CONTENTS

1 STRUCTURAL COEFFECT LANGUAGE

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

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

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

1.1 INTRODUCTION

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

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

1.1.1 *Contributions*

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

- We present a *structural coeffect calculus* with a type system that is parameterized by a *structural coeffect algebra* and can be instantiated to obtain all of the three examples discussed (Section 1.2).
- We give the equational theory of the calculus. We prove the type-preservation property for all structural calculi for both call-by-name and call-by-value (Section 1.4).
- We show how to extend indexed comonads introduced in the previous section to *structural indexed comonads* and use them to give the seman-

tics of structural coeffect calculus (Section 1.3). As with the flat version, the categorical semantics provides a motivation for the design of the calculus.

1.1.2 *Related work*

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

The work in this chapter is more closely related to sub-structural type systems [84]. While sub-structural systems remove some or all of *weakening*, *contraction* and *exchange* rules, our systems keep them, but use them to manipulate both the context and its annotations.

Our work follows the language semantics style in that we provide a structural semantics to the terms of ordinary λ-calculus. The most closely related work has been done in the meta-langauge style, which extends the terms and types with constructs working with the context explicitly. This includes Contextual Modal Type Theory (CMTT) [47], where variables may be of a type $A[\Psi]$ denoting a value of type A that requires context Ψ. In CMTT, $A[\Psi]$ is a first-class type, while structural coeffect systems do not expose coeffect annotations as stand-alone types.

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

1.2 STRUCTURAL COEFFECT CALCULUS

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

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

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

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

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

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

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

1.2.1 *Structural coeffect algebra*

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

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

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

In addition, the following distributivity axioms hold:

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

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

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

Definition 2. *A* structural coeffect algebra *is formed by a structural coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *equipped with the following additional structures:*

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

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

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

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

1.2.2 *Structural coeffect types*

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

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

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

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

STRUCTURAL RULES. The remaining rules, shown in Figure 1 (b), are not syntax-directed. They allow different transformation of the free variable context. We include sub-coeffecting (*sub*) as one of the rules, allowing sub-coeffecting on coeffect scalars belonging to individual variables. The remaining rules capture *weakening*, *exchange* and *contraction* known from sub-structural systems.

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

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

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

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

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

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

b.) Structural rules for context manipulation:

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

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

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

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

Figure 1: Type system for the structural coeffect calculus

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

1.2.3 *Understanding structural coeffects*

The type system for structural coeffects appears more complicated when compared to the flat version, but it is in many ways simpler – it removes the ambiguity arising from the use of $\wedge$ in lambda abstraction and, as discussed in Section 1.4, has a cleaner equational theory.

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

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

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

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

Remark 1 (Let binding)**.** *In a structural coeffect calculus, the typing of* $(\lambda x.e_2)\ e_1$ *is equivalent to the typing of* $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$.

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

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

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

1.2.4 *Examples of structural coeffects*

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

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

Example 1 (Structural liveness)**.** *The structural coeffect algebra for liveness is formed by* $(\mathcal{L}, \sqcap, \sqcup, \mathsf{L}, \mathsf{D}, \sqsubseteq)$*, where* $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ *is the same two-point lattice as in the flat version, that is* $\mathsf{D} \sqsubseteq \mathsf{L}$ *with a join* $\sqcup$ *and meet* $\sqcap$.

Example 2 (Structural data-flow)**.** *In data-flow, context is annotated with natural numbers and the flat coeffect algebra is formed by* $(\mathbb{N}, +, max, 0, 0, \leqslant)$.

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

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

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

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

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

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

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

Example 3 (Bounded variable reuse)**.** *The structural coeffect algebra for tracking bounded variable use is given by* $(\mathbb{N}, *, +, 1, 0, \leqslant)$

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

1.3 CATEGORICAL MOTIVATION

When introducing structural coeffect systems in Section **??**, we included a concrete semantics of structural liveness and bounded variable reuse. In this section, we generalize the examples using the notion of *structural indexed comonad*, which is an extension of *indexed comonad* structure. As in the previous chapter, the main aim of this section is to motivate and explain the design of the structural coeffect calculus shown in Section 1.2. The semantics highlights the similarities and differences between the two systems.

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

1.3.1 *Semantics of vectors*

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

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

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

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

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

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

1.3.2 *Indexed comonads, revisited*

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

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

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

Definition 3. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with a point-wise extension of the* $\circledast$ *operator to a vector (written as* $t \circledast \mathbf{s}$*) and an operation lifting scalars to vectors* $\langle - \rangle$*, an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^{\mathbf{r}}, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{s,\mathbf{r}})$*:*

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

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

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

1.3.3 *Structural indexed comonads*

The flat indexed comonad structure extends indexed comonads with operations $\mathsf{merge}_{r,s}$ and $\mathsf{split}_{r,s}$ that combine or split the additional (flat) context and are annotated with the flat coeffect operations $\wedge$ and $\oplus$, respectively. In the structural version, we use corresponding operations that operate on variable vectors represented using $\hat{\times}$ and are annotated with a tensor $\times$ which mirrors the variable structure.

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

Definition 4. *Given a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *with operations* $\langle - \rangle$ *and* $\circledast$*, a* structural indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$, $\mathsf{dup}_{r,s}$ *and* $\mathsf{lift}_{r',r}$ *where:*

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

Such that the following equalities hold:

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

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

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

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

1.3.4 *Semantics of structural caluculus*

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

The latter aspect is fully captured by the semantics shown in this section. The earlier point is left to the concrete notion of structural coeffect. Our model still gives us the flexibility of defining the concrete representation of variable vectors. We explore a number of examples in Section 1.3.5 and start by looking at the unified categorical semantics defined in terms of *structural indexed comonads*.

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

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

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

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

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

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

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

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

Assuming the following auxiliary definitions:

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

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

Figure 2: Categorical semantics of the structural coeffect calculus

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

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

Note that the instances of flat indexed comonad ignored the fact that the variable context wrapped in the data structure is a product. This is not generally the case for the structural indexed comonads – the definitions shown in Section 1.3.5 are given specifically for $C^{\langle r_1, \ldots, r_n \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ rather than more generally for $C^{r}\alpha$. The need for examining the structure of the variable context is another reason for using $-\hat{\times}-$ when interpreting expressions in contexts.

EXPRESSIONS. The semantics of strucutural coeffect calculi is shown in Figure 2. As in the previous chapter, the semantics is written in a programming language style using constructs such as let-binding rather than using a categorical (point-free) notation. As before, the semantics can be written using standard primitives (currying, uncurrying, function pairing etc.).

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

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

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

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

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

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

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

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

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

Remark 2 (Correspondence)**.** *In all of the typing rules of the structural coeffect system, the context annotations r and s of typing judgements $\Gamma @ r \vdash e : \tau$ and function types $\tau_1 \xrightarrow{s} \tau_2$ correspond to the indices of mappings C^r and $C^{\langle s \rangle}$ in the corresponding semantic function defined by $[\![\Gamma @ r \vdash e : \tau]\!]$.*

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

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

1.3.5 *Examples of structural indexed comonads*

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

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

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

Example 4 (Indexed list for data-flow)**.** *The indexed list model of data-flow computations is defined over a structural coeffect algebra $(\mathbb{N}, +, max, 0, 0, \leqslant)$. The data type $C^{\langle n_1, \dots, n_k \rangle}$ is indexed by required number of past variables for each individual variable. It is defined over a vector of variables $\alpha_1 \hat{\times} \dots \hat{\times} \alpha_k$ and it keeps a product containing a current value followed by n_i past values:*

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

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

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

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

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

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

$$\begin{aligned}&\mathsf{cobind}_{m,\langle n_1,\ldots,n_k\rangle}\ f\ \langle\langle a_{1,0},\ldots a_{1,m+n_1}\rangle,\ldots,\langle a_{k,0},\ldots a_{k,m+n_k}\rangle\rangle = \\ &\quad\langle\langle\ f\langle\langle a_{1,0},\ldots a_{1,n_1}\rangle,\ldots,\langle a_{k,0},\ldots a_{k,n_k}\rangle\rangle,\ \ldots\ , \\ &\qquad f\langle\langle a_{1,m},\ldots a_{1,m+n_1}\rangle,\ldots,\langle a_{k,m},\ldots a_{k,m+n_k}\rangle\rangle\ \ \rangle\rangle\end{aligned}$$

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

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

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

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

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

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

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

Example 5 (Indexed list for bounded reuse)**.** *The indexed list model of bounded variable reuse is defined over a structural coeffect algebra* $(\mathbb{N}, *, +, 1, 0, \leqslant)$. *The data type* $C^{\langle n_1,\ldots,n_k\rangle}$ *is a vector containing* n_i *values of* i-*th variable:*

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

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

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

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

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

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

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

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

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

Example 6 (Structural indexed option)**.** *Given a structural coeffect algebra formed by* $(\{\mathsf{L},\mathsf{D}\},\sqcap,\sqcup,\mathsf{L},\mathsf{D},\sqsubseteq)$ *and the indexed option data type* D^l*, such that* $D^{\mathsf{D}}\alpha = 1$ *and* $D^{\mathsf{L}}\alpha = \alpha$*, the data type for structural indexed option comonad is:*

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

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

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

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

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

The dup operation resembles with the flat version of split – this is expected as duplication in the flat calculus is performed by first duplicating the variable context (using map) and then applying split. Here, the duplication returns a pair which may or may not contain value, depending on the annotations.

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

1.4 EQUATIONAL THEORY

??

1.4.1 *Properties of reductions*

Similarly to the flat version, the λ_{Cs} calculus is defined abstractly. We cannot define its operational meaning, because that will differ for every concrete application. For example, when tracking array accesses, variables are interpreted as arrays and $a_{[n]}$ denotes access to a specified element.

Just like previously, we can state general properties of the reductions. As the syntax of expressions is the same for λ_{Cs} as for λ_{Cf}, the substitution and reduction $\twoheadrightarrow_\beta$ are also the same and can be found in Figure **??**.

The structural coeffect calculus λ_{Cs} associates information with individual variables. This means that when an expression requires certain context, we know from what scope it comes – the context must be provided by a scope that defines the associated variable, which is either a lambda abstraction or global scope. This distinguishes the structural system from the flat system where context could have been provided by any scope and the lambda rule allowed arbitrary splitting of context requirements between the two scopes (or declaration and caller site).

INTERNALIZED SUBSTITUTION. Before looking at properties of the evaluation, we consider let binding, which can be viewed as internalized substitution. The typing rule λ_{Cs} -Tlet can be derived from application and abstraction as follows.

Lemma 1 (Definition of let binding)**.** *If* $C^r\Gamma \vdash (\lambda x.e_2)\ e_1 : \tau_2$ *then* $C^r\Gamma \vdash$ **let** $x = e_1$ **in** $e_2 : \tau_2$.

Proof. The premises and conclusions of a typing derivation of $(\lambda x.e_2)\ e_1$ correspond with the typing rule λ_{Cs} -Tlet:

$$\dfrac{\dfrac{C^{r \times s}(\Gamma_1, \nu : \tau_1) \vdash e_2 : \tau_2 \qquad \nu \notin \Gamma_1}{C^r\Gamma_1 \vdash \lambda\nu.e_2 : C^s\tau_1 \to \tau_2} \qquad C^t\Gamma_2 \vdash e_1 : \tau_1}{C^{r \times (s \vee t)}(\Gamma_1, \Gamma_2) \vdash (\lambda\nu.e_2)\ e_1 : \tau_2} \qquad \square$$

The term e_2 which is substituted in e_1 is checked in a different variable and coeffect context $C^t\Gamma_2$. This is common in sub-structural systems where a variable cannot be freely used repeatedly. The context Γ_2 is used in place of the variable that we are substituting for. The let binding captures substitution for a specific variable (the context is of a form $C^{r \times s}\Gamma, \nu : \tau$). For a general substitution, we need to define the notion of context with a hole.

SUBSTITUTION AND HOLES. In λ_{Cs}, the structure of the variable context is not a set, but a tree. When substituting for a variable, we need to replace the variable in the context with the context of the substituted expression. In general, this can occur anywhere in the tree. To formulate the statement, we define contexts with holes, written $\Delta[-]$. Note that there is a hole in the free variable context and in a corresponding part of the coeffect tag:

$$\begin{array}{rcll}
\Delta[-] & ::= & C^1() & \\
& \mid & C^r(x : \tau) & \\
& \mid & C^-(-) & \\
& \mid & C^{r_1 \times r_2}(\Gamma_1, \Gamma_2) & (\text{where } C^{r_i}\Gamma_i \in \Delta[-])
\end{array}$$

Assuming we have a context with hole $C^r\Gamma \in \Delta[-]$, the hole filling operation $C^r\Gamma[r'|\Gamma']$ fills the hole in the variable context with Γ' and the corresponding

$$\begin{aligned}
C^1()[r'|\Gamma'] &= C^1() \\
C^r(x:\tau)[r'|\Gamma'] &= C^r(x:\tau) \\
C^-(-)[r'|\Gamma'] &= C^{r'}\Gamma' \\
C^{r_1 \times r_2}(\Gamma_1, \Gamma_2)[r'|\Gamma'] &= C^{r_1' \times r_2'}(\Gamma_1', \Gamma_2') \\
& \quad \text{where } C^{r_i'}\Gamma_i' = C^{r_i}\Gamma_i[r'|\Gamma']
\end{aligned}$$

Figure 3: The definition of hole filling operation for $\Delta[-]$

coeffect tag hole with r'. The operation is defined in Figure 3. Using contexts with holes, we can now formulate the general substitution lemma for λ_{Cs}.

Lemma 2 (Substitution Lemma). *If* $C^r\Gamma[R|v : \tau'] \vdash e : \tau$ *and* $C^S\Gamma' \vdash e' : \tau'$ *then* $C^r\Gamma[R \vee S|\Gamma'] \vdash e[v \leftarrow e'] : \tau$.

Proof. Proceeds by rule induction over $\vdash$ using the properties of structural coeffect tag structure $(S, \vee, 0, \times, 1)$ (see Appendix **??**). □

Theorem 1 (Subject reduction). *If* $C^r\Gamma \vdash e_1 : \tau$ *and* $e_1 \twoheadrightarrow_\beta e_2$ *then* $C^r\Gamma \vdash e_2 : \tau$.

Proof. Direct consequence of Lemma 2 (see Appendix **??**). □

LOCAL SOUNDNESS AND COMPLETENESS. As with the previous calculus, we want to guarantee that the introduction and elimination rules (λ_{Cs} -Tfun and λ_{Cs} -Tapp) are appropriately strong. This can be done by showing *local soundness* and *local completeness*, which correspond to β-reduction and η-expansion. Former is a special case of subject reduction and the latter is proved by a simple derivation:

Theorem 2 (Local soundness). *If* $C^r\Gamma \vdash (\lambda x.e_2)\ e_1 : \tau$ *then* $C^r\Gamma \vdash e_2[x \leftarrow e_1] : \tau$.

Proof. Special case of subject reduction (Theorem 1). □

Theorem 3 (Local completeness). *If* $C^r\Gamma \vdash f : C^s\tau_1 \rightarrow \tau_2$ *then* $C^r\Gamma \vdash \lambda x.fx : C^s\tau_1 \rightarrow \tau_2$.

Proof. The property is proved by the following typing derivation:

$$\dfrac{\dfrac{C^r\Gamma \vdash f : C^s\tau_1 \rightarrow \tau_2 \qquad C^0(x:\tau_1) \vdash x : \tau_1}{C^{r \times (s \vee 0)}(\Gamma, x:\tau_1) \vdash f\ x : \tau_2}}{C^r\Gamma \vdash \lambda x.fx : C^s\tau_1 \rightarrow \tau_2}$$ □

In the last step, we use the *lower bound* property of structural coeffect tag, which guarantees that $s \vee 0 = s$. Recall that in λ_{Cf}, the typing derivation for $\lambda x.fx$ required for local completeness was not the only possible derivation. In the last step, it was possible to split the coeffect tag arbitrarily between the context and the function type.

In the λ_{Cs} calculus, this is not, in general, the case. The $\times$ operator is not required to be associative and to have units and so a unique splitting may exist. For example, if we define $\times$ as the operator of a *free magma*, then it is invertible and for a given t, there are unique r and s such that $t = r \times s$. However, if the $\times$ operation has additional properties, then there may be other possible derivation.

1.5 EXAMPLES OF STRUCTURAL COEFFECTS

1.5.1 *Example: Liveness analysis*

1.5.2 *Example: Data-flow (revisited)*

TODO: Also, consider additional language features that we consider for flat coeffects (mainly recursion and possibly conditionals)

1.6 CONCLUSIONS

TODO: (...)

APPENDIX A 2

2.1 INTERNALIZED SUBSTITUTION

2.1.1 *First transformation*

$$(\mathbf{glet}\ x = e_1\ \mathbf{in}\ e_2)\ e_3 \rightsquigarrow \mathbf{glet}\ x = e_1\ \mathbf{in}\ (e_2\ e_3)$$

$$\textit{(app)}\ \dfrac{\textit{(glet)}\ \dfrac{\Gamma @ s \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 @ r \vdash e_2 : \tau_3 \xrightarrow{t} \tau_2}{\Gamma @ r \oplus (s \circledast r) \vdash \mathbf{glet}\ x = e_1\ \mathbf{in}\ e_2 : \tau_3 \xrightarrow{t} \tau_2} \qquad \Gamma @ u \vdash e_3 : \tau_3}{\Gamma @ (r \oplus (s \circledast r)) \oplus (u \circledast t) \vdash (\mathbf{glet}\ x = e_1\ \mathbf{in}\ e_2)\ e_3 : \tau_2}$$

to

$$\textit{(glet)}\ \dfrac{\Gamma @ s \vdash e_1 : \tau_1 \qquad \textit{(app)}\ \dfrac{\Gamma, x{:}\tau_1 @ r \vdash e_2 : \tau_3 \xrightarrow{t} \tau_2 \qquad \Gamma @ u \vdash e_3 : \tau_3}{\Gamma @ r \oplus (u \circledast t) \vdash e_2\ e_3 : \tau_2}}{\Gamma @ (r \oplus (u \circledast t)) \oplus (s \circledast (r \oplus (u \circledast t))) \vdash \mathbf{glet}\ x = e_1\ \mathbf{in}\ (e_2\ e_3) : \tau_2}$$

meaning

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

2.1.2 *Second transformation*

Second transformation

$$\textit{(glet)}\ \dfrac{\Gamma @ s \vdash e_s : \tau_s \qquad \Gamma, x{:}\tau_1 @ r \vdash e_r : \tau_r \qquad \Gamma, x{:}\tau_1 @ t \vdash e_t : \tau_t}{\Gamma @ t \oplus ((r \oplus (s \circledast r)) \circledast t) \vdash \mathbf{glet}\ x_r = (\mathbf{glet}\ x_s = e_s\ \mathbf{in}\ e_r)\ \mathbf{in}\ e_t : \tau_t}$$

or

$$\textit{(glet)}\ \dfrac{\Gamma @ s \vdash e_s : \tau_s \qquad \Gamma, x{:}\tau_1 @ r \vdash e_r : \tau_r \qquad \Gamma, x{:}\tau_1 @ t \vdash e_t : \tau_t}{\Gamma @ (t \oplus (r \circledast t)) \oplus (s \circledast (t \oplus (r \circledast t))) \vdash \mathbf{glet}\ x_s = e_s\ \mathbf{in}\ (\mathbf{glet}\ x_r = e_r\ \mathbf{in}\ e_t) : \tau_t}$$

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

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

require

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

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

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

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

[16] 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.

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

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

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

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

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

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

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

[24] 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.

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

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

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

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

[36] 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.

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

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

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

[40] 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.

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

[42] 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.

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

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

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

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

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

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

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

[50] D. Orchard. Programming contextual computations.

[51] D. Orchard and A. Mycroft. A notation for comonads. In *Post-Proceedings of IFL'12 (to appear)*, LNCS. Springer Berlin / Heidelberg, 2012.

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

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

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

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

[56] 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.

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

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

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

[60] 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.

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

[62] 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.

[63] 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.

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

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

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

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

[68] 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.

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

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

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

[72] 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.

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

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

[75] 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.

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

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

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

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

[80] 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.

[81] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.

[82] 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.

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

[84] D. Walker. *Substructural Type Systems*, pages 3–43. MIT Press.