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 may 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]. This may partly be why no unified context tracking mechanism has been developed so far.

In Chapter **??**, we argued that context-awareness had, so far, only limited influence on the design of programming languages because it is a challenge that is not easy to see. However, many of the properties that this thesis treats uniformly as *coeffects* have been previously tracked by other means. This includes special-purpose type systems, systematic approaches arising from modal logic S4, as well as techniques based on abstractions designed for other purpose, most frequently monads.

In this chapter, we describe a number of simple calculi for tracking a wide range of contextual properties. The systems are adapted from existing work, but the uniform presentation in this chapter is a novel contribution. The fact that we find a common structure in all systems presented here lets us develop unified coeffect calculi in the upcoming three chapters.

1.1 STRUCTURE OF COEFFECT SYSTEMS

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

Effect systems generally use judgements of the form $\Gamma \vdash e : \tau \,\&\, \sigma$, associating effects σ with the output type. In contrast, we choose to 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. Besides the notation, there are more important differences between effects and coeffects.

1.1.1 *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 (*abs-eff*) in Figure 1. Wadler and Thiemann [53] 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.*

1 Research programme, as introduced by Lakatos [22], is a network of scientists sharing the same basic assumptions and techniques.

$$(\textit{abs-pure})\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2 1} \qquad (\textit{abs-eff})\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \,\&\, \sigma}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\sigma} \tau_2 \,\&\, \emptyset}$$

Figure 1: Lambda abstraction for pure and effectful computations

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 [40] 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.

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

The standard lambda abstraction (*abs-pure*), shown in Figure 1, splits the free-variable context of an expression into two parts. 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). Here, the splitting is determined syntactically – the notation $\lambda x.e$ names the variable whose value comes from the call site.

The flat and structural coeffects behave in the same way. They also split context-requirements between the declaration site and call site, but they do it in two different ways.

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 context of such expression will be $\Gamma @ \{ \mathsf{gps}, \mathsf{time} \}$.

The interesting case is when we construct a lambda function $\lambda x.e$, marshall it and send it to another node. In that case, 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 send 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 capture a *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 $\Gamma @ \{ \text{gps} \}$. The current time is required from the caller of the function. So, the context requirement on the call site (dynamic scope) will be $r = \{ \text{time} \}$. 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, in case of distributed programming, the resources can be freely distributed between 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 related to variables in programs.

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 $x[\textbf{cursor}] = y[\textbf{cursor} - 1] + y[\textbf{cursor} + 1]$.

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 **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 **??**.

Unlike in flat coeffects, in the structural systems, splitting of context determined by the syntax. For example, consider a function that takes y and contains the above body: $\lambda y.x[\textbf{cursor}] = y[\textbf{cursor} - 1] + y[\textbf{cursor} + 1]$. 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 of flat coeffect systems.

Before looking at concrete flat and structural systems in mode details, we briefly overview some notation used in this thesis. As structural coeffects keep annotations as *vectors*, we 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 free variables of types τ_i can be modelled by a function $\tau_1 \times \ldots \times \tau_n \to \tau$ with a product on the left, but a single value on the right. Effect systems attach effect annotations to the result τ. In coeffect systems, we attach a coeffect annotation 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* such that $r_1, \ldots, r_n \in \mathcal{C}$. 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 lower-case letters r, s, t for scalars[2]. We also say that a *shape* of a vector $[\mathbf{r}]$ (or more generally any container) is the set of *positions* in a vector. So, a vector of length n has shape $\{1, 2, \ldots, n\}$. We discuss containers and shapes further in Chapter **??** and also discuss how our use relates to containers of Abbott, Altenkirch and Ghani [2].

Just as in the usual multiplication of a vector by scalar, we lift any binary operation on scalars into a scalar-vector one. For any binary operation on scalars $\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, 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 $\Gamma_1 \times \langle x{:}\tau \rangle \times \Gamma_2$.

1.2 FLAT COEFFECT SYSTEMS

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

In this section, we look at a number of examples ranging from Haskell's type constraints and implicit parameters to distributed computing. For three of our examples – implicit parameters, liveness analysis and data-flow – we show an ad-hoc type system that captures their properties. This serves as a basis for Chapter 2, which develops a unified flat coeffect calculus.

1.2.1 *Implicit parameters and type classes*

Haskell provides two examples of flat coeffects – type class and implicit parameter constraints [52, 23]. 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 no give a full type system.

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

IMPLICIT PARAMETERS. Implicit parameters are a special kind of variables that support dynamic scoping. They can be used 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 (which uses it) 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[3] **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)
```

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 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:

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

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

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. As a result, the latter 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

3 Haskell uses **let** ?p $= e_1$ **in** e_2, but we use a different keyword to avoid confusion.

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

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

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

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

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

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

Figure 2: Coeffect rules for tracking implicit parameters

$\Gamma\,@\,r \vdash e : \tau$ 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 sets of pairs consisting of implicit parameter names and types, i. e. $r, s, t \subseteq \mathsf{Names} \times \mathsf{Types}$. The expressions include $?\mathsf{param}$ 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 $?\mathsf{param}$ 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 ($\tau_1 \xrightarrow{s} \tau_2$).

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 ?:

$$\Gamma\,@\,\{?a : \mathsf{int}\} \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?b:\mathsf{int}\}} \mathsf{int} \quad (1)$$

$$\Gamma\,@\,\{?b : \mathsf{int}\} \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?a:\mathsf{int}\}} \mathsf{int} \quad (2)$$

$$\Gamma\,@\,\{?a : \mathsf{int}\} \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?a:\mathsf{int},?b:\mathsf{int}\}} \mathsf{int} \quad (3)$$

$$\Gamma\,@\,\emptyset \vdash \lambda x.?a + ?b : \mathsf{int} \xrightarrow{\{?a:\mathsf{int},?b:\mathsf{int}\}} \mathsf{int} \quad (4)$$

The first two examples demonstrate why 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.

In (3), the parameter $?a$ has to be provided from both the declaration-site and call-site. We describe system that supports dynamic rebinding, meaning that when the caller provides a value, it hides the value that may be available from the declaration-site. This means that 4 is a more precise typing modelling the same situation.

The semantics is defined inductively over the typing derivation:

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

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

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

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

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

Monadic semantics using the reader monads differs as follows:

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

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

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

Figure 3: Semantics of a language with implicit parameters

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 **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 the same 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, our interpretation is a function that takes a product of variables from Γ together with a hidden dictionary of implicit parameters and returns τ:

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

The hidden 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 a set of those that are specified by the type system.

A hidden dictionary also needs to be attached to all functions. 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. Let binding can be viewed as a syntactic sugar for $(\lambda x.e_2)\ e_1$ and so it is omitted for brevity.

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.

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.

The (*rdabs*) rule in Figure 3 shows a semantics that matches the usual monadic semantics using the reader monad. 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 *unit*.

As we discuss later in Section 3.?, 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 [52]. They provide a principled form of ad-hoc polymorphism (overloading). When a 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 a hidden 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:

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

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. 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, 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. [6]: *"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 is *rebound* to the resource on the new machine. This is captured by dynamic scoping rules. 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 lexical scoping rules.

A practical language that supports rebindable resources is for example Acute [34]. In the following example, we use the construct **access** Res to represent access to a rebindable resource named Res. The following simple function accessed a database and a current date and filters values based on the date:

```
let recentEvents = 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; can 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 recentEvents 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 resources non-deterministically 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 [48].

Links [9], F# WebTools and WebSharper [38, 29], ML5 and QWeSST [27, 33] and Hop [24] 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 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

a.) Set based type system for cross-compilation, inspired by Links [9]

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

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

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

b.) Version number based type system, inspired by Android API level [11]

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

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

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

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

and performs a remote call to the server-side function (how this is implemented 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 [9], functions can be annotated as client-side, server-side and database-side. F# WebTools [29] 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 [11] to cross-compile for multiple versions of the same platform.

TYPE SYSTEMS. Cross-compilation may seem similar to the tracking of resources (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 of required parameters using union. However, this symmetry does not hold for lambda abstraction (*abs*), which still uses *union*. This models the case where the function can be executed in two different ways:

- The function is represented as executable code for an environment available at the call-site and is executed there, possibly after it is marshalled and transferred to another machine.
- The function body is compiled for the environment available at the declaration-site; 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 and the implementation (and semantics) needs to be clarified. However, the example shows that the algebraic structure of coeffect annotations may be more complex. Here, using $\cap$ 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.

In addition to the work discussed already, ML5 [27] is another important work that looks at tracking of execution environments. It uses 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. We return to this topic in Chapter **??**.

1.2.3 *Liveness analysis*

Live variable analysis (LVA) [3] 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 [51] 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 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:

$$\textbf{let}\ \mathsf{constant42} = \lambda x \to 42$$

$$\textbf{let}\ \mathsf{constant} = \lambda \mathsf{value} \to \lambda x \to \mathsf{value}$$

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 declaration-site as live (this will be refined in structural liveness system).

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

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

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

$$f : \tau_1 \xrightarrow{r} \tau_2 \qquad g : \tau_2 \xrightarrow{s} \tau_3 \qquad g \circ f : \tau_1 \xrightarrow{r \sqcup s} \tau_3$$

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

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

$$f : \tau_1 \xrightarrow{r} \tau_2 \qquad h : \tau_1 \xrightarrow{s} \tau_3 \qquad \langle f, h \rangle : \tau_1 \xrightarrow{r \sqcap s} \tau_2 \times \tau_3$$

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

Figure 6: Liveness annotations with sequential and pointwise composition

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*).

The distinction between *may* and *must* analyses demonstrates the importance of interaction between contextual properties and certain language constructs such as conditionals. We discuss this in Section **??**

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 (which may be used) using the (*sub*) rule. This captures the *may* nature of the analysis.

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

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

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

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

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

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

Figure 7: Coeffect rules for tracking whole-context liveness

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 $(\lambda x.e_2)\ e_1$. Briefly, this follows from the simple observation that $r \sqcup (s \sqcap r) = r$.

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

$$[\![\Gamma \mathbin{@} \mathsf{D} \vdash c_i : \tau_i]\!] = \lambda() \to \delta(c_i) \qquad (\textit{const})$$

$$[\![\Gamma \mathbin{@} \mathsf{L} \vdash e : \tau]\!] = \lambda x \to [\![\Gamma \mathbin{@} \mathsf{D} \vdash e : \tau]\!]\ () \qquad (\textit{sub-1})$$

$$[\![\Gamma \mathbin{@} r \vdash e : \tau]\!] = \lambda x \to [\![\Gamma \mathbin{@} r \vdash e : \tau]\!]\ x \qquad (\textit{sub-2})$$

$$[\![\Gamma \mathbin{@} \mathsf{L} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!] = \lambda(x_1, \ldots, x_n) \to \lambda y \to [\![\Gamma, y : \tau_1 \mathbin{@} \mathsf{L} \vdash e : \tau_2]\!]\ (x_1, \ldots, x_n, y) \qquad (\textit{abs-1})$$

$$[\![\Gamma \mathbin{@} \mathsf{D} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!] = \lambda() \to \lambda() \to [\![\Gamma, y : \tau_1 \mathbin{@} \mathsf{D} \vdash e : \tau_2]\!]\ () \qquad (\textit{abs-2})$$

$$[\![\Gamma \mathbin{@} r \vdash e_1\ e_2 : \tau_2]\!] = \lambda x \to \mathbf{let}\ g = [\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2]\!]\ x\ \mathbf{in}\ g\ () \qquad (\textit{app-1})$$

$$[\![\Gamma \mathbin{@} \mathsf{L} \vdash e_1\ e_2 : \tau_2]\!] = \lambda x \to \mathbf{let}\ g = [\![\Gamma \mathbin{@} \mathsf{L} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2]\!]\ x\ \mathbf{in}\ g\ ([\![\Gamma \mathbin{@} \mathsf{D} \vdash e_2 : \tau_1]\!]\ ()) \qquad (\textit{app-2})$$

$$[\![\Gamma \mathbin{@} r \sqcup (s \sqcap t) \vdash e_1\ e_2 : \tau_2]\!] = \lambda x \to \mathbf{let}\ g = [\![\Gamma \mathbin{@} r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]\ x\ \mathbf{in}\ g\ ([\![\Gamma \mathbin{@} s \vdash e_2 : \tau_1]\!]\ x) \qquad (\textit{app-3})$$

Figure 8: Semantics of a language with liveness analysis

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 ?:

$$(\lambda x \to 42)\ y \qquad (1)$$
$$\mathsf{twoTimes}\ 42 \qquad (2)$$
$$(\lambda x \to x)\ 42 \qquad (3)$$

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. The type system presented above requires the semantics to *implement* dead code elimination. This means that when a function does not require an input (it is marked as dead), the semantics does not evaluate the argument and passes an empty value as the input instead.

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 X. For now, it suffices to define the semantics using two separate rules:

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

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:

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

Unlike with implicit parameters, the coeffect system for liveness tracking cannot be modelled using monads. Any monadic semantics would express functions as $\tau_1 \to M\,\tau_2$. Unless laziness is already built-in in the semantics, 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 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, the 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 thunked.

The semantics of liveness reveals a number of interesting properties too. Firstly, the semantics cannot be captured by any monad. 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 at length how the semantics of application arises from *sequential* and *pointwise* composition. This is another important aspect of coeffect systems – categorical semantics typically builds on *sequential* composition, but to model full λ calculus it needs more. For coeffect systems, we need *pointwise* composition where the same context is shared by multiple sub-expressions.

1.2.4 *Data-flow languages*

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 [50] 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* [5] build on the data-flow paradigm. The following example is inspired by the Lustre [17] language and implements 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. Formally, the constructs are defined as follows (writing x_n for n-th element of a stream x):

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

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 [49] originally proposed to use monads, while Uustalu and Vene later used comonads [44]. In Chapter 2, we extend the latter approach. However, the present chapter presents a sketch of a 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. A type system that tracks the maximal number of accessed past values annotates the context with a single integer. The current value is

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

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

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

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

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

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

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

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, **prev** (**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 data-flow, the operations combine annotations using $+$ and *max* operations:

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

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

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

$$[\![\Gamma \circledast n+1 \vdash \textbf{prev}\ e : \tau]\!] = \lambda\langle \mathbf{v}_0, \ldots, \mathbf{v}_{n+1}\rangle \to [\![\Gamma \circledast n \vdash e : \tau]\!]\ \langle \mathbf{v}_1, \ldots, \mathbf{v}_{n+1}\rangle \qquad (\textit{prev})$$

$$[\![\Gamma \circledast n \vdash e : \tau]\!] = \lambda\langle \mathbf{v}_0, \ldots, \mathbf{v}_n\rangle \to [\![\Gamma \circledast n' \vdash e : \tau]\!]\ \langle \mathbf{v}_0, \ldots, \mathbf{v}_{n'}\rangle \qquad (\textit{sub})$$

$$[\![\Gamma \circledast n \vdash \lambda y.e : \tau_1 \xrightarrow{n} \tau_2]\!] = \lambda\langle \mathbf{v}_0, \ldots \mathbf{v}_n\rangle \to \lambda(y, g) \to [\![\Gamma, y : \tau_1 \circledast n \vdash e : \tau_2]\!]\ \langle(\mathbf{v}_0, y_0), \ldots, (\mathbf{v}_n, y_n)\rangle \qquad (\textit{abs})$$

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

Figure 10: Semantics of a simple data-flow language

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).

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 [45] model causal data-flow computations using a non-empty list NeList $\tau = \tau \times (\text{NeList}\ \tau + 1)$ over the input. A function $\tau_1 \to \tau_2$ is thus modelled as NeList $\tau_1 \to \tau_2$. 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:

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

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 pairs*. To make the rules easier to follow, we write $\langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$ for an n-element list containing pairs. Pairs 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 from the declaration-site $\langle \mathbf{v}_0, \ldots, \mathbf{v}_n \rangle$ and one containing the argument provided by the call-site $\langle y_0, \ldots, v_n \rangle$. The semantics applies the well-known *zip* operation on the lists and passes the result to the body.

Finally, application (*abs*) 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. The most interesting point about the data-flow example 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, it may also track *permissions* or *capabilities* to perform some operation. Liveness can be seen 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 [13] as one, more advanced example. Calculus of capabilities of Cray et al. [10] is discussed later in Section **??**.

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 mutable variable 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 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:

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

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 perhaps 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 extension which we discuss in Section ?. The extension also lets us capture systems with effect-style lambda abstraction such as safe locking.

1.3 STRUCTURAL COEFFECT SYSTEMS

In structural coeffect systems, the additional information are 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 a number of examples. We first consider an example inspired by linear logic. Then we revisit liveness and data-flow, for which the structural system provides a more precise analysis. Finally, we look at a number of other practical uses including security, tainting and provenance tracking.

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. In this section, 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:

$$\textbf{let}\ \mathsf{constant} = \lambda y \to \lambda x \to y$$
$$\textbf{let}\ \mathsf{answer} = (\lambda x \to x)\ 42$$

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 **??**) which provide 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 Chapter **??**, but the alternative is briefly explored in Chapter **??**.

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. Other 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 point-wise 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

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

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

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 11: Structural coeffect liveness analysis

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 $\tau_1 \xrightarrow{\mathbf{t}} \tau_2$. The composition is defined as follows:

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

This definition is only for illustration and is revised in Chapter **??**. 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 $(\lambda x.e_2)\ e_1$.

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 $(\lambda x \to v)\ y$. 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:

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

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:

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

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 wrapped the entire context, i.e. $1 + (\tau_1 \times \ldots \times \tau_n)$. In the structural version, the semantics wraps 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. More formally:

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

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

Note that the product of the free variables is not an ordinary tuple of our language, but a special construction. 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:

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

a.) Semantics of ordinary expressions

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

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

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

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

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

b.) Semantics of structural context manipulation

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

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

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

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

Figure 12: Semantics of structural liveness

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 a product with a number of variables represented as another product $\mathbf{v}$ with one more variable x at the end. We shall be more precise in Chapter **??**.

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 $()$.

In application, we again need to "implement" dead code elimination. When 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 the 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.

The structural systems discussed in the upcoming sections are remarkably similar to the one shown here. We discuss two more examples in details 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. [16] 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 **??**, 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 ($*$) and addition ($+$).

Variable access (*var*) annotates a variable with 1, meaning that it has been used once. An unused variable (*weak*) is annotated with 0. Multiple occur-

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

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 13: Structural coeffect bounded reuse analysis

rences 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 $\mathbf{t}$-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 $(\mathbf{t} * \mathbf{s})$ (where $*$ is a 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.

Finally, the sub-coeffecting rule (*sub*) safely overapproximates the number of uses 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:

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

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:

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

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. [16] 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:

$$[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mathbin{@} \langle r_1, \ldots, r_n\rangle \vdash e : \tau]\!] \ :\ (\tau_1^{r_1} \times \ldots \times \tau_n^{r_n}) \to \tau$$
$$\text{where } \tau_i^{r_i} = \underbrace{\tau_i \times \ldots \times \tau_i}_{r_i\text{−times}}$$

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 variable names $\mathbf{v}_i$ for context containing multiple variables (where each variable may be available multiple times), i. e. have a type $\tau_1^{r_1} \times \ldots \times \tau_m^{r_m}$; 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 $s = \langle s_1, \ldots, s_n\rangle$:

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

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

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 differs from bounded linear logic (BLL) [16]. Using the terminology from Section **??**, our system is written in the *language semantics* style, while BLL is written in the *meta-language* style.

This means that the terms and types of our system are the terms and types of 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 $\tau_1, ..., \tau_n \mathbin{@} \langle k_1, ..., k_n \rangle$ 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:

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

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* and also revisit the BLL example in Chapter **??**.

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 **??**.

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. Our approach has two benefits – it lets you unify flat and structural systems (Chapter **??**) and it also makes it possible to build composed systems that mix both flat and structural properties.

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

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

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

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

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

Figure 14: Structural coeffect bounded reuse analysis

1.3.3 *Data-flow languages revisited*

When discussing data-flow languages in the previous 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:

$$\textbf{let}\ \mathsf{offsetZip} = \mathsf{left} + \textbf{prev}\ \mathsf{right}$$

The value offsetZip 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). As discussed in Section X, the structural system is necessary when we allow arbitrary recursion and cannot (easily) determine the number of required values statically.

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 point-wise 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 $\lambda x.\textbf{prev}\ (y + x)$ applied to an argument $\textbf{prev}\ (\textbf{prev}\ y)$. 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:

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

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 $\textbf{prev}\ (\textbf{prev}\ y)$ 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:

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

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 $\textbf{prev}\ (y + (\textbf{prev}\ (\textbf{prev}\ y)))$. Semantically, this performs stream lookups $y[1]$ and $y[3]$ where the indices are the number of enclosing **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 some wrapper (list or option type), we now wrap 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:

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

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 **prev** construct and of the structural contraction rule:

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

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

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 **prev** requires only s_i past values and so the semantics simply drops the last value.

In the (*contr*) rule, the semantic function receives $max(s, t)$ 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 properties 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 as a runtime mark (e.g. in the Perl language) or statically using a type system. Tainting can be viewed as a special case of *provenance tracking*, known from database systems [8], where values are annotated with more detailed information about their source.

Statically typed systems that based on tainting have been use to prevent cross-site scripting attacks [46] and a well known attack known as SQL injection [19, 18]. In the latter chase, 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 that stores 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. The checking of tainting 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. [47] provide the first (provably)

sound type system that guarantees non-inference and Sabelfeld et al. [32] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the FlowCaml [35] language. Finally, Russo et al. and Swamy et al. [31, 37] show that the 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:

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

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).

Abadi et al. realized that there is a number of analyses similar to secure information flow and proposed to unify them using a single model called Dependency Core Calculus (DCC) [1]. It captures other cases where some information about expression relies on properties of variables in the context where it executes. The DCC captures, for example, *binding time analysis* [41], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [42] 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 tracking of tainting, security and provenance in details, 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*. Simply accessing variable does not introduce taint, but using a variable in certain contexts – such as in arguments of query – does introduce a taint. This is captured using the standard application rule (*app*):

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

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* mod-

elling 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 meta-data (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 be evaluating certain code block in a different scope (e.g. by wrapping it in prev in data-flow), but also by calling a function that, for example, acquires new capabilities and returns those to the caller. 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. [10] 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 [43] 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.

Calculus of capabilities differs in two ways. First, it allows explicit allocation and deallocation of memory regions (and so region lifetimes do not follow strict LIFO ordering). Second, it uses continuation-passing style. We ignore the latter aspect. The following example is almost identical as 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. To type check such programs, we do not only need to know what context is required to call `calculate`. We also need to know what effects it has on the context when it evaluates and the current context meeds to be updated after a function call.

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:

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

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:

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

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, we briefly consider the problem of active contexts in the Section X of the future work chapter.

SOFTWARE UPDATING. Another example of a system that uses contextual information actively is dynamic software updating (DSU) [14, 20]. The DSU systems have the ability to update programs at runtime without stopping them. For example, Proteus developed by Stoyle et al. [36] investigates what language support is needed to enable safe dynamic software updating in C-like languages. The system is based on the idea of capabilities.

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 need to 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 from principled approaches based on comonads, ad-hoc type system extensions and static analyses as well as approaches based on monads. We looked at a number of applications inclding 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 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 **??**, respectively). We can, however, unify the two variants into a single system discussed in Chapter **??**.

2 FLAT COEFFECT CALCULUS

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

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

2.1 INTRODUCTION

In the previous chapter, we looked at three important examples of systems that track whole-context properties. The type systems for whole-context liveness (Section 1.2.3) and whole-context data-flow (Section 1.2.4) have a very similar structure – their lambda abstraction duplicates the requirements and their application arises from the combination of *sequential* and *point-wise* composition.

The system for tracking of implicit parameters (Section 1.2.1), and similar systems for rebindable resources, differ in two ways. In lambda abstraction, they split the context requirements between the declaration-site and the call-site and they use only a single operator on the indices, typically $\cup$.

2.1.1 *Contributions*

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

- We present a *flat coeffect calculus* with a type system that is parameterized by a *flat coeffect algebra* and can be instantiated to obtain all of the three examples discussed (Section 2.2).
- We give the equational theory of the calculus and discuss type-preservation for call-by-name and call-by-value reduction (Section 2.4). We also extend the calculus with pairs and recursion (Section 2.5).
- We present the semantics of the calculus in terms of *indexed comonads*, which is a generalization of comonads, a category-theoretical dual of monads (Section 2.3). The semantics provides deeper insight into how (and why) the calculus works.
- We develop an alternative presentation of the system in terms of a simple structure (semi-lattice) and use it to develop a type inference algorithm for flat coeffect calculus.

2.1.2 *Related work*

The development in this chapter can be seen as a counterpart to the well-known development of *effect systems* [15] and the use of *monads* [26] in programming languages. The syntax and type system of the flat coeffect calculus follows similar style as effect systems [25, 39], but differs in the structure, as explained in the previous chapter, most importantly in lambda abstraction.

Wadler and Thiemann famously show a correspondence between effect systems to monads [53], relating effectful functions $\tau_1 \xrightarrow{\sigma} \tau_2$ to monadic computations $M^{\sigma}\tau_1 \to \tau_2$. In this chapter, we show a similar correspondence between *coeffect systems* and *comonads*. However, due to the asymmetry of λ-calculus, this is not a simple mechanical dualization.

The main purpose of the comonadic semantics presented in this chapter is to provide a semantic motivation for the flat coeffect calculus. The semantics is inspired by the work of Uustalu and Vene [45] who present the semantics of contextual computations (mainly for data-flow) in terms of comonadic functions $C\tau_1 \to \tau_2$. Our *indexed comonads* annotate the structure with information about the required context, i. e. $C^{\sigma}\tau_1 \to \tau_2$. This is similar to the recent work on *parameterized monads* by Katsumata [21].

2.2 FLAT COEFFECT CALCULUS

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

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

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

2.2.1 *Reconciling lambda abstraction*

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

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

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

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

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

2.2.2 *Flat coeffect algebra*

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

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

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

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

In addition, the following distributivity axioms hold:

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

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

- The monoid $(\mathcal{C}, \circledast, \mathsf{use})$ represents *sequential* composition of (semantic) functions. The laws of a monoid are required in order to form a category structure in the categorical model (Section 2.3).
- The monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ represents *point-wise* composition, i. e. the case when the same context is passed to multiple (independent) computations. The monoid laws guarantee that usual syntactic transformations on tuples and the unit value (Section 2.5) preserve the coeffect.
- For the $\wedge$ operation, we require associativity and idempotence. The idempotence requirement makes it possible to duplicate the coeffects and place the same requirement on both call-site and declaration-site,

i. e. it makes the (*abs-df*) rule admissible. In some cases, the operator forms a monoid with the unit being the greatest element of the set. This alternative is discussed when we consider recursion (Section 2.5).

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

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

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

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

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

2.2.3 *Understanding flat coeffects*

Before looking at the type system in Figure 15, let us clarify how the rules should be understood. The coeffect calculus provides both analysis of context dependence (type system) and semantics for context (how it is propagated). These two aspects provide different ways of reading the judgements $\Gamma\,@\,r \vdash e : \tau$ and the typing rules used to define it.

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

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

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

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

The reason for this asymmetry follows from the fact that the context appears in a *negative position* in the semantic model (Section 2.3). It means that we need to be careful about using the words *split* and *merge*, because they can be read as meaning opposite things. To disambiguate, we always use the term *context requirements* when using the syntactic view and *context capabilities* or just *available context* when using the semantic view.

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

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

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

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

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

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

Figure 15: Type system for the flat coeffect calculus

2.2.4 *Flat coeffect types*

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. Following the top-down syntactic reading, the (*sub*) rule allows us to treat an expression with fewer context requirements as an expression with more context requirements.

The (*abs*) rule is defined as discussed in Section 2.2.1. The body is annotated with context requirements $r \wedge s$, which are then split between the context-requirements on the declaration-site r and context-requirements on the call-site s. Examples of the $\wedge$ operator are discussed in the next section.

In function application (*app*), context requirements of both expressions and the function are combined as discussed in Chapter 1. The pointwise composition $\oplus$ is used to combine the context requirements of the expression representing a function r and the context requirements of the argument, sequentially composed with the context-requirements of the function $s \circledast t$.

The type system also includes a rule for let-binding. The rule is *not* equivalent to the derivation for $(\lambda x.e_2)\ e_1$, but it represents one admissible typing derivation. We return to let-binding after looking a number of examples. Additional constructs such as recursion and tuples are covered in Section 2.5.

2.2.5 *Examples of flat coeffects*

The flat coeffect calculus generalizes the 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. The following summary defines the systems for implicit parameters, liveness and data-flow. For the latter two, we obtain more general (but compatible) rule for lambda abstraction.

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

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

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

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

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

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

As in Section 1.2.3, sequential composition $\circledast$ is modelled by the join operation $\sqcup$ and point-wise composition $\oplus$ is modelled by meet $\sqcap$. Two-point lattice is a commutative, idempotent monoid. The distributivity $(r \sqcap s) \sqcup t = (r \sqcup t) \sqcap (s \sqcup t)$ does not hold for *every* lattice, but it trivially holds for a two-point lattice used here.

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

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

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

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

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

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

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

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

2.2.6 *Typing of let binding*

Recall the (*let*) rule in Figure 15. It annotates the expression $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ with context requirements $s \oplus (s \circledast r)$. This is a special case of typing of an expression $(\lambda x.e_2)\ e_1$, using the idempotence of $\wedge$ as follows:

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

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

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

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

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

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

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

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

2.3 CATEGORICAL MOTIVATION

a

2.10 COEFFECT SEMANTICS USING INDEXED COMONADS

The approach of *categorical semantics* interprets terms as morphisms in some category. For typed calculi, typing judgments $x_1 : \tau_1 \dots x_n : \tau_n \vdash e : \tau$ are usually mapped to morphisms $[\![\tau_1]\!] \times \dots \times [\![\tau_n]\!] \to [\![\tau]\!]$. Moggi showed the semantics of various effectful computations can be captured generally using the (*strong*) *monad* structure [26]. Dually, Uustalu and Vene showed that (*monoidal*) *comonads* capture various kinds of context-dependent computation [45].

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

INDEXED COMONADS. Uustalu and Vene's approach interprets well-typed terms as morphisms $C(\tau_1 \times \dots \times \tau_n) \to \tau$, where C encodes contexts and has a comonad structure [45]. Indexed comonads comprise a *family* of object mappings C^r indexed by a coeffect r describing the contextual requirements satisfied by the encoded context. We interpret judgments $C^r(x_1 : \tau_1, \dots, x_n : \tau_n) \vdash e : \tau$ as morphisms $C^r([\![\tau_1]\!] \times \dots \times [\![\tau_n]\!]) \to [\![\tau]\!]$.

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

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

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

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

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

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

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

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

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

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

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

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

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

2.10.1 *Monoidal indexed comonads.*

Indexed comonads provide a semantics to sequential composition, but additional structure is needed for the semantics of the full coeffect calculus. Uustalu and Vene [45] additionally require a (*lax semi-*) *monoidal comonad* structure, which provides a monoidal operation $m : CA \times CB \to C(A \times B)$ for merging contexts (used in the semantics of abstraction).

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

- $m_{r,s} : C^r A \times C^s B \to C^{(r \wedge s)}(A \times B)$ natural in A, B;
- $n_{r,s} : C^{(r \vee s)}(A \times B) \to C^r A \times C^s B$ natural in A, B;

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

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

For the sake of brevity, we elide the indexed versions of the laws required by Uustalu and Vene (e. g. most importantly, merging two contexts and then adding the third is equivalent to merging the last two and then adding the first; similar rule holds is required for splitting).

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

$$m_{L,L}(x, y) = (x, y) \qquad n_{D,D}\ () = ((), ()) \qquad n_{D,L}(x, y) = ((), y)$$
$$m_{r,s}\ (x, y) = () \qquad n_{L,D}(x, y) = (x, ()) \qquad n_{L,L}(x, y) = (x, y)$$

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

Figure 16: Categorical semantics for the coeffect calculus

EXAMPLE Uustalu and Vene model causal dataflow computations using the non-empty list comonad $\mathsf{NEList}\ A = A \times (1 + \mathsf{NEList}\ A)$ [45]. Whilst this comonad implies a trivial indexed comonad, we define an indexed comonad with integer indices for the number of past values demanded of the context.

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

2.10.2 *Categorical Semantics.*

Figure 16 shows the categorical semantics of the coeffect calculus using additional operations π_i for projection of the i^{th} element of a product, usual *curry* and *uncurry* operations, and $\Delta : A \to A \times A$ duplicating a value. While C^r is a family of object mappings, it is promoted to a family of functors with the derived morphism mapping $C^r(f) = (f \circ \varepsilon)^\dagger_{\mathsf{use},r}$.

The semantics of variable access and abstraction are the same as in Uustalu and Vene's semantics, modulo coeffects. Abstraction uses $\mathsf{m}_{r,s}$ to merge the outer context with the argument context for the context of the function body. The indices of use for ε and r, s for $\mathsf{m}_{r,s}$ match the coeffects of the terms. The semantics of application is more complex. It first duplicates the free-variable values inside the context and then splits this context using $\mathsf{n}_{r,s\oplus t}$. The two contexts (with different coeffects) are passed to the two subexpressions, where the argument subexpression, passed a context $(s \oplus t)$, is coextended to produce a context s which is passed into the parameter of the function subexpression (*cf.* given $f : A \to (B \to C)$, $g : A \to B$, then $\mathit{uncurry}\, f \circ (\mathrm{id} \times g) \circ \Delta : A \to C$).

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

2.11 SYNTAX-BASED EQUATIONAL THEORY

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

In a pure λ-calculus, β and η equality for functions (also called *local soundness* and *completeness* respectively [30]) describe how pairs of abstraction and application can be eliminated: $(\lambda x.e_2)e_1 \equiv_\beta e_1[x \leftarrow e_2]$ and $(\lambda x.e\,x) \equiv_\eta e$. The β equality rule, using the usual Barendregt convention of syntactic substitution, implies a *reduction*, giving part of an operational semantics for the calculus.

The call-by-name evaluation strategy modelled by β-reduction is not suitable for impure calculi therefore a restricted β rule, corresponding to call-by-value, is used, i. e. $(\lambda x.e_2)v \equiv e_2[x \leftarrow v]$. Such reduction can be encoded

by a *let*-binding term, $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$, which corresponds to sequential composition of two computations, where the resulting pure value of e_1 is substituted into e_2 [12, 26].

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

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

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

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

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

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

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

Our calculus can be extended with *let*-binding and (2). However, we also consider the cases when a syntactic substitution $e_2[x \leftarrow e_1]$ has the coeffects specified by the above rule (2) and prove *subject reduction* theorem for certain coeffect calculi. We consider two common special cases when the coeffect of variables `use` is the greatest ($\top$) or least ($\bot$) element of the semi-lattice $(S, \vee)$ and derive additional conditions that have to hold about the coeffect algebra:

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

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

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

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

Proof. A direct consequence of Lemma 1. □

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

However, the coeffect calculus also captures context-dependence in languages with more complex evaluation strategies than *call-by-name* reduction

based on syntactic substitution. In particular, syntactic substitution does not provide a suitable evaluation for dataflow (because a substituted expression needs to capture the context of the original scope).

Nevertheless, the above results show that – unlike effects – context-dependent properties can be integrated with *call-by-name* languages. Our work also provides a model of existing work, namely Haskell implicit parameters [23].

2.12 RELATED AND FURTHER WORK

This paper follows the approaches of effect systems [15, 39, 53] and categorical semantics based on monads and comonads [26, 45]. Syntactically, *coeffects* differ from *effects* in that they model systems where λ-abstraction may split contextual requirements between the declaration-site and call-site.

Our *indexed (monoidal) comonads* (§2.10) fill the gap between (non-indexed) *(monoidal) comonads* of Uustalu and Vene [45] and indexed monads of Atkey [4], Wadler and Thiemann [53]. Interestingly, *indexed* comonads are *more general* than comonads, capturing more notions of context-dependence (§**??**).

COMONADS AND MODAL LOGICS. Bierman and de Paiva [7] model the $\Box$ modality of an intuitionistic S4 modal logic using monoidal comonads, which links our calculus to modal logics. This link can be materialized in two ways.

Pfenning et al. and Nanevski et al. derive term languages using the Curry-Howard correspondence [30, 7, 28], building a *metalanguage* (akin to Moggi's monadic metalanguage [26]) that includes $\Box$ as a type constructor. For example, in [30], the modal type $\Box\tau$ represents closed terms. In contrast, the *semantic* approach uses monads or comonads *only* as a semantics. This has been employed by Uustalu and Vene and (again) Moggi [26, 45]. We follow the semantic approach.

Nanevski et al. extend an S4 term language to a *contextual* modal type theory (CMTT) [28]. The *context* is a set of variables required by a computation, which makes CMTT useful for meta-programming and staged computations. Our contextual types are indexed by a coeffect algebra, which is more general and can capture variable contexts, but also integers, two-point lattices, *etc.*.

The work on CMTT suggests two extensions to coeffects. The first is developing the logical foundations. We briefly considered special cases of our system that permits local soundness in §2.11 and local completeness can be treated similarly. The second problem is developing the coeffects *metalanguage*. The use of coeffect algebras would provide an additional flexibility over CMTT, allowing a wider range of applications.

RELATING EFFECTS AND COEFFECTS. The difference between effects and coeffects is mainly in the (*abs*) rule. While the semantic model (monads vs. comonads) is very different, we can consider extending the two to obtain equivalent syntactic rules. To allow splitting of implicit parameters in lambda abstraction, the reader monad needs an operation that eagerly performs some effects of a function: $(\tau_1 \to M^{r \oplus s} \tau_2) \to M^r(\tau_1 \to M^s \tau_2)$. To obtain a pure lambda abstraction for coeffects, we need to restrict the $m_{r,s}$ operation of indexed comonads, so that the first parameter is annotated with use (meaning no effects): $C^{\mathsf{use}} A \times C^r B \to C^r(A \times B)$.

STRUCTURAL COEFFECTS. To make the liveness analysis practical, we need to associate information with individual variables (rather than the entire context). We can generalize the calculus from this paper by adding a

product operation $\times$ to the coeffect algebra. A variable context $x : \tau_1, y : \tau_2, z : \tau_3$ is then annotated with $r \times s \times t$ where each component of the tag corresponds to a single variable. The system then needs to be extended with structural rules such as:

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

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

The structural coeffect system is related to bunched typing [**?**] (but generalizes it by adding indices). We are currently investigating how to use structural coeffects to capture fine-grained context-dependence properties such as secure information flow [47] or, more generally, those captured by dependency core calculus [**?**].

2.13 CONCLUSIONS

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

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

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

BIBLIOGRAPHY

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

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] A. Filinski. Monads in action. In *Proceedings of POPL*, 2010.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

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

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

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

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

[31] 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.

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

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

[34] 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.

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

[36] 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.

[37] 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.

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

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

[40] 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.

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

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

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

[44] 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.

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

[46] 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.

[47] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[48] J. Vouillon and V. Balat. From bytecode to javassript: the js_of_ocaml compiler. *Software: Practice and Experience*, 2013.

[49] B. Wadge. Monads and intensionality. In *International Symposium on Lucid and Intensional Programming*, volume 95, 1995.

[50] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[51] 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.

[52] 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.

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