

In this section, we have seen how this affects both the type system and the semantics of the language. In the type system, the (*abs*) rule places context-requirements on both the declaration site and the call site. For implicit parameters, this rule introduces non-determinism in the type-inference, because the parameters can be split arbitrarily. However, as we show in the next section, this is not always the case. Semantically, lambda abstraction *merges* two parts of context (implicit parameter dictionaries) that are provided by the call site and declaration site.

1.2.2 *Distributed computing*

Distributed programming was used as one of the motivating examples for coeffects in Chapter **??**. This section explores the use case. We look at rebindable resources and cross-compilation. The structure of both is very similar to implicit parameters and type class constraints, but they demonstrate that there is a broader use for coeffect systems.

REBINDABLE RESOURCES. The need for parameters that support dynamic scoping also arises in distributed computing. To quote an example discussed by Bierman et al. [11]: *“Dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources.”*

Rebindable parameters are identifiers that refer to some specific resource. When a function value is marshalled and sent to another machine, rebindable resources can be handled in two ways. First, if the resource is available on the target machine, the parameter may be *rebound* to the resource on the new machine. This is captured by the dynamic scoping rule. Second, if the resource is not available on the target machine, the resource is either marshalled or a *remote reference* is created. This is captured by the lexical scoping rule.

A practical language that supports rebindable resources is for example Acute [99]. In the following example, we use the construct access Res to represent access to a rebindable resource named Res. The following simple function accesses a database together with a current date; then it filters from the database based on the date:

```
let recentNews = λ() →
  let db = access News in
  query db "SELECT * WHERE Date > %1" (access Clock)
```

```
// Checks that input is valid; can run on both server and client
let validateInput = λname →
  name ≠ "" && forall isLetter name

// Searches database for a product; must run on the server-side
let retrieveProduct = λname →
  if validateInput name then Some(queryProductDb name)
  else None

// Client-side function to show price or error (for invalid inputs)
let showPrice = λname →
  if validateInput name then
    match (remote retrieveProduct()) with
    | Some p → showPrice (getPrice p)
    | None  → showError "Invalid input on the server"
  else showError "Invalid input on the client"
```

Figure 4: Sample client-server application with input validation

When recentNews is created on the server and sent to the client, a remote reference to the database (available only on the server) must be captured. If the client device supports a clock, then Clock can be locally *rebound*, e. g., to accommodate time-zone changes. Otherwise, the date and time needs to be obtained from the server too.

The type system and semantics for rebindable resources are essentially the same as those for implicit parameters. Primitive requirements are introduced by the access keyword. Lambda abstraction splits the requirements between declaration site (capturing remote reference) and call site (representing rebinding). For this reason, we do not discuss the system in details and instead look at other uses.

CROSS-COMPILATION. A related issue with distributed programming is the need to target increasing number of diverse platforms. Modern applications often need to run on multiple platforms (iOS, Android, Windows or as JavaScript) or multiple versions of the same platform. Many programming languages are capable of targeting multiple different platforms. For example, functional languages that can be compiled to native code and JavaScript include, among others, F#, Haskell and OCaml [120].

Links [25], F# WebTools and WebSharper [105, 81], ML5 and QWeSST [69, 95] and Hop [62] go further and allow including code for multiple distinct platforms in a single source file. A single program is then automatically split and compiled to multiple target runtimes. This posses additional challenges – it is necessary to check where each part of the program can run and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

We demonstrate the problem by looking at input validation. In applications that communicate over an unsecured HTTP channel, user input needs to be validated interactively on the client-side (to provide immediate response) and then again on the server-side (to guarantee safety).

Consider the client-server example in Figure 4. The retrieveProduct function represents the server-side, while showPrice is called on the client-side and performs a remote call to the server-side function (how this is implea.) Set-based type system for cross-compilation, inspired by Links [25]

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

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

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

b.) Version-based type system, inspired by Android API level [30]

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

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

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

Figure 5: Two variants of coeffect typing rules for cross-compilation

mented is not our concern here). To ensure that the input is valid *both* functions call validateInput – however, this is fine, because validateInput uses only basic functions and language features that can be cross-compiled to both client-side and server-side.

In Links [25], functions can be annotated as client-side, server-side and database-side. F# WebTools [81] supports cross-compiled (mixed-side) functions similar to validateInput. However, these are single-purpose language features and they are not extensible. A practical implementation needs to be able to capture multiple different patterns – sets of environments (client, server, mobile) for distributed computing, but also Android API level [30] to cross-compile for multiple versions of the same platform.

TYPE SYSTEMS. Cross-compilation may seem similar to resource tracking (and thus to the tracking of implicit parameters), but it actually demonstrates a couple of new ideas that are important for flat coeffect systems. Unlike with implicit parameters, we will not present a specific existing system in this section – instead we briefly look at two examples that let us explore the range of possibilities.

In the first system, shown in Figure 5 (a), the coeffect annotations are sets of execution environments, i. e. $r, s, t \subseteq \{\mathsf{client}, \mathsf{server}, \mathsf{database}\}$. Sub-coeffecting (*sub*) lets us ignore some of the supported execution environments; application (*app*) can be only executed in the *intersection* of the environments required by the two expressions and the function value.

Sub-coeffecting and application are trivially dual to the rules for implicit parameters. We just track supported environments using intersection as opposed to tracking required parameters using union. However, this symmetry does not hold for lambda abstraction (*abs*), which still uses *union*. This models the case when there are two ways of executing the function:

- The function is represented as executable code for an call site environment and is executed there, possibly after it is marshalled and transferred to another machine.
- The function body is compiled for the declaration site environment; the value that is returned is a remote reference to the code and function calls are performed as remote invocations.

This example ignores important considerations – for example, it is likely desirable to make this difference explicit (e. g. using explicit wrapping of unevaluated expressions) and the implementation also needs to be clarified. For a system that does this, see e. g. ML5 [69]). The key point of our brief example is that the algebraic structure of coeffect annotations may be more complex and use, for example, $\cap$ for application and $\cup$ for abstraction.

The second system, shown in Figure 5 (b) is inspired by the API level requirements in Android. Coeffect annotations are simply numbers representing the level ($r, s, t \in \mathbb{N}$). Levels are ordered increasingly, so we can always require higher level (*sub*). The requirement on function application (*app*) is the highest level of the levels required by the sub-expressions and the function. The system uses yet another variant of lambda abstraction (*abs*) – the requirements of the body are duplicated and placed on *both* the declaration site and the call site.

The ML5 language [69] mentioned above served as an inspiration for our example. It tracks execution environments using modalities of modal S4 to represent the environment – this approach is similar to coeffects, both from the practical perspective, but also through deeper theoretical links. However, it is based on the *meta-language* style of embedding modalities rather than on the *language-semantics* style (see Section **??**). We return to this topic in Section **??**.

1.2.3 *Liveness analysis*

Our next example shows the idea of coeffects from a different perspective. Rather than keeping additional information independent of the variable context, we track properties about how variables are used. Nevertheless, we still look at the left-hand side of $\vdash$ and the structure of the typing rules and semantics will be very similar.

Live variable analysis (LVA) [6] is a standard technique in compiler theory. It detects whether a free variable of an expression may be used by a program during its evaluation (it is *live*) or whether it is definitely not needed (it is *dead*). As an optimization, compiler can remove bindings to dead variables as they are never accessed. Wadler [124] describes the property of a variable that is dead as the *absence* of a variable.

FLAT LIVENESS ANALYSIS. In this section, we discuss a restricted form of liveness analysis. We do not track liveness of *individual* variables, but of the *entire* variable context. This is not practically useful, but it provides an interesting insight into how flat coeffects work. A per-variable liveness analysis can be captured using structural coeffects and is discussed in Section 1.3.1. Consider the following two examples:

```
let constant42 = λx → 42
let constant = λvalue → λx → value
```

a.) The operations of a two-point lattice $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ where $\mathsf{D} \sqsubseteq \mathsf{L}$ are defined as:

$$\begin{array}{lclclclclclcl}
\mathsf{L} \sqcup \mathsf{L} & = & \mathsf{L} & \quad & \mathsf{L} \sqcup \mathsf{D} & = & \mathsf{D} & \quad & \mathsf{L} \sqcap \mathsf{L} & = & \mathsf{L} & \quad & \mathsf{L} \sqcap \mathsf{D} = \mathsf{L} \\
\mathsf{D} \sqcup \mathsf{L} & = & \mathsf{D} & \quad & \mathsf{D} \sqcup \mathsf{D} & = & \mathsf{D} & \quad & \mathsf{D} \sqcap \mathsf{L} & = & \mathsf{L} & \quad & \mathsf{D} \sqcap \mathsf{D} = \mathsf{D}
\end{array}$$

b.) Sequential composition of (semantic) functions composes annotations using $\sqcup$:

$$\begin{array}{llll}
f : \tau_1 \xrightarrow{r} \tau_2 & g : \tau_2 \xrightarrow{s} \tau_3 & g \circ f : \tau_1 \xrightarrow{r \sqcup s} \tau_3 & \\
 & & & \\
f : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 & g : \tau_2 \xrightarrow{\mathsf{L}} \tau_3 & g \circ f : \tau_1 \xrightarrow{\mathsf{L}} \tau_3 & (1) \\
f : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 & g : \tau_2 \xrightarrow{\mathsf{L}} \tau_3 & g \circ f : \tau_1 \xrightarrow{\mathsf{D}} \tau_3 & (2) \\
f : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 & g : \tau_2 \xrightarrow{\mathsf{D}} \tau_3 & g \circ f : \tau_1 \xrightarrow{\mathsf{D}} \tau_3 & (3) \\
f : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 & g : \tau_2 \xrightarrow{\mathsf{D}} \tau_3 & g \circ f : \tau_1 \xrightarrow{\mathsf{D}} \tau_3 & (4)
\end{array}$$

c.) Pointwise composition of (semantic) functions composes annotations using $\sqcap$:

$$\begin{array}{llll}
f : \tau_1 \xrightarrow{r} \tau_2 & h : \tau_1 \xrightarrow{s} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{r \sqcap s} \tau_2 \times \tau_3 & \\
 & & & \\
f : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 & h : \tau_1 \xrightarrow{\mathsf{D}} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 \times \tau_3 & (1) \\
f : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 & h : \tau_1 \xrightarrow{\mathsf{L}} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \times \tau_3 & (2) \\
f : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 & h : \tau_1 \xrightarrow{\mathsf{D}} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \times \tau_3 & (3) \\
f : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 & h : \tau_1 \xrightarrow{\mathsf{L}} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \times \tau_3 & (4)
\end{array}$$

Figure 6: Liveness annotations with sequential and pointwise composition

The body of the first function is just a constant 42 and so the context of the body is marked as *dead*. The parameter (call site) of the function is not used and can also be marked as dead. Similarly, no variables from the declaration site are used and so they are also marked as dead.

In contrast, the body of the second function accesses a variable value and so the body of the function is marked as *live*. In the flat system, we do not track *which* variable was used and so we have to mark both the call site and the declaration site as live (this will be refined in a structural version of the system).

FORWARD VS. BACKWARD & MAY VS. MUST. Static analyses can be classified as either *forward* or *backward* (depending on how they propagate information) and as either *must* or *may* (depending on what properties they guarantee). Liveness is a *backward* analysis – the requirements are propagated from variables to their declarations. The distinction between *must* and *may* is apparent when we look at an example with conditionals:

```
let defaultArg  = λcond → λinput →
  if cond then 42 else input
```

Liveness analysis is a *may* analysis meaning that it marks variable as live when it *may* be used and as dead if it is *definitely* not used. This means that the variable input is *live* in the example above. A *must* analysis would mark the variable only if it was used in both of the branches (this is sometimes called *neededness* or *very busy* variable/expression).

The distinction between *may* and *must* analyses demonstrates the importance of interaction between contextual properties and certain language constructs such as conditionals.

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

$$(\mathit{const})\ \frac{c : \tau \in \Delta}{\Gamma @ \mathsf{D} \vdash c : \tau}$$

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

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

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

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

Figure 7: Coeffect rules for tracking whole-context liveness

TYPE SYSTEM. A type system that captures whole-context liveness annotates the context with value of a two-point lattice $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$. The annotation L marks the context as *live* and D stands for a *dead* context. Figure 6 (a) defines the ordering $\sqsubseteq$, meet $\sqcup$ and join operations $\sqcap$ of the lattice.

The typing rules for tracking whole-context liveness are shown in Figure 7. The language now includes constants $c : \tau \in \Delta$. Accessing a constant (*const*) annotates the context as dead using D. This contrasts with variable access (*var*), which marks the context as live using L. A dead context (definitely not needed) can be treated as live context using the (*sub*) rule. This captures the *may* nature of the analysis.

The (*app*) rule is best understood by discussing its semantics. The semantics uses *sequential composition* to compose the semantics of e_2 with the function obtained as the result of e_1. However, we need more than just sequential composition. The same input context is passed to the expression e_1 (in order to get the function value) and to the sequentially composed function (to evaluate e_2 followed by the function call). This is captured by *pointwise composition*.

Consider first *sequential composition* of (semantic) functions f, g annotated with r, s. The composed function $g \circ f$ is annotated with $r \sqcup s$ as shown in Figure 6 (b). The argument of the function $g \circ f$ is live only when the arguments of both f and g are live (1). When the argument of f is dead, but g requires τ_2 (2), we can evaluate f without any input and obtain τ_2, which is then passed to g. When g does not require its argument (3, 4), we can just evaluate g, without evaluating f. Here, the semantics *implements* the dead code elimination optimization.

Secondly, a *pointwise composition* passes the same argument to f and h. The parameter is live if either the parameter of f or h is live. The pointwise composition is written as $\langle f, h\rangle$ and it combines annotations using $\sqcap$ as shown in Figure 6 (c). Here, the argument is not needed only when both f and h do not need it (1). In all other cases, the parameter is needed and is then used either once (2, 3) or twice (4). The rule for function application (*app*) combines the two operations. The context Γ is live if it is needed by e_1 (which always needs to be evaluated) *or* when it is needed by the function value *and* by e_2.

The (*abs*) rule duplicates the annotation of the body, similarly to the cross-compilation example in Figure 5. When the body accesses any variables, it requires both the argument and the variables from declaration site. When it does not use any variables, it marks both as dead. Finally, the (*let*) rule annotates the composed expression with the liveness of the expression e_2 – if the context of e_2 is live, then it also requires variables from Γ; if it is dead, then it does not require Γ or x. As further discussed later in Section ?, the (*let*) rule is again just a syntactic sugar for $(\lambda x.e_2)\ e_1$. This follows from the simple observation that $r \sqcup (s \sqcap r) = r$.

EXAMPLES. Before looking at the semantics, we consider a number of simple examples to demonstrate the key aspects of the system. Full typing derivations are shown in Appendix **??**:

$$
\begin{aligned}
&(\lambda x.42)\ y && (1)\\
&\mathsf{twoTimes}\ 42 && (2)\\
&(\lambda x.x)\ 42 && (3)
\end{aligned}
$$

In the first case, the context is dead. In (1), the function's parameter is dead and so the overall context is dead, even though the argument uses a variable y – the semantics evaluates the function without passing it an actual argument. In the second case (2), the function is a variable that needs to be obtained and so the context is live. In the last case (3), the function accesses a variable and so its declaration site is marked as requiring the context (*abs*). This is where structural coeffect analysis would be more precise – the system shown here cannot capture the fact that x is a bound variable.

SEMANTICS. As showed in the examples, the type system for the liveness coeffect calculus marks the context of an expression $(\lambda x.42)\ y$ as dead. This means that the semantics of the above expression must not evaluate the argument y. In other words, the type system is only sound if the semantics includes dead code elimination.

To capture dead code elimination in the semantics, we add a special empty value and pass it as an argument to a function whose argument is not needed, so $(\lambda x.42)$ will be called with an empty value as argument (because it does not need its argument).

We can represent such empty values using the option type (known as Maybe in Haskell). We use the notation $\tau + 1$ to denote option types. Given a context with variables x_i of type τ_i, the semantics is a function taking $(\tau_1 \times \ldots \times \tau_n) + 1$. When the context is live, it will be called with the left value (product of variable assignments); when the context is dead, it will be called with the right value (containing no information).

However, ordinary option type is not sufficient. We need to capture the fact that the representation depends on the annotation – in other words, the type is *indexed* by the coeffect annotation. The indexing is discussed in details in Section **??**. For now, it suffices to define the semantics using two separate rules:

$$
\begin{aligned}
[\![x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n\ @\ \mathsf{L} \vdash e:\tau]\!] \quad &: \quad (\tau_1 \times \ldots \times \tau_n) \to \tau\\
[\![x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n\ @\ \mathsf{D} \vdash e:\tau]\!] \quad &: \quad 1 \to \tau
\end{aligned}
$$

The semantics of functions is defined similarly. When the argument of a function is live, the function takes the input value; when the argument is dead, the semantic function takes a unit as its argument:

$$
\begin{aligned}
&[\![\tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!] = \tau_1 \to \tau_2 \\
&[\![\tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!] = 1 \to \tau_2
\end{aligned}
$$

Unlike with implicit parameters, the coeffect system for liveness tracking cannot be modelled using monads. Any monadic semantics would express functions as $\tau_1 \to \mathsf{M}\,\tau_2$. Unless laziness is already built-in, there is no way to call such function without first obtaining a value τ_1. The above semantics makes this possible by taking a unit 1 when the argument is not live.

In Figure 8, we define the semantics directly. We write $()$ for the only value of type 1. This appears, for example, in (*const*) which takes $()$ as the input and returns a constant using a global dictionary δ. In (*var*), the context is live and so the semantics performs a projection. Sub-coeffecting is captured by two rules. A dead context can be treated as live using (*abs-1*); in other cases, the annotation is not changed (*abs-2*).

Lambda abstraction can be annotated in just two ways. When the body requires context (*abs-1*), the value of a bound variable y is added to the context Γ before passing it to the body. When the body does not require context (*abs-2*), it is called with $()$ as the input.

For application, there are 8 possible combinations of annotations. The semantics of some of them is the same, so we only need to show 3 cases. The rules should be read as ML-style pattern matching, where the last rule handles all cases not covered by the first two. In (*app-1*), we handle the case when the function g does not require its argument – e_2 is not used and instead, the function is called with $()$ as the argument. The case (*app-2*) covers the case when the expression e_1 does not require a context, but e_1 does. Finally, in (*app-3*), the same input (which may be either tuple of variables or unit) is propagated uniformly to both e_1 and e_2.

SUMMARY. Unlike with implicit parameters, lambda abstraction for liveness analysis does not introduce non-determinism. It simply duplicates the context requirements. However, this still matches the property of coeffects that impurities cannot be delayed or thunked and attached just to the function arrow – we place requirements on both call site and declaration site.

The semantics of liveness reveals three interesting properties. Firstly, the coeffect calculus for liveness cannot be modelled as a monadic computation of the form $\tau_1 \to \mathsf{M}\tau_2$. Secondly, the system would not work without the coeffect annotations. The shape of the semantic function depends on the annotation (the input is either 1 or τ) and is *indexed* by the annotation.

Finally, we discussed how the semantics of application arises from *sequential* and *pointwise* composition. This is an important aspect of coeffect systems – categorical semantics typically builds on *sequential* composition, but to model full λ calculus it needs more. For coeffects, we need *pointwise* composition where the same context is shared by multiple sub-expressions.

1.2.4 *Data-flow languages*

We used implicit parameters as our first example, because they show the simplest form of coeffects. Liveness requires a richer coeffect annotation structure, but the flat version is not practical. In this section, we look at a system with a structure similar to liveness that is not a toy example.

$$\llbracket \Gamma @ \mathsf{L} \vdash x_i : \tau_i \rrbracket = \lambda(x_1, \ldots, x_n) \to x_i \qquad (\textit{var})$$

$$\llbracket \Gamma @ \mathsf{D} \vdash c_i : \tau_i \rrbracket = \lambda() \to \delta(c_i) \qquad (\textit{const})$$

$$\llbracket \Gamma @ \mathsf{L} \vdash e : \tau \rrbracket = \lambda x \to \llbracket \Gamma @ \mathsf{D} \vdash e : \tau \rrbracket \; () \qquad (\textit{sub-1})$$

$$\llbracket \Gamma @ \mathsf{r} \vdash e : \tau \rrbracket = \lambda x \to \llbracket \Gamma @ \mathsf{r} \vdash e : \tau \rrbracket \; x \qquad (\textit{sub-2})$$

$$\begin{aligned} &\llbracket \Gamma @ \mathsf{L} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \rrbracket = \lambda(x_1, \ldots, x_n) \to \\ &\quad \lambda y \to \llbracket \Gamma, y : \tau_1 @ \mathsf{L} \vdash e : \tau_2 \rrbracket \; (x_1, \ldots, x_n, y) \end{aligned} \qquad (\textit{abs-1})$$

$$\begin{aligned} &\llbracket \Gamma @ \mathsf{D} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 \rrbracket = \lambda() \to \\ &\quad \lambda() \to \llbracket \Gamma, y : \tau_1 @ \mathsf{D} \vdash e : \tau_2 \rrbracket \; () \end{aligned} \qquad (\textit{abs-2})$$

$$\begin{aligned} &\llbracket \Gamma @ \mathsf{r} \vdash e_1 \; e_2 : \tau_2 \rrbracket = \lambda x \to \\ &\quad \texttt{let} \; g = \llbracket \Gamma @ \mathsf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 \rrbracket \; x \; \texttt{in} \; g \; () \end{aligned} \qquad (\textit{app-1})$$

$$\begin{aligned} &\llbracket \Gamma @ \mathsf{L} \vdash e_1 \; e_2 : \tau_2 \rrbracket = \lambda x \to \\ &\quad \texttt{let} \; g = \llbracket \Gamma @ \mathsf{L} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \rrbracket \; x \; \texttt{in} \; g \; (\llbracket \Gamma @ \mathsf{D} \vdash e_2 : \tau_1 \rrbracket \; ()) \end{aligned} \qquad (\textit{app-2})$$

$$\begin{aligned} &\llbracket \Gamma @ \mathsf{r} \sqcup (\mathsf{s} \sqcap \mathsf{t}) \vdash e_1 \; e_2 : \tau_2 \rrbracket = \lambda x \to \\ &\quad \texttt{let} \; g = \llbracket \Gamma @ \mathsf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{t}} \tau_2 \rrbracket \; x \; \texttt{in} \; g \; (\llbracket \Gamma @ \mathsf{s} \vdash e_2 : \tau_1 \rrbracket \; x) \end{aligned} \qquad (\textit{app-3})$$

Figure 8: Semantics that implements dead code elimination for λ-calculus

The Section **??** briefly demonstrated that we can treat array access as an operation that accesses a context. In case of arrays, the context is neighbourhood of a current location in the array specified by a cursor. In this section, we make the example more concrete, using a simpler and better studied programming model, data-flow languages.

Lucid [123] is a declarative data-flow language designed by Wadge and Ashcroft. In Lucid, variables represent streams and programs are written as transformations over streams. A function application square(x) represents a stream of squares calculated from the stream of values x.

The data-flow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [9] build on the data-flow paradigm. The following example is inspired by the Lustre [43] language and implements a program to count the number of edges on a Boolean stream:

```
let edge = false fby (input && not (prev input))

let edgeCount =
  0 fby ( if edge then 1 + (prev edgeCount)
          else prev edgeCount )
```

The construct `prev` x returns a stream consisting of previous values of the stream x. The second value of `prev` x is first value of x (and the first value is undefined). The construct y `fby` x returns a stream whose first element is the first element of y and the remaining elements are values of x. Note that in Lucid, the constants such as false and 0 are constant streams.

$$(\mathit{var})\quad \frac{x:\tau \in \Gamma}{\Gamma @ 0 \vdash x:\tau}$$

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

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

$$(\mathit{app})\quad \frac{\Gamma @ m \vdash e_1:\tau_1 \xrightarrow{p} \tau_2 \qquad \Gamma @ n \vdash e_2:\tau_1}{\Gamma @ \max(m, n+p) \vdash e_1\ e_2:\tau_2}$$

$$(\mathit{let})\quad \frac{\Gamma @ m \vdash e_1:\tau_1 \qquad \Gamma, x:\tau_1 @ n \vdash e_2:\tau_2}{\Gamma @ n+m \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2:\tau_2}$$

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

Figure 9: Coeffect rules for tracking context-usage in data-flow language

Formally, the constructs are defined as follows (writing x_n for n-th element of a stream x):

$$(\mathtt{prev}\ x)_n = \begin{cases} \mathit{nil} & \text{if } n = 0 \\ x_{n-1} & \text{if } n > 0 \end{cases} \qquad (y\ \mathtt{fby}\ x)_n = \begin{cases} y_0 & \text{if } n = 0 \\ x_n & \text{if } n > 0 \end{cases}$$

When reading data-flow programs, we do not need to think about variables in terms of streams – we can see them as simple values. Most of the operations perform calculation just on the *current* value of the stream. However, the operation `fby` and `prev` are different. They require additional *context* which provides past values of variables (for `prev`) and information about the current location in the stream (for `fby`).

The semantics of Lucid-like languages can be captured using a number of mathematical structures. Wadge [122] originally defined a monadic semantics, while Uustalu and Vene later used comonads [114]. In Chapter 2, we extend the latter approach. The present chapter presents a sketch of a concrete data-flow semantics defined directly on streams.

In the introductory example with array access patterns, we used coeffects to track the range of values accessed. In this section, we look at a simpler example – we only consider the `prev` operation and track the maximal number of *past values* needed. This is an important information for efficient implementation of data-flow languages. When we can guarantee that at most x past values are accessed, the values can be stored in a pre-allocated buffer rather than using e. g. on-demand computed lazy streams.

TYPE SYSTEM. We can use a coeffect type system to track the maximal number of accessed past values. Here, the context is annotated with a single integer. The current value is always present, so 0 means that no past values are needed, but the current value is still available. The typing rules of the system are shown in Figure 9.

Variable access (*var*) annotates the context with 0; sub-coeffecting (*sub*) allows us to require more values than is actually needed. Primitive context-requirements are introduced in (*prev*), which increments the number of past values by one. Thus, for example, $\mathtt{prev}\ (\mathtt{prev}\ x)$ requires 2 past values.

The (*app*) rule follows the same intuition as for liveness. It combines *sequential* and *pointwise* composition of semantic functions. In case of dataflow, the operations combine annotations using $+$ and *max* operations:

$$\begin{array}{lll} f : \tau_1 \xrightarrow{m} \tau_2 & g : \tau_2 \xrightarrow{n} \tau_3 & g \circ f : \tau_1 \xrightarrow{m+n} \tau_3 \\ f : \tau_1 \xrightarrow{m} \tau_2 & h : \tau_1 \xrightarrow{n} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{\max(m,s)} \tau_2 \times \tau_3 \end{array}$$

Sequential composition adds the annotations. The function f needs m past values to produce a single τ_2 value. To produce two τ_2 values, we thus need $m+1$ past values of τ_1; to produce three τ_2 values, we need $m+2$ past values of τ_1, and so on. To produce n past values that are required as the input of g, we need $m+n$ past values of type τ_1. The pointwise composition is simpler. It uses the same stream to evaluate functions requiring m and n past values, and so it needs maximum of the two at most.

In summary, function application (*app*) requires maximum of the values needed to evaluate e_1 and the number of values needed to evaluate the argument e_2, sequentially composed with the function.

In function abstraction (*abs*), the requirements of the body are duplicated on the declaration site and the call site as in liveness analysis. If the body requires n past values, it may access n values of any variables – including those available in Γ, as well as the parameter x. Finally, the (*let*) rule simply adds the two requirements. This corresponds to the sequential composition operation, but it is also a rule that we obtain by treating let-binding as a syntactic sugar for $(\lambda x.e_2)\ e_1$.

EXAMPLE. As with the liveness example, the application rule might require more explanation. The following example is somewhat arbitrary, but it demonstrates the rule well. We assume that counter is a stream of positive integers (starting from zero) and tick flips between 0 and 1. The full typing derivation is shown in Appendix **??**:

```
( if (prev tick) = 0
  then (λx → prev x)
  else (λx → x) )      (prev counter)
```

The left-hand side of the application returns a function depending on the *previous* value of tick. The resulting stream of functions flips between a function returning a current value and a function returning the previous value. If the current tick is 0, and the function is applied to a stream $\langle \ldots, 4, 3, 2, 1 \rangle$ (where 1 is the current value), it yields the stream $\langle \ldots, 4, 4, 2, 2 \rangle$.

To obtain the function, we need one past value from the context (for prev tick). The returned function needs either none or one past value (thus a subtyping rule is required to type it as requiring one past value). So, the annotations for (*app*) are $m = 1, p = 1$. The function is called with prev counter as an argument, meaning that the result is either the first or second past element. Given counter $= \langle \ldots, 5, 4, 3, 2, 1 \rangle$, the argument is $\langle \ldots, 5, 4, 3, 2 \rangle$ and so the overall result is a stream $\langle \ldots, 5, 5, 3, 3 \rangle$. From the argument, we get the requirement $n = 1$.

Using the (*app*) rule, we get that the overall number of past elements needed is $max(1, 1+1) = 2$. This should match the intuition about the code – when the first function is applied to the argument, the computation will first access prev tick (using one past value) and then prev (prev counter)) (using two past values).

$$[\![\Gamma @ 0 \vdash x_i : \tau_i]\!] = \lambda \langle (x_0, \ldots, x_n) \rangle \rightarrow x_i \qquad (\mathit{var})$$

$$\begin{aligned}&[\![\Gamma @ n+1 \vdash \texttt{prev}\ e : \tau]\!] = \lambda \langle \mathbf{v}_0, \ldots, \mathbf{v}_{n+1} \rangle \rightarrow \\ &\quad [\![\Gamma @ n \vdash e : \tau]\!]\ \langle \mathbf{v}_1, \ldots, \mathbf{v}_{n+1} \rangle\end{aligned} \qquad (\mathit{prev})$$

$$\begin{aligned}&[\![\Gamma @ n \vdash e : \tau]\!] = \lambda \langle \mathbf{v}_0, \ldots, \mathbf{v}_{n} \rangle \rightarrow \\ &\quad [\![\Gamma @ n' \vdash e : \tau]\!]\ \langle \mathbf{v}_0, \ldots, \mathbf{v}_{n'} \rangle\end{aligned} \qquad (\mathit{sub})$$

$$\begin{aligned}&[\![\Gamma @ n \vdash \lambda y.e : \tau_1 \xrightarrow{n} \tau_2]\!] = \lambda \langle \mathbf{v}_0, \ldots \mathbf{v}_{n} \rangle \rightarrow \\ &\quad \lambda(y, g) \rightarrow [\![\Gamma, y : \tau_1 @ n \vdash e : \tau_2]\!]\ \langle (\mathbf{v}_0, y_0), \ldots, (\mathbf{v}_n, y_n) \rangle\end{aligned} \qquad (\mathit{abs})$$

$$\begin{aligned}&[\![\Gamma @ \max(m, n+p) \vdash e_1\ e_2 : \tau_2]\!] = \lambda(\mathbf{v}_0, \ldots, \mathbf{v}_{\max(m,n+p)}) \rightarrow \\ &\quad \texttt{let}\ g = [\![\Gamma @ m \vdash e_1 : \tau_1 \xrightarrow{p} \tau_2]\!]\ (\mathbf{v}_0, \ldots, \mathbf{v}_m) \\ &\quad \texttt{in}\ g\ (\ [\![\Gamma @ n \vdash e_2 : \tau_1]\!]\ (\mathbf{v}_0, \ldots, \mathbf{v}_n), \ldots, \\ &\qquad\quad [\![\Gamma @ n \vdash e_2 : \tau_1]\!]\ (\mathbf{v}_p, \ldots, \mathbf{v}_{n+p})\ )\end{aligned} \qquad (\mathit{app})$$

Figure 10: Semantics showing how past values are accessed in a data-flow language

SEMANTICS. The sample language discussed in this section is a *causal* data-flow language. This means that a computation can access *past* values of the stream (but not future values). In the semantics, we again need richer structure over the input.

Uustalu and Vene [115] model causal data-flow computations using a non-empty list $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$ over the input. A function $\tau_1 \rightarrow \tau_2$ is thus modelled as $\mathsf{NeList}\ \tau_1 \rightarrow \tau_2$. This model is difficult to implement efficiently, as it creates unbounded lists of past elements.

The coeffect system tracks maximal number of past values and so we can define the semantics using a list of fixed length. As with liveness, this is a data structure *indexed* by the coeffect annotation. We write τ^n for a list containing n elements (which can be also viewed as an n-element product $\tau \times \ldots \times \tau$).

As with the previous examples, our semantics interprets a judgement using a (semantic) function; functions in the language are modelled as functions taking a list of inputs:

$$\begin{aligned}[\![x_1 : \tau_1, \ldots, x_n : \tau_n @ n \vdash e : \tau]\!] \quad &: \quad (\tau_1 \times \ldots \times \tau_n)^{n+1} \rightarrow \tau \\ [\![\tau_1 \xrightarrow{n} \tau_2]\!] \quad &: \quad \tau_1^{n+1} \rightarrow \tau_2\end{aligned}$$

Note that the semantics requires one more value than is the number of past values. This is because the first value is the current value and has to be always available, even when the annotation is zero as in (*var*).

The rules defining the semantics are shown in Figure 10. The semantics of the context is a *list of products*. To make the rules easier to follow, we write $\langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$ for an n-element list containing products. Products that model the entire context such as $\mathbf{v}_1$ are written in bold. When we access individual variables, we write $\mathbf{v} = (x_1, \ldots, x_m)$ where x_i denote individual variables of the context.

In (*var*), the context is a singleton-list containing a product of variables, from which we project the right one. In (*prev*) and (*sub*), we drop some of the elements from the history (from the front and end, respectively) and then evaluate the original expression.

Lambda abstractions (*abs*) receives two lists of the same size – one containing values of the variables (list of products) from the declaration site

$\langle \mathbf{v}_0, \ldots, \mathbf{v}_n \rangle$ and one containing the argument (list of values) provided by the call site $\langle y_0, \ldots, y_n \rangle$. The semantics applies the well-known *zip* operation on the lists and passes the result to the body.

Finally, application (*app*) uses the input context in two ways, which gives rise to the two requirements combined using *max*. First, it evaluates the expression e_1 which is called with the past m values. The resulting function g is then sequentially composed with the semantics of e_2. To call the function, we need to evaluate e_2 repeatedly – namely, $p + 1$ times, which results in the overall requirement for $n + p$ past values.

SUMMARY. Type systems have been used in the context of data-flow languages, for example to check initialization properties [24], but to our knowledge, not for checking the maximal number of required past values. Thus this section serves not just as an example, but also shows how coeffects can lead to novel results.

The most interesting point about the data-flow system is that it is remarkably similar to our earlier liveness example. In the type system, abstraction (*abs*) duplicates the context requirements and application (*abs*) arises from sequential and pointwise composition. We capture this striking similarity in Chapter 2. Before doing that, we look at one more example and then explore the *structural* class of systems.

1.2.5 *Permissions and safe locking*

In the implicit parameters and data-flow examples, the context provides additional resources or values that may be accessed at runtime. However, coeffects can also track *permissions* or *capabilities* to perform some operation. We can invert the intuition behind liveness and use it as a trivial example – when the context is live, it contains a *permission* to access variables. In this section, we briefly consider a system for safe locking of Flanagan and Abadi [34] as one, more advanced example. Calculus of capabilities of Cray et al. [27] is discussed later in Section 1.4.

SAFE LOCKING. The system for safe locking prevents race conditions (by only allowing access to mutable state under a lock) and avoids deadlocks (by imposing strict partial order on locks). The following program uses a mutable state under a lock:

```
newlock l : ρ in
let state = ref_ρ 10 in
sync l (!state)
```

The declaration newlock creates a lock l protecting memory region ρ. We can than allocate mutable variables in that memory region (second line). An access to one or more mutable variables is only allowed in scope that is protected by a lock. This is done using the sync keyword, which locks a lock and evaluates an expression in a context that contains permission to access memory region of the lock (ρ in the above example).

The type system for safe locking associates a list of acquired locks with the context. Interestingly, the original presentation of the system by Flanagan and Abadi [34] uses a coeffect-style judgements of a form $\Gamma; p \vdash e : \tau$ where p is a list of accessible regions (protected by an acquired lock). Using our notation, the rule for sync looks as follows:

$$(\textit{sync})\;\frac{\Gamma@p \vdash e_1 : m \qquad \Gamma@p \cup \{m\} \vdash e_2 : \tau}{\Gamma@p \vdash \texttt{sync}\; e_1\; e_2 : \tau}$$

The rule requires that e_1 yields a value of a singleton type m. The type is added as an indicator of the locked region to the context $p \cup \{m\}$ which is then used to evaluate the expression e_2.

SUMMARY. Despite attaching annotations to the variable context, the system for safe locking uses effect-style lambda abstraction. Lambda abstraction associates all requirements with the call site – a lambda function created under a lock cannot access protected memory available at the time of creation. It will be executed later and can only access the memory available then. This suggests that safe locking is better seen as an effect system.

Another interesting aspect is the extension to avoid deadlocks. In that case, the type system needs to reject programs that acquire locks in an invalid order. One way to model this is to replace $p \cup \{m\}$ with a *partial* operation $p \uplus \{m\}$ which is only defined when the lock m can be added to the set p. Supporting partial operations on coeffect annotations is an interesting future extension for coeffect systems.

1.3 STRUCTURAL COEFFECT SYSTEMS

In structural coeffect systems, the additional information is associated with individual variables. This is very often information about how the variables are used, or, in which contexts they are used. In Chapter **??**, we introduced the idea using an example that tracks array access patterns. Each variable is annotated with a range specifying which elements of the corresponding array may be accessed.

In this section, we look at three examples in detail – we revisit liveness and show a practically useful structural version of the system; we consider an example inspired by linear logic; finally, we revisit data-flow to get a more precise analysis. Although quite different, the common pattern among these three examples is somewhat easier to see, because they all track information about variable usage. We finish the section with a brief outline of several other applications.

1.3.1 *Liveness analysis revisited*

The flat system for liveness analysis presented in Section 1.2.3 is interesting from a theoretical perspective, but it is not practically useful. Here, we revisit the problem and define a structural system that tracks liveness per-variable.

STRUCTURAL LIVENESS. Recall two examples discussed earlier where the flat liveness analysis marked the whole context as (syntactically) live, despite the fact part of it was (semantically) dead:

```
let constant = λy → λx → y
let answer = (λx → x) 42
```

In the first case, the variable x is dead, but was marked as live. In the second example, the declaration site of the answer value is dead, but was marked as live. This is because in both of the expressions, *some* variable is accessed. However, the (*abs*) rule of flat liveness has no way of determining *which* variables are used by the body – and, in particular, whether the accessed variable is the *bound* variable or some of the *free* variables.

As discussed earlier, we can resolve this by attaching a *vector* of liveness annotations to a *vector* of variables. In the first example, the available variables are y and x, so the variable context Γ is a vector $\langle y:\tau, x:\tau\rangle$. Only the variable y is used and so the annotated context is: $y:\tau, x:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$. When writing the contexts, we omit angle brackets around variables, but it should still be viewed as a vector. There are two important points:

- The fact that variables are now a vector means that we cannot freely reorder them. This guarantees that $x:\tau, y:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$ can not be confused with $y:\tau, x:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$. We need to define the type system in a way that is similar to sub-structural systems (discussed in Section **??**) and add explicit rules for manipulating the context.

- We choose to attach a vector of annotations to a vector of variables, rather than attaching individual annotations to individual variables. This lets us unify and combine flat and structural systems as discussed in Section **??**, but the alternative is briefly explored in Section **??**.

TYPE SYSTEM. The structural system for liveness uses the same two-point lattice of annotations $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ that was used by the flat system. We also use the $\sqcup, \sqcap$ and $\sqsubseteq$ operators that are defined in Figure 6.

The rules of the system are split into two groups. Figure 11 (a) shows the standard syntax-driven rules plus sub-coeffecting. In (*var*), the context contains just the single accessed variable, which is annotated as live. Unused variables can be introduced using weakening. A constant (*const*) is accessed in an empty context, which also carries no annotations. The sub-coeffecting rule (*sub*) uses a pointwise extension of the $\sqsubseteq$ relation over two vectors as defined in Section 1.1.3.

In the (*abs*) rule, the variable context of the body $\Gamma, x:\tau_1$ is annotated with a vector $\mathbf{r}\times\langle s\rangle$, where the vector $\mathbf{r}$ corresponds to Γ and the singleton annotation s corresponds to the variable x. Thus, the function is annotated with s. Note that the free-variable context is annotated with vectors, but functions take only a single input and so are annotated with primitive annotations.

The (*app*) rule is similar to function applications in flat systems, but there is an important difference. In structural systems, the two sub-expressions have separate variable contexts Γ_1 and Γ_2. Therefore, the composed expression just concatenates the variables and their corresponding annotations. (We can still use the same variable in both sub-expressions thanks to the structural contraction rule.)

The context Γ_1 is used to evaluate e_1 and is thus annotated with $\mathbf{r}$. The annotation for Γ_2 is more interesting. It is a result of sequential composition of two semantic functions – the first one takes the (multi-variable) context Γ_2 and evaluates e_2; the second takes the result of type τ_1 and passes it to the function $\tau_1 \xrightarrow{t} \tau_2$. The composition is defined as follows:

$$g : \tau_1 \times \ldots \times \tau_n \xrightarrow{\mathbf{s}} \sigma \qquad f : \sigma \xrightarrow{t} \tau \qquad f \circ g : \tau_1 \times \ldots \times \tau_n \xrightarrow{t \sqcup \mathbf{s}} \tau$$

This definition is only for illustration and is revised in Chapter 4. The function g takes a product of multiple variables (and is annotated with a vector). The function f takes just a single value and is annotated with the scalar. As in the flat system, sequential composition is modelled using $\sqcup$ – but here, we use a scalar-vector extension of the operation. Finally, the (*let*) rule follows similar reasoning (and also corresponds to the typing of $(\lambda x.e_2)\ e_1$).

a.) Ordinary, syntax-driven rules along with sub-coeffecting

$$(\mathit{var})\quad \frac{}{x{:}\tau @ \langle \mathsf{L} \rangle \vdash x : \tau}$$

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

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

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

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

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

b.) Structural rules for context manipulation

$$(\mathit{weak})\quad \frac{\Gamma @ \mathbf{r} \vdash e : \sigma}{\Gamma, x{:}\tau @ \mathbf{r} \times \langle \mathsf{D} \rangle \vdash e : \sigma}$$

$$(\mathit{exch})\quad \frac{\Gamma_1, x{:}\tau', y{:}\tau, \Gamma_2 @ \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \sigma}{\Gamma_1, y{:}\tau, x{:}\tau', \Gamma_2 @ \mathbf{r} \times \langle t, s \rangle \times \mathbf{q} \vdash e : \sigma} \quad \begin{array}{l} \mathit{len}(\Gamma_1) = \mathit{len}(\mathbf{r}) \\ \mathit{len}(\Gamma_2) = \mathit{len}(\mathbf{s}) \end{array}$$

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

Figure 11: Structural coeffect liveness analysis

STRUCTURAL TYPING RULES. The structural typing rules are shown in Figure 11 (b). They mirror the rules know from sub-structural type systems (Section **??**). Weakening (*weak*) extends the context with a single unused variable x and adds the D annotation to the vector of coeffects.

The variable is always added to the end as in the (*abs*) rule. However, the exchange rule (*exch*) lets us arbitrarily reorder variables. It flips the variables x and x' and their corresponding coeffect annotations in the vector. This is done by requiring that the lengths of the remaining, unchanged, parts of the vectors match.

Finally, contraction (*contr*) makes it possible to use a single variable multiple times. Given a judgement that contains variables y and z, we can derive a judgement for an expression where both z and y are replaced by a single variable x. Their annotations s, t are combined into $s \sqcap t$, which means that x is live if either z or y were live in the original expression.

EXAMPLE. To demonstrate how the system works, we consider the expression $(\lambda x \rightarrow v)\ y$. This is similar to an example where flat liveness mistakenly marks the entire context as live. Despite the fact that the variable y is accessed (syntactically), it is not live – because the function that takes it as an argument always returns v.

The typing derivation for the body uses (*var*) and (*abs*). However, we also need (*weak*) to add the unused variable x to the context:

$$(\mathit{weak})\ \dfrac{\dfrac{\dfrac{}{v{:}\tau @ \langle \mathsf{L}\rangle \vdash v : \tau}\ (\mathit{var})}{v{:}\tau, x{:}\tau @ \langle \mathsf{L}, \mathsf{D}\rangle \vdash v : \tau}}{v{:}\tau @ \langle \mathsf{L}\rangle \vdash (\lambda x \to v) : \tau \xrightarrow{\mathsf{D}} \tau}\ (\mathit{abs})$$

The interesting part is the use of the (*app*) rule in the next step. Although the variable y is live in the expression y, it is marked as dead in the overall expression, because the function is annotated with D:

$$(\mathit{app})\ \dfrac{\dfrac{v{:}\tau @ \langle \mathsf{L}\rangle \vdash (\lambda x \to v) : \tau \xrightarrow{\mathsf{D}} \tau \qquad \dfrac{}{y{:}\tau @ \langle \mathsf{L}\rangle \vdash y : \tau}\ (\mathit{var})}{v{:}\tau, y{:}\tau @ \langle \mathsf{L}\rangle \times (\mathsf{D} \sqcup \langle \mathsf{L}\rangle) \vdash (\lambda x \to v)\ y : \tau}}{v{:}\tau, y{:}\tau @ \langle \mathsf{L}, \mathsf{D}\rangle \vdash (\lambda x \to v)\ y : \tau}$$

The application is written in two steps – the first one directly applies the (*app*) rule and the second one simplifies the coeffect annotation. The key part is the use of the scalar-vector operator $\mathsf{D} \sqcup \langle \mathsf{L}\rangle$. Using the definition of the scalar-vector extension, this equals $\langle \mathsf{D} \sqcup \mathsf{L}\rangle$ which is $\langle \mathsf{D}\rangle$.

SEMANTICS. When defining the semantics of flat liveness calculus, we used an indexed form of the option type $1 + \tau$ (which is 1 for dead contexts and τ for live contexts). In the semantics of expressions, the type constructor was applied to the entire context, i. e. $1 + (\tau_1 \times \ldots \times \tau_n)$. In the structural version, the semantics applies the option type constructor to individual elements of the free-variable context pair: $(1 + \tau_1) \times \ldots \times (1 + \tau_n)$. For each variable, the type is indexed by the corresponding annotation:

$$[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n @ \langle r_1, \ldots, r_n\rangle \vdash e : \tau]\!] \;:\; (\tau'_1 \times \ldots \times \tau'_n) \to \tau$$

$$\text{where } \tau'_i = \begin{cases} \tau_i & (r_i = \mathsf{L}) \\ 1 & (r_i = \mathsf{D}) \end{cases}$$

Note that the product of the free variables is not an ordinary tuple of our language, but a special construction (we return to this topic in Section 4.5). This follows from the asymmetry of λ-calculus, as discussed in Section 1.1.3. Functions take just a single input and so they are interpreted in the same way as in flat calculus:

$$[\![\tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!] = \tau_1 \to \tau_2 \qquad\qquad [\![\tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!] = 1 \to \tau_2$$

The rules that define the semantics are shown in Figure 12. To make the definition simpler, we are somewhat vague when working with products. We write variables of product type such as $\mathbf{v}$ in bold-face and individual values like x in normal face. We freely re-associate products and so $(\mathbf{v}, x)$ should not be seen as a nested product, but simply as a product containing all variables from the product $\mathbf{v}$ together with one additional variable x at the end. We shall be more precise in Chapter 4.

In (*var*), the context contains just a single variable and so we do not even need to apply projection; (*cosnt*) receives no variables and uses global constant lookup function δ. In (*abs*), we obtain two parts of the context and combine them into $(\mathbf{v}, x)$. This works the same way regardless of whether the variables are live or dead. For simplicity, we omit sub-coeffecting, which just turns some of the available values v_i to unit values $()$.

As dictated by the semantics, the application again needs to "implement" dead code elimination (otherwise the type system would be unsound). When a.) Semantics of ordinary expressions

$$[\![x\!:\!\tau\, @\, \langle \mathsf{L}\rangle \vdash x : \tau]\!] = \lambda(x) \to x \qquad (\textit{var})$$

$$[\![()\, @\, \langle\rangle \vdash c : \tau]\!] = \lambda() \to \delta(c) \qquad (\textit{const})$$

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

$$\begin{aligned}&[\![\Gamma_1, \Gamma_2\, @\, \mathbf{r}\times(\mathsf{L} \sqcup \mathbf{s}) \vdash e_1\, e_2 : \tau_2]\!] = \lambda(\mathbf{v_1}, \mathbf{v_2}) \to \\ &\quad \texttt{let}\ g = [\![\Gamma_1\, @\, \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!]\ \mathbf{v_1} \\ &\quad \texttt{in}\ g\ ([\![\Gamma_2\, @\, \mathbf{s} \vdash e_2 : \tau_1]\!]\ \mathbf{v_2})\end{aligned} \qquad (\textit{app-1})$$

$$\begin{aligned}&[\![\Gamma_1, \Gamma_2\, @\, \mathbf{r}\times(\mathsf{D} \sqcup \mathbf{s}) \vdash e_1\, e_2 : \tau_2]\!] = \lambda(\mathbf{v_1}, \mathbf{v_2}) \to \\ &\quad \texttt{let}\ g = [\![\Gamma_1\, @\, \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!]\ \mathbf{v_1}\ \texttt{in}\ g\ ()\end{aligned} \qquad (\textit{app-2})$$

b.) Semantics of structural context manipulation

$$[\![\Gamma, x\!:\!\tau\, @\, \mathbf{r}\times\langle \mathsf{D}\rangle \vdash e : \sigma]\!] = \lambda(\mathbf{v}, ()) \to [\![\Gamma\, @\, \mathbf{r} \vdash e : \sigma]\!]\ \mathbf{v} \qquad (\textit{weak})$$

$$\begin{aligned}&[\![\Gamma_1, y\!:\!\tau, x\!:\!\tau', \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{t}, \mathsf{s}\rangle\times\mathbf{q} \vdash e : \sigma]\!] = \lambda(\mathbf{v_1}, y, x, \mathbf{v_2}) \to \\ &\quad [\![\Gamma_1, x\!:\!\tau', y\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{s}, \mathsf{t}\rangle\times\mathbf{q} \vdash e : \sigma]\!]\ (\mathbf{v_1}, x, y, \mathbf{v_2})\end{aligned} \qquad (\textit{exch})$$

$$\begin{aligned}&[\![\Gamma_1, x\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{D}\rangle\times\mathbf{q} \vdash e[z \leftarrow x][y \leftarrow x] : \sigma]\!] = \lambda(\mathbf{v_1}, (), \mathbf{v_2}) \to \\ &\quad [\![\Gamma_1, y\!:\!\tau, z\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{D}, \mathsf{D}\rangle\times\mathbf{q} \vdash e : \sigma]\!]\ (\mathbf{v_1}, (), (), \mathbf{v_2})\end{aligned} \qquad (\textit{contr-1})$$

$$\begin{aligned}&[\![\Gamma_1, x\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{L}\rangle\times\mathbf{q} \vdash e[z \leftarrow x][y \leftarrow x] : \sigma]\!] = \lambda(\mathbf{v_1}, x, \mathbf{v_2}) \to \\ &\quad \begin{cases} [\![\Gamma_1, y\!:\!\tau, z\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{L}, \mathsf{L}\rangle\times\mathbf{q} \vdash e : \sigma]\!]\ (\mathbf{v_1}, x, x, \mathbf{v_2}) \\ [\![\Gamma_1, y\!:\!\tau, z\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{D}, \mathsf{L}\rangle\times\mathbf{q} \vdash e : \sigma]\!]\ (\mathbf{v_1}, (), x, \mathbf{v_2}) \\ [\![\Gamma_1, y\!:\!\tau, z\!:\!\tau, \Gamma_2\, @\, \mathbf{r}\times\langle \mathsf{L}, \mathsf{D}\rangle\times\mathbf{q} \vdash e : \sigma]\!]\ (\mathbf{v_1}, x, (), \mathbf{v_2}) \end{cases}\end{aligned} \qquad (\textit{contr-2})$$

Figure 12: Semantics of structural liveness

the input parameter of the function g is live (*app-1*), we first evaluate e_2 and then pass the result to g. When the parameter is dead (*app-2*), we do not need to evaluate e_2 and so all values in $\mathbf{v_2}$ can be dead, i. e. $()$.

In the structural rules, (*weak*) receives context containing a dead variable as the last one. It drops the $()$ value and evaluates the expression in a context $\mathbf{v}$. Exchange (*exch*) simply swaps two variables. In contraction, we either duplicate a dead value (*contr-1*), or a live value (*contr-2*). In the latter, one of the duplicates may be dead and so we need to consider three separate cases.

Summary. The structural liveness calculus is a typical example of a system that tracks per-variable annotations. In a number of ways, the system is simpler than the flat coeffect calculi. In lambda abstraction, we simply annotate function with the annotation of a matching variable (this rule is the same for all upcoming systems). In application, the *pointwise* composition is no longer needed, because the sub-expressions use separate contexts. On the other hand, we had to add weakening, contraction and exchange rules to let us manipulate contexts.

The semantics of weakening demonstrates an important point about coeffects that may be quite confusing. When we read the *typing rule* from top to bottom, weakening adds a variable to the context. When we read the *semantic rule*, weakening drops a variable value from the context! This duality is caused by the fact that coeffects talk about context – they describe how to build the context required by the sub-expressions and so the semantics implements transformation from the context in the (typing) conclusion to the (typing) assumption. How should coeffects be understood, in general, is discussed further in Section 2.2.3.

The structural systems discussed in the upcoming sections are remarkably similar to the one shown here. We discuss two more examples to explore the design space, but we shall omit details that are the shared with the system in this section.

1.3.2 *Bounded variable use*

Liveness analysis checks whether a variable is used or unused. With structural coeffects, we can go further and track how many times is the variable accessed. Girard et al. [41] coined this idea as *bounded linear logic* and use it to restrict well-typed programs to polynomial-time algorithms. We first introduce the system in our, coeffect, style and then relate it with the original formulation.

BOUNDED VARIABLE USE. The system discussed in this section tracks the number of times a variable is accessed in the call-by-name evaluation. Although we look at an example that tracks *variable usage*, the same system could be used to track access to resources that are always passed as a reference (and behave effectively as call-by-name) and so the system is relevant for call-by-value languages too. To demonstrate the idea, consider the following term:

$$(\lambda v.x + v + v)\ (x + y)$$

When evaluated, the body of the function directly accesses x once and then twice indirectly, via the function argument. Similarly, y is accessed twice indirectly. Thus, the overall expression uses x three times and y twice.

As discussed in Chapter 4, the system preserves type and coeffect annotations under the β-reduction. Reducing the expression in this case gives $x + (x + y) + (x + y)$. This has the same bounds as the original expression – x is used three times and y twice.

TYPE SYSTEM. The type system in Figure 13 annotates contexts with vectors of integers. The rules have the same structure as those of the system for liveness analysis. The only difference is how annotations are combined. Here, we use integer multiplication ($*$) for sequential composition and addition ($+$) for point-wise composition.

Variable access (*var*) annotates a variable with 1, meaning that it has been used once. An unused variable (*weak*) is annotated with 0. Multiple occurrences of the same variable are introduced by contraction (*contr*), which adds the numbers of the two contracted variables.

As previously, application (*app*) and let binding (*let*) combine two separate contexts. The second part applies a function that uses its parameter t-times to an argument that uses variables in Γ_2 at most $\mathbf{s}$-times (here, $\mathbf{s}$ is a vector of integers with an annotations for each variable in Γ_2). The sequential composition (modelling call-by-name) multiplies the uses, meaning that the total number of uses is $(t * \mathbf{s})$ (where $*$ is a point-wise multiplication of a vector by a scalar). This models the fact that for each use of the function parameter, we replicate the variable uses in e_2.

a.) Ordinary, syntax-driven rules along with sub-coeffecting

$$(\textit{var})\quad \frac{}{x{:}\tau @ \langle 1\rangle \vdash x : \tau}$$

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

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

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

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

b.) Structural rules for context manipulation

$$(\textit{weak})\quad \frac{\Gamma @ \mathbf{r} \vdash e : \sigma}{\Gamma, x{:}\tau @ \mathbf{r}\times\langle 0\rangle \vdash e : \sigma}$$

$$(\textit{exch})\quad \frac{\Gamma_1, x{:}\tau', y{:}\tau, \Gamma_2 @ \mathbf{r}\times\langle s, t\rangle\times\mathbf{q} \vdash e : \sigma}{\Gamma_1, y{:}\tau, x{:}\tau', \Gamma_2 @ \mathbf{r}\times\langle t, s\rangle\times\mathbf{q} \vdash e : \sigma} \quad \begin{array}{l} \textit{len}(\Gamma_1) = \textit{len}(\mathbf{r}) \\ \textit{len}(\Gamma_2) = \textit{len}(\mathbf{s}) \end{array}$$

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

Figure 13: Structural coeffect bounded reuse analysis

Finally, the sub-coeffecting rule (*sub*) safely overapproximates the number of accesses using the pointwise $\leqslant$ relation. We can view any variable as being used a greater number of times than it actually is.

EXAMPLE. To type check the expression $(\lambda v.x + v + v)\ (x + y)$ discussed earlier, we need to use abstraction, application, but also the contraction rule. Assuming the type judgement for the body, abstractions yields:

$$(\textit{abs})\quad \frac{x{:}\mathbb{Z}, v : \mathbb{Z} @ \langle 1, 2\rangle \vdash x + v + v : \mathbb{Z}}{x{:}\mathbb{Z} @ \langle 1\rangle \vdash (\lambda v.x + v + v) : \mathbb{Z} \xrightarrow{2} \mathbb{Z}}$$

To type-check the application, the contexts of e_1 and e_2 need to contain disjoint variables. For this reason, we α-rename x to x' in the argument $(x + y)$ and later join x and x' using the contraction rule. Assuming $(x' + y)$ is checked in a context that marks x' and y as used once, the application rule yields a judgement that is simplified as follows:

$$\frac{x{:}\mathbb{Z}, x'{:}\mathbb{Z}, y{:}\mathbb{Z} @ \langle 1\rangle \times (2 * \langle 1, 1\rangle) \vdash (\lambda v.x + v + v)\ (x' + y) : \mathbb{Z}}{(\textit{contr})\quad \dfrac{x{:}\mathbb{Z}, x'{:}\mathbb{Z}, y{:}\mathbb{Z} @ \langle 1, 2, 2\rangle \vdash (\lambda v.x + v + v)\ (x' + y) : \mathbb{Z}}{x{:}\mathbb{Z}, y{:}\mathbb{Z} @ \langle 3, 2\rangle \vdash (\lambda v.x + v + v)\ (x + y) : \mathbb{Z}}}$$

The first step performs scalar multiplication, producing the vector $\langle 1, 2, 2\rangle$. In the second step, we use contraction to join variables x and x' from the function and argument terms respectively.

SEMANTICS. In the previous examples, we defined the semantics – somewhat informally – using a simple λ-calculus language to encode the model. More formally, this could be a Cartesian-closed category. In that model, we can reuse variables arbitrarily and so it is not a good fit for modelling bounded reuse. Girard et al. [41] model their bounded linear logic in an (ordinary) linear logic where variables can be used at most once.

Following the same approach, we could model a variable τ, annotated with r as a product containing r copies of τ, that is τ^r:

$$\llbracket x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n \mathbin{@} \langle r_1,\ldots,r_n\rangle \vdash e:\tau\rrbracket \;:\; (\tau_1^{r_1}\times\ldots\times\tau_n^{r_n}) \to \tau$$

$$\text{where } \tau_i^{r_i} = \underbrace{\tau_i\times\ldots\times\tau_i}_{r_i-\text{times}}$$

The functions are interpreted similarly. A function $\tau_1 \xrightarrow{t} \tau_2$ is modelled as a function taking t-element product of τ_1 values: $\tau_1^t \to \tau_2$.

The rules that define the semantics of bounded calculus are mostly the same as (or easy to adapt from) the semantic rules of liveness in Figure 12. The ones that differ are those that use sequential composition (application and let binding) and the contraction rule, which represents pointwise composition.

In the following, we use vector names $\mathbf{v}_i$ for contexts containing multiple variables i. e. have a type $\tau_1^{r_1}\times\ldots\times\tau_m^{r_m}$. Each vector contains multiple copies of each variable, to model the fact that variables are used in an affine way (at most once). We do not explicitly write the sizes of these vectors (number of variables in a context; number of instances of a variable) as these are clear from the coeffect annotations. We assume that Γ_2 contains n variables and that $s = \langle s_1,\ldots,s_n\rangle$:

$$\begin{aligned}
&\llbracket \Gamma_1, x\!:\!\tau, \Gamma_2 \mathbin{@} \mathbf{r}\times\langle s+t\rangle\times\mathbf{q} \vdash e[z\leftarrow x][y\leftarrow x] : \sigma\rrbracket = \\
&\quad \lambda(\mathbf{v_1},(x_1,\ldots,x_{s+t}),\mathbf{v_2}) \to \\
&\qquad \llbracket \Gamma_1, y\!:\!\tau, z\!:\!\tau, \Gamma_2 \mathbin{@} \mathbf{r}\times\langle s,t\rangle\times\mathbf{q} \vdash e:\sigma\rrbracket \\
&\qquad\quad (\mathbf{v_1},(x_1,\ldots,x_s),(x_{s+1},\ldots,x_{s+t}),\mathbf{v_2})
\end{aligned} \qquad (\textit{contr})$$

$$\begin{aligned}
&\llbracket \Gamma_1,\Gamma_2 \mathbin{@} \mathbf{r}\times(t*\mathbf{s}) \vdash e_1\,e_2 : \tau_2\rrbracket = \\
&\quad \lambda(\mathbf{v_1},((x_{1,1},\ldots,x_{1,t*s_1}),\ldots,(x_{n,1},\ldots,x_{n,t*s_n})) \to \\
&\qquad \texttt{let } g = \llbracket \Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2\rrbracket\; \mathbf{v_1} \\
&\qquad \texttt{let } \mathbf{y}_1 = ((x_{1,1},\ldots,x_{1,s_1}),\ldots,(x_{n,1},\ldots,x_{1,s_n})) \\
&\qquad \texttt{let } \ldots \\
&\qquad \texttt{let } \mathbf{y}_t = ((x_{1,(t-1)*s_1+1},\ldots,x_{1,t*s_1}),\ldots, \\
&\qquad\qquad\qquad (x_{n,(t-1)*s_n+1},\ldots,x_{1,t*s_n})) \\
&\qquad \texttt{in } g\;(\llbracket\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1\rrbracket\;\mathbf{y}_1,\ldots,\llbracket\Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1\rrbracket\;\mathbf{y}_t)
\end{aligned} \qquad (\textit{app})$$

In the (*contr*) rule, the semantic function is called with $s+t$ copies of a value for the x variable. The values are split between s and t separate copies of variables y and z, respectively.

The (*app*) rule is similar in that it needs to split the input variable context. However, it needs to split values of multiple variables – in $x_{i,j}$, the index i stands for an index of the variable while j is an index of one of multiple copies of the value. In the semantic function, the second part of the context consists of n variables where the multiplicity of each value is specified by the annotation s_i multiplied by t. The rule needs to evaluate the argument e_2 t-times and each call requires s_i copies of the i^{th} variable. To do this, we create contexts $\mathbf{y}_1$ to $\mathbf{y}_t$, each containing s_i copies of the variable (and so we require $s_i * t$ copies of each variable). Note that the contexts are created such that each value is used exactly once.

It is worth noting that the (*var*) rule requires exactly one copy of a variable and so the system tracks precisely the number of uses. However, the (*sub*) rule lets us ignore additional copies of a value. Thus, permitting (*sub*) rule is only possible if the underlying model is *affine* rather than *linear*.

BOUNDED LINEAR LOGIC. The system presented in this section is based on the idea of bounded linear logic (BLL) [41], but it is adapted to follow the structure of other coeffect systems discussed in this chapter. This elucidates the connection between BLL and coeffects.

The big difference, using the terminology from Section **??**, is that our system is written in *language semantics* style, while BLL is written in *meta-language* style. We briefly consider the original BLL formulation.

The terms and types of our system are the terms and types of an ordinary λ-calculus, with the only difference that functions carry coeffect annotations. In BLL, the language of types is extended with a type constructor $!_k A$ (where A is a proposition, corresponding to a type τ in our system). The type denotes a value A that can be used at most k times.

As a result, BLL does not need to attach additional annotation to the variable context as a whole. The requirements are attached to individual variables and so our context $\tau_1, ..., \tau_n @ \langle k_1, ..., k_n \rangle$ corresponds to a BLL assumption $!_{k_1} A_1, ..., !_{k_n} A_n$. Using the formulation of bounded logic (and omitting the terms), the weakening and contraction rules are written as follows:

$$(\textit{weak})\ \frac{\Gamma \vdash B}{\Gamma, !_0 A \vdash B} \qquad\qquad (\textit{contr})\ \frac{\Gamma, !_n A, !_m A \vdash B}{\Gamma, !_{n+m} A \vdash B}$$

The system captures the same idea as the structural coeffect system presented above. Variable access in bounded linear logic is simply an operation that produces a value $!_n A$ and so the system further introduces *dereliction* rule which lets us treat $!_1 A$ as a value A. We further explore difference between *language semantics* and *meta-language* in Section **??**.

SUMMARY. Comparing the structural coeffect calculus for tracking liveness and for bounded variable reuse reveals which parts of the systems differ and which parts are shared. In particular, both systems use the same vector operations ($\times$, $\langle - \rangle$) and also share the lambda abstraction rule (*abs*). They differ in the primitive values used to annotate used and unused variables (L, D and 1, 0, respectively) and in the operators used for sequential composition and contraction ($\sqcup$, $\sqcap$ and $*$, $+$, respectively). The algebraic structure capturing these operators is developed in Chapter 4.

The brief overview of bounded linear logic shows an alternative approach to tracking properties related to individual variables – we could attach annotations to the variables themselves rather than attaching a *vector* of annotations to the entire context. The main benefit of our approach is that it lets us unify flat and structural systems (Chapter **??**).

1.3.3 *Data-flow languages revisited*

When discussing data-flow languages in an earlier section, we said that the context provides past values of variables. In Section 1.2.4, we tracked this as a *flat* property, which gives us a system that keeps the same number of past values for all variables. However, data-flow can also be adapted to a structural system which keeps the number of required past values individually for each variable. Consider the following example:

$$(\textit{var})\quad \frac{}{x\!:\!\tau \,@\, \langle 0\rangle \vdash x : \tau}$$

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

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

$$(\textit{weak})\quad \frac{\Gamma \,@\, \mathbf{r} \vdash e : \sigma}{\Gamma, x\!:\!\tau \,@\, \mathbf{r} \times \langle 0\rangle \vdash e : \sigma}$$

$$(\textit{contr})\quad \frac{\Gamma_1, y\!:\!\tau, z\!:\!\tau, \Gamma_2 \,@\, \mathbf{r} \times \langle s, t\rangle \times \mathbf{q} \vdash e : \sigma}{\Gamma_1, x\!:\!\tau, \Gamma_2 \,@\, \mathbf{r} \times \langle \max(s,t)\rangle \times \mathbf{q} \vdash e[z \leftarrow x][y \leftarrow x] : \sigma}$$

Figure 14: Structural coeffect bounded reuse analysis

$$\mathtt{let}\ \mathsf{offsetAdd} = \mathsf{left} + \mathtt{prev}\ \mathsf{right}$$

The value offsetAdd adds values of left with previous values of right. To evaluate a current value of the stream, we need the current value of left and one past value of right. Flat system is not able to capture this level-of-detail and simply requires 1 past values of both streams in the variable context.

Turning a flat data-flow system to a structural data-flow system is a change similar to the one between flat ans structural liveness. In case of liveness analysis, we included the flat system only as an illustration (it is not practically useful). For data-flow, the flat system is less precise, but still practically useful (simplicity may outweigh precision).

TYPE SYSTEM. The type system in Figure 14 annotates the variable context with a vector of integers. This is similar as in the bounded reuse system, but the integers *mean* a different thing. Consequently, they are also calculated differently. We omit rules that are the same for all structural coeffect systems (exchange, lambda abstraction).

In data-flow, we annotate both used variables (*var*) and unused variables (*weak*) with 0, meaning that no past values are required. This is the same as in flat data-flow, but different from bounded reuse and liveness (where difference between using and not using a variable matters). Primitive requirements are introduced by the (*prev*) rule, which increments the annotation of all variables in the context.

In flat data-flow, we identified sequential composition and pointwise composition as two primitive operations that were used in the (flat) application. In the structural system, these are used in (*app*) and (*contr*), respectively. Thus application combines coeffect annotations using $+$ and contraction using *max*. This contrasts with bounded reuse, which uses $*$ and $+$, respectively.

EXAMPLE. As an example, consider a function $\lambda x.\mathtt{prev}\ (y + x)$ applied to an argument $\mathtt{prev}\ (\mathtt{prev}\ y)$. The body of the function accesses the past value of two variables, one free and one bound. The (*abs*) rule splits the annotations between the declaration site and call site of the function:

$$(\textit{abs})\quad \frac{y\!:\!\mathbb{Z}, x\!:\!\mathbb{Z} \,@\, \langle 1, 1\rangle \vdash \mathtt{prev}\ (y + x) : \mathbb{Z}}{y\!:\!\mathbb{Z} \,@\, \langle 1\rangle \vdash \lambda x.\mathtt{prev}\ (y + x) : \mathbb{Z} \xrightarrow{1} \mathbb{Z}}$$

The expression always requires the previous value of y and adds it to a previous value of the parameter x. Evaluating the value of the argument $\texttt{prev}\ (\texttt{prev}\ y)$ requires two past values of y and so the overall requirement for the (free) variable y is 3 past values. In order to use the contraction rule, we rename y to y' in the argument:

$$\dfrac{\dfrac{y{:}\mathbb{Z} @ \langle 1 \rangle \vdash \lambda x.\,(\ldots) : \mathbb{Z} \xrightarrow{1} \mathbb{Z} \qquad x{:}\mathbb{Z} @ \langle 2 \rangle \vdash (\texttt{prev}\ (\texttt{prev}\ y') : \mathbb{Z}}{y{:}\mathbb{Z}, y'{:}\mathbb{Z} @ \langle 1, 3 \rangle \vdash (\lambda x.\texttt{prev}\ (y + x))\ (\texttt{prev}\ (\texttt{prev}\ y')) : \mathbb{Z}}}{y{:}\mathbb{Z} @ \langle 3 \rangle \vdash (\lambda x.\texttt{prev}\ (y + x))\ (\texttt{prev}\ (\texttt{prev}\ y)) : \mathbb{Z}}$$

The derivation uses (*app*) to get requirements $\langle 1, 3 \rangle$ and then (*contr*) to take the maximum, showing three past values are sufficient.

Note that we get the same requirements when we perform β reduction of the expression. Substituting the argument for x yields the expression $\texttt{prev}\ (y + (\texttt{prev}\ (\texttt{prev}\ y)))$. Semantically, this performs stream lookups $y[1]$ and $y[3]$ where the indices are the number of enclosing $\texttt{prev}$ constructs.

SEMANTICS. To define the semantics of our structural data-flow language, we can use the same approach as when adapting flat liveness to structural liveness. Rather than wrapping the whole context in a type constructor (list or option), we now wrap the individual components of the product representing the variables in the context.

The result is similar as the structure used for bounded reuse. The only difference is that, given a variable annotated with r, we need $1 + r$ values. That is, we need the current value, followed by r past values:

$$\begin{aligned}
&[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n @ \langle r_1, \ldots, r_n \rangle \vdash e : \tau]\!] \ :\ (\tau_1^{(r_1+1)} \times \ldots \times \tau_n^{(r_n+1)}) \to \tau \\
&[\![\tau_1 \xrightarrow{s} \tau_2]\!] \ =\ \tau_1^{(s+1)} \to \tau_2
\end{aligned}$$

Despite the similarity with the semantics for bounded reuse, the values here *represent* different things. Rather than providing multiple copies of a value (out of which each can be used just once), the pair provides past values (that can be reused and freely accessed). To illustrate the behaviour we consider the semantics of the $\texttt{prev}$ construct and of the structural contraction rule:

$$\begin{aligned}
&[\![\Gamma @ \langle (s_1 + 1), \ldots, (s_n + 1) \rangle \vdash \texttt{prev}\ e : \tau]\!] = \\
&\quad \lambda((x_{1,0}, \ldots, x_{1,s_1+1}), \ldots, (x_{n,0}, \ldots, x_{n,s_n+1})) \to \\
&\qquad [\![\Gamma @ \langle s_1, \ldots, s_n \rangle \vdash e : \tau]\!] \\
&\qquad\quad ((x_{1,0}, \ldots, x_{1,s_1}), \ldots, (x_{n,0}, \ldots, x_{n,s_n}))
\end{aligned} \qquad (\textit{prev})$$

$$\begin{aligned}
&[\![\Gamma_1, x{:}\tau, \Gamma_2 @ \mathbf{r} \times \langle \max(s, t) \rangle \times \mathbf{q} \vdash e[z \leftarrow x][y \leftarrow x] : \sigma]\!] = \\
&\quad \lambda(\mathbf{v_1}, (x_0, x_1, \ldots, x_{\max(s,t)}), \mathbf{v_2}) \to \\
&\qquad [\![\Gamma_1, y{:}\tau, z{:}\tau, \Gamma_2 @ \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \sigma]\!] \\
&\qquad\quad (\mathbf{v_1}, (x_0, \ldots, x_s), (x_0, \ldots, x_t), \mathbf{v_2})
\end{aligned} \qquad (\textit{contr})$$

In (*prev*), the semantic function is called with an argument that stores values of n variables, such that a variable x_i has values ranging from $x_{i,0}$ to x_{i,s_i+1}. Thus, there is one current value, followed by $s_i + 1$ past values. The expression e nested under $\texttt{prev}$ requires only s_i past values and so the semantics simply drops the last value.

In the (*contr*) rule, the semantic function receives $\mathit{max}(s, t)$ values of a specific variable x. It needs to produce values for two separate variables, y and z that require s and t past values. Both of these numbers are certainly smaller than (or equal to) the number of values available. Thus we simply take the first values. Unlike in the contraction for BLL, the values are duplicated and the same values are used for both variables.

SUMMARY. Two of the structural examples shown so far (liveness and data-flow) extend an earlier flat version of a similar system. We discuss this relation in general later. However, a flat system can generally be turned into a structural one – although this only gives a useful system when the flat version captures statically scoped properties, i. e. related to variables.

The data-flow example demonstrates that the a flat system can also be turned into structural system. In general, this only works for systems where lambda abstraction duplicates context requirements (as in Figure 7).

1.3.4 *Security, tainting and provenance*

Tainting is a mechanism where variables coming from potentially untrusted sources are marked (*tainted*) and the use of such variables is disallowed in contexts where untrusted input can cause security issues or other problems. Tainting can be done dynamically using a runtime mark (e. g. in the Perl language) or using a static type system. Tainting can be viewed as a special case of *provenance tracking*, known from database systems [20], where values are annotated with more detailed information about their source.

Static typed systems based on tainting have been use to prevent cross-site scripting attacks [118] and SQL injection attacks [45, 44]. In the latter case, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables id and msg:

```
let name = query("SELECT Name WHERE Id = %1", id)
msg + name
```

In this example, id must not come directly from a user input, because query requires untainted string. Otherwise, the attacker could specify values such as `"1; DROP TABLE Users"`. The variable msg may or may not be tainted, because it is not used in protected context (i.e. to construct an SQL query).

In runtime checking, all (string) values need to be wrapped in an object with a Boolean flag (for tainting) or more complex data (for provenance). In static checking, the information need to be associated with the variables in the variable context.

CORE DEPENDENCY CALCULUS. Taint checking is a special case of checking of the *non-interference* property in *secure information flow*. There, the aim is to guarantee that sensitive information (such as credit card number) cannot be leaked to contexts with low secrecy (e. g. sent via an unsecured network channel). Volpano et al. [119] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [94] surveys more recent work. Information flow checking has been also integrated (as a single-purpose extension) in the FlowCaml [100] language. Finally, Russo et al. and Swamy et al. [93, 102] show that such properties can be checked using a monadic library.

Systems for secure information flow typically define a lattice of security classes $(\mathcal{S}, \leqslant)$ where $\mathcal{S}$ is a finite set of classes and an ordering. For example a set $\{\mathsf{L}, \mathsf{H}\}$ represents low and high secrecy, respectively with $\mathsf{L} \leqslant \mathsf{H}$ meaning that low security values can be treated as high security (but not the other way round).

IMPLICIT FLOWS. An important aspect of secure information flow is called *implicit flows*. Consider the following example which returns either y or zero, depending on the value of x:

$$\texttt{let}\ z = \texttt{if}\ x > 0\ \texttt{then}\ y\ \texttt{else}\ 0$$

If the value of y is high-secure, then z becomes high-secure after the assignment (this is an *explicit* flow). However, if x is high-secure, then the value of z becomes high-secure, regardless of the security level of y, because the fact whether an assignment is performed or not performed leaks information in its own (this is an *implicit* flow).

Although we do not describe a coeffect calculus for information flow checking, it is worth noting that Abadi et al. [1] realized that there is a number of analyses similar to secure information flow and unified them using a single model called Dependency Core Calculus (DCC). This would be a useful basis for coeffect-based information flow checking.

The DCC captures other cases where some information about expression relies on properties of variables in the context where it executes. This includes, for example, *binding time analysis* [110], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [111] that identifies parts of programs that contribute to the output of an expression.

COEFFECT SYSTEMS. The work outlined in this section is another area where coeffect systems could be applied. We do not develop coeffect systems for taint tracking, security and provenance in detail, but briefly mention some examples in the upcoming chapters.

The systems work in the same way as the examples discussed already. For example, consider the tainting example with the query function calling an SQL database. To capture such tainting, we annotate variables with T for *tainted* and with U for *untainted*. Accessing a variable marks it as untainted, but using an expression that depends on some variable in certain dangerous contexts – such as in arguments of query – does introduce a taint on all the variables contributing to the expression. This is captured using the standard application rule (*app*):

$$(\mathit{app})\ \frac{\Gamma @ r \vdash \mathsf{query} : \mathsf{string} \xrightarrow{\mathsf{T}} \mathsf{Table} \qquad \mathsf{id} : \mathsf{string} @ \langle \mathsf{U} \rangle \vdash \mathsf{id} : \mathsf{string}}{\Gamma, \mathsf{id} : \mathsf{string} @ r \times \langle \mathsf{T} \rangle \vdash \mathsf{query}(\texttt{"..."}, \mathsf{id}) : \mathsf{Table}}$$

The derivation assumes that query is a standard function that requires the parameters to be tainted (it does not have to be a built-in language construct). The argument is a variable and so it is not tainted in the assumptions.

In the conclusion, we need to derive an annotation for the variable id. To do this, we combine T (from the function) and U (from the argument). In case of tainting, the variable is tainted whenever it is already tainted *or* the function marks it as tainted. For different kinds of annotations, the composition would work differently – for example, for provenance, we could union the *set* of possible data sources, or even combine *probability distributions* modelling the influence of different sources on the value. However, expanding such ideas is beyond the scope of this thesis.

1.4 BEYOND PASSIVE CONTEXTS

In both flat and structural systems discussed so far, the context provides additional data (resources, implicit parameters, historical values) or metadata (security, provenance). However, *within* the language, it is impossible to write a function that modifies the context. We use the term *passive* context for such applications.

There is a number of systems that also capture contextual properties, but that make it possible to *change* the context – not just by evaluating certain code block in a locally modified context (e. g.by wrapping it in prev in dataflow), but also by calling a function that, for example, acquires new capabilities and returns those to the caller. Such actions appear to be closer to effects than to coeffects. While this thesis focuses on systems with passive contexts, we briefly consider the most important examples of the *active* variant.

CALCULUS OF CAPABILITIES. Crary et al. [27] introduced the Calculus of Capabilities to provide a sound system with region-based memory management for low-level code that can be easily compiled to assembly language. They build on the work of Tofte and Talpin [112] who developed an effect system (as discussed in Section **??**) that uses lexically scoped *memory regions* to provide an efficient and controlled memory management.

In the work of Tofte and Talpin, the context is *passive*. They extend a simple functional language with the letrgn construct that defines a new memory region, evaluates an expression (possibly) using memory in that region and then deallocates the memory of the region:

```
let calculate = λinput →
  letrgn ρ in
  let x = ref_ρ input in
  x := !x + 1; !x
```

The memory region ρ is a part of the context, but only in the scope of the body of letrgn. It is only available to the last two lines which allocate a memory cell in the region, increment a value in the region and then read it. The region is de-allocated when the execution leaves its lexical scope – there is no way to allocate a region inside a function and pass it back to the caller.

The calculus of capabilities differs in two ways. First, it allows explicit allocation and deallocation of memory regions (and so region lifetimes do not necessarily follow strict LIFO ordering). Second, it uses continuation-passing style. We ignore the latter aspect. The following example is almost identical to the previous one:

```
let calculate = λinput →
  letrgn ρ in
  let x = ref_ρ input in
  x := !x + 1; x
```

The difference is that the example does not return the *value* of a reference using !x, but returns the reference x itself. The reference is allocated in a newly created region ρ. Together with the value, the function returns a *capability* to access the region ρ.

This is where systems with active context differ from systems with passive context. To type check such programs, we do not only need to know what context is required to call calculate (i. e. context on the left-hand side of $\vdash$). We also need to know what effects an expression has on the context when it evaluates and the current context meeds to be updated after a function call. This is an effectful property that would appear on the right-hand side of $\vdash$.

ACTIVE CONTEXTS. In a systems with passive contexts, we only need an annotation that specifies the required context. In semantics, this is reflected by having some structure (data type) $\mathcal{C}$ over the *input* of the function. Without giving any details, the semantics generally has the following structure (with comonad to model coeffects on the left):

$$[\![\tau_1 \xrightarrow{r} \tau_2]\!] = \mathcal{C}^r \tau_1 \to \tau_2$$

Systems with active contexts require two annotations – one that specifies the context required before the call is performed and one that specifies how the context changes after the call (this could be either a *new* context or *update* to the original context). Thus the structure of the semantics would look as follows (with comonad to model coeffects on the left and monad to model effects on the right):

$$[\![\tau_1 \xrightarrow{r,s} \tau_2]\!] = \mathcal{C}^r \tau_1 \to \mathcal{M}^s \tau_2$$

In case of Calculus of Capabilities, both of the structures could be the same and they could carry a set of available memory regions. In this thesis, we focus only on passive contexts. However, capturing active contexts is an interesting future work.

SOFTWARE UPDATING. Another example of a system that uses contextual information actively is dynamic software updating (DSU) [36, 48]. The DSU systems have the ability to update programs at runtime without stopping them. For example, Proteus developed by Stoyle et al. [101] investigates what language support is needed to enable safe dynamic software updating in C-like languages. The work is based on capabilities and follows a structure similar to the Calculus of Capabilities [27].

The system distinguishes between *concrete* uses and *abstract* uses of a value. When a value is used concretely, the program examines its representation (and so it is not safe to change the representation during an update). An abstract use of a value does not examine the representation and so updating the value does not break the program.

The Proteus system uses capabilities to restrict what types may be used concretely after any point in the program. All other types, not listed in the capability, can be dynamically updated as this will not change concrete representation of types accessed later in the evaluation.

Similarly to Capability Calculus, capabilities in DSU can be changed by a function call. For example, calling a function that may update certain types makes it impossible to use those types concretely following the function call. This means that DSU uses the context *actively* and not just *passively*.

1.5 SUMMARY

This chapter served two purposes. The first aim was to present existing work on programming languages and systems that include some notion of *context*. Because there was no well-known abstraction capturing contextual properties, the languages use a wide range of formalisms – including principled approaches based on comonads and modal S4, ad-hoc type system extensions and static analyses as well as approaches based on monads. We looked at a number of applications including Haskell's implicit parameters and type classes, data-flow languages such as Lucid, liveness analysis and also a number of security properties.

The second aim of this chapter was to re-formulate the existing work in a more uniform style and thus reveal that all *context-dependent* languages share a common structure. In the upcoming three chapters, we identify the common structure more precisely and develop three calculi to capture it. We will then be able to re-create many of the examples discussed in this chapter just by instantiating our unified calculi.

This chapter was divided into two major sections. First, we looked at *flat* systems, which track whole-context properties. Next, we look a *structural* systems, which track per-variable properties. Both of the variants are useful and important – for example, implicit parameters can only be expressed as *flat* system, but liveness analysis is only useful as *structural*. For this reason, we explore both of these variants in this thesis (Chapter 2 and Chapter 4, respectively). We can, however, unify the two variants into a single system discussed in Chapter **??**.

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 2) 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 3). 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 2 and Chapters 3, highlighting the split between general definitions and properties (about the coeffect calculus) and concrete definitions and properties (about concrete context-awre language). The Chapter 4 mirrors the same development for *structural coeffect systems*.

CHAPTER 4

	Coeffect calculus	Language-specific
Syntax	Coeffect λ-calculus (Section 2.2)	Extensions such as ?param and prev (Section 2.2.4)
Type system	Abstract coeffect algebra (Section 2.2.1)	Concrete instances of the coeffect algebra (Section 2.2.4)
	Coeffect type system parameterized by the coeffect algebra (Section 2.2.2)	Typing for language-specific extensions (Section 2.2.4)
		Procedure for determining a unique typing derivation (Section 2.3)
Properties	Syntactic properties of coeffect λ-calculus (Section 2.4)	Uniqueness of the above (Section 2.3)

CHAPTER 5

	Coeffect calculus	Language-specific
Categorical	Indexed comonads (Sectiuon 3.2.4)	Examples including indexed product, list and maybe comonads (Section 3.2.5)
	Categorical semantics of coeffect λ-calculus (Section 3.2.6)	
Translational	Functional target language (Section 3.3.1)	
	Translation from coeffect λ-calculus to target language (Section 3.3.3)	Translation for language-specific extensions (prev, ?p) (Sections 3.4.1 and 3.4.2)
Operational	Abstract comonadically-inspired primitives (Section 3.3.3)	Concrete reduction rules for comonadically-inspired primitives (Sections 3.4.1 and 3.4.2)
		Reduction rules for language-specific extensions (prev, ?p) (Sections 3.4.1 and 3.4.2)
	Sketch of generalized syntactic soundness (Section 3.5)	Syntactic soundness (Sections 3.4.1 and 3.4.2)

2 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 2 and Chapter 3) and *structural coeffects* (Chapter 4) 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 2.2). We show that the system can be instantiated to obtain three of the systems discussed in Section 1.2, 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 2.3).

- We discuss equational properties of the calculus, covering type-preservation for call-by-name and call-by-value reduction (Section 2.4). We also extend the calculus with subtyping and pairs (Section 2.5).

2.1 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 1.2.3) and whole-context dataflow (Section 1.2.4) 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 1.2.1) 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 2.1.1) 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* [39]. Chapter 3 then links *coeffects* with *comonads* in the same way in which effect systems have been linked with monads [68]. The syntax and type system of the flat coeffect calculus follows a similar style as effect systems [63, 107], but differs in the structure of lambda abstraction as discussed briefly here and in Section 1.1.1 (the relationship with monads is further discussed in Section 3.6).

2.1.1 *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):

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

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:

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

The rule (*df1*) is admissible in a system that includes the (*df2*) rule. 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*).

2.2 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:

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

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 2.3 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 2.5. 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.

2.2.1 *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 2.5) 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 2.1.1, 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 2.5.1, 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 4), 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 2.4, when this is the case, we are able to prove certain syntactic properties of the calculus.

2.2.2 *Type system*

The type system for flat coeffect calculus is shown in Figure 15. Variables (*var*) and constants (*const*) are annotated with special values provided by the coeffect algebra.

The (*abs*) rule is defined as discussed in Section 2.1.1. 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 1, 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 $r \oplus (s \circledast t)$.

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

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

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

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

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

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

Figure 15: 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 2.5.2, 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 2.5. When discussing procedure for choocing unique typing in Section 2.3, we consider only the syntax-directed part of the system.

2.2.3 *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 2), it provides analysis of context dependence. As a semantics (Chapter 3), 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 3.2). 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 2, and *context capabilities* or just *available context* when using the semantic view, especially in Chapter 3.

2.2.4 *Examples of flat coeffects*

The flat coeffect calculus generalizes the three flat systems discussed in Section 1.2 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:

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

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

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

The (*param*) rule specifies that the accessed parameter $?p$ needs to be in the set of required parameters r. As discussed earlier, we use the same type num for all parameters, but it is 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 1.2.1. 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 1.2.3, 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:

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

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

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 $\bot$ element, that is $\mathbb{N}_\bot = \{\bot, 0, 1, 2, 3, \ldots\}$ such that $\forall n \in \mathbb{N}. \bot \leqslant n$. The flat coeffect algebra is $(\mathbb{N}_\bot, +, \mathit{max}, \mathit{min}, 0, \bot, \leqslant)$ where $m + n$ is $\bot$ whenever $m = \bot$ or $n = \bot$ and* min, max *treat $\bot$ as the least element.*

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

2.3 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 1.2.1), 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 2.2 reintroduces similar ambiguity because it allows multiple valid typing derivations.

Furthermore, the semantics of context-aware languages in Chapter **??** and also in Chapter 3 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 2.2.2 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.
- $\mathsf{f} : \mathsf{num} \xrightarrow{\{?x\}} \mathsf{num}$ – 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.

2.3.1 *Implicit parameters*

For implicit parameters we choose to follow the behaviour implemented by Haskell [61] 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 16. 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*)

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

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

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

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

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

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

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

Figure 16: 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 16 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 3 and we also implement this algorithm as discussed in Chatper **??**.

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

In order to prove the uniqueness of typing theorem (Theorem 2), we follow the standard approach [89] and first give the inversion lemma (Lemma 1).

Lemma 1 (Inversion lemma for implicit parameters)**.** *For the type system defined in Figure 16:*

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 16. □

Theorem 2 (Uniqueness of coeffect typing for implicit parameters)**.** *In the type system for implicit parameters defined in Figure 16, 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 (*app*), (*let*) and (*letpar*) similarly to (*abs*). □

Finally, we note that unique typing derivations obtained using the type system given in Figure 16 are valid typing derivation under the original flat coeffect type system in Figure 15.

Theorem 3 (Admisibility of unique typing for implicit parameters)**.** *If* $\Gamma;\Delta @ r \vdash e : \tau$ *(using the rules in Figure 16) then also* $\Gamma @ r \vdash e : \tau$ *(using the rules in Figure 15 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 16 is a special case of splitting two sets using $\cup$. □

2.3.2 *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:

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

This is the rule that we originally used for liveness and dataflow computations in Chapter 1. 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 $r = min(s, t)$. 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 2.5.3 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 15 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 3) and it is used in the implementation (Chatper **??**).

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 39 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 15 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 16. □

Theorem 5 (Uniqueness of coeffect typing for liveness and dataflow). *In the type system for liveness and dataflow defined in Figure 15 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*): $e = \lambda x{:}\tau_1.e_1$. Then $\tau = \tau_1 \xrightarrow{c} \tau_2$ 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 $\tau = \tau'$.

Cases (*var*), (*const*) are direct consequence of Lemma 4.

Cases (*app*) and (*let*) similarly to (*abs*). □

2.4 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.

2.4.1 *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:

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

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 2.4.2) and call-by-name (Section 2.4.3) 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. [17]. We give formal semantics of implicit parameters and dataflow in Chapter 3 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 [127], 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 2.4.2.

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.

2.4.2 *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:

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

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. □

2.4.3 *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 2.4.1, 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 $\lambda x.y$:

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

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 4.

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:

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

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:

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

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 1.2.2. 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 4 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 $\to_\eta$ 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* $e \to_\eta e'$ *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 [88].

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

$$(\textit{sub-fun})\ \frac{\tau'_1 <: \tau_1 \qquad \tau_2 <: \tau'_2 \qquad r' \geqslant r}{\tau_1 \xrightarrow{r} \tau_2 <: \tau'_1 \xrightarrow{r'} \tau'_2}$$

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

Figure 17: Subtyping rules for flat coeffect calculus

2.5 SYNTACTIC PROPERTIES AND EXTENSIONS

The flat coeffect algebra introduced in Section 2.2 requires a number of axioms. The axioms are required for three reasons – to be able to define the categorical structure in Section 3.2, to prove equational properties in Section 2.4 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 2.5.1), discuss what syntactic equivalences are permitted by the properties of $\wedge$ (Section 2.5.3) and we extend the calculus with pairs and units and discuss their syntactic properties (Section 2.5.4).

2.5.1 *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 15 (recall that for implicit parameters $\leqslant = \subseteq$):

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

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 17, 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:

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

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

2.5.2 *Typing of let binding*

Recall the (*let*) rule in Figure 15. It annotates the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ 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:

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

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 $\texttt{let}$ 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.

2.5.3 *Properties of lambda abstraction*

In Section 2.1.1, 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 2.3.2, 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*:

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

□

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 2.3.2), 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.

2.5.4 *Language with pairs and unit*

To focus on the key aspects of flat coeffect systems, the calculus introduced in Section 2.2 consists only of variables, abstraction, application and let bind-

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

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

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

Figure 18: 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:

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

The typing rules for pairs and the unit value are shown in Figure 18. 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.

2.6 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 1.2.

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 2.3 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.

3 SEMANTICS OF FLAT COEFFECTS

The *flat coeffect calculus* introduced in the previous chapter uniformly captures a number of context-aware systems outlined in Chapter 1. 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 [127] 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 3.2) 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 3.3). 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 3.4)

- 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 3.6).

3.1 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 2, define its *abstract comonadic semantics* and use it to define a translation that gives a *concrete operational semantics* to a number of concrete context-aware languages. The type system is used to guarantee that the resulting programs are correct. Finally, the development in this chapter is closely mirrored by the implementation presented in Chapter **??**, which implements the translation together with an interpreter for the target language.

The key claim of this thesis is that writing context-aware programs using coeffects is easier and 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 $\mathsf{int} \xrightarrow{\{?\mathsf{one},?\mathsf{two}\}} \mathsf{int}$. We call it in a context containing ?one and so the coeffect of the program is {?two}. The safety property for implicit parameters (Theorem **??**) guarantees that, when executed in a context that provides a value for the implicit parameter ?one, the program reduces to a value of 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 $\mathsf{num} \xrightarrow{1} \mathsf{num}$ meaning that it requires one past value (in addition to the current value). The safety property for dataflow (Theorem **??**) shows that, when called with a context that contains the required number of past values as captured by the coeffect type system, the function does not get stuck.

In summary, 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.

3.2 CATEGORICAL MOTIVATION

The type system of the flat coeffect calculus arises syntactically, as a generalization of the examples discussed in Chapter 1, 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 3.3.

3.2.1 *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 [127] and further explored by Atkey [7] and Vazou and Leijen [73]). 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 [115] 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 [53] who parameterizes monads in a similar way to our indexed comonads.

3.2.2 *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* [38] 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 2.3.

As a best known example, Moggi [68] 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 [115] showed that (*monoidal*) *comonads* uniformly capture the semantics of various kinds of context-dependent computations [115]. 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 $\tau_1 \to \tau_2$ performs *no* effects (requires no context) whereas $\tau_1 \to M\tau_2$ performs *some* effects and $C\tau_1 \to \tau_2$ requires *some* context[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 $C^r\tau_1 \to \tau_2$ using an *indexed comonad* C^r.

3.2.3 *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)* $\mathsf{C} : \mathcal{C} \to \mathcal{C}$
- counit *is a mapping* $\mathsf{C}\,\alpha \to \alpha$
- cobind *is a mapping* $(\mathsf{C}\,\alpha \to \beta) \to (\mathsf{C}\,\alpha \to \mathsf{C}\,\beta)$

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

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

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 $\mathsf{C}\,\alpha \to \beta$ into a function that takes a value with context, applies the context-dependent function to value(s) in the context and then propagates the context.

As mentioned earlier, Uustalu and Vene [115] 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 $\tau_1 \to M_1(M_2\ \tau_2)$. 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:*

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

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

3.2.4 *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*

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

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:

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

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 1.2.3, 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* $\mathsf{C}^{r}\alpha = \alpha \times (r \to \mathsf{num})$ *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:*

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

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.

3.2.5 *Flat indexed comonads*

Because of the asymmetry of λ-calculus (discussed in Section 1.1), 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 [68] requires a *strong* monad which has an additional operation $\mathsf{strength} : \alpha \times \mathsf{M}\beta \to \mathsf{M}(\alpha \times \beta)$. 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 [115] require *lax semi-monoidal* comonad. This structure requires an additional monoidal operation:

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

The m operation is needed in the semantics of lambda abstraction. 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* $\mathsf{C}^{r}\alpha \times \mathsf{C}^{s}\beta \to \mathsf{C}^{r \wedge s}(\alpha \times \beta)$
- $\mathsf{split}_{r,s}$ *is a family of mappings* $\mathsf{C}^{r \oplus s}(\alpha \times \beta) \to \mathsf{C}^{r}\alpha \times \mathsf{C}^{s}\beta$

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 4 and it makes the connection between the two easier to see.

EXAMPLES. All the examples of *indexed comonads* discussed in Section 3.2.4 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 [115]. 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:*

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

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* $\alpha \times (r \rightarrow \mathsf{num})$ *where* num *is the type of implicit parameter values. The additional operations are defined as:*

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

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 1.2.1. 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* $(\mathbb{N}, +, \mathit{max}, \mathit{min}, 0, 0, \leqslant)$*. 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:*

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

$$
\begin{aligned}
&\mathsf{cobind}_{m,n}\ f\langle a_0, \ldots a_{m+n} \rangle = \\
&\quad \langle f\langle a_0, \ldots, a_m \rangle, \ldots, f\langle a_n, \ldots, a_{m+n} \rangle \rangle \\[1ex]
&\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m \rangle, \langle b_0, \ldots, b_n \rangle) = \\
&\quad \langle (a_0, b_0), \ldots, (a_{\mathit{min}(m,n)}, b_{\mathit{min}(m,n)}) \rangle \\[1ex]
&\mathsf{split}_{m,n}\langle (a_0, b_0), \ldots, (a_{\mathit{max}(m,n)}, b_{\mathit{max}(m,n)}) \rangle = \\
&\quad (\langle a_0, \ldots, a_m \rangle, \langle b_0, \ldots, b_n \rangle)
\end{aligned}
$$

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:

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

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

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

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

Assuming the following auxiliary operations:

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

Figure 19: 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.

3.2.6 *Semantics of flat calculus*

In Section 1.2, 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 2.3, 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 2.3.

CONTEXTS AND TYPES. The modelling of contexts and functions generalizes the concrete examples discussed in Chapter 1. We use the family of mappings C^r as an (indexed) data-type that wraps the product of free variables of the context and the arguments of functions:

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

EXPRESSIONS. The definition of the semantics is shown in Figure 19. For consistency with earlier work [115, 76], 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 3.3).

The semantics of variable access and abstraction are the same as in the semantics of Uustalu and Vene [115], 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 3.3 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 2.2.2 – 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 19, 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 19. 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

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

REDUCTION RULES

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

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 20: Common syntax and reduction rules of the target language

to provide. In Section 3.5, 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 1.

Theorem 16 (Generalization). *Consider a typing derivation obtained according to the rules for finding unique typing derivations as specified in Section 2.3 for a coeffect language with liveness, dataflow or implicit parameters.*

The semantics obtained by instantiating the rules in Figure 19 with the concrete operations defined in Example 10, Example 11 or Example 12 is the same as the one defined in Figure 8, Figure 3 and Figure 10, respectively.

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

3.3 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.

3.3.1 *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 20. 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 20, 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 20.

3.3.2 *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 3.4.

We use the standard syntactic approach to type safety introduced by Milner [67]. Following Wright, Felleisen and Pierce [89, 129], 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* $\tau = \tau_1 \to \tau_2$ *then* $e = \lambda x.e'$ *for some* x, e'
3. *If* $\tau = \tau_1 \times \ldots \times \tau_n$ *then* $e = (v_1, \ldots, v_n)$ *for some* v_i

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

Lemma 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* $e \to e'$ *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:

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

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

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

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

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

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

Figure 21: Comonadically-inspired extensions for the target language

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

3.3.3 *Comonadically-inspired translation*

In Section 3.2, 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 21. We consider examples later in Section 3.4.

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:

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

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

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

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

Assuming the following auxiliary operations:

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

Figure 22: 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 21 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:

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

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

EXPRESSIONS. The rules shown in Figure 22 define how expressions of the source language are translated into the target language. The rules are very similar to those shown earlier in Figure 19. 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 19 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 **??**), 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 3.4, 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):*

$$\begin{array}{ccc} \dfrac{[\![\ (\ldots)\ ]\!]}{[\![@\,r \vdash e : \tau]\!]} & \begin{array}{c} = \\ = \end{array} & \dfrac{(\ldots)}{f} \end{array}$$

Then f *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. Given a judgement $x_1 : \tau_1 \ldots x_n : \tau_n @\,c \vdash e : \tau$, the translation constructs a function of type $C^c([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$.

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

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

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

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

3.4 SAFETY OF CONTEXT-AWARE LANGUAGES

The language defined in Figure 20 and Figure 21 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 3.4.1) and for implicit parameters (Section 3.4.2). 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 3.5.

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 $\mathsf{C}^{r}\tau$ and reduction rules for the comonadically-inspired operations ($\mathsf{cobind}_{s,r}$, $\mathsf{counit}_{\mathsf{use}}$, etc.).

3.4.1 *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 $\mathsf{C}^{r}\tau$ 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 23 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 and we also add a typing rule (*df*) that checks the types of the elements of the stream and also guarantees that the number of elements in the stream matches the number in the coeffect. The additional reduction rules mirror the semantics that we discussed 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 20 and comonadically-inspired primtives defined in Figure 21 and also concrete notion of comonadically-inspired value and reduction rules for dataflow as defined in Figure 23.

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 22 and Figure 23 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 $\mathsf{C}^{n+1}\tau$ 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:*

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

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

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

TRANSLATION

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

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 23: Additional constructs for modelling dataflow in the target language

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

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

Lemma 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 $\Gamma \vdash f : C^{m}\tau_1 \to \tau_2$ and $\Gamma \vdash \mathsf{Df}\langle v_0, \ldots v_{m+n}\rangle : C^{m+n}\tau$. The reduced expression has a type $C^{n}\tau_2$:

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

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

Theorem 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. □

3.4.2 *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 20) with comonadically-inspired extensions (Figure 21) 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 24 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 $\mathsf{Impl}(\nu, \{?p_1 \mapsto \nu_1, \ldots ?p_n \mapsto \nu_n\})$ which contains a value ν 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 24 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 2.3 always split implicit parameters into *disjoint sets*, so preferences do not come into play.

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 20) with the comonadically-inspired extensions (Figure 21) and the domain-specific extensions for implicit parameters defined in Figure 24. 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*

LANGUAGE SYNTAX

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

TYPING RULES

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

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

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

TRANSLATION

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

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

REDUCTION RULES

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

Figure 24: Additional constructs embedding implicit parameters into the language

program f *obtained using the rules in Figure 22 and Figure 24 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 $\{?\mathsf{p}\}$ which includes the parameter $?\mathsf{p}$ as required in order to use the (*lookup*) typing rule.

Case (*letimpl*): The type of *ctx* matches with the input type of $\mathsf{map}_{r\cup(s\setminus\{?\mathsf{p}\})}$. After duplication and splitting the context, ctx_1 and ctx_2 have types $C^{r}(\ldots)$ and $C^{s\setminus\{?\mathsf{p}\}}(\ldots)$, 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* $\tau = C^{\{?\mathsf{p}_1,\ldots,?\mathsf{p}_n\}}\tau_1$ *then* $e = \mathsf{Impl}(v,\{?\mathsf{p}_1 \mapsto v_1, \ldots, ?\mathsf{p}_n \mapsto v_n\})$

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

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*): $e = \mathsf{counit}_{\emptyset}(\mathsf{Impl}(v,\{\ \}))$. 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*): $e = \mathsf{cobind}_{r,s}\ \mathsf{f}\ (\mathsf{Impl}(v,\{?\mathsf{p}_1 \mapsto v_1, \ldots, ?\mathsf{p}_n \mapsto v_n\}))$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^{s}\tau_2$ and assumptions $\Gamma \vdash \mathsf{Impl}(v,\{?\mathsf{p}_1 \mapsto v_1, \ldots, ?\mathsf{p}_n \mapsto v_n\}) : C^{r}\tau$ and $\Gamma \vdash \mathsf{f} : C^{r}\tau_1 \to \tau_2$. The reduced expression has a type $C^{s}\tau_2$:

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{f} : C^{r}\tau_1 \to \tau_2 \qquad \Gamma \vdash \mathsf{Impl}(v,\{?\mathsf{p}_i \mapsto v_i \mid \mathsf{p}_i \in r\}) : C^{r}\tau_1}{\Gamma \vdash \mathsf{f}\ (\mathsf{Impl}(v,\{?\mathsf{p}_i \mapsto v_i \mid \mathsf{p}_i \in r\})) : \tau_2}}{\Gamma \vdash \mathsf{Impl}(\mathsf{f}\ (\mathsf{Impl}(v,\{?\mathsf{p}_i \mapsto v_i \mid \mathsf{p}_i \in r\})),\{?\mathsf{p}_i \mapsto v_i \mid \mathsf{p}_i \in s\}) : C^{s}\tau_2}$$

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

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, $e_1 = \mathsf{Impl}(v, \{\,\})$ 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 $\mathsf{cobind}_{r,s}\ v\ \_$. If both are values, then from Lemma 30, we have $e_2 = \mathsf{Impl}(v, \{?p_i \mapsto v_i \mid ?p_i \in r \cup s\})$ 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 $(\mathsf{Impl}(v, \{?p_i \mapsto v_i \mid ?p_i \in r\}), \mathsf{Impl}(v', \{?p_i \mapsto v_i \mid ?p_i \in s\}))$ 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 $\mathsf{Impl}(v, \{?p_1 \mapsto v_1, \ldots, ?p_{i-1} \mapsto v_{i-1}, ?p_i \mapsto \_, ?p_{i+1} \mapsto v_{i+1}, \ldots ?p_n \mapsto e_n\})$. 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. □

3.5 GENERALIZED SAFETY OF COMONADIC EMBEDDING

In Section **??** and Section 3.4.2, 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 3.4 would be guaranteed directly by the target language. This includes dependently-typed languages such as Idris or Agda [14, 13], but type-level numerals and sets can also be encoded in the Haskell type system [80].

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 2. 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:

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

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 $\mathsf{C}^{\mathsf{L}}\tau$ while a dead context has a type $\mathsf{C}^{\mathsf{D}}\tau$.

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

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:

$$(\mathit{counit}) \quad \mathsf{counit}_{\mathsf{L}}\ (\mathsf{Just}\ v) \rightsquigarrow v$$

$$\begin{array}{ll}
(\mathit{cobind}\text{-}1) & \mathsf{cobind}_{\mathsf{L},\mathsf{L}}\ \mathsf{f}\ (\mathsf{Live}\ v) \rightsquigarrow \mathsf{Live}\ (\mathsf{f}\ v) \\
(\mathit{cobind}\text{-}2) & \mathsf{cobind}_{\mathsf{D},\mathsf{L}}\ \mathsf{f}\ (\mathsf{Live}\ v) \rightsquigarrow \mathsf{Live}\ (\mathsf{f}\ \mathsf{Dead}) \\
(\mathit{cobind}\text{-}3) & \mathsf{cobind}_{\mathsf{L},\mathsf{D}}\ \mathsf{f}\ v \rightsquigarrow \mathsf{Dead} \\
(\mathit{cobind}\text{-}4) & \mathsf{cobind}_{\mathsf{D},\mathsf{D}}\ \mathsf{f}\ v \rightsquigarrow \mathsf{Dead}
\end{array}$$

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 $\mathsf{C}^{\mathsf{L}}\tau$ and will always be a value Live v.

Similarly, the when reducing $\mathsf{cobind}_{r,s}$, the typing ensures that the value passed as the second argument is of type $\mathsf{C}^{r \circledast s}\tau_1$. 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 $\mathsf{C}^{s}\tau_2$, 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 [64, 131, 19], which encode some of the features known from dependently-typed languages such as Agda [13]. 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 21:

```
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 [60] 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 M a 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 C r a where the annotation r specifies what context may be required. Without the annotation, a type C a would represent a comonadic context that has *all possible context* available, which is rarely useful in practice.

3.6 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 3.2.4) that deserve additional discussion.

In Section 3.6.1, we discuss related approaches to adding indices to categorical structures (mostly monads). In Section 3.6.2, we discuss a question that often arises when discussing coeffects and that is *when is a coeffect (not) an effect?*

3.6.1 *Indexed categorical structures*

Ordinary comonads have the *shape preservation* property [79]. 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 fail to form a comonad without indexing. In the rest of the section, we look at work that discusses indexing in the context of *monads.*

FAMILIES OF MONADS. When linking effect systems and monads, Wadler and Thiemann [68] propose a *family of monads* as the categorical structure. The dual structure, *family of comonads*, is defined as follows.

Definition 8. *A* family of comonads *is formed by triples* $(C^r, \mathsf{cobind}_r, \mathsf{counit}_r)$ *for all* r *such that each triple forms a comonad. Given* r, r' *such that* $r \leqslant r'$*, there is also a mapping* $\iota_{r',r} : C^{r'} \to C^r$ *satisfying certain coherence conditions.*

A *family of comonads* is not as expressive as an *indexed comonads*. Many indexed comonads cannot be captured by a family of comonads. This is 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 $C^{D}\alpha = 1$). However, given a family of comonads and indices such that $r \leqslant r \circledast s$, we can define an indexed comonad. Briefly, to define $\mathsf{cobind}_{r,s}$ of an indexed comonad, we use $\mathsf{cobind}_{r \circledast s}$ from the family, together with two lifting operations: $\iota_{r \circledast s, r}$ and $\iota_{r \circledast s, s}$.

PARAMETERIC EFFECT MONADS. Parametric effect monads introduced by Katsumata [53] (independently to our indexed comonads) are closely related to our definition. Although presented in a more general categorical framework (and using monads), the model (i) defines the unit operation only on the unit of a monoid and (ii) the bind operation composes effect annotations using the provided monoidal structure.

3.6.2 *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 dataflow). In terms of program analysis [57], monas capture forward dataflow analyses and comonads correspond to backward dataflow analyses.

- 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). Effect systems may use a richer algebraic structure to support additional language constructs such as conditionals [72, 92], but not for abstraction and application.

- 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 lambda abstraction in monadic effect systems always defer all effects – creating a function value has no effect.

Despite the differences, our implicit paremeters resemble, in many ways, the *reader* monad. As discussed in Section 3.6.3, the *reader* monad is semantically equivalent to the *product* comonad when we consider just sequential composition. For a language with lambda abstraction, we need a slight extension to the usual treatment of monads in order to model implicit parameters using a monad.

3.6.3 *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 $r \to \texttt{num}$ 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 [77], 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 shows an intriguing relationship between the *product comonad* and *reader monad*, but it cannot be extended beyond that. In particular, comonads that model dataflow computations or liveness do not have a corresponding monadic structure. 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 $\texttt{merge}_{r,s}$ operation to model context merging. This can be supported in monadic computations by adding an aditional operation discussed next.

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

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

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:

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

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 $\mathsf{M}^{r}\tau$:

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

This suggests that the *reader monad* is a special case among monads. Our work suggests that passing read-only information to a computation is better captured by a product comonad, which also matches the intuition – read-only information is a *contextual capability*.

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 [108].

3.7 CONCLUSIONS

In the previous chapter, we defined a *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 **??**.

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 implementations of comonadic operations for dataflow and implicit parameters and we presented a 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 1.

4 THE STRUCTURAL COEFFECT CALCULUS

In Chapter 1, we discussed two notions of context. Context-aware programming languages that capture whole-context properties were generalized by the *flat coeffect calculus* in Chapters 2 and 3. Here, we consider per-variable contextual properties and we introduce the *structural coeffect calculus*.

The flat coeffect system captures a number of interesting use-cases (implicit parameters, liveness and dataflow), but provides relatively imprecise approximation and weak syntactical properties. Some of those examples can be also seen as per-variable properties. For those, structural coeffect systems give a more precise information about the context. However, we also look at other applications that arise from the work on sub-structural logics, a pathway to coeffects discussed in Section **??**.

We mirror the development for flat coeffect calculus and develop a small calculus with a type system that captures per-variable contextual properties. We outline its categorical semantics and use it as a basis for a translation that turns well-typed programs in context-aware languages into well-typed programs in a simple target functional language. We prove syntactic safety for a sample target language, showing that "well-typed context-aware programs do not get stuck".

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We present a *structural coeffect calculus* as a type system that is parameterized by a *structural coeffect algebra* (Section 4.2. We show how the system captures fine-grained liveness and dataflow information, as well as a calculus for bounded reuse (checking how many times is a variable accessed).

- We present a syntax-directed version of the calculus that is used to obtain unique typing derivation for programs in structural coeffect calculus (Section 4.3). Unlike in flat systems, the procedure for choosing a unique typing derivation is common to all structural systems.

- We discuss the equational theory of the calculus. For structural coeffects, the substitution lemma holds for all instances of the calculus. Thus, we prove the type-preservation property for all structural calculi for both call-by-name and call-by-value (Section 4.4).

- We extend indexed comonads introduced in the previous chapter to *structural indexed comonads* and use them to provide the semantics of structural coeffect calculus (Section 4.5). As with the flat version, the theory serves as a motivation for syntactic translational semantics.

- We give a *translational semantics* (Section 4.6) that translates programs from the structural coeffect calculus into a simple functional language with uninterpreted comonadically-inspired primitivies. We give concrete operational semantics for the target language for one of our sample langauges and show that well-typed programs, produced by translation from the coeffect calculus do not get stuck.

4.1 INTRODUCTION

Compared to Chapter 2, structural coeffect calculi we consider are more homogeneous and so finding the common pattern is easier. However, the systems are more complicated as they need to keep annotations attached to individual variables and thus require explicit structural rules. Before looking at the system, we briefly consider the most important related work.

4.1.1 *Related work*

In the previous chapter, we discussed the correspondence between coeffects and effects (and between comonads and monads). As noted in Section 1.1.3, the λ-calculus is asymmetric in that an expression has multiple inputs (variables in the context), but just a single result (the resulting value) and so monads and effects have no notion directly corresponding to structural coeffect systems.

The work in this chapter is more closely related to sub-structural type systems [128]. Sub-structural systems remove some or all of *weakening*, *contraction* and *exchange* rules. In contrast, our systems keep all three structural rules, but use them to manipulate the coeffect annotations in a way that matches the variable manipulations.

Our work follows the "language semantics" style in that we provide a structural semantics to the terms of ordinary λ-calculus. By contrast the most closely related work has been done in the meta-language style, which extends the terms and types of a language with constructs for explicitly manipulating the context. This includes Contextual Modal Type Theory (CMTT) [71], where variables may be of a type $A[\Psi]$ denoting a value of type A that requires context Ψ. In CMTT, $A[\Psi]$ is a first-class type, while structural coeffect systems do not expose coeffect annotations as stand-alone types (indexed comonads only appear in the semantics).

Our structural coeffect systems annotate the whole variable context with a *vector* of annotations. For example, a context with variables x and y annotated with s and t, respectively is written as $x:\tau_1, y:\tau_2 @ \langle s, t \rangle$. A benefit of this approach is that the typing judgements have the same structure as those of the flat coeffect calculus. As discussed in Chapter **??**, this makes it possible to unify the two systems.

4.2 STRUCTURAL COEFFECT CALCULUS

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

Scalar annotations are written as r, s, t (following the style used in the previous chapter). Functions always have exactly one input variable and so they are annotated with a single coeffect scalar. Thus the expressions and types of structural coeffect calculi are the same as in the previous chapter. The only difference is that r, s, t now range over values of a *structural coeffect* *scalar* rather than over values of a *flat coeffect algebra* (the definitions are related, but not the same).

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

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

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

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

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

4.2.1 *Structural coeffect algebra*

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

Definition 9. *A* structural coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, binary operations* $\circledast, \oplus$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *and* $(\mathcal{C}, \oplus, \mathsf{ign})$ *are monoids and a binary relation* $\leqslant$ *such that* $(\mathcal{C}, \leqslant)$ *is a pre-order. That is, for all* $r, s, t \in \mathcal{C}$*:*

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

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 addition, we require the following two properties of $\mathsf{ign}, \oplus$ *and* $\leqslant$*:*

$$\begin{aligned} & r \leqslant r' \ \Rightarrow \ \forall s.\ (r \circledast s) \leqslant (r' \circledast s) \\ & \mathsf{ign} \leqslant (\mathsf{ign} \circledast r) \end{aligned}$$

The structural coeffect scalar structure resembles flat coeffect algebra, but it differs in a number of important ways:

- The $\oplus$ operation of structural coeffect scalar is not required to be idempotent. In structural systems, we can track individual variable accesses and not requiring idempotence allows interesting systems such as the one for bounded reuse (Section 4.5.5).

- In the flat coeffect calculus, we used the $\wedge$ operation to merge the annotations of contexts available from the declaration site and the call site or, in the syntactic reading, to split the context requirements. In structural systems, this is handled directly by vectors and so $\wedge$ is no longer needed.
- We now require two properties that relate the ordering $\leqslant$ with coeffects composed via $\oplus$ and with ign. These were previously required for bottom-pointed substitution lemma (Lemma 9). Here, we require them for all structural coeffects as it holds for all our examples; it is needed for the structural version of the substitution lemma (Lemma 40).

In the structural coeffect calculus, the scalar coeffect structure is supplemented by a vector structure. Combining and splitting of coeffects becomes just vector concatenation or splitting, respectively; these are provided by the tensor product. The operations on vectors are indexed by integers representing the lengths of the vectors. The additional structure required by the type system for structural coeffect calculi is given by the following definition.

Definition 10. *A structural coeffect algebra is formed by a structural coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *equipped with the following additional structures:*

- *Coeffect vectors* $\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t}$*, ranging over structural coeffect scalars indexed by vector lengths* $m, n \in \mathbb{N}$*.*
- *An operation that constructs a vector from scalars indexed by the vector length* $\langle - \rangle_n : \mathcal{C} \times \ldots \times \mathcal{C} \rightarrow \mathcal{C}^n$ *and an operation that returns the vector length such that* $len(\boldsymbol{r}) = n$ *for* $\boldsymbol{r} : \mathcal{C}^n$
- *A pointwise extension of the* $\circledast$ *operator written as* $t \circledast \boldsymbol{s}$ *such that* $t \circledast \langle r_1, \ldots, r_n \rangle = \langle t \circledast r_1, \ldots, t \circledast r_n \rangle$*.*
- *An indexed tensor product* $\times_{n,m} : \mathcal{C}^n \times \mathcal{C}^m \rightarrow \mathcal{C}^{n+m}$ *that is used in both directions – for vector concatenation and for splitting – which is defined as* $\langle r_1, \ldots, r_n \rangle \times_{n,m} \langle s_1, \ldots, s_m \rangle = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$

The fact that the tensor product $\times_{n,m}$ is indexed by the lengths of the two vectors means that we can use it unambiguously for both concatenation of vectors and for splitting of vectors, provided that the lengths of the resulting vectors are known. In the following text, we usually omit the indices and write just $\mathbf{r} \times \mathbf{s}$, because the lengths of the coeffect vectors can be determined from the lengths of the matching free variable context vectors. More generally, we could see the the coeffect annotations as *containers* [3]. This approach is used in Chapter **??** to unify flat and structural systems.

4.2.2 *Structural coeffect types*

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

a.) Syntax-driven typing rules:

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

$$(\mathit{const})\ \frac{}{()\,@\,\langle\rangle \vdash n : \mathsf{num}}$$

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

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

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

b.) Structural rules for context manipulation:

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

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

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

c.) Non-syntax-directed subcoeffecting rule:

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

Figure 25: Type system for the structural coeffect calculus

SYNTAX-DRIVEN RULES. The syntax-driven rules of the type system are shown in Figure 25 (a). The variable access rule (*var*) annotates the corresponding variable as being accessed using use. Note that, as in sub-structural systems, the free variable context contains *only* the accessed variable. Other variables can be introduced using explicit weakening. Constants (*const*) are type checked in an empty variable context, which is annotated with an empty vector of coeffect annotations.

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

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

STRUCTURAL RULES. The rules shown in Figure 25 (b) are not syntax-directed and allow different transformation of the free variable context. They correspond to the transformations known as *weakening*, *exchange* and *contraction* from sub-structural systems.

Rule (*weak*) allows adding a variable to the context, extending the coeffect vector with ign to mark it as unused, (*exch*) provides a way to rearrange variables in the context, performing the same reordering on the coeffect vector. Finally recall that variables in the free variable context are required to be *unique*. The (*contr*) rule allows re-using a variable. We can type check sub-expressions using two separate variables and then unify them using substitution. The resulting variable is annotated with $\oplus$ and it is the only place in the structural coeffect system where context requirements are combined (semantically, this is where the available context is shared).

SUBCOEFFECTING. Finbally, Figure 25 (c) shows the subcoeffecting rule (*sub*). As with the flat coeffect calculus, we include it mainly to guide the understanding of the system, but we treat it as an additional rule and do not discuss it in the subsequent theory. The rule would be needed in languages with conditionals and other language features. It is worth noting that allowing subcoeffecting on coeffect scalars belonging to individual variables is sufficient for structural systems.

4.2.3 *Understanding structural coeffects*

The type system for structural coeffects appears more complicated when compared to the flat version, but it is in many ways simpler – it removes the ambiguity arising from the use of $\wedge$ in lambda abstraction and, as discussed in Section 4.4, has a cleaner equational theory.

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

This suggests that $\wedge$ and $\oplus$ serve two roles in flat coeffects. First, they are used as over-approximations and under-approximations of $\times$. This is demonstrated by the (*approximation*) requirement introduced in Section 2.4.2, which requires that $r \wedge t \leqslant r \oplus t$. Semantically, flat abstraction combines two values representing the available context, potentially discarding parts of it (under-approximation), while flat application splits the available context (a single value), potentially duplicating parts of it (over-approximation)[1].

Secondly, the operator $\oplus$ is used when the semantics passes a given context to multiple sub-expressions. In flat systems, the context is shared in (*app*) and (*pair*), because the sub-expressions may share variables. In structural systems, the sharing is isolated into an explicit contraction rule.

LET BINDING. The other aspect that makes structural systems simpler is that they remove the need for separate let binding. As discussed in Section 2.5.2, flat calculi include let binding that gives a *more precise* typing than combination of abstraction and application. Structural coeffect systems avoid this issue.

[1] Because of this duality, earlier version of coeffect systems [84] used $\wedge$ and $\vee$.

Remark 36 (Let binding)**.** *In a structural coeffect calculus, the typing of* `let` $x = e_1$ `in` e_2 *can be seen as a derived rule, i. e. its typing is equivalent to the typing of the expression* $(\lambda x.e_2)\ e_1$.

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

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

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

4.2.4 *Examples of structural coeffects*

The structural coeffect calculus above can be instantiated to obtain the 3 structural coeffect calculi presented in Section 1.3. Two of them – structural dataflow and structural liveness provide a more precise tracking of properties that can be tracked using flat systems. Formally, any flat coeffect algebra can be turned into a structural coeffect scalar (by dropping the $\wedge$ operator), but this does not always give us a meaningful system – for example, implicit parameters could be attached to individual variables, but, to our knowledge, this has not been used in any practical system.

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

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

Example 14 (Structural dataflow)**.** *In dataflow, context is annotated with natural numbers and the structural coeffect scalar is formed by* $(\mathbb{N}, +, max, 0, 0, \leqslant)$.

For the two examples that have both flat and structural version, obtaining the structural coeffect algebra is easy. As shown by the examples above, we simply omit the $\wedge$ operation. The laws required by a structural coeffect algebra are the same as those required by the flat version and so the above definitions are both valid. Similar construction can be used for the *optimized dataflow* example from Section 2.2.4.

It is important to note that this gives us a systems with *different* properties. The information are now tracked per-variable rather than for entire contexts. For dataflow, we also need to adapt the typing rule for the `prev` construct. Here, we write $+$ for a pointwise extension of the $+$ operator, such that $\langle r_1, \ldots, r_n \rangle + k = \langle r_1 + k, \ldots, r_n + k \rangle$.

$$(\mathit{prev})\ \dfrac{\Gamma @ \mathbf{r} \vdash e : \tau}{\Gamma @ \mathbf{r} + 1 \vdash \texttt{prev}\ e : \tau}$$

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

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

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

$$(\mathit{const})\ \frac{}{(\,) \mathbin{@} \langle\rangle \vdash n : \mathsf{num}}$$

$$(\mathit{app})\ \frac{\Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \qquad \Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1}{\Gamma \mathbin{@} \mathbf{c} \vdash e_1\ e_2 : \tau_2} \quad \Gamma \mathbin{@} \mathbf{c} = \mathsf{mergevars}(t, \Gamma_1 \mathbin{@} \mathbf{r}, \Gamma_2 \mathbin{@} \mathbf{s})$$

$$(\mathit{abs})\ \frac{\Gamma_1 \mathbin{@} \mathbf{t} \vdash e : \tau_2}{\Gamma_2 \mathbin{@} \mathbf{r} \vdash \lambda x : \tau_1 . e : \tau_1 \xrightarrow{s} \tau_2} \quad (\Gamma_2 \mathbin{@} \mathbf{r}), \tau_1, s = \mathsf{findvar}_x(\Gamma_1 \mathbin{@} \mathbf{t})$$

$$\begin{array}{l}
\mathsf{exch}_x(\Gamma \mathbin{@} \mathbf{t}) \ = \ (\Gamma_1, \Gamma_2 \mathbin{@} \mathbf{t}_1 \times \mathbf{t}_2), \tau, s \quad \text{where} \\
\qquad \mathit{len}(\Gamma_1) = \mathit{len}(\mathbf{t}_1) \text{ and } \mathit{len}(\Gamma_2) = \mathit{len}(\mathbf{t}_2) \\
\qquad x{:}\tau \in \Gamma \text{ and } \Gamma \mathbin{@} \mathbf{t} = \Gamma_1, x{:}\tau, \Gamma_2 \mathbin{@} \mathbf{t}_1 \times \langle s \rangle \times \mathbf{t}_2 \\[1ex]
\mathsf{findvar}_x(\Gamma \mathbin{@} \mathbf{t}) \ = \ (\Gamma \mathbin{@} \mathbf{t}), \tau, \mathsf{ign} \qquad (\text{otherwise})
\end{array}$$

$$\begin{array}{l}
\mathsf{mergevars}(t, \Gamma_1 \mathbin{@} \mathbf{r}, \Gamma_2 \mathbin{@} \mathbf{s}) \ = \ \Gamma_1', \Gamma_2', \Gamma \mathbin{@} \mathbf{r}' \times (t \circledast \mathbf{s}') \times \mathbf{c} \quad \text{where} \\
\qquad \Gamma_1 \mathbin{@} \mathbf{r} = x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle \\
\qquad \Gamma_2 \mathbin{@} \mathbf{s} = y_1{:}\tau_1', \ldots, y_m{:}\tau_n' \mathbin{@} \langle s_1, \ldots, s_m \rangle \\
\qquad \Gamma \mathbin{@} \mathbf{c} = z_1{:}\tau_1'', \ldots, z_k{:}\tau_k'' \mathbin{@} \langle c_1, \ldots, c_k \rangle \\
\qquad\quad \text{such that } \forall l \in \{1 \ldots k\}\ \exists i, j.\ (z_l{:}\tau_l'' = x_i{:}\tau_i = y_j{:}\tau_j') \\
\qquad\qquad \text{and } c_l = r_i \oplus (t \circledast s_j) \\
\qquad \Gamma_1 \mathbin{@} \mathbf{r} = x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle \text{ such that } x_i{:}\tau_i \notin \Gamma \\
\qquad \Gamma_2 \mathbin{@} \mathbf{s} = y_1{:}\tau_1', \ldots, y_m{:}\tau_n' \mathbin{@} \langle s_1, \ldots, s_m \rangle \text{ such that } y_i{:}\tau_i' \notin \Gamma
\end{array}$$

Figure 26: Syntax-directed type system for the structural coeffect calculus

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

Similarly to the structural calculus for dataflow, the calculus for bounded variable reuse annotates each variable with an integer. However, the integer denotes how many times is the variable *accessed* rather than how many *past values* are needed. The resulting type system is the one shown in Figure 11 in Chapter 1.

4.3 CHOOSING A UNIQUE TYPING

In the structural coeffect calculus, the lambda abstraction rule does not introduce ambiguity in the typing. This is in contrast with flat coeffect systems (most importantly, the one for implicit parameters), where lambda abstraction allowed arbitrary splitting of context requirements. In structural coeffect systems, the context requirements placed on the call-site (attached to the function type) are those of the bound variable.

However, the type system for structural coeffect calculus in Figure 25 introduces another kind of ambiguity due to the fact that non syntax-directed structural rules can be applied repeatedly and in arbitrary order. As with the semantics for flat coeffect calculus in Chapter 3, we define the semantics of the structural coeffect calculus over a *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.

4.3.1 *Syntax-directed type system*

In order to choose a unique typing derivation, we follow the example of sub-structural type systems [128] and introduce a syntax-directed version of the type system (also called *algorithmic*). This replaces the non-syntax-directed rules for weakening, exchange and contraction with additional logic in places where contexts are combined. Given a typing derivation in the syntax-directed type system, we can then produce a matching typing derivation in the original type system.

The syntax-directed version of the type system is shown in Figure 26. The typing rules for variables (*var*) and constants (*const*) are the same as before. The two interesting rules are lambda abstraction and application.

LAMBDA ABSTRACTION. In lambda abstraction (*abs*) in Figure 25, we assume that the bound varible is the last variable of the context. In the syntax-directed system, we do not make the same assumption. Instead, we use an auxiliary relation $\mathsf{findvar}_x$ that takes a typing context $\Gamma @ \mathbf{t}$ and returns a context with the variable x removed together with its type and coeffect.

The $\mathsf{findvar}_x$ relation is defined by two disjoint cases. If the variable x is not present in the context, the context is returned as is and the returned coeffect is ign. This case corresponds to the (*weak*) structural rule. If the variable is not found in the context, the relation does not place any restrictions on the type of the variable and so τ can be any type (this is the only reason why $\mathsf{findvar}$ is a relation rather than a function). If the variable x appears in the context then $\mathsf{findvar}_x$ removes it together with its corresponding coeffect annotation and returns it as the result.

FUNCTION APPLICATION. In the (*app*) rule in Figure 25, we assume that the variable contexts of the two sub-expressions can be merged. This requires that they contain disjoint variables, which can be always obtained by exchange and contraction. In the syntax-driven systenm, we merge coeffects of shared variables explicitly. This is done in the $\mathsf{mergevars}$ function.

The $\mathsf{mergevars}$ function returns context consisting of three parts. Parts Γ_1 and Γ_2 represent variables that appear only in the first or the second context; part Γ contains common variables. The coeffect annotations corresponding to Γ_1 are the original annotations from $\mathbf{r}$; the coeffects corresponding to Γ_2 are composed with the coeffect of the function value $t \circledast \mathbf{s}'$ as in the original (*app*) rule. Finally, for shared variables, the coeffect is obtained by pointwise composition (as in contraction) of the coeffect for the two contexts $r_i \oplus (t \circledast s_j)$. The first coeffect corresponds to the context requirements in the sub-expression e_1 and the second coeffect corresponds to the function argument e_2 sequentially composed with the coeffect t of the function (as in ordinary application rule).

4.3.2 *Properties*

The syntax-directed type checking presented in the previous section gives a unique typing derivation that can be automatically turned into a valid typing derivation of the original type system presented in Figure 25. This gives us a unique typing derivation for the structural coeffect calculus. As with the unique typing derivation for the flat coeffect system, this is used to give semantics of terms of the structural coeffect calculus. We also note that a well-typed program in the original type system has a typing derivation in the syntax-driven version.

As when discussing uniqueness of typing for flat coeffect systems (Section 2.3), we first give the inversion lemma (Lemma 37) and then prove the uniqueness of typing theorem (Theorem 38).

Lemma 37 (Inversion lemma for syntax-directed structural coeffects)**.** *For the type system defined in Figure 26:*

1. *If* $\Gamma @ \boldsymbol{c} \vdash x : \tau$ *then* $\Gamma = x{:}\tau$ *and* $\boldsymbol{c} = \langle \mathsf{use} \rangle$.
2. *If* $\Gamma @ \boldsymbol{c} \vdash n : \tau$ *then* $\Gamma = ()$ *and* $\tau = \mathsf{num}$ *and* $\boldsymbol{c} = \langle\rangle$.
3. *If* $\Gamma @ \boldsymbol{c} \vdash e_1\ e_2 : \tau_2$ *then there is some* $\Gamma_1, \Gamma_2, \tau_1$ *and some* $t, \boldsymbol{r}, \boldsymbol{s}$ *such that* $\Gamma_1 @ \boldsymbol{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2$ *and* $\Gamma_2 @ \boldsymbol{s} \vdash e_2 : \tau_1$ *and also* $\Gamma @ \boldsymbol{c} = \mathsf{mergevars}(t, \Gamma_1 @ \boldsymbol{r}, \Gamma_2 @ \boldsymbol{s})$.
4. *If* $\Gamma @ \boldsymbol{c} \vdash \lambda x{:}\tau_1.e : \tau$ *then there is some* Γ', τ_2 *and some* $s, \boldsymbol{t}$ *such that* $\Gamma' @ \boldsymbol{t} \vdash e : \tau_2$ *and* $\tau = \tau_1 \xrightarrow{s} \tau_2$ *and also* $(\Gamma @ \boldsymbol{c}), \tau_1, s = \mathsf{findvar}_x(\Gamma_1 @ \boldsymbol{t})$.

Proof. Follows from the individual rules given in Figure 26. □

Theorem 38 (Uniqueness of syntax-directed structural coeffects)**.** *In the syntax-directed type system for structural coeffects defined in Figure 26, when* $\Gamma @ \boldsymbol{r} \vdash e : \tau$ *and* $\Gamma @ \boldsymbol{r}' \vdash e : \tau'$ *then* $\tau = \tau'$ *and* $\boldsymbol{r} = \boldsymbol{r}'$.

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

Case (*abs*): $e = \lambda x : \tau_1.e_1$. Then $\tau = \tau_1 \xrightarrow{c} \tau_2$ for some τ_2 and $\Gamma' @ \mathbf{t} \vdash e : \tau_2$ for some $\Gamma', \mathbf{t}$ and also $(\Gamma @ \mathbf{c}), \tau_1, s = \mathsf{findvar}_x(\Gamma' @ \mathbf{t})$. By case (4) of Lemma 37, the final rule of the derivation (B) must have also been (*abs*) and this derivation has a sub-derivation with a conclusion $\Gamma @ \mathbf{c}' \vdash e : \tau_2'$. By the induction hypothesis $\tau_2 = \tau_2'$ and $\mathbf{c} = \mathbf{c}'$ and therefore also so $\tau = \tau'$. Although $\mathsf{findvar}_x$ is a relation, it allows only one possible result (because the type of the bound variable matches the type annotation).

Cases (*var*), (*const*) are direct consequence of Lemma 37.

Casea (*app*) similarly to (*abs*). □

As noted earlier, unique typing derivations obtained using the type system given in Figure 26 can be automatically turned into typing derivations of the original (non-syntax-directed) structural coeffect type system in Figure 25. The following remark provides the details.

Remark 39 (Admisibility of unique typing for implicit parameters)**.** *If* $\Gamma @ \boldsymbol{r} \vdash e : \tau$ *(using the rules in Figure 26) then there is a uniqye typing derivation using the typing rules from Figure 25 with a conclusion* $\Gamma @ \boldsymbol{r} \vdash e : \tau$ *obtained by induction over the original typing derivation as folows:*

Case (var), (const)*: The resulting typing derivation uses the corresponding rule of the non-syntax-directed type system.*

Case (abs)*: Take the typing derivation for the sub-expression* e*. If the variable* x *does not appear in* Γ_1*, apply* (weak) *followed by* (abs)*. Otherwise assume* $\Gamma_1 = x_1 : \tau_1, \ldots, x_n : \tau_n$ *and* $x = x_i$*. Apply* (exch) *repeatedly on variables* x_i, x_{i+1} *then* x_i, x_{i+2} *and so on until it is applied on* x_i, x_n*. At this point,* x_i *is the* *last variable of the vector and we can apply* (abs)*. This produces the same consequent as the one in the original typing derivation.*

Case (app)*: Take the typing derivations for the sub-expressions e_1 and e_2 in free-variable contexts Γ_1 and Γ_2. For each variable x that appears in both Γ_1 and Γ_2, rename the variable to a fresh name x' in e_1 and to another fresh name x'' in e_2 and their typing derivations. Now we have disjoint contexts and we can apply* (app) *on the target derivations.*

Next, apply (exch) *until x' and x'' are last two variables in the vector and apply* (contr)*, renaming both x' and x'' to the original name x. Repeat this step for all variables that were renamed. The resulting variable context is Γ and the resulting coeffect annotation is the same as in the original typing derivation.*

4.4 SYNTACTIC EQUATIONAL THEORY

The properties of the structural coeffect algebra guarantee that certain equational properties hold in all instances of the structural coeffect calculus. In this section, we look at these common properties that we get "for free" from the structural coeffect calculus. We start the discussion by briefly considering the key aspects that make the equational theory of flat and structural coeffects different.

4.4.1 *From flat coeffects to structural coeffects*

When discussing syntactic reductions for the flat calculus in Section 2.4, we noted that call-by-name reduction does not, in general, preserve typing for all flat coeffect calculi. In the structural coeffect calculus, β-reduction and also η-expansion preserve typing for all instances of the calculus. Using the terminology of Pfenning and Davies [88], the structural coeffect calculus satisfies both the *local soundness* and the *local completeness* properties.

SUBSTITUTION FOR FLAT COEFFECTS. The less obvious (*top-pointed*) variant of the substitution lemma for flat coeffects (Lemma 9) required all operations of the flat coeffect algebra to coincide. This enables substitution to preserve the type of expressions, because all additional requirements arising as the result of the substitution can be associated with the declaration context. For example, consider the following example where implicit parameter ?offset is substituted for the variable y:

$$\begin{array}{rll} y : \mathsf{int} @ \emptyset \vdash \lambda x.y + \mathsf{?total} & : \mathsf{int} \xrightarrow{\{\mathsf{?total}\}} \mathsf{int} & (\textit{before}) \\ () @ \{\mathsf{?offset}\} \vdash \lambda x.\mathsf{?offset} + \mathsf{?total} & : \mathsf{int} \xrightarrow{\{\mathsf{?total}\}} \mathsf{int} & (\textit{after}) \end{array}$$

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

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

SUBSTITUTION FOR STRUCTURAL COEFFECTS. In contrast, the substitution lemma (Lemma 40, shown later on page 107) for structural coeffects can be proven because structural coeffect systems provide enough *precision* to identify exactly with which variable should a context requirement be associated.

The following example shows a situation similar to the previous one. Here, we use structural dataflow calculus (writing $\mathsf{prev}\ e$ to obtain previous value of the expression e) and we substitute $w+z$ for y:

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

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

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

4.4.2 *Holes and substitution lemma*

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

Consider the expression $\lambda x.x+x$. It is type-checking by type-checking $x_1 + x_2$, contracting x_1 and x_2 and then applying lambda abstraction. During the reduction of $(\lambda x.x+x)\ (y+z)$ we need to substitute y_1+z_1 for x_1 and y_2+z_2 for x_2. This is similar to substitution lemma in other structural variants of λ-calculus, such as the bunched typing system [74]. To express the substitution lemma later in this section, we first define the notion of a *context with holes*:

Definition 11 (Context with holes)**.** *A* context with holes *is a context such as* $x_1 : \tau_1, \ldots, x_k : \tau_k \mathbin{@} \langle r_1, \ldots, r_k\rangle$*, where some of the variable typings* $x_i : \tau_i$ *and corresponding coeffects* r_i *are replaced by* holes *written as* $-$*.*

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

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

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

Definition 12 (Hole filling). *Given a context with* n *holes* $\Delta @ \boldsymbol{s} \in \Delta[-@-]_n$*, the hole filling operation written as* $\Delta @ \boldsymbol{s}[\Gamma_1 @ \mathbf{r_1} \mid \ldots \mid \Gamma_n @ \mathbf{r_n}]$*, which replaces the holes by the specified variables and corresponding coeffect annotations, is defined as:*

$$\begin{aligned} -, \Delta @ \langle - \rangle \times \boldsymbol{s}\; [\Gamma_1 @ \boldsymbol{r}_1 \mid \Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \; &= \; \Gamma_1, \Gamma_2 @ \boldsymbol{r}_1 \times \boldsymbol{r}_2 \\ \text{where } \Gamma_2 @ \boldsymbol{r}_2 &= \Delta @ \boldsymbol{s}[\Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \\[1em] x_1 : \tau, \Delta @ \langle r_1 \rangle \times \boldsymbol{s}\; [\Gamma_1 @ \boldsymbol{r}_1 \mid \Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \; &= \; x_1 : \tau, \Gamma_2 @ \langle r_1 \rangle \times \boldsymbol{r}_2 \\ \text{where } \Gamma_2 @ \boldsymbol{r}_2 &= \Delta @ \boldsymbol{s}[\Gamma_1 @ \boldsymbol{r}_1 \mid \Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \\[1em] () @ \langle\rangle\; [\,] \; &= \; () @ \langle\rangle \end{aligned}$$

When we substitute an expression with coeffects $\mathbf{t}$ (associated with variables Γ) for a variable that has coeffects s, the resulting coeffects of Γ need to combine $\mathbf{t}$ and s. Unlike in the flat coeffect systems, the structural substitution does not require all coeffect algebra operations to coincide and so the combination is more interesting than in the bottom-pointed substitution for flat coeffects, where it used the only available operator (Lemma 9).

Substitution can be seen as a form of sequencing, i. e. the term $e_1[x \leftarrow e_2]$ should have the same type as the term `let` $x = e_2$ `in` e_1. In other words, we first need to obtain the value of the expression (requiring $\mathbf{t}$) and then use it in context with requirements s. Thus the free variables of the expression *after* substitution are annotated with $s \circledast \mathbf{t}$, using the (scalar-vector extension) of the sequential composition operator $\circledast$.

An operational reading of the resulting coeffect annotation is that, to evaluate each of the variables annotated with coeffects in the vector $\mathbf{t}$, we first need to evaluate the substituted expression with coeffects s, followed by the rest of the expression with coeffects specified by $\mathbf{t}$.

Lemma 40 (Multi-nary substitution). *Given an expression with multiple holes filled by variables* $x_i : \tau_i$ *with coeffects* s_k*:*

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

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

$$\Gamma_1 @ \boldsymbol{t}_1 \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma_k @ \boldsymbol{t}_k \vdash e_k : \tau_k$$

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

$$\Gamma @ \boldsymbol{r}\; [\Gamma_1 @ s_1 \circledast \boldsymbol{t}_1 \mid \ldots \mid \Gamma_k @ s_k \circledast \boldsymbol{t}_k] \vdash e_r[x_1 \leftarrow e_1] \ldots [x_k \leftarrow e_k] : \tau_r$$

Proof. By induction over $\vdash$, using the multi-nary aspect of the substitution in the proof of the contraction case (see Appendix **??**). □

4.4.3 *Reduction and expansion*

In the Chapter 2, we discussed call-by-value separately from call-by-name, because the proof of call-by-value substitution has fewer prerequisites. In this section, we consider full β-reduction, which encompasses both call-by-value and call-by-name. We also show that η-expansion preserves the types. Both of the properties hold for a system with any structural coeffect algebra.

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

Theorem 41 (Type preservation)**.** *In a structural coeffect system with a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and operations* $\langle - \rangle$ *and* $\circledast$*, if* $\Gamma @ \boldsymbol{r} \vdash e : \tau$ *and* $e \to_\beta e'$ *using the full* β*-reduction then* $\Gamma @ \boldsymbol{r} \vdash e' : \tau$.

Proof. Consider the typing derivation for the redex $(\lambda x.e_r)\, e_s$ before the reduction (note that this is similar to the typing of let binding discussed in Remark 36, but we do not swap the two parts of the free-variable context):

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

For the substitution lemma, we first rewrite the typing judgement for e_r, i. e. $\Gamma_r, x : \tau_s @ \mathbf{r} \times \langle t \rangle \vdash e_r : \tau_r$ as a context with a single hole filled by the x variable: $\Gamma_r, - @ \mathbf{r} \times - \ [x : \tau_s @ \langle t \rangle] \vdash e_r : \tau_r$. Now we can perform the substitution using Lemma 40:

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

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

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

EXPANSION THEOREM. Structural coeffect systems also exhibit η-equality, therefore satisfying both *local soundness* and *local completeness* as required by Pfenning and Davies [88]. Informally, this means that abstraction does not introduce too much, and application does not eliminate too much.

Theorem 42 (η-expansion)**.** *In a structural coeffect system with a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and operations* $\langle - \rangle$ *and* $\circledast$*, if* $\Gamma @ \boldsymbol{r} \vdash e : \tau$ *and* $e \to_\eta e'$ *using the full* η*-reduction then* $\Gamma @ \boldsymbol{r} \vdash e' : \tau$.

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

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

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

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

4.5 CATEGORICAL MOTIVATION

To define the semantics of structural coeffect calculus, we follow the same approach as for flat coeffect calculus in Chapter 3. In this section, we define categorical semantics for the calculus in terms of *structural indexed comonad*, which is an extension of the *indexed comonad* structure. Similarly to *flat indexed comonad*, the structural variant adds operations that are needed to embed full λ-calculus, this time with per-variable contexts.

We use the semantics to guide the *categorically-inspired translation* discussed in Section 4.6, which translates context-aware programs from the structural coeffect calculus to a simple target functional language with uninterpreted comonadically-inspired primitives (that correspond to operations of the structural indexed comonads). We then give operational semantics for a concrete context-aware language by giving the domain-specific reduction rules for the comonadically-inspired primitives. As an example, we use this to prove syntactic type safety of structural dataflow language in Section 4.6.3.

4.5.1 *Semantics of vectors*

Recall that in the flat coeffect calculus, the context is interpreted as a product and so a typing judgement $x_1 : \tau_1, \ldots, x_n : \tau_n \,@\, r \vdash e : \tau$ is interpreted as a morphism $C^r(\tau_1 \times \ldots \times \tau_n) \to \tau$. In this model, we can freely transform the value contained in the context modelled using an indexed comonad C^r.

Previously, we defined the map function (in terms of cobind and counit), which transforms the value inside the context without affecting the coeffect annotation. Thus, we can use $\mathsf{map}_r\ \pi_i$ to transform a context containing product of variables $C^r(\tau_1 \times \ldots \times \tau_n)$ into a context containing a single value $C^r\tau_i$. This changes the carried value without affecting the coeffect r.

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

The operator is a bifunctor, but it is *not* a product in the categorical sense. In particular, there is no way to turn $\tau_1 \hat{\times} \ldots \hat{\times} \tau_n$ into τ_i (the structure does not have projections) and so there is also no way of turning $C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ into $C^{\langle r_1,\ldots,r_n\rangle}\tau_i$, which would break the correspondence between coeffect annotations and variable structure.

The structure created using $-\hat{\times}-$ can be manipulated only using operations provided by the *structural indexed comonad*, which operate over variable contexts contained in an indexed comonad C^r and are designed to preserve the correspondence between vectors and annotations.

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

4.5.2 *Indexed comonads, revisited*

The semantics of structural coeffect calculus reuses the definition of *indexed comonad* with a minimal change. The additional structure that is required for context manipulation (merging and splitting) is different and is here provided by the *structural indexed comonad* structure that we introduce in this section.

Recall the definition from Section 3.2.4, which defines an indexed comonad over a monoid $(\mathcal{C}, \circledast, \mathsf{use})$ as a triple $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{r,s})$. The triple consists of a family of object mappings C^r, and two mappings that involve context-dependent morphisms of the form $C^r\tau_1 \to \tau_2$.

In the structural coeffect calculus, we work with morphisms of the form $C^r\tau_1 \to \tau_2$ representing function values (appearing in the language), but also of the form $C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n) \to \tau$, modelling expressions in a context. To capture this, we need to revisit the definition and use *coeffect vectors* in some of the operations.

Definition 13. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with a pointwise extension of the* $\circledast$ *operator to a vector (written as* $\mathsf{t} \circledast \boldsymbol{s}$*) and an operation lifting scalars to vectors* $\langle - \rangle$*, an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^{\mathbf{r}}, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{\mathsf{s},\boldsymbol{r}})$*:*

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

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

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

4.5.3 *Structural indexed comonads*

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

In the structural version, the corresponding operations operate convert between a (wrapped) vector of values represented using $\hat{\times}$ and ordinary pairs of contexts containing parts of the vector. The vectors of coeffect annotations are split or merged using $\times$ of the structural coeffect algebra, in a way that mirrors the wrapped vectors (variable structure).

The following definition includes $\mathsf{dup}_{r,s}$ which models duplication of a variable in a context needed for the semantics of contraction:

Definition 14. *Given a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *with operations* $\langle - \rangle$ *and* $\circledast$*, a* structural indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{\boldsymbol{r},\boldsymbol{s}}$*,* $\mathsf{split}_{\boldsymbol{r},\boldsymbol{s}}$ *and* $\mathsf{dup}_{r,s}$ *where:*

- $\mathsf{merge}_{r,s}$ *is a family of mappings* $C^{r}\alpha \times C^{s}\beta \rightarrow C^{r\times s}(\alpha \hat{\times} \beta)$
- $\mathsf{split}_{r,s}$ *is a family of mappings* $C^{r\times s}(\alpha \hat{\times} \beta) \rightarrow C^{r}\alpha \times C^{s}\beta$
- $\mathsf{dup}_{r,s}$ *is a family of mappings* $C^{\langle r \oplus s\rangle}\alpha \rightarrow C^{\langle r,s\rangle}(\alpha \hat{\times} \alpha)$

Here, the following equalities must hold:

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

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

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

4.5.4 *Semantics of structural calculus*

The concrete semantics for liveness and bounded variable use shown in Sections 1.3.1 and 1.3.2 suggests that semantics of structural coeffect calculi tend to be more complex than semantics of flat coeffect calculi. The complexity comes from the fact that we need a more expressive representation of the variable context – e. g. a vector of optional values. Additionally, the structural system needs to pass separate variable contexts to the sub-expressions.

The latter aspect is fully captured by the semantics shown in this section. The earlier point is left to the concrete notion of structural coeffect. Our model still gives us the flexibility of defining the concrete representation of variable vectors. We explore a number of examples in Section 4.5.5 and start by looking at a unified categorical semantics defined in terms of *structural indexed comonads*.

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

$$\begin{aligned} [\![x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n @ \langle r_1,\ldots,r_n\rangle \vdash e : \tau]\!] \;&:\; C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n) \rightarrow \tau \\ [\![\tau_1 \xrightarrow{r} \tau_2]\!] \;&=\; C^{\langle r\rangle}\tau_1 \rightarrow \tau_2 \end{aligned}$$

Note that the instances of flat indexed comonad ignored the fact that the variable context wrapped in the data structure is a product. This is not generally the case for the structural indexed comonads – the definitions shown in Section 4.5.5 are given specifically for $C^{\langle r_1,\ldots,r_n\rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ rather than more generally for $C^{\mathbf{r}}\alpha$. The need to examine the structure of the variable context is another reason for using $-\hat{\times}-$ when interpreting expressions in contexts.

The semantics is defined over a typing derivation:

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

$$\frac{}{[\![() @ \langle\rangle \vdash n : \mathsf{num}]\!]} \;=\; \frac{}{\mathsf{const}\ n} \qquad (\textit{num})$$

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

$$\frac{[\![\Gamma_1 @ \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!] = f \qquad [\![\Gamma_2 @ \mathbf{s} \vdash e_2 : \tau_1]\!] = g}{[\![\Gamma_1, \Gamma_2 @ \mathbf{r} \times (t \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2]\!] = \mathsf{app} \circ f \times (\mathsf{cobind}_{t,\mathbf{s}}\ g) \circ \mathsf{split}_{\mathbf{r},t \circledast \mathbf{s}}} \qquad (\textit{app})$$

$$\frac{[\![\Gamma @ \mathbf{r} \vdash e : \tau]\!]}{[\![\Gamma, x{:}\tau_1 @ \mathbf{r} \times \langle \mathsf{ign} \rangle \vdash e : \tau]\!]} \;=\; \frac{f}{f \circ \mathsf{snd} \circ \mathsf{split}_{\mathbf{r},\langle \mathsf{ign} \rangle}} \qquad (\textit{weak})$$

$$\frac{\begin{array}{r}[\![\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \\ @ \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau]\!]\end{array}}{\begin{array}{r}[\![\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \\ @ \mathbf{r} \times \langle t, s \rangle \times \mathbf{q} \vdash e : \tau]\!]\end{array}} \;=\; \frac{f}{\begin{array}{l}f \circ \mathsf{nest}_{\mathbf{r},\langle t,s \rangle,\langle s,t \rangle,\mathbf{q}} \\ \quad (\mathsf{merge}_{\langle s \rangle,\langle t \rangle} \circ \mathsf{swap} \circ \mathsf{split}_{\langle t \rangle,\langle s \rangle})\end{array}} \qquad (\textit{exch})$$

$$\frac{\begin{array}{r}[\![\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \\ @ \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau]\!]\end{array}}{\begin{array}{r}[\![\Gamma_1, x{:}\tau_1, \Gamma_2 @ \mathbf{r} \times \langle s \oplus t \rangle \times \mathbf{q} \\ \vdash e[z, y \leftarrow x] : \tau]\!]\end{array}} \;=\; \frac{f}{f \circ \mathsf{nest}_{\mathbf{r},\langle s \oplus t \rangle,\langle s,t \rangle,\mathbf{q}}\ \mathsf{dup}_{s,t}} \qquad (\textit{contr})$$

Assuming the following auxiliary operations:

$$\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}}\ f \;=\; \mathsf{merge}_{\mathbf{r},\mathbf{s}' \times \mathbf{t}} \circ \mathsf{id} \times (\mathsf{merge}_{\mathbf{s}',\mathbf{t}} \circ f \times \mathsf{id} \circ \mathsf{split}_{\mathbf{s},\mathbf{t}}) \circ \mathsf{split}_{\mathbf{r},\mathbf{s} \times \mathbf{t}}$$

$$\begin{aligned}
\mathsf{id}\ x &= x \\
\mathsf{const}\ v &= \lambda x.v \\
\mathsf{curry}\ f\ x\ y &= \lambda f.\lambda x.\lambda y.f\ (x, y) \\
\mathsf{fst}\ (x, y) &= x \\
\mathsf{swap}\ (x, y) &= (y, x) \\
f \times g &= \lambda(x, y).(f\ x, g\ y) \\
\mathsf{app}\ (f, x) &= f\ x
\end{aligned}$$

Figure 27: Categorical semantics of the structural coeffect calculus

EXPRESSIONS. A semantics of structural coeffect calculi is shown in Figure 27. The semantics is written as composition of morphisms using a number of auxiliary definitions. Due to the eqivalence between Cartesian Closed Categories and the λ-calculus, we will treat it as specifying translation to a target functional langauge in Section 4.6.

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

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

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

  A value $C^{\langle t \rangle}\tau_1$ required in order to call the function is obtained by applying $\mathsf{cobind}_{t,s}$ to the semantics of e_2. The result $C^{t \circledast s}(\ldots \hat{\times} \ldots \hat{\times} \ldots) \rightarrow C^{\langle t \rangle}\tau_1$ is then called with the latter part of the split input context.

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

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

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

The transformations on the middle part are quite simple. The (*exch*) rule swaps two single-variable contexts and the (*contr*) rule uses the $\mathsf{dup}_{s,t}$ operation to duplicate a variable while splitting its additional context.

PROPERTIES. As in the flat calculus, the main reason for defining the categorical semantics in this chapter is to provide validation for the design of the calculus. The following correspondence theorem states that the annotations in the typing rules of the structural coeffect calculus correspond to the indices of the semantics. Thus, the calculus captures a context-dependent property if it can be modelled by a *structural indexed comonad*. As we show in the next section, this is the case for all three discussed examples (liveness, dataflow, bounded variable reuse).

Theorem 43 (Correspondence)**.** *In all of the typing rules of the structural coeffect system, the context annotations r and s of typing judgements $\Gamma @ r \vdash e : \tau$ and function types $\tau_1 \xrightarrow{s} \tau_2$ correspond to the indices of mappings C^{r} and $C^{\langle s \rangle}$ in the corresponding semantic function defined by $[\![\Gamma @ r \vdash e : \tau]\!]$.*

Proof. By analysis of the semantic rules in Figure 27. □

4.5.5 *Examples of structural indexed comonads*

The categorical semantics for structural coeffect calculus is easily instantiated to give semantics for a concrete calculus. In this section, we revisit the three examples discussed throughout this chapter – structural liveness, dataflow and bounded variable reuse. Some aspects of the first two examples will be similar to flat versions discussed in Section 3.2.5 – they are based on the same data structures (option and a list, respectively), but the data structures are composed differently. Generally speaking, rather than having a data structure over a product of variables, we now have a vector of variables over a specific data structure.

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

DATAFLOW. It is interesting to note that the semantics of dataflow and bounded variable reuse (discussed next) both keep a product of multiple values for each variable, so they are both built around an *indexed list* data structure. However, their cobind and dup operations work differently. We start by looking at the structure modelling dataflow computations. For readability, variables in bold face (such as $\mathbf{a_1}$) range over vectors while ordinary notation (such as a_1) is used for individual values.

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

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

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

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

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

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

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

$$\begin{array}{l}\mathsf{cobind}_{m,\langle n_1,\ldots,n_k \rangle} \; f \; \langle\langle a_{1,0}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,m+n_k} \rangle\rangle = \\ \quad \langle\langle \; f\langle\langle a_{1,0}, \ldots a_{1,n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,n_k} \rangle\rangle, \; \ldots \;, \\ \qquad f\langle\langle a_{1,m}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,m}, \ldots a_{k,m+n_k} \rangle\rangle \;\; \rangle\rangle\end{array}$$

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

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

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

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

The $\mathsf{dup}_{m,n}$ operation needs to produce a two-variable context containing m and n values, respectively, of the input variable. The input provides $\mathit{max}(m, n)$ values, so the definition is simply a matter of restriction. Finally, counit extracts the value of its single variable.

bounded reuse. As mentioned earlier, the semantics of calculus for bounded reuse is also based on the indexed list structure. Rather than representing possibly different past values that can be shared (see the definition of dup), the list now represents multiple copies of the same value as each value can only be accessed once. This semantics follows that of Girard [41].

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

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

The merge *and* split *operations are defined as in Example* **??**. *The operations that capture the behaviour of bounded reuse are:*

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

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

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

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

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

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

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

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

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

$$\begin{aligned}
\mathsf{dup}_{\mathsf{D},\mathsf{D}}\langle()\rangle &= \langle (), ()\rangle\\
\mathsf{dup}_{\mathsf{L},\mathsf{D}}\langle a\rangle &= \langle a, ()\rangle \qquad\qquad \mathsf{counit}_{\mathsf{L}}\ \langle a\rangle = a\\
\mathsf{dup}_{\mathsf{D},\mathsf{L}}\langle a\rangle &= \langle (), a\rangle\\
\mathsf{dup}_{\mathsf{L},\mathsf{L}}\langle a\rangle &= \langle a, a\rangle
\end{aligned}$$

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

The dup operation resembles with the flat version of split – this is expected as duplication in the flat calculus is performed by first duplicating the variable context (using map) and then applying split. Here, the duplication returns a pair. Depending on the required coeffect annotations, this may copy (duplicate) the value, or it may produce an empty context.

Finally, counit extracts a value which is always present as guaranteed by the type $C^{\langle \mathsf{L}\rangle}\alpha \rightarrow \alpha$. The lifting operation models sub-coeffecting which can turn a context with a value into a dead context (second case); otherwise it behaves as identity.

PROPERTIES. The concrete categortical semantics presented in this section is a generalization of the concrete semantics given when introducing context-aware programming languages in Chapter 1.

Theorem 44 (Generalization)**.** *Consider a typing derivation obtained according to the rules for finding unique typing derivations as specified in Section 4.3 for a coeffect language with liveness, dataflow or bounded variable use.*

The semantics obtained by instantiating the rules in Figure 27 with the concrete operations defined in Example 18, Example 16 or Example 17 is the same as the one defined in Figure 12, Section 1.3.3 and Section 1.3.2, respectively.

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

4.6 TRANSLATIONAL SEMANTICS

In the previous section, we used category theory to give a unified model capable of capturing the semantics of our three context-aware language based on the structural coeffect calculus. Although the categorical model is interesting on its own, we use it in the same way as in Chapter 3 – to define a translation from source context-aware languages to a simple target functional language. As for flat coeffect calculus, we show that the translation produces well-typed programs in the target language. For a sample context-aware programming language, we then show that well-typed programs (produced by the translation) do not get stuck.

This section mirrors the development presented in Chapter 3 for flat coeffect calculus. We extend the simple target language with additional constructs inspired by structural indexed comonads (Section 4.6.1), define the

LANGUAGE SYNTAX. Given a structural coeffect algebra, extend the programming language syntax with the following constructs:

$$\begin{array}{rcl} e & = & \ldots \mid \mathsf{cobind}_{s,\mathbf{r}}\ e_1\ e_2 \mid \mathsf{counit}_{\mathsf{use}}\ e \mid \mathsf{merge}_{\mathbf{r},\mathbf{s}}\ e \mid \mathsf{split}_{\mathbf{r},\mathbf{s}}\ e \mid \mathsf{dup}_{r,s}\ e \\ \tau & = & \ldots \mid C^{\mathbf{r}}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k) \\ K & = & \ldots \mid \mathsf{cobind}_{s,\mathbf{r}}\ \_\ e \mid \mathsf{cobind}_{s,\mathbf{r}}\ v\ \_ \mid \mathsf{counit}_{\mathsf{use}}\ \_ \\ & & \mid \mathsf{merge}_{\mathbf{r},\mathbf{s}}\ \_ \mid \mathsf{split}_{\mathbf{r},\mathbf{s}}\ \_ \mid \mathsf{dup}_{r,s}\ \_ \end{array}$$

TYPING RULES. Given a structural coeffect algebra, add the typing rules:

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

$$(\mathit{cobind})\ \frac{\Gamma \vdash e_1 : C^{\mathbf{r}}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k) \to \tau \qquad \Gamma \vdash e_2 : C^{s \circledast \mathbf{r}}\tau_1(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{cobind}_{s,\mathbf{r}}\ e_1\ e_2 : C^{\langle s \rangle}\tau}$$

$$(\mathit{merge})\ \frac{\Gamma \vdash e : C^{\mathbf{r}}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_l) \times C^{\mathbf{s}}(\tau_{l+1} \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{merge}_{\mathbf{r},\mathbf{s}}\ e : C^{\mathbf{r} \times \mathbf{s}}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)}$$

$$(\mathit{split})\ \frac{\Gamma \vdash e : C^{\mathbf{r} \times \mathbf{s}}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{split}_{\mathbf{r},\mathbf{s}}\ e : C^{\mathbf{r}}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_l) \times C^{\mathbf{s}}(\tau_{l+1} \hat{\times} \ldots \hat{\times} \tau_k)}$$

$$(\mathit{dup})\ \frac{\Gamma \vdash e : C^{\langle r \oplus s \rangle}\tau}{\Gamma \vdash \mathsf{dup}_{r,s}\ e : C^{\langle r,s \rangle}(\tau \hat{\times} \tau)}$$

Figure 28: Comonadically-inspired extensions for structural coeffects

translation to the target langauge (Section 4.6.2) and prove safety of one sample language (Section 4.6.3) – we choose structural dataflow to allow easy comparison with the flat system.

4.6.1 *Comonadically-inspired language extensions*

In Section 3.3.1, we defined the syntax, typing rules and operational semantics for a simple functional programming language. We then extended it with uninterpreted constructs inspired by the *flat indexed comonad* structure and used it as the translation target for the flat coeffect calculus. In this section, we take the same core language and extend it with constructs inspired by the *structural indexed comonad*.

Given a coeffect language with a structural coeffect algebra formed by $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ and operations $\langle - \rangle$ and $\circledast, \times$, we extend the core functional language with operations shown in Figure 28. The syntax extensions add comonadically-inspired operations that mirror those defined in Section 4.5.3. The typing for the operations corresponds to their categorical counterparts.

We also include an uninterpreted type $C^{\mathbf{r}}\tau_1 \hat{\times} \ldots \hat{\times} \tau_k$, which models a contextual (comonadic) value indexed by a vector of annotations. As in the categorical model for structural coeffects, context consisting of multiple variables is not modelled as ordinary tuple – it can only be manipulated by the

The translation is defined over a typing derivation:

$$\frac{}{\llbracket x{:}\tau \,@\, \langle \mathsf{use} \rangle \vdash x : \tau \rrbracket} = \frac{}{\lambda ctx.\mathsf{counit}_{\mathsf{use}}\ ctx} \qquad (var)$$

$$\frac{}{\llbracket () \,@\, \langle\rangle \vdash n : \mathsf{num} \rrbracket} = \frac{}{\lambda ctx.n} \qquad (num)$$

$$\frac{\llbracket \Gamma, x{:}\tau_1 \,@\, \mathbf{r} \times \langle s \rangle \vdash e : \tau_2 \rrbracket}{\llbracket \Gamma \,@\, \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2 \rrbracket} = \frac{f}{\lambda ctx.\lambda v.f\ (\mathsf{merge}_{\mathbf{r},\langle s \rangle}\ (ctx, v))} \qquad (abs)$$

$$\frac{\begin{array}{r} \llbracket \Gamma_1 \,@\, \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \rrbracket = f \\ \llbracket \Gamma_2 \,@\, \mathbf{s} \vdash e_2 : \tau_1 \rrbracket = g \end{array}}{\llbracket \Gamma_1, \Gamma_2 \,@\, \mathbf{r} \times (t \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2 \rrbracket = \begin{array}{l} \lambda ctx. \\ \quad \mathtt{let}\ (ctx_1, ctx_2) = \mathsf{split}_{\mathbf{r},t \circledast \mathbf{s}}\ ctx \\ \quad f\ ctx_1\ (\mathsf{cobind}_{t,\mathbf{s}}\ g\ ctx_2) \end{array}} \qquad (app)$$

$$\frac{\llbracket \Gamma \,@\, \mathbf{r} \vdash e : \tau \rrbracket}{\llbracket \Gamma, x{:}\tau_1 \,@\, \mathbf{r} \times \langle \mathsf{ign} \rangle \vdash e : \tau \rrbracket} = \frac{f}{\begin{array}{l} \lambda ctx. \\ \quad \mathtt{let}\ (ctx_1, \_) = \mathsf{split}_{\mathbf{r},\langle \mathsf{ign} \rangle}\ ctx \\ \quad f\ ctx_1 \end{array}} \qquad (weak)$$

$$\frac{\begin{array}{l} \llbracket \Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \\ \quad @\, \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau \rrbracket \end{array}}{\begin{array}{l} \llbracket \Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \\ \quad @\, \mathbf{r} \times \langle t, s \rangle \times \mathbf{q} \vdash e : \tau \rrbracket \end{array}} = \frac{f}{\begin{array}{l} \lambda ctx.\ f(\mathsf{nest}_{\mathbf{r},\langle t,s \rangle,\langle s,t \rangle,\mathbf{q}}\ (\lambda ctx'. \\ \quad \mathtt{let}\ (ctx_1, ctx_2) = \mathsf{split}_{\langle t \rangle,\langle s \rangle} \\ \quad \mathsf{merge}_{\langle s \rangle,\langle t \rangle}\ (ctx_2, ctx_1)\ )) \end{array}} \qquad (exch)$$

$$\frac{\begin{array}{l} \llbracket \Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \\ \quad @\, \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau \rrbracket \end{array}}{\begin{array}{l} \llbracket \Gamma_1, x{:}\tau_1, \Gamma_2 \,@\, \mathbf{r} \times \langle s \oplus t \rangle \times \mathbf{q} \\ \quad \vdash e[z, y \leftarrow x] : \tau \rrbracket \end{array}} = \frac{f}{\begin{array}{l} \lambda ctx.f\ (\mathsf{nest}_{\mathbf{r},\langle s \oplus t \rangle,\langle s,t \rangle,\mathbf{q}} \\ \qquad \mathsf{dup}_{s,t}\ ctx) \end{array}} \qquad (contr)$$

Assuming the following auxiliary definition:

$$\begin{array}{l} \mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}} = \lambda f.\lambda ctx. \\ \quad \mathtt{let}\ (ctx_1, ctx') = \mathsf{split}_{\mathbf{r},\mathbf{s} \times \mathbf{t}}\ ctx \\ \quad \mathtt{let}\ (ctx_2, ctx_3) = \mathsf{split}_{\mathbf{s},\mathbf{t}}\ ctx' \\ \quad \mathsf{merge}_{\mathbf{r},\mathbf{s}' \times \mathbf{t}}\ (ctx_1, \mathsf{merge}_{\mathbf{s}',\mathbf{t}}\ (f\ ctx_2, ctx_3)) \end{array}$$

Figure 29: Trnaslation from a structural coeffect calculus

comonadically-inspired operations. In the target language, this is done by defining the $C^{\mathbf{r}}$ type over zero or more underlying types. Here, our syntactic treatment differs slightly from the categorical model where objects created by $\hat{\times}$ were first-class values. In Figure 28, we need to explicitly specify the types of individual components in typing rules for (*merge*), (*split*) and (*cobind*).

As with flat coeffects, the extensions described here are common for all concrete instances of structural context-aware languages. For each concrete language, we need to provide values of type $C^{\mathbf{r}}\tau$ and reduction rules for comonadically-inspired operations.

4.6.2 *Comonadically-inspired translation*

When translating context-aware programs to the functional language, variable contexts become values of comonadically-inspired data types containing a value for each variable in the context. Function inputs become comonadically-inspired values containing exactly one variable. More formally:

$$\begin{aligned}
[\![x_1\!:\!\tau_1,\ldots,x_n\!:\!\tau_n \,@\, \langle r_1,\ldots,r_n\rangle]\!] &= \mathsf{C}^{\langle r_1,\ldots,r_n\rangle}([\![\tau_1]\!]\,\hat{\times}\ldots\hat{\times}\,[\![\tau_n]\!])\\
[\![\tau_1 \xrightarrow{r} \tau_2]\!] &= \mathsf{C}^{\langle r\rangle}[\![\tau_1]\!] \rightarrow [\![\tau_2]\!]\\
[\![\mathsf{num}]\!] &= \mathsf{num}
\end{aligned}$$

The definition differs from the one for flat coeffects (Section 3.3.3) in that the comonadically-inspired data type takes multiple type parameters (separated using $\hat{\times}$), rather than wrapping a regular tuple in the target language.

The translation rules are defined in Figure 29. As in the case of the flat coeffect calculus, the definition directly follows the categorical semantics shown in Figure 29. We expand the definitions so that the result is a valid program in the target language rather than a composition of morphisms.

As with flat coeffects, the correspondence property of the semantics (Theorem 43) can now be adapted into well-typedness of the translation. Given a well-typed program in the structural coeffect calculus, the translation produces a well-typed program in the target language. This is true for all context-aware languages based on the structural coeffect calculus and it provides us with the first part of type safety theorem. The second part is type safety of the target language with concrete domain-specific extensions as discussed in Section 4.6.3.

Theorem 45 (Well-typedness of the translation)**.** *Given a typing derivation for a well-typed closed expression* $@\langle\rangle \vdash e : \tau$ *written in a structural context-aware programming langugae that is translated to the target language as (we write ... for the omitted part of the translation tree):*

$$\frac{[\![\ (\ldots)\ ]\!]}{[\![@\langle\rangle \vdash e : \tau]\!]} \;\; \begin{matrix} = \\ = \end{matrix} \;\; \frac{(\ldots)}{f}$$

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

Proof. By rule induction over the derivation of the translation. Given a judgement $x_1 : \tau_1 \ldots x_n : \tau_n \,@\, \mathbf{c} \vdash e : \tau$ where $\mathbf{c} = \langle c_1,\ldots,c_n\rangle$, the translation constructs a function of type $\mathsf{C}^{\mathbf{c}}([\![\tau_1]\!]\,\hat{\times}\ldots\hat{\times}\,[\![\tau_n]\!]) \rightarrow [\![\tau]\!]$.

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

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

Case (*abs*): The type of *ctx* is $\mathsf{C}^{\mathbf{r}}(\ldots)$ and the type of v is $\mathsf{C}^{\langle s\rangle}\tau_1$, calling $\mathsf{merge}_{\mathbf{r},\langle s\rangle}$ produces a context of type $\mathsf{C}^{\mathbf{r}\times\langle s\rangle}(\ldots\,\hat{\times}\,\tau_1)$ as expected by f.

Case (*app*): After applying $\mathsf{split}_{\mathbf{r},t\circledast\mathbf{s}}$ the types of *ctx*$_1$,*ctx*$_2$ are $\mathsf{C}^{\mathbf{r}}(\ldots)$ and $\mathsf{C}^{t\circledast\mathbf{s}}(\ldots)$, respectively. g requires $\mathsf{C}^{\mathbf{s}}(\ldots)$ and so the result of $\mathsf{cobind}_{t,\mathbf{s}}$ is $\mathsf{C}^{\langle t\rangle}\tau_1$ as required by f.

Case (*weak*): After applying $\mathsf{split}_{\mathbf{r},\langle \mathsf{ign}\rangle}$ the type of *ctx*$_1$ is $\mathsf{C}^{\mathbf{r}}(\ldots)$ as required.

Case (*exch*), (*contr*): The auxiliary definition $\mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s}',\mathbf{t}}$ keeps parts of the context corresponding to coeffect annotations $\mathbf{r},\mathbf{q}$ unchanged and transforms the nested part. In (*exch*), the provided lambda function is of type $\mathsf{C}^{\langle s,t\rangle}(\tau_1\,\hat{\times}\,\tau_2) \rightarrow \mathsf{C}^{\langle t,s\rangle}(\tau_2\,\hat{\times}\,\tau_1)$. In (*contr*), the type of $\mathsf{dup}_{s,t}$ is $\mathsf{C}^{\langle s\oplus t\rangle}\tau \rightarrow \mathsf{C}^{\langle s,t\rangle}(\tau\,\hat{\times}\,\tau)$ (assuming nest is expanded, rather than treated as a value within the language). □

4.6.3 *Structural coeffect language for dataflow*

The target language with comonadically-inspired primitives provides a framework that can be used to model a variety of structural context-aware languages and prove their safety. As outlined in Section 3.5, the key principle that guarantees the safety of the target language is a correspondence between coeffect annotations and the values they represent. In a more expressive target language (such as Haskell or Agda), it would be sufficient to provide an implementation of the comonadically-inspired primitives for the concrete domain-specific language.

Our simple target language is not expressive enough to capture the correspondence in the type system and so we instead follow the same methodology as when discussing safety for flat coeffect languages in section Section 3.4 and show safety for a sample concrete context-aware language. We consider the structural coeffect language for dataflow. The definitions can be compared with the flat version discussed in Section 3.4.1, which highlights the similarities and differences between the flat and structural notion of context.

DOMAIN-SPECIFIC EXTENSIONS. The Figure 30 extends the target functional langauge with constructs, typing rules, translation rules and reduction rules needed for modelling structural dataflow. In the structural model, the type $C^{\langle r_0,\ldots,r_n\rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_n)$ represents a structure that provides values and additional contexts for variables of types $\tau_0,\ldots,\tau_n$ with contextual capabilities as specified by corresponding coeffect annotations $r_0,\ldots,r_n$.

In case of dataflow, the comonadically-inspired structure keeps vectors of past values for each of the variables in the context. In the syntax, we write **v** and **e** for vectors of values and expressions, respectively. The expression $\mathsf{Df}\langle \mathbf{e}_0,\ldots,\mathbf{e}_k\rangle$ is then formed by a vector of variable assignments where each variable assignment is a vector of current and past values. When reducing expressions to values, we reduce values from left to right and from currentmost value to the last past value. This is specified by the context K.

When type-checking expressions that create the Df values, we use an auxiliary judgement $\Gamma \vdash_{\mathsf{vec}} e : \tau, n$. The judgement checks that a vector of expressions **e** contains exactly n expressions of type τ. This is caputred by the (*vec*) rule, which is then used to check individual elements of the context in the (*df*) rule.

PROPERTIES. Now consider a target language consisting of the core (ML-subset) defined by the syntax, reduction rules and typing rules given in Figure 20 (Chapter 3) with primitives inspired by structural indexed comonads defined in Figure 28 and also concrete notion of comonadically-inspired value and reduction rules for dataflow as defined in Figure 30.

As with the examples discussed in Chapter 3, the resulting language is type safe. Together with the well-typedness of the translation (Theorem 45), this guarantees type safety of the structural coeffect calculus for dataflow. 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 **??**) and progress (Theorem **??**) theorems.

LANGUAGE SYNTAX

$$\begin{array}{rcl@{\qquad}rcl}
v & = & \ldots \mid \mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle & \mathbf{v} & = & \langle v_0, \ldots, v_k \rangle \\
e & = & \ldots \mid \mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_k \rangle \mid \mathsf{prev}_{n_0,\ldots,n_k}\ e & \mathbf{e} & = & \langle e_0, \ldots, e_k \rangle \\
K & = & \ldots \mid \mathsf{Df}\langle \mathbf{v}_0, \ldots, \langle v_{j,0}, \ldots, v_{j,i-1}, \_, e_{j,i+1} \ldots, e_{j,n} \rangle, \ldots, \mathbf{e}_k \rangle & & & \\
 & & \ldots \mid \mathsf{prev}_{n_0,\ldots,n_k}\ \_ & & &
\end{array}$$

TYPING RULES

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

$$(df)\ \frac{\forall i \in \{0 \ldots k\}.\ \Gamma \vdash_{\mathsf{vec}} \mathbf{e}_i : \tau_i, n_i}{\Gamma \vdash \mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_k \rangle : C^{\langle n_0,\ldots,n_k \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)}$$

$$(prev)\ \frac{\Gamma \vdash e : C^{\langle n_0+1,\ldots,n_k+1 \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)}{\Gamma \vdash \mathsf{prev}_{n_0,\ldots,n_k}\ e : C^{\langle n_0,\ldots,n_k \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)}$$

TRANSLATION

$$\begin{array}{rcl}
\underline{[\![\Gamma @ \langle n_0+1, \ldots, n_k+1 \rangle \vdash e : \tau]\!]} & = & \underline{f \qquad\qquad\qquad\qquad} \\
[\![\Gamma @ \langle n_0, \ldots, n_k \rangle \vdash \mathtt{prev}\ e : \tau]\!] & = & \lambda ctx.\mathsf{prev}_{n_0,\ldots,n_k}\ ctx
\end{array}$$

REDUCTION RULES

$$\begin{array}{rl}
(counit) & \mathsf{counit}_0(\mathsf{Df}\langle\langle v_0 \rangle\rangle) \leadsto v_0 \\
(cobind) & \mathsf{cobind}_{m,\langle n_1,\ldots,n_k \rangle}\ f\ \mathsf{Df}\langle\langle a_{1,0}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,m+n_k} \rangle\rangle \leadsto \\
 & \quad \mathsf{Df}\langle\langle\ f(\mathsf{Df}\langle\langle a_{1,0}, \ldots a_{1,n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,n_k} \rangle\rangle),\ \ldots\ , \\
 & \qquad\qquad f(\mathsf{Df}\langle\langle a_{1,m}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,m}, \ldots a_{k,m+n_k} \rangle\rangle)\ \ \rangle\rangle \\
(merge) & \mathsf{merge}_{\langle m_0,\ldots,m_k \rangle,\langle n_0,\ldots,n_l \rangle}((\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle), (\mathsf{Df}\langle \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle)) \leadsto \\
 & \quad \mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k, \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle \\
(split) & \mathsf{split}_{\langle m_0,\ldots,m_k \rangle,\langle n_0,\ldots,n_l \rangle}(\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k, \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle) \leadsto \\
 & \quad (\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle,\ \mathsf{Df}\langle \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle) \\
(prev) & \mathsf{prev}_{\langle n_1,\ldots,n_k \rangle}(\mathsf{Df}\langle\langle v_{1,0}, \ldots, v_{1,n_1}, v_{1,n_1+1} \rangle, \ldots, \langle v_{k,0}, \ldots, v_{k,n_k}, v_{k,n_k+1} \rangle\rangle) \leadsto \\
 & \quad \mathsf{Df}\langle\langle v_{1,0}, \ldots, v_{1,n_1} \rangle, \ldots, \langle v_{k,0}, \ldots, v_{k,n_k} \rangle\rangle \\
(dup) & \mathsf{dup}_{m,n}(\mathsf{Df}\langle\langle v_0, \ldots, v_{max(m,n)} \rangle\rangle) \leadsto \mathsf{Df}\langle\langle v_0, \ldots, v_m \rangle, \langle v_0, \ldots, v_n \rangle\rangle
\end{array}$$

Figure 30: Additional constructs for modelling structural dataflow

Theorem 46 (Well-typedness of the `prev` translation). *Given a typing derivation for a well-typed closed expression* $@\langle\rangle \vdash e : \tau$*, the translated program* f *obtained using the rules in Figure 29 and Figure 30 is well-typed, i.e. in the target language:* $\vdash f : [\![() @ \langle\rangle]\!] \to [\![\tau]\!]$.

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

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

Case (*prev*): Type of *ctx* is $C^{\langle n_0+1,\ldots,n_k+1 \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)$ and so we can apply the (*prev*) rule to obtain $C^{\langle n_0,\ldots,n_k \rangle}(\tau_0 \hat{\times} \ldots \hat{\times} \tau_k)$ as required by f. □

Lemma 47 (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* $e = (\nu_1, \ldots, \nu_n)$ *for some* ν_i
4. *If* $\tau = C^{\langle n_1, \ldots, n_k \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)$ *then* $e = \mathsf{Df}\langle v_0, \ldots v_n \rangle$ *for some* v_i *such that* $v_i = \langle \nu_0, \ldots, \nu_{n_i} \rangle$.

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

Lemma 48 (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 $\mathsf{prev}_{\mathbf{n}}$. □

Theorem 49 (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 48 for (*fn*).

Case (*counit*): $e = \mathsf{counit}_0(\mathsf{Df}\langle\langle \nu_0 \rangle\rangle)$. The last rule in the type derivation of e must have been (*counit*) with $\Gamma \vdash \mathsf{Df}\langle \nu_0 \rangle\rangle : C^{\langle 0 \rangle}\tau$ and therefore $\Gamma \vdash \nu_0 : \tau$.

Case (*cobind*): $e = \mathsf{cobind}_{m,\langle n_1, \ldots, n_k \rangle}\ f\ (\mathsf{Df}\langle \mathbf{v}_0, \ldots \mathbf{v}_k \rangle)$ such that $\forall i \in \{1 \ldots k\}$ $\mathbf{v}_i = \langle \nu_0, \ldots, \nu_{n_i} \rangle$. The last rule in the type derivation of e must have been (*cobind*) with a type $\tau = C^{\langle m \rangle}\tau'$ and assumptions:

- $\Gamma \vdash f : C^{\langle n_1, \ldots, n_k \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k) \rightarrow \tau_2$ and
- $\Gamma \vdash \mathsf{Df}\langle \mathbf{v}_0, \ldots \mathbf{v}_k \rangle : C^{\langle m+n_1, \ldots, m+n_k \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_k)$

Using the (*df*) rule, the reduced expression has a type $C^{\langle m \rangle}\tau'$.

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

Theorem 50 (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 47) 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 47, $e_1 = \mathsf{Df}\langle\langle \nu \rangle\rangle$ and so we can apply (*counit*) reduction rule.

Case (*cobind*): $e = \mathsf{cobind}_{m,\langle n_1, \ldots, n_k \rangle}\ e_1\ e_2$. If e_1 is not a value, reduce using (*ctx*) with context $\mathsf{cobind}_{m,\langle n_1, \ldots, n_k \rangle}\ \_\ e$. If e_2 is not a value reduce using (*ctx*) with context $\mathsf{cobind}_{m,\langle n_1, \ldots, n_k \rangle}\ \nu\ \_$. If both are values then we have $e_2 = \mathsf{Df}\langle\langle a_{1,0}, \ldots a_{1,m+n_1} \rangle, \ldots, \langle a_{k,0}, \ldots a_{k,m+n_k} \rangle\rangle$ from Lemma 47 and so we can apply the (*cobind*) reduction.

Case (*merge*): $e = \mathsf{merge}_{\langle m_0, \ldots, m_k \rangle, \langle n_0, \ldots, n_l \rangle} e_1$. If e_1 is not a value, reduce using (*ctx*) with context $e = \mathsf{merge}_{\langle m_0, \ldots, m_k \rangle, \langle n_0, \ldots, n_l \rangle}\ \_$. If e_1 is a value, it must be a pair of vectors $(\mathsf{Df}\langle \mathbf{v}_0, \ldots, \mathbf{v}_k \rangle, \mathsf{Df}\langle \mathbf{v}'_0, \ldots, \mathbf{v}'_l \rangle)$ using Lemma 47 and it can reduce using (*merge*) reduction.

Case (*df*): $e = \mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_n \rangle$. If $\mathbf{e}_i$ is not a value then reduce using (*ctx*) with the context $\mathsf{Df}\langle \ldots \rangle$. Otherwise, $\mathbf{e}_0, \ldots, \mathbf{e}_n$ are values and so $\mathsf{Df}\langle \mathbf{e}_0, \ldots, \mathbf{e}_n \rangle$ 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 51 (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 49 and Theorem 50. □

4.7 CONCLUSIONS

This chapter completes the key development of this thesis – the presentation of the coeffect framework, consisting of two calculi capturing properties of context-aware computations introduced in Chapter 1. In Chapters 2 and 3, we focused on whole-context properties of computations and we developed *flat coeffect calculus* to capture them. This chapter develops *structural coeffect calculus*, which captures *per-variable* contextual properties. The system provides a precise analysis of liveness and dataflow and allows other interesting uses such as tracking of variable accesses based on bounded linear logic.

Following the structure of the previous two chapters, the structural coeffect calculus is parameterized by a *structural coeffect algebra*. The two definitions are similar – both require operations $\circledast$ and $\oplus$ that model sequential and pointwise composition, respectively. For flat coeffects, we required $\wedge$ to model context merging. For structural coeffects, we instead use a vector (free monoid) with the $\times$ operation – which serves a similar purpose as $\wedge$. In order to keep track of separate annotations for each variable, we use a system with explicit structural rules (contraction, weakening and exchange) that manipulate the structure of variables and the structure of annotations at the same time.

The structural coeffect calculus has desirable equational properties that are satisfied only by certain flat coeffect calculi. In particular, we show that β-reduction and η-expansion have the typing preservation property for any instance of the structural coeffect calculus. These two strong properties are desirable for programming languages, but are often not satisfied (e. g. by languages with effects).

Finally, we discuss the semantics of the structural coeffect calculus in in terms of *structural indexed comonads*. As in Chapter 3, we first use the categorical semantics to unify the different notions of context (bounded reuse, dataflow and liveness) and then use it as a basis for translation that turns well-typed programs written in the structural coeffect calculus into well-typed programs of a simple functional language. We give concrete reduction rules for the target language, modelling *structural dataflow* and prove type safety of the language.

BIBLIOGRAPHY

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

[2] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, pages 23–38. Springer, 2003.

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

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

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

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

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

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

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

[21] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 957–964. ACM, 2009.

[22] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *Proceedings of ICFP*, ICFP '13, pages 403–416, 2013.

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

[24] J.-L. Cola&#x00e7;o and M. Pouzet. Type-based initialization analysis of a synchronous dataflow language. *Int. J. Softw. Tools Technol. Transf.*, 6(3):245–255, Aug. 2004.

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

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

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

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

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

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

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

[32] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 429–442. ACM, 2013.

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

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

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

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

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

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

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

[40] G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. *ACM SIGPLAN Notices*, 46(12):13–22, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

[53] 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.

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

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

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

[57] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

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

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

[60] D. Leijen and E. Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[61] 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.

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

[63] 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.

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

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

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

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

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

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

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

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

[72] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, pages 114–136. Springer, 1999.

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

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

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

[76] D. Orchard. Programming contextual computations.

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

[78] D. Orchard and A. Mycroft. Efficient and correct stencil computation via pattern matching and static typing. In *Proceedings of DSL 2011*, arXiv preprint arXiv:1109.0777, 2011.

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

[80] 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.

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

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

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

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

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

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

[87] 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.

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

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

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

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

[92] C. W. Probst, C. Hankin, and R. R. Hansen, editors. *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*. Springer, 2016.

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

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

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

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

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

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

[99] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *J. Funct. Program.*, 17(4-5):547–612, July 2007.

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

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

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

[103] 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.

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

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

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

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

[108] 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.

[109] The F# Software Foundation. F#. See http://fsharp.org, 2014.

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

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

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

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

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

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

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

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

[118] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

[119] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[120] J. Vouillon and V. Balat. From bytecode to javassript: the js_of_ocaml compiler. *Software: Practice and Experience*, 2013.

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

[122] B. Wadge. Monads and intensionality. In *International Symposium on Lucid and Intensional Programming*, volume 95, 1995.

[123] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[124] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM, 1988.

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

[126] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

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

[128] D. Walker. *Substructural Type Systems*, pages 3–43. MIT Press.

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

[130] H. Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(02):215–286, 2007.

[131] 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.