ABSTRACT

The development of programming languages needs to reflect important changes in the way programs execute. In recent years, this has included the development of parallel programming models (in reaction to the multi-core revolution) or improvements in data access technologies. This thesis is a response to another such revolution – the diversification of devices and systems where programs run.

The key point made by this thesis is the realization that an execution environment or a *context* is fundamental for writing modern applications and that programming languages should provide abstractions for programming with context and verifying how it is accessed.

We identify a number of program properties that were not connected before, but model some notion of context. Our examples include tracking different execution platforms (and their versions) in cross-platform development, resources available in different execution environments (e. g. GPS sensor on a phone and database on the server), but also more traditional notions such as variable usage (e. g. in liveness analaysis and linear logics) or past values in stream-based dataflow programming. Our first contribution is the discovery of the connection between the above examples and their novel presentation in the form of calculi (*coeffect systems*). The presented type systems and formal semantics highlight the relationship between different notions of context.

Our second contribution is the definition of two unified coeffect calculi that capture the common structure of the examples. In particular, our *flat coeffect calculus* models languages with contextual properties of the execution environment and our *structural coeffect calculus* models languages where the contextual properties are attached to the variable usage. We define the semantics of the calculi in terms of category theoretical structure of an *indexed comonad* (based on dualisation of the well-known monad structure), use it to define operational semantics and prove safety result for the calculi.

Our third contribution is a novel presentation of our work in the form of web-based *interactive essay*. This provides a simple implementation of three context-aware programming languages and lets the reader write and run simple context-aware programs, but also explore the theory behind the implementation including the typing derivation and semantics.

CONTENTS

i CONTEXT-AWARE PROGRAMMING 1

ii COEFFECT CALCULI 3

iii TOWARDS PRACTICAL COEFFECTS 7

1 IMPLEMENTATION 9
- 1.1 From theory to implementation . . . 10
  - 1.1.1 Type checking and inference . . . 10
  - 1.1.2 Execution of context-aware programs . . . 11
  - 1.1.3 Supporting additional context-aware languages . . . 12
- 1.2 Case studies . . . 12
  - 1.2.1 Typing context-aware programs . . . 13
  - 1.2.2 Comonadically-inspired translation . . . 13
- 1.3 Interactive essay . . . 15
  - 1.3.1 Explorable language implementation . . . 16
  - 1.3.2 Implementation overview . . . 18
- 1.4 Related work . . . 19
- 1.5 Summary . . . 20

2 FURTHER WORK 21
- 2.1 The unified coeffect calculus . . . 21
  - 2.1.1 Shapes and containers . . . 22
  - 2.1.2 Structure of coeffects . . . 22
  - 2.1.3 Unified coeffect type system . . . 25
  - 2.1.4 Structural coeffects . . . 27
  - 2.1.5 Flat coeffects . . . 28
  - 2.1.6 Semantics of unified calculus . . . 30
- 2.2 Towards practical coeffects . . . 31
  - 2.2.1 Embedding contextual computations . . . 32
  - 2.2.2 Coeffect annotations as types . . . 32
  - 2.2.3 Alternative formulation using coeffect lattice . . . 33
- 2.3 Coeffect meta-language . . . 35
  - 2.3.1 Coeffects and contextual modal type theory . . . 35
  - 2.3.2 Coeffect meta-language . . . 36
  - 2.3.3 Embedding flat coeffect calculus . . . 37
- 2.4 Further work . . . 38
- 2.5 Summary . . . 39

3 CONCLUSIONS 41
- 3.1 Overview of contributions . . . 41
- 3.2 Summary . . . 43

BIBLIOGRAPHY 45

Part I

CONTEXT-AWARE PROGRAMMING

The computing ecosystem is becoming increasingly heterogeneous and rich. Modern programs need to run on a variety of devices that are all different, but provide unique rich capabilities. For example, application running on a phone can access the GPS sensor, while application running in the cloud can access GPU computing resources. Both diversity and richness can only be expected to increase with trends such as the internet of things.

In this thesis, we argue that the creating programming languages that allow the programmer to better work with the environment or *context* in which applications execute is the next big challenge for programming language designers.

We start with a detailed discussion of the motivation for the thesis and an overview of our methodology (Chapter **??**). Next, we discuss previous programming language research that leads to the work presented in this thesis (Chapter **??**) and we examine a number of practical context-aware systems in detail (Chapter **??**), identifying two kinds of context that we later capture by *flat* and *structural coeffects*.

Part II

COEFFECT CALCULI

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

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

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

CHAPTER 4		
	COEFFECT CALCULUS	LANGUAGE-SPECIFIC
SYNTAX	Coeffect λ-calculus (Section **??**)	Extensions such as `?param` and `prev` (Section **??**)
TYPE SYSTEM	Abstract coeffect algebra (Section **??**)	Concrete instances of the coeffect algebra (Section **??**)
	Coeffect type system parameterized by the coeffect algebra (Section **??**)	Typing for language-specific extensions (Section **??**)
		Procedure for determining a unique typing derivation (Section **??**)
PROPERTIES	Syntactic properties of coeffect λ-calculus (Section **??**)	Uniqueness of the above (Section **??**)

CHAPTER 5		
	COEFFECT CALCULUS	LANGUAGE-SPECIFIC
CATEGORICAL	Indexed comonads (Sectiuon **??**)	Examples including indexed product, list and maybe comonads (Section **??**)
	Categorical semantics of coeffect λ-calculus (Section **??**)	
TRANSLATIONAL	Functional target language (Section **??**)	
	Translation from coeffect λ-calculus to target language (Section **??**)	Translation for language-specific extensions (`prev`, `?p`) (Sections **??** and **??**)
OPERATIONAL	Abstract comonadically-inspired primitives (Section **??**)	Concrete reduction rules for comonadically-inspired primitives (Sections **??** and **??**)
		Reduction rules for language-specific extensions (`prev`, `?p`) (Sections **??** and **??**)
	Sketch of generalized syntactic soundness (Section **??**)	Syntactic soundness (Sections **??** and **??**)

Part III

TOWARDS PRACTICAL COEFFECTS

In the first part of the thesis, we argued for the importance of *context* in programming languages. As programs execute in increasingly diverse and rich environments, languages need to understand and check how programs use such context. In the second part, we developed theoretical foundations (type system and semantics) for context-aware programming languages. What remains to be done if context-aware programming languages are to become "the next big thing"?

In this part, we explore practical aspects of implementing context-aware programming languages based on coeffects and related future work. We discuss a prototype implementation (Chapter 1), which links together all parts of the theory discussed in the previous part. Building a production-ready programming language is outside the scope of the thesis, so we instead focus on conveying the concept of coeffects to broader audience and make the implementation available as a web-based interactive essay (Section 1.3) at: http://tomasp.net/coeffects.

In further work (Chapter 2), we outline unification of flat and structural coeffects (Section 2.1), which may be more suited for embedding in practical languages and we discuss alternative approaches for using coeffects in programming languages (2.3).

1 IMPLEMENTATION

In the previous three chapters, we presented two coeffect calculi that capture two kinds of contextual properties. The calculi are parameterized and can be instantiate to capture concrete notions of context. They consist of type systems (parameterized by coeffect algebra) and semantics (parameterized by small number of comonadically-inspired primitives). The theory can be seen as a framework that simplifies the implementation of safe context-aware programming languages. To support this claim, this chapter presents a prototype implementation of the coeffect framework and uses it to build three concrete languages – language with implicit parameters and both flat and structural versions of a dataflow language.

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

The main goal of the implementation is to show that the theory is practically useful and to present it in a more practical way. However, we do not intend to build a complete real-world programming language. For this reason, the implementation is available primarilly as an interactive essay[1] online at http://tomasp.net/coeffects.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We discuss how the implementation follows the theory (Section 1.1) presented earlier. This applies to the implementation of the *type checker* and the implementation of the *translation* to a simple target language that is then iterpereted. We also discuss how the common framework makes it easy to implement additional context-aware languages (Section 1.1.3).
- We consider a number of case studies (Section 1.2) that illustrate interesting aspects of the theories discussed earlier. This includes the typing of lambda abstraction and the difference between flat and structural systems (Section 1.2.1) and the comonadically-inspired translation (Section 1.2.2).
- The implementation is available not just as downloadable code, but also in the format of interactive essay (Section 1.3), which aims to make coeffects accessible to a broader audience. We discuss the most interesting aspects of the web-based presentation and briefly discuss some of the interesting implementation detail (Section 1.3.2).

1 The *interactive essay* format is based on Bret Victor's work on *explorable explanations* [117] and is further explained in Section 1.3.1

1.1 FROM THEORY TO IMPLEMENTATION

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

1.1.1 *Type checking and inference*

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

The type inference works in the standard way [91, 28] by generating type constraints and solving them. Solving of type constraints is done in the standard way, but we additionally collect and solve *coeffect constraints*. In this section, we focus only on coeffect constraints (as generation and solving of type constraints is standard). The following (*abs*) rule demonstrates the notation used in this section:

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

The judgement $\Gamma \mathbin{@} r \vdash e : \tau \mid C$ denotes that an expression e in a context $\Gamma \mathbin{@} r$ has a type τ and produces coeffect constraints C. In the (*abs*) rule, we annotate the function body, function type and declaration site with new coeffect variables r, t and s, respectively and we generate a coeffect constraint $t = r \wedge s$ that captures the (*abs*) rule from flat coeffect calculus (Figure **??**).

In order to obtain unique type derivation for each term (using the algorithms discussed in Section **??**), we generate additional coeffect constraints for lambda abstraction of each flat coeffect language, as this is where flat coeffects permit multiple possible typings.

Example 1 (Flat implicit parameters.)**.** *As discussed in Section* **??***, when choosing unique typing derivation for implicit parameters, we keep track of the implicit parameters available in the lexical scope (written as* Δ*). In the lambda abstraction rule, the implicit parameters required by the body (tracked by* r*) are split so that all parameters available in lexical scope are captured and only the remaining parameters (*$r \setminus \Delta$*) are required from the caller of the function.*

From the presentation in Section **??***, it might appear that resolving the ambiguity related to lambda abstraction for implicit parameters requires a type system different from the type system shown earlier in Section* **??***. This is not the case. We track implicit parameters in scope* Δ*, but the rest of the* (abs) *rule from the implementation only generates an additional coeffect constraint. The adapted* (abs) *rule for implicit parameters looks as follows:*

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

Given a typing derivation for the body, we generate an additional constraint that restricts r *(declaration site demands) to those available in the current static scope* Δ *and a constraint that restricts the delayed (call site) demands* s *to* $t \backslash \Delta$*.*

Example 2 (Flat dataflow)**.** *In a context-aware language for dataflow (and in language with liveness tracking), the inherent ambiguity of the* (abs) *rule is resolved by placing the context requirements of the body on both the declaration site and the call site. In Section* **??***, this was defined by replacing the standard coeffect* (abs) *rule with a rule* (idabs) *that uses an annotation* $\mathtt{r}$ *for the body of the function, declaration site coeffect and call site coeffect.*

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

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

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

1.1.2 *Execution of context-aware programs*

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

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

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

In the interactive essay (Section 1.3), we further use the coeffects to automatically generate a user interface that requires the user to provide the required contextual capabilities (past values for individual variables, or values for implicit parameters).

Another benefit of using the common framework is that the implementation can be easily extended to support further languages with additional contextual properties.

1.1.3 *Supporting additional context-aware languages*

The prototype implementation supports two of the context-aware languages discussed in this thesis: implicit parameters and dataflow. Those are sufficient to demonstrate all important aspects of the system, but the implementation could be extended to support the remaining languages discussed in the thesis. Thanks to the fact that the implementation is based on the common coeffect framework, this requires minimal amount of work. In order to extend the prototype implementation with support for liveness, bounded reuse tracking or other context-aware language, the following four additions are required:

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

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

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

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

The list mirrors a list of steps that need to be done when supporting a new effectful computation in a language that supports monadic do-notation. The step (3) corresponds to implementing a new monad and (4) corresponds to adding monad-specific effectful operations. The step (2) applies when using indexing to track effects more precisely [80]. The only step that does not have a counterpart in effectful/monadic languages is (1).

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

1.2 CASE STUDIES

The prototype implementation illustrates a number of interesting aspects of coeffect systems. Those appear as examples in the interactive essay (discussed in Section 1.3), but we briefly review them in this section.

1.2.1 *Typing context-aware programs*

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

ABSTRACTION FOR IMPLICIT PARAMETERS. As discussed in Section 1.1.1, the implementation of the language with implicit parameters resolves the ambiguity in the lambda abstraction by generating a coeffect constraint that restricts the set of parameters required from the declaration site to those that are lexically available. Remaining parameters are required from the call site. This is illustrated by the following example:

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

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

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

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

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

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

1.2.2 *Comonadically-inspired translation*

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

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

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

Translating the code to the target language produces the code below. The reader is encouraged to view the translation in the interactive essay (Section 1.3), which displays the types and coeffect annotations of the individual values and primitives. As in the theory, the comonadically-inspired primitives are families of operations indexed by the coeffects (we omit the annotations here):

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

The `finput` value on the first line models an empty context in which the expression is evaluated and is of type $C^{\mathsf{ign}}\mathsf{unit}$. The ign annotation captures the fact that there are no implicit parameters in the context and the type unit specifies that the context carries no variables.

The context is duplicated and the second part ctx3 is not needed, because 10 is a constant. The first part ctx2 is passed to letimpl, which assigns an implicit parameter value in the newly returned context ctx1 of type $C^{\{?\mathsf{param}\}}\mathsf{unit}$ (the hidden dictionary now contains a value for the implicit parameter ?param, but the context does not contain value bindings for any ordinary variables (the unit type can be seen as an empty tuple).

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

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

```
fun x → fun y → prev x
```

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

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

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

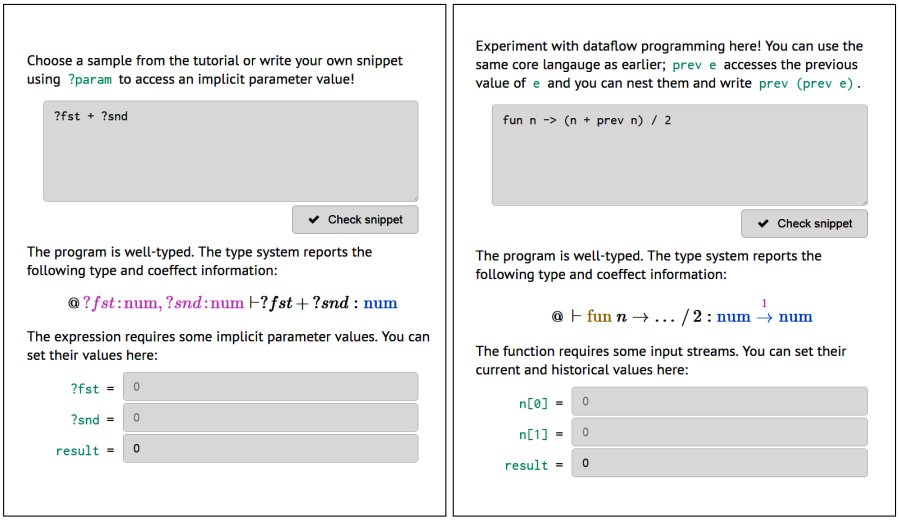

Figure 1: Interactive evaluation of implicit parameters (left) and dataflow (right)

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

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

1.3 INTERACTIVE ESSAY

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

- ACCESSIBLE. Anyone interested should be able to experiment with the implemented languages without downloading the source code and compiling it and without installing specialized software.
- EXPLORABLE. It should be possible to explore the inner workings – how is the typing derived, how is the source code translated to the target language and how is it evaluated.

To make the work *accessible*, we implement sample context-aware languages in a way that makes it possible to use them in any standard web browser with JavaScript support (Section 1.3.2) without requiring any server-side component. Following the idea that "the medium is the message" [65], we choose a medium that encourages *exploration* and make the implementation available not just as source code that can be compiled and run locally, but also in the format of an interactive essay (Section 1.3.1). The live version of the essay can be found at: http://tomasp.net/coeffects.

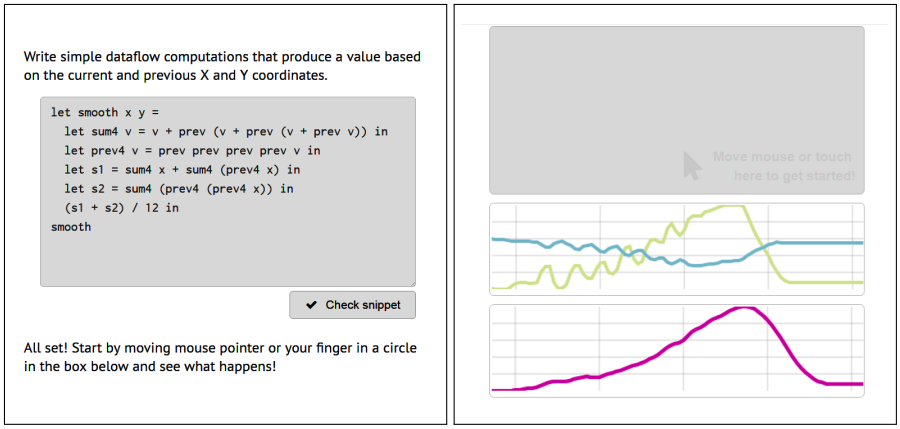

Figure 2: Function smoothing the X coordinate (left) with a sample run (right).

1.3.1 *Explorable language implementation*

The interactive essay format used of the implementation is based on Bret Victor's work on *explorable explanations* [117]. Bret Victor describes what the interactivity enables as follows:

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

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

The idea of active reading is older and has been encouraged in the context of art Josef Albers' classic 1963 work on color [5] (which was turned into an interactive essay 60 years later [90]). More recently, similar formats have been used to explain topics in areas such as signal processing [96] (explaining Fourier transformations) and sociology [47] (visualizing and explaining game theoretical model of segregation in the society [97]). To our best knowledge, the format has not previously been used in the area of programming language theory and so we briefly outline some of the interesting features that our essay provides in order to encourage active reading.

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

The panels for implicit parameters and for dataflow computations are shown in Figure 1. The sample code on the left adds two implicit parameters and the generated UI lets the user enter implicit parameter values as required by the context in the typing judgement. The sample on the right

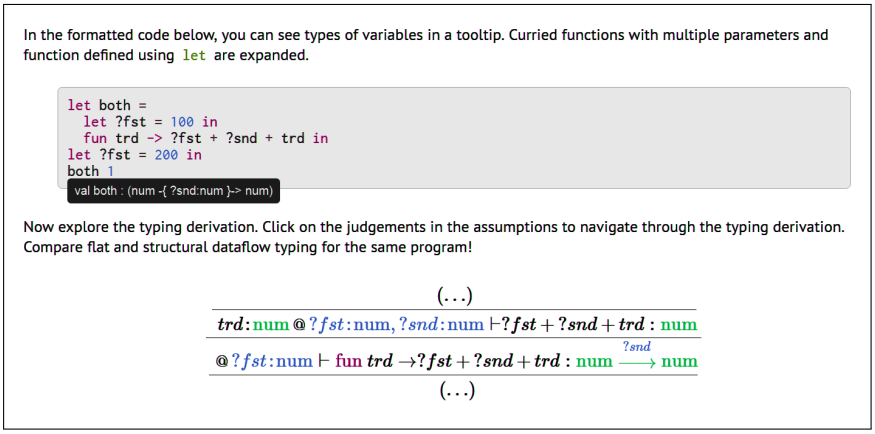

Figure 3: Source code with type information and explorable typing derivation.

calculates the average of the current and past value in a dataflow; the UI lets the user enter past values required by the type of the function.

The essay guides the readers through a number of programs (shown in Section 1.2.1) and encourages them to run and try modifying them. For implicit parameters, this includes the case where an implicit parameter is available at both declaration site and call site (showing that the declaration site value is captured). For dataflow, the examples include the comparison between the inferred type when using flat and structural type systems.

REACTIVE DATAFLOW. The interactive program execution lets the reader run sample programs, but not in a particularly realistic context. To show a more real-world scenario, the essay includes a widget shown in Figure 2. This lets the user write a function taking a stream of X and Y coordinates and calculate value based on the current and past values of the mouse pointer. The X and Y values, together with the result are plotted using a live chart.

In the example run shown above, the sample program calculates the average of the last 12 values of the X coordinate (green line in Figure 2). The example also illustrates one practical use of the coeffect type system – when running, the widget keeps the coordinates in a pre-allocated fixed-size array, because the coeffect type system guarantees that at most 12 past values will be accessed.

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

The presented interactive essay provides two features to help the reader understand and explore typing derivations. First, when reading code samples in the text, tooltips show the typing of individual identifiers, most interestingly functions (Figure 3, above).

Second, a later part of the essay provides a type checker that lets the user enter a source code in a context aware programming language and produces an explorable typing derivation for the program. The output (shown in Figure 3, below) displays a typing judgement with assumptions and conclu-

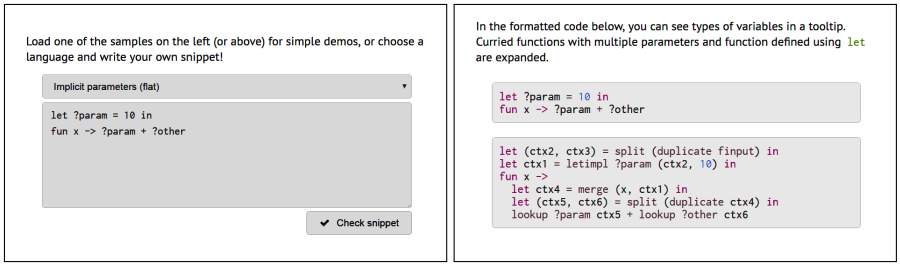

Figure 4: Source code with comonadically-inspired translation

sions and lets the reader navigate through the typing derivation by clicking on the assumptions or conclusions. This way, the reader can see how is the final typing derivation obtained, exploring interesting aspects, such as the abstraction rule (shown in Figure 3).

COMONADICALLY-INSPIRED TRANSLATION. The last interactive element of the essay lets the reader explore the translation of source context aware language to a target simple functional language (Section 1.1.2). Compared with the monadic do-notation [51], the comonadic translation is more complex for two reasons.

First, it merges all variables into a single (comonadic) value representing the context. Second, there is a flat and structural variation of the system. For these two reasons, understanding the translation based just on the rules is harder than for monads. The essay lets readers see translation for carefuly chosen case studies that illustrate the key aspects of the translation (some discussed in Section 1.2.2), but also for their own programs encouraging *active reading*. Sample input and output are shown in Figure 4.

1.3.2 *Implementation overview*

The core part of our implementation mostly follows standard techniques for implementing type checkers and interpreters for statically-typed functional programming languages. Two interesting aspects that are worth discussing is the JavaScript targetting (for running the language implementation in a web browser) and integration with client-side (JavaScript) libraries for building the user interface. The full source code can be obtained from https://github.com/coeffects/coeffects-playground. It is structured as follows:

- Parsing is implemented using a simple parser combinator library; ast.fs defines the abstract syntax tree for the languages; parsec.fs implements the parser combinators and lexer.fs with parser.fs parse the source code by first tokenizing the input and then parsing the stream of tokens.
- The type checking is implemented by typechecker.fs, which annotates an untyped AST with types and generates set of type and coeffect constraints. The constraints are later solved (using domain-specific algorithms for each of the languages) in solver.fs.
- Type-checked programs in context-aware languages are translated to simple target functional subset of the language in translation.fs; evaluation.fs then interprets programs in the target language. The interpretation does not handle the source language and so programs containing context-aware contstructs cannot be interpreted directly.

- The user interface of the interactive essay (Section 1.3.1) is implemented partly in F# and partly in JavaScript. The two most important components include a pretty-printer pretty.fs which formats source code (with type tooltips) and typing derivations and gui.fs that implements user interaction (e.g. navigation in explorable typing derivations).

As discussed in Section 1.1.3, the implementation can be easily extended to support additional context-aware programming languages. This is due to the fact that it is based on the unified theory of coeffects. In practice, adding support for liveness tracking would require adding a domain-specific constraint solver in solver.fs and extending the interpreter in evaluation.fs with a new kind of comonadically-inspired data type (indexed maybe comonad) with its associated operations.

1.4 RELATED WORK

To our best knowledge, combining Bret Victor's idea of explorable explanations with programming language theory (as discussed in Section 1.3) is a novel contribution of our work. On the technical side, we build on a number of standard methods.

PARSING AND TYPE CHECKING. The implementation of the parser, type checker and interpreter follows standard techniques for implementing functional programming languages. In order to be able to compile the implementation to JavaScript (see below), we built a small parser combinator library [50] rather than using one of the already available libraries [113].

TARGETTING JAVASCRIPT. In order to make the implementation accessible to a broad audience, it can be executed in a web browser. This is achieved by automatically translating the implementation from F# to JavaScript. We use an F# library called FunScript [15] (which is a more recent incarnation of the idea developed by the author [81]). We choose F#, but similar tools exist for other functional languages such as OCaml [121]. It is worth noting that FunScript is implemented as a *library* rather than as a compiler extension. This is done using the meta-programming capabilities of F# [103].

CLIENT-SIDE LIBRARY INTEGRATION. An interesting aspect of the interactive essay user interface is the integration with third-party JavaScript libraries. We use a number of libraries including JQuery (for web browser DOM manipulation) and MathJax [18] (for rendering of typing derivation). In order to call those from F# code, we use a number of mapping methods described in [87]. For example, the following F# declaration is used to invoke the Queue function in MathJax:

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

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

1.5 SUMMARY

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

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

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

2 FURTHER WORK

The main goal of this thesis is to provide a *unified* calculus for tracking context dependence. We have not achieved this goal yet. In Chapter **??**, we identified two kinds of contextual properties that we further covered separately. Flat coeffects in Chapter **??** track whole-context properties, while structural coeffects, covered in Chapter **??**, track per-variable properties. In this chapter, we unify the two notions.

Although the results presented in this thesis are mainly of a theoretical nature, we indeed believe that coeffects should be integrated in main-stream programming languages. This is addressed in the second part of the chapter, where we outline possible approach for a practical implementation of coeffect systems. Finally, we conclude the chapter with a brief discussion of an alternative approach to coeffect system, based on the meta-language style.

CHAPTER STRUCTURE AND CONTRIBUTIONS

- We start by unifying the two kinds of coeffects discussed so far. In Section 2.1, we introduce the *unified coeffect calculus*, which generalizes flat and structural systems and can be instantiated to track both flat and structural properties.
- We outline one possible approach for practical implementations of coeffects (Section 2.2). The technique is based on embedding comonadic computations via a lightweight syntactic extension and using rich type systems for capturing the structure of coeffect algebra.
- Finally, we discuss an alternative approach to defining coeffect systems based on the meta-language style, which highlights the relationship between our work and related work arising from modal logics such as CMTT [71] (Section 2.3).

The unified coeffect calculus presented in the first part of the chapter is a novel result that completes the theory developed in this thesis. The next two sections present important future and related work, respectively.

2.1 THE UNIFIED COEFFECT CALCULUS

The type systems of the flat coeffect calculus (Figure **??**) and the structural coeffect calculus (Figure **??**) differ in a number of ways. Understanding the differences is the key to reconciling the two systems:

- The structural coeffect calculus contains explicit rules for context manipulation (weakening, contraction, exchange). In the flat coeffect calculus, these rules are not defined explicitly, but are admissible.
- In the structural coeffect calculus, the variable context is treated as a vector and is annotated with a vector of (scalar) coeffects. In the flat coeffect calculus, the variable context is a finite partial function (from names to types) and is annotated with a single (scalar) coeffect.

- In the flat coeffect calculus, we distinguish between *splitting* context requirements and *merging* of context requirements ($\oplus$ and $\wedge$, respectively). In the structural coeffect calculus, the operations (which model appending and splitting vectors) are invertible and so the structural coeffect algebra uses the same tensor product $\times$.

In the unified calculus presented in this section, we address the three differences as follows. We include explicit rules for context manipulation in the calculus; in systems that arise from flat calculi, the structural rules do not change coeffects and can thus be applied freely. We generalize the structure of coeffect annotations using the notion of a "container" which can be specialized to obtain a single annotation or a vector of annotations. This could potentially be used to capture other structures such as trees in bunched typing [74]. Finally, we distinguish between splitting and merging of context requirements (using the notation $\curlywedge$ and $\curlyvee$, respectively). For structural coeffect calculi, the two operators coincide, but for flat coeffect calculi, they provide the needed flexibility.

2.1.1 *Shapes and containers*

Our generalization of coeffect structure using a *coeffect container* is based on containers of Abbott et al. [3]. Interestingly, containers have later been linked to comonads by Ahman et al. [4]. Intuitively a container describes data types such as lists, trees or streams. A container is formed by shapes (e. g. lengths of lists)[1]. For every shape, we can obtain a set of positions in the container (e. g. offsets in a list of a specified length). More formally:

Definition 1. *A* container *consists of a pair* (S, P)*, usually written as* $\mathsf{S} \triangleleft \mathsf{P}$*. Here,* S *is a set of shapes and* P *is a shape-indexed family* $\mathsf{P} : \mathsf{S} \to \mathsf{Set}$ *of positions.*

Well-known examples of containers include lists, non-empty lists, (unbounded) streams, trees and the singleton data type (which contains exactly one element). Containers relevant to our work are lists and singleton data types:

- The container representing lists is given by $\mathsf{S} \triangleleft \mathsf{P}$ where shapes are integers $\mathsf{S} = \mathsf{Nat}$ (lengths of a list). The set of positions for a given length n is the set of indices $\mathsf{P}(n) = \{1 \ldots n\}$.
- The container representing the singleton data type is given by $\mathsf{S} \triangleleft \mathsf{P}$ where shapes are given by a singleton set $\mathsf{S} = \{*\}$ and the set of positions for the shape $*$ contains exactly one position. To follow the intuition based on offset or index, we write the single position as 0, that is $\mathsf{P}(*) = \{0\}$.

In the unified coeffect calculus, the structure of coeffect annotations is defined by a container with additional operations (discussed later) that links it with the free-variable context Γ.

2.1.2 *Structure of coeffects*

In the structural coeffect calculus, the coeffect annotation was formed by a vector of coeffect scalars. The annotations in the unified coeffect calculus are similar, but a *vector* is replaced with a more general *container*. The primitive

1 Shapes can also be intuitively thought of as sizes, but this is not fully precise as some containers may have multiple shapes of the same length, e. g. differently balanced trees.

coeffect annotations in the unified calculus are formed of *coeffect scalars*. The coeffect scalars are equipped with the same structure as in structural coeffect calculus (Definition **??**). In this section, we refer to it as *unified coeffect scalar* (and we repeat the definition below). Then we define *unified coeffect containers* which determines how coeffect scalar values are attached to the free-variable context. Finally, we define the *unified coeffect algebra* which consists of shape-indexed coeffect scalar values.

As in the structural coeffect calculus, the contexts in the unified calculus are annotated with shape-indexed coeffects, written as $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$; functions take just a single input parameter and so are annotated with scalar coeffect values $\sigma \xrightarrow{r} \tau$.

COEFFECT SCALARS. The following definition of the coeffect scalar structure repeats the Definition **??** from the previous chapter.

Definition 2. *A unified 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* $r, s, t \in \mathcal{C}$*:*

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

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

$$r \leqslant r' \;\Rightarrow\; \forall s.\ (r \circledast s) \leqslant (r' \circledast s)$$
$$\mathsf{ign} \leqslant (\mathsf{ign} \circledast r)$$

As previously, the monoid $(\mathcal{C}, \circledast, \mathsf{use})$ models sequential composition; the laws guarantee an underlying category structure; use and ign represent an accessed and unused variable, respectively.

The $\oplus$ operation models combining of context requirements arising from multiple parts of a program. The meaning depends on the coeffect container. The operation can either combine requirements of individual variables (structural coeffects) or requirements attached to the whole context of multiple sub-expressions (flat coeffects).

COEFFECT CONTAINERS. The coeffect container is a container that determines how are scalar coeffects attached to free-variable contexts. In addition to a container $S \triangleleft P$ formed by shapes and shape-indexed positions, the coeffect container provides a mapping that returns the shape corresponding to a free-variable context.

The mapping between the shape of the variable context and the shape of the coeffect annotation is not necessarily bijective. For example, coeffect annotations in flat systems have just a single shape $S = \{*\}$.

In the coeffect judgment $\Gamma \mathbin{@} \mathbf{r} \vdash e : \tau$, the coeffect annotation $\mathbf{r}$ is drawn from the set of coeffect scalars $\mathcal{C}$ indexed by the shape corresponding to Γ. We write $s = \mathit{len}(\Gamma)$ for the shape corresponding to Γ. The operation $P(s)$ returns a *set* of positions and so we can write $\mathbf{r} \in P(s) \to \mathcal{C}$ as a mapping from positions (defined by the shape) to scalar coeffects. We usually write this as

the exponent $\mathbf{r} \in \mathcal{C}^{\mathsf{P}(s)}$. The coeffect container is also equipped with an operation that appends shapes (when we concatenate variable contexts) and two special shapes in S representing the empty context and the singleton context.

Definition 3. *A coeffect container structure $(\mathsf{S} \triangleleft \mathsf{P}, \diamond, \hat{0}, \hat{1}, \mathit{len}(-))$ comprises a container $\mathsf{S} \triangleleft \mathsf{P}$ with a binary operation $\diamond$ on S for appending shapes, a mapping from free-variable contexts to shapes $\mathit{len}(\Gamma) \in \mathsf{S}$, and elements $\hat{0}, \hat{1} \in \mathsf{S}$ such that $(\mathsf{S}, \diamond, \hat{0})$ is a monoid.*

The elements $\hat{0}$ and $\hat{1}$ represent the shapes of the empty and the singleton free-variable context, respectively. The $\diamond$ operation corresponds to concatenation of free-variable contexts. Given Γ_1 and Γ_2 such that $s_1 = \mathit{len}(\Gamma_1), s_2 = \mathit{len}(\Gamma_2)$, we require that $s_1 \diamond s_2 = \mathit{len}(\Gamma_1, \Gamma_2)$.

As discussed earlier, we use two kinds of coeffect containers that describe the structure of vectors (for structural coeffects) and the shape of trivial singleton container (for flat coeffects):

Example 3. *Structural coeffect container is defined as $(\mathsf{S} \triangleleft \mathsf{P}, |\text{–}|, +, 0, 1)$ where $\mathsf{S} = \mathbb{N}$ and $\mathsf{P}(n) = \{1 \ldots n\}$. The shape mapping $|\Gamma|$ returns the number of variables in Γ. Empty and singleton contexts are annotated with 0 and 1, respectively, and shapes of combined contexts are added so that $|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|$.*

Therefore, a coeffect annotation is a vector *$\boldsymbol{r} \in \mathcal{C}^{\mathsf{P}(n)}$ and assigns a coeffect scalar $\boldsymbol{r}(i) \in \mathcal{C}$ for each position (corresponding to a variable x_i in the context).*

Example 4. *Flat coeffect container is defined as $(\mathsf{S} \triangleleft \mathsf{P}, |-|, \diamond, \star, \star)$. The container is defined as a singleton data type $\mathsf{S} = \{\star\}$ and $\mathsf{P}(\star) = \{0\}$ with a constant function $|\Gamma| = \star$ and a trivial operation $\star \diamond \star = \star$.*

That is, there is a single shape $\star$ with a single position and all free-variable contexts have the same singleton shape. Therefore, a coeffect annotation is drawn from $\mathcal{C}^{\{\star\}}$ which is isomorphic to $\mathcal{C}$ and so a coeffect scalar $r \in \mathcal{C}$ is associated with every free-variable context.

Example 5. *Similarly to the previous example, we can also define a coeffect container with* no *positions, i.e. $(\mathsf{S} \triangleleft \mathsf{P}, |-|, \diamond, \star, \star)$ where $\mathsf{S} = \{\star\}$, $\mathsf{P}(\star) = \emptyset$, $|\Gamma| = \star$ and $\star \diamond \star = \star$. This reduces our system to the simply-typed λ-calculus with no context annotations, because $\mathsf{P}(\star) = \emptyset$ and so coeffect annotations would be from the empty set $\mathcal{C}^{\emptyset}$.*

UNIFIED COEFFECT ALGEBRA. The unified coeffect calculus annotates typing judgments with shape-indexed coeffect annotations. The *unified coeffect algebra* combines a coeffect scalar and a coeffect container to define shape-indexed coeffects and operations for manipulating these.

The definition here reconciles the third point discussed in Section 2.1 – the fact that flat coeffects use separate operations for splitting and merging ($\oplus$ and $\wedge$) while structural coeffects use the tensor $\times$. In the unified calculus, we use two operators that can, however, coincide.

Definition 4. *Given a unified coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ and a coeffect container $(\mathsf{S} \triangleleft \mathsf{P}, \mathit{len}(-), \diamond, \hat{0}, \hat{1})$ a* unified coeffect algebra *extends the two structures with $(\curlyvee, \curlywedge, \bot)$ where $\bot \in \mathcal{C}^{\mathsf{P}(\hat{0})}$ is a coeffect annotation for the empty context and $\curlyvee, \curlywedge$ are families of operations that combine coeffect annotations indexed by shapes. That is $\forall n, m \in \mathsf{S}$:*

$$\curlywedge_{m,n}, \curlyvee_{m,n} \;:\; \mathcal{C}^{\mathsf{P}\,m} \times \mathcal{C}^{\mathsf{P}\,n} \to \mathcal{C}^{\mathsf{P}(m \diamond n)}$$

A coeffect algebra induces the following three additional operations:

$$\begin{aligned} \langle - \rangle &: \mathcal{C} \to \mathcal{C}^{P(\hat{1})} \\ \langle x \rangle &= \lambda\_.x \end{aligned}$$

$$\begin{aligned} \circledast_m &: \mathcal{C} \times \mathcal{C}^{P(m)} \to \mathcal{C}^{P(m)} \\ r \circledast \mathbf{s} &= \lambda s.r \circledast \mathbf{s}(s) \end{aligned}$$

$$\begin{aligned} \mathit{len}(-) &: \mathcal{C}^{P(m)} \to m \\ \mathit{len}(\mathbf{r}) &= m \end{aligned}$$

The $\langle - \rangle$ operation lifts a scalar coeffect to a shape-indexed coeffect that is indexed by the shape of a singleton context. The $\circledast_m$ operation is a left multiplication of a vector by a scalar. As we always use bold face for vectors and ordinary face for scalars (as well as a distinct colour), using the same symbol is not ambiguous. We also tend to omit the subscript m and write $\circledast$.

Finally, we define $\mathit{len}(-)$ as an operation that returns the shape of a given shape-indexed coeffect. The only purpose is to simplify notation, as we tend to avoid subscripts, but often need to specify that shapes of variable context and coeffect match, e. g. $\mathit{len}(\mathbf{r}) = \mathit{len}(\Gamma)$.

SPLITTING AND MERGING COEFFECTS. The operators $\between$ and $\curlywedge$ combine shape-indexed coeffects associated with two contexts. For example, assume we have Γ_1 and Γ_2 with coeffects $\mathbf{r} \in \mathcal{C}^{P(m)}$ and $\mathbf{s} \in \mathcal{C}^{P(n)}$. In the structural system, the context shapes m, n denote the number of variables in the two contexts. The combined context Γ_1, Γ_2 has a shape $m \diamond n$ and the combined coeffects $\mathbf{r} \curlywedge \mathbf{s}, \mathbf{r} \between \mathbf{s} \in \mathcal{C}^{P(m \diamond n)}$ are indexed by that shape.

For structural coeffect systems such as bounded reuse, both $\between$ and $\curlywedge$ are just the tensor product $\times$ of vectors. For flat coeffect systems, the operations can be defined independently, letting $\curlywedge = \wedge$ and $\between = \oplus$.

The difference between $\curlywedge$ and $\between$ is clarified by the semantics (Section 2.1.6), where $\mathbf{r} \curlywedge \mathbf{s}$ is an annotation of the *codomain* of a morphism that merges the capabilities provided by two contexts (in the syntactic reading, splits the context requirements), while $\mathbf{r} \between \mathbf{s}$ is an annotation of the *domain* of a morphism that splits the capabilities of a single context into two parts (in the syntactic reading, merges their context requirements). Syntactically, this means that we always use $\curlywedge$ in the rule *assumptions* and $\between$ in *conclusions*.

2.1.3 *Unified coeffect type system*

The unified coeffect system in Figure 5 resembles the structural type system shown in Figure **??**. Rather than explaining the rules one-by-one, we focus on how the unified system differs from the structural system.

The type system for the unified coeffect calculus is parameterized by a coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ together with a coeffect algebra $(\curlywedge, \between, \bot)$ and the derived constructs $\langle - \rangle$, $\mathit{len}(-)$ and $\circledast$. As in the structural system, free-variable contexts Γ are treated as vectors modulo duplicate use of variables – the associativity is built-in. The order of variables matters, but can be changed using the exchange rule. The context annotations $\mathbf{r}, \mathbf{s}, \mathbf{t}$ are shape-indexed coeffects (rather than simple vectors as before). As before, functions are annotated with coeffects scalars.

a.) Syntax-driven typing rules:

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

$$(const)\ \frac{c{:}\tau \in \Delta}{() \mathbin{@} \bot \vdash c : \tau}$$

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

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

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

b.) Structural rules for context manipulation:

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

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

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

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

Figure 5: Type system for the unified coeffect calculus

SYNTAX-DRIVEN RULES. The (*var*) rule is syntactically the same as in the structural system, but it should be read differently. The $\langle - \rangle$ operation does not create a *vector*, but a *shape-indexed coeffect* that returns `use` for all positions in the singleton shape $\hat{1}$. The (*const*) rule annotates empty context with a special annotation of the shape $\hat{0}$.

In a structural system, the two annotations correspond to a singleton and empty vector, respectively. However, for a singleton shape with one position, the annotations are equivalent to annotating variables with a scalar `use` and constants with a scalar `ign`.

In the (*app*), (*abs*) and (*let*) rules, the only change from the structural system is that the vector concatenation $\times$ is now replaced with context splitting/merging of the unified coeffect algebra. As already mentioned, we use splitting of context requirements $\barwedge$ in rule *assumptions* and merging of context requirements $\between$ in rule *conclusions*.

Note that we use the terms *merging* and *splitting* in the syntactic (top-down) sense. As discussed in Chapter **??**, Section **??**, we can also read the rules in semantic (bottom-up) sense, in which case assumptions merge available contextual information and conclusions split available contextual information.

STRUCTURAL RULES. The merging/splitting operations in the structural rules are changed in the same way as in the rules above. It is also worth noting that structural coeffect system used vectors of multiple elements. For example, $\langle s, t\rangle$ and $\langle t, s\rangle$ in the (*exch*) rule denoted two-element vectors. In the unified system, this is replaced with merging/splitting of two lifted scalars: $\langle s\rangle \between \langle t\rangle$ and $\langle t\rangle \bowtie \langle s\rangle$.

In structural systems, the two notations mean the same thing – we are simply concatenating or splitting two singleton vectors. However, this generalization allows us to capture flat coeffect systems as well. The lifting operation in flat systems simply returns the lifted scalar and operators $\between$ and $\bowtie$ correspond to operations on coeffect scalars. As discussed in Section 2.1.5, thanks to the properties of coeffect scalars, contraction, weakening and exchange that do not affect the coeffect annotation are admissible for all flat systems embedded in the unified coeffect calculus.

2.1.4 *Structural coeffects*

The unified coeffect system uses a general notion of context shape, but it has been designed with structural and flat systems in mind. In this and the next section, we show how it captures the two coeffect systems from Chapter **??** and Chapter **??**.

The unified calculus closely resembles the structural system and so using it to model structural systems is easy – given a coeffect scalar, we use the coeffect container that describes a *vector* of annotations (Example 3) and define a coeffect algebra formed by a vector (free monoid) of scalars.

Definition 5. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *a* structural coeffect system *is defined by:*

- *A coeffect container* $(S \triangleleft P, |\text{–}|, +, 0, 1)$ *where* $S = \mathbb{N}$ *and* $P(n) = \{1 \ldots n\}$ *and* $|x_1 : \tau_1, \ldots, x_n : \tau_n| = n$.
- *A coeffect algebra* $(\times, \times, \epsilon)$ *where* $\times$ *and* ϵ *are shape-indexed versions of the binary operation and the unit of a free monoid over* $\mathcal{C}$. *That is* $\epsilon : \mathcal{C}^{P(0)}$ *is an empty vector and* $\times : \mathcal{C}^{P(n)} \times \mathcal{C}^{P(m)} \to \mathcal{C}^{P(n+m)}$ *appends vectors.*

The definition is valid since the shape operations form a monoid $(\mathbb{N}, +, 0)$ and $\mathit{len}(\text{–})$ (calculating the length of a list) is a monoid homomorphism from the free monoid to the monoid of shapes.

PROPERTIES. An important property of the unified system is that, when used in a structural way as discussed above, it gives calculi with the same properties as the structural system described in Chapter **??**. This can be easily seen by comparing the Figure 5 with the Figure **??** and using the free monoid interpretation of the unified coeffect algebra.

Remark 1. *The system described in Definition 5 is equivalent to the structural coeffect system described in Figure* **??**. *That is, a typing derivation using a structural coeffect embedded in the unified system is valid if and only if the corresponding derivation is valid in the structural system.*

Using the above definition, our unified coeffect system can capture per-variable coeffect properties discussed in Section **??**. This includes the system for bounded reuse (which is only used in the structural form) and precise tracking of per-variable dataflow and liveness.

2.1.5 *Flat coeffects*

The same unified coeffect system can be used to capture systems that track whole-context (flat) coeffects such as implicit parameters. This is achieved using a singleton-shaped container for coeffect annotations. The resulting system has explicit structural rules (and is syntactically different from the standard flat coeffect system), but we show that they are equivalent.

Flat coeffect systems are characterised by a singleton set of shapes (Example 4). In this setting, the context annotations $\mathcal{C}^{\mathsf{P}(\star)} = \{0\} \to \mathcal{C}$ are equivalent to coeffect scalars $\mathcal{C}$. In addition to the coeffect scalar structure, we need to define $\rtimes$ and $\ltimes$. Our examples of flat coeffects use $\oplus$ (merging of scalar coeffects) for $\rtimes$ (merging of shaped coeffect annotations). The $\ltimes$ operation (corresponding to $\wedge$ in flat coeffect calculus) needs to be provided explicitly.

Definition 6. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a binary operation* $\wedge : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *such that* $(r \wedge s) \leqslant (r \oplus s)$*, the unified coeffect algebra modelling a flat coeffect systems consists of:*

- *A flat coeffect container* $(S \triangleleft \mathsf{P}, |-|, \diamond, \star, \star)$ *as defined in Example 4.*
- *A flat coeffect algebra* $(\wedge, \oplus, \mathsf{ign})$*, i.e.* $\rtimes = \oplus$ *and* $\bot = \mathsf{ign}$ *with an additional binary operation* $\ltimes = \wedge$*.*

Intuitively, the requirement $(r \wedge s) \leqslant (r \oplus s)$, which could be also written as $(r \ltimes s) \leqslant (r \rtimes s)$, denotes that splitting context requirements and then re-combining them preserves all the requirements from the assumptions. The system may be imprecise and conclusions can overapproximate assumptions, but it cannot lose requirements. This is fundamental for showing that exchange and contraction are admissible in the unified system.

PROPERTIES. To show that the typing of flat properties in the unified system is equivalent to the typing in the flat system, we show that a valid typing judgement in the first system is a valid typing judgement in the second system and vice versa.

In one direction, we show that the unified system (capturing flat properties) permits weakening, contraction and exchange rules that do not change the coeffect annotations. This guarantees that a valid judgement in flat system is also valid in the unified system.

Lemma 2. *A unified coeffect calculus capturing flat properties admits weakening that does not change the coeffect annotation.*

Proof. The rule is admissible using the following derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma \circledast r \vdash e : \tau}{\Gamma, x{:}\tau_1 \circledast r \rtimes \langle \mathsf{ign} \rangle \vdash e : \tau}}{\Gamma, x{:}\tau_1 \circledast r \oplus \mathsf{ign} \vdash e : \tau}}{\Gamma, x{:}\tau_1 \circledast r \vdash e : \tau}$$

We write r rather than $\mathbf{r}$, because we are tracking flat properties. The first step is an application of the (*weak*) rule. Next, we use the fact that $\rtimes = \oplus$ and $\langle \mathsf{ign} \rangle = \mathsf{ign}$, which is the unit of the monoid $(\mathcal{C}, \oplus, \mathsf{ign})$. □

Lemma 3. *A unified coeffect calculus capturing flat properties admits exchange that does not change the coeffect annotation.*

Proof. We use the idempotence of $\wedge$ and $\oplus$ together with the (*exch*) rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \circledast r \vdash e : \tau}{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \circledast r \wedge r \wedge r \wedge r \vdash e : \tau}}{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \circledast r \asymp \langle r \rangle \asymp \langle r \rangle \asymp r \vdash e : \tau}}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \circledast r \asymp \langle r \rangle \asymp \langle r \rangle \asymp r \vdash e : \tau}}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \circledast r \oplus r \oplus r \oplus r \vdash e : \tau}}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \circledast r \vdash e : \tau}$$

Using idempotence, we first duplicate the annotation r to get a coeffect in the form $r \asymp \langle r \rangle \asymp \langle r \rangle \asymp r$ as required by the assumption of the (*exch*) rule. Note that $\langle - \rangle$ can be added freely as $\langle r \rangle$ is equivalent to r. After applying (*exch*), we use idempotence of $\asymp$. □

Lemma 4. *A unified coeffect calculus capturing flat properties admits contraction that does not change the coeffect annotation.*

Proof. Similarly to exchange, the proof uses idempotence of $\wedge$ and $\oplus$:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \circledast r \wedge r \wedge r \wedge r \vdash e : \tau}{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \circledast r \asymp \langle r \rangle \asymp \langle r \rangle \asymp r \vdash e : \tau}}{\Gamma_1, x{:}\tau_1, \Gamma_2 \circledast r \asymp \langle r \oplus r \rangle \asymp r \vdash e[z, y \leftarrow x] : \tau}}{\Gamma_1, x{:}\tau_1, \Gamma_2 \circledast r \oplus r \oplus r \oplus r \vdash e[z, y \leftarrow x] : \tau}$$ □

In the last two cases, we need to turn the coeffect into a form that is required by the exchange and contraction rules. Aside from idempotence, we could use the unit property and obtain e. g. $\mathsf{ign} \wedge r \wedge \mathsf{ign} \wedge \mathsf{ign}$. However, this approach does not work because $\oplus$ and $\wedge$ may have different unit elements (in fact, we do not ever require the existence of a unit for $\wedge$).

In the other direction, we need to show that any valid judgement in the unified system (tracking flat properties) is also valid in the flat system. The syntax-directed rules are the same in both systems, but we need to show that any use of (explicit) weakening, contraction and exchange can be derived in the flat system.

Lemma 5. *Weakening, as defined in a unified coeffect calculus capturing flat properties, is admissible in the flat coeffect calculus.*

Proof. Similar to the proof in Lemma 2. The property follows from the fact that $\bot = \langle \mathsf{ign} \rangle$ is the unit of $\oplus$. □

Lemma 6. *Contraction, as defined in a unified coeffect calculus capturing flat properties, is admissible in the flat coeffect calculus.*

Proof. The application of (*contr*) rule has the following form:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \circledast r \wedge s \wedge t \wedge q \vdash e : \tau}{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \circledast r \asymp \langle s \rangle \asymp \langle t \rangle \asymp q \vdash e : \tau}}{\Gamma_1, x{:}\tau_1, \Gamma_2 \circledast r \asymp \langle s \oplus t \rangle \asymp q \vdash e[z, y \leftarrow x] : \tau}}{\Gamma_1, x{:}\tau_1, \Gamma_2 \circledast r \oplus s \oplus t \oplus q \vdash e[z, y \leftarrow x] : \tau}$$

From Definition 6, we know that $(r \wedge s \wedge t \wedge q) \leqslant (r \oplus s \oplus t \oplus q)$. Thus, the judgement can be derived using the (*sub*) rule of flat coeffect calculus. □

Lemma 7. *Exchange, as defined in a unified coeffect calculus capturing flat properties, is admissible in the flat coeffect calculus.*

Proof. The application of (*exch*) rule has the following form:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \circledast r \wedge s \wedge t \wedge q \vdash e : \tau}{\Gamma_1, x{:}\tau_1, y{:}\tau_2, \Gamma_2 \circledast r \between \langle s \rangle \between \langle t \rangle \between q \vdash e : \tau}}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \circledast r \betwixt \langle t \rangle \betwixt \langle s \rangle \betwixt q \vdash e : \tau}}{\Gamma_1, y{:}\tau_2, x{:}\tau_1, \Gamma_2 \circledast r \oplus t \oplus s \oplus q \vdash e : \tau}$$

From idempotence of $\wedge$, we get that $r \wedge s \wedge t \wedge q = r \wedge t \wedge s \wedge q$. Thus the judgement can be derived using (*sub*) as in contraction. □

A consequence of the equivalence discussed above is that the unified coeffect system can capture all properties that can be captured by the flat coeffect system – including implicit parameters, rebindable resources, Haskell type classes (discussed by Orchard [76]), dataflow and variable liveness.

2.1.6 *Semantics of unified calculus*

The semantics of the unified calculus can be obtained by generalizing the semantics of the structural calculus in the same way as we generalized the type system. However, it is worth exploring the type signatures of the operations of the underlying (comonad-based) structure.

The semantics of the unified coeffect calculus follows the same pattern as the semantics of the structural calculus (Section **??**) with a number of differences. The index $\mathbf{r}$ of the object mapping $C^{\mathbf{r}}$ is an element of the structure $\mathcal{C}^{P(n)}$ for a specified container $S \triangleleft P$ and a shape n determined as $n = \mathit{len}(\Gamma)$. To match the type system discussed in Section 2.1.3, we need to modify the domains and codomains of the merge, split and dup operations as follows.

$$\begin{aligned}
\mathsf{merge}_{\mathbf{r},\mathbf{s}} &: C^{\mathbf{r}}\alpha \times C^{\mathbf{s}}\beta \to C^{\mathbf{r} \between \mathbf{s}}(\alpha \hat{\between} \beta) \\
\mathsf{split}_{\mathbf{r},\mathbf{s}} &: C^{\mathbf{r} \betwixt \mathbf{s}}(\alpha \hat{\between} \beta) \to C^{\mathbf{r}}\alpha \times C^{\mathbf{s}}\beta \\
\mathsf{dup}_{\mathbf{r},\mathbf{s}} &: C^{\langle \mathbf{r} \oplus \mathbf{s} \rangle}\alpha \to C^{\langle \mathbf{r} \rangle \between \langle \mathbf{s} \rangle}(\alpha \hat{\between} \alpha)
\end{aligned}$$

The key difference is that the vector concatenation $\times$ is replaced with $\betwixt$ in the *domain* of the operations (in split) and with $\between$ in the codomain (in merge, but also in dup which models contraction). This means that the splitting and merging the context can change the coeffect annotation:

$$\mathsf{merge}_{\mathbf{r},\mathbf{s}} \circ \mathsf{split}_{\mathbf{r},\mathbf{s}} \;:\; C^{\mathbf{r} \betwixt \mathbf{s}}(\alpha \hat{\between} \beta) \to C^{\mathbf{r} \between \mathbf{s}}(\alpha \hat{\between} \beta)$$

For flat coeffect calculi where $(r \between s) \leqslant (r \betwixt s)$, splitting and then merging a context may lose some of the available contextual capabilities (and increase context requirements). This illuminates why our proofs of Lemma 6 and Lemma 7 relied on subcoeffecting in the flat system.

RELATED AND FUTURE WORK. The semantics presented here serves mainly to inform the design of the type system. For this reason, we keep the semantics simple and close to the concrete notions used in the type system such as flat, structural and unified coeffect algebras. In particular, the indexed comonad structure consists of object mapping C^{r} that is indexed by a corresponding algebra:

- In the flat coeffect calculus, C^{r} is indexed by scalar coeffects $r \in \mathcal{C}$
- In the structural coeffect calculus, the object mapping $C^{\mathbf{r}}$ is indexed by vectors of scalars $\mathbf{r} \in \bigcup_{m \in \mathbb{N}} \mathcal{C}^{m}$
- In unified coeffect calculus with a coeffect container $S \triangleleft P$, the mapping $C^{\mathbf{r}}$ is indexed by shape-indexed scalars $\mathbf{r} \in \bigcup_{s \in S} \mathcal{C}^{P(m)}$

In this treatment, the indices are formed by objects that are modelled outside of category theory. A fully categorical semantics can be found in a joint paper with Orchard [85]. There, the index (coeffect scalars and shape-indexed scalars) are modelled as categories too. The definition is written using the following notation:

- $[\mathbb{C}, \mathbb{D}]$ is a category of functors between categories $\mathbb{C}$ and $\mathbb{D}$
- B^A is an exponential object $A \Rightarrow B$ in a Cartesian-closed category

In the categorical semantics of coeffect calculi, coeffect scalars are modelled as a category of scalars $\mathbb{I}$ such that $obj(\mathbb{I}) = \mathcal{C}$. The structure of the context (indexed comonad) over the input then becomes a functor indexed by the free-variable context shape $len(\Gamma)$ and coeffect annotations $\mathbf{r}$, i. e. $C_{\mathbf{r}}^{len(\Gamma)}$:

$$[\![\Gamma \circledast R \vdash e : \tau]\!] : C_R^{len(\Gamma)}[\![\Gamma]\!] \to [\![\tau]\!]$$

The structure C can be thought of as a dependent product of functors C^n over possible shapes $n \in S$ where:

$$C^n : \mathbb{I}^{P(n)} \to [\mathbb{C}^{P(n)}, \mathbb{C}]$$

For a fixed context shape n the functor $C^n : \mathbb{I}^{P(n)} \to [\mathbb{C}^{P(n)}, \mathbb{C}]$ maps a coeffect indexed by positions $P(n)$ to a functor from a context $\mathbb{C}^n$ to an object in $\mathbb{C}$. That is, given a coeffect annotation (matching the shape of the context), we get a functor $\in [\mathbb{C}^n, \mathbb{C}]$. From a programming perspective, this functor defines a data structure that models the additional context provided to the program. The shape of this data structure depends on the coeffect annotation $\mathbb{I}^n$. For example, in bounded reuse, the annotation defines the number of values needed for each variable and the functor will be formed by lists of length matching the required number.

In the model above, shapes and positions are still treated as ordinary sets. Indeed, in the joint paper [85], we treat shapes as just sets of sets. In this chapter, we instead define *coeffect container* as an extension of containers, which have a well-defined categorical structure [2]. A future work is adapting the semantics of the unified coeffect calculus so that it is based on the semantics of containers. However, note that context is *not* a (simple) container, because containers are used as indices (coeffect annotations) of the object that wrap the free-variable context.

2.2 TOWARDS PRACTICAL COEFFECTS

As discussed earlier, the main focus of this thesis is the development of the much needed *theory of context-aware programming languages* and so discussing the details of a practical implementation of coeffect tracking is beyond the scope of the thesis. However, this section briefly outlines one possible pathway towards this goal.

Many of the examples of contextual computation that we discussed earlier have been implemented as a single-purpose programming language feature (e. g. implicit parameters [61] or distributed computations [69, 25]). However, the main contribution of this thesis is that it captures *multiple* different notions of context-aware computations using just a *single* common abstraction. For this reason, we advocate that future practical implementations of coeffects should not be single-purpose language features, but rather reusable abstractions that can be instantiated with a concrete *coeffect algebra* specified by the user.

In order to do this, programming languages need to provide two features; one that allows embedding of context-aware computations themselves in programs akin to the "do" notation in Haskell (Section 2.2.1) and one that allows tracking of the contextual information in the type system (Section 2.2.2). To make constraint resolution in type inference easier, we also discuss how the coeffect algebra can be simplified (Section **??**)

2.2.1 *Embedding contextual computations*

The embedding of *contextual* computations into programming languages can follow the successful model of *effectful* computations. In purely functional programming languages such as Haskell, effectful computations are embedded by *implementing* their model within the language and inserting the necessary (monadic) plumbing. This is made easier by the "do" notation or monad comprehensions [51, 40], which insert the monadic operations automatically.

The recently proposed "codo" notation [79] provides similar automation for context-aware computations based on comonads. The notation follows the semantics of our flat coeffect calculus (Chapter **??**). Extending the "codo" notation to support calculi based on the structural coeffect calculus (Chapter **??**) is an interesting future work – this would require explicitly manipulating individual context variables and application of structural rules, which is not needed in flat coeffects.

In ML-like languages, effects (and many coeffects) are built-in into the language semantics, but they can still benefit from a special notation for explicitly marking effectful (coeffectful) blocks of code. In F#, this is done using *computation expressions* [86] that differ from the "do" notation in two ways. First, they support wider range of constructs, making it possible to wrap existing F# code within a block without other changes. Second, they support other abstractions including monads, applicative functors and monad transformers. It would be interesting to see if computation expressions can be extended to support programming with computations based on flat/structural indexed comonads.

More lightweight syntax for effectful computation can be obtained using techniques that automatically insert the necessary monadic plumbing (without special syntax). This has been done for effectful computations [102] and a similar approach would be worth exploring for coeffects.

2.2.2 *Coeffect annotations as types*

The other aspect of practical implementation of coeffects is that of tracking the context requirements (coeffect annotations) in the type system. To achieve this (without resorting to a single-purpose language feature) the type system needs to be able to capture various kinds of coeffect algebras. The structures required in this thesis include sets (with union or intersection), natural numbers (with addition, maximum, minimum and multiplication), two-point lattice (for liveness) and free monoids (vectors of annotations).

The work [80] on embedding effect systems in Haskell demonstrated that the recent additions to the Haskell type system provide enough power to implement structures such as sets at the type level. Using these to embed coeffect systems in Haskell is one fruitful future direction for applied coeffects.

In dependently-typed programming languages such as Agda or Idris [13, 14], the embedding of coeffects can be implemented more directly (as terms implementing sets or lattices can be lifted to the type level). However, we believe that coeffect tracking does not require full dependent types and can be made accessible in more mainstream languages. Dependent ML [130] provides an interesting example of a language with some dependent typing (e. g. computations with integers) which is still close to its non-dependently-typed predecessor ML.

Another approach for embedding computations into the type system has been pioneered by F# *type providers* [104]. Technically, type providers are compiler extensions that generate types based on external data sources or other information in order to provide easy access to data or services. A similar approach could be used for embedding *algberas* such as coeffect algebras into the type system. An *algebra provider* would then be a library that specifies the objects of the algebra, its equational laws and generalization rules for type inference. This could provide an easy-to-use way of embedding coeffect tracking in pragmatic languages such as F#. It is worth noting that the mechanism could also subsume F# units of measure [55]; these could be alternatively provided via one such *algebra provider*.

2.2.3 *Alternative formulation using coeffect lattice*

Compared to typical effect systems based on sets, coeffect systems have richer structure consisting of four operations ($\circledast, \wedge, \oplus$ and a relation $\leqslant$). This allows capturing interesting properties, but it makes working with coeffect annotations difficult – it complicates tasks such as type generalization and constraint solving in type inference. Here, we look at one possible simplification of the structure.

- As mentioned in Section **??**, when the monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ is a semilattice, the $\leqslant$ relation can often be expressed in terms of the $\oplus$ operation ($r \leqslant s \iff r \oplus s = s$). This does not give a definition equivalent to our, but it is a simplification that is consistent with all examples in this thesis.
- Another approach that works for most (but not all) systems we discussed is to require the structure $(\mathcal{C}, \oplus, \wedge)$ to form a lattice. This is consistent with all our structural coeffect systems as well as flat systems for liveness and dataflow, but not with the system for tracking of implicit parameters (or sets of rebindable resources).

The system for tracking implicit parameters does not fit the lattice-based model, because it uses the same operation $\cup$ for both $\oplus$ and $\wedge$.

This also means that the system for implicit parameters cannot be used with a system where lambda abstraction duplicates the context requirements (Section **??**).

LATTICE-BASED COEFFECTS. Putting implicit parameters aside, there are many other coeffect systems for which we could use the following definition of coeffect scalars (for simplicity, we only consider flat systems):

Definition 7. *A flat coeffect lattice* $(\mathcal{C}, \circledast, \leqslant, \mathsf{use}, \mathsf{ign})$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$ *and an ordering* $\leqslant$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *is a monoid and* $(\mathcal{C}, \leqslant)$ *is a lattice with the least element* ign.

a.) Lattice-based flat systems (selected rules):

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

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

b.) Lattice-based structural systems (selected rules):

$$(contr)\ \frac{\Gamma_1, y{:}\tau_1, z{:}\tau_1, \Gamma_2 \,@\, \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau}{\Gamma_1, x{:}\tau_1, \Gamma_2 \,@\, \mathbf{r} \times \langle u \rangle \times \mathbf{q} \vdash e[z, y \leftarrow x] : \tau}\ (s \leqslant u, t \leqslant u)$$

Figure 6: Flat and structural coeffects using lattice-based formulation

Assuming $\oplus$ *and* $\wedge$ *are the* least upper bound *and* greatest lower bound *of the lattice, we require the following distributivity axioms:*

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

The definition provides all the operations of a flat coeffect algebra, but it defines a more rigid structure. In particular, $\wedge$ and $\oplus$ are both defined in terms of $\leqslant$. This simplifies the system as the operations of the underlying indexed comonad structure can be expressed using just $\circledast$ and $\leqslant$:

$$\begin{array}{lll}
\mathsf{counit}_{\mathsf{use}} & : C^{\mathsf{use}}\alpha \to \alpha & \\
\mathsf{cobind}_{r,s} & : (C^{r}\alpha \to \beta) \to (C^{r \circledast s}\alpha \to C^{s}\beta) & \\
\mathsf{merge}_{r,s,t} & : C^{r}\alpha \times C^{s}\beta \to C^{t}(\alpha \times \beta) & (t \leqslant r, t \leqslant s) \\
\mathsf{split}_{r,s,t} & : C^{t}(\alpha \times \beta) \to C^{r}\alpha \times C^{s}\beta & (r \leqslant t, s \leqslant t) \\
\mathsf{lift}_{r,s} & : C^{r}\alpha \to C^{s}\alpha & (s \leqslant r)
\end{array}$$

Note that we still need *two distinct* operators in the system. Analysis of our examples shows that the operator used for sequential composition $\circledast$ is not necessarily related in any way to the ordering on the lattice provided by $\leqslant$.

However, the operations merge, split and lift can now all use ordering. For example, the merge operation previously returned a result with greatest lower bound $r \wedge s$. Now, we require any element that is smaller than r and s. This adds an implicit subcoeffecting to the operation – for any t such that $t \leqslant r$ and $t \leqslant s$, it is also the case that $t \leqslant (r \wedge s)$. This means that we can compose the original merge operation which returns greatest lower bound with lift to get the (less precise) operation required here.

LATTICE-BASED TYPE SYSTEM. Using lattice-based coeffect structure leads to the typing rules shown in Figure 6. In the flat coeffect system, this affects both application and abstraction – in abstraction, the context-requirements of the body should be smaller than the context available on the call site and declaration site. In application, we require more than what is required by the sub-expressions.

Structural coeffect systems only require least upper bound $\oplus$ in the (*contr*) rule. This can be rewritten similarly to the flat (*app*) rule – when contracting two variables, the context requirements of the resulting variable are greater than the requirements associated with each of the original variables.

In summary, the lattice-based coeffect system has some advantages over the design used in this thesis. In particular, it uses better-known structures and is simpler. However, we do not use it as it rules out one of the important motivating examples (implicit parameters) and, even with the simplification, solving constraints with $\circledast$ and $\leqslant$ is still an open question.

2.3 COEFFECT META-LANGUAGE

In Section **??**, we discussed two ways of using monads in programming language semantics as introduced by Moggi [68]. The first approach is to use monads in the *semantics* of an effectful language. The second approach is to extend the language with (additional) monadic constructs that can then be used for writing effectful monads explicitly.

In this thesis, we focused on the first approach. In both flat and structural coeffect calculi, the term language is that of a simply-typed λ-calculus, and we used (flat or structural) indexed comonads to give the semantics for the language and to derive type system for it.

In this section, we briefly discuss the other technique. That is, we embed indexed comonads into a λ-calculus as additional constructs. To do that, we introduce the type constructor $C^r\tau$ which represents a value τ wrapped in additional context (semantically, this corresponds to an indexed comonad) and we add language constructs that correspond to the operations of indexed comonads.

This section provides a brief sketch of the coeffect meta-language in order to highlight the relationship between coeffects and closely related work on contextual modal type theory (CMTT) [71]. Developing the system further is an interesting future research direction.

2.3.1 *Coeffects and contextual modal type theory*

As discussed in Section **??**, context-aware computations are related to modal logics – comonads have been used to model the $\Box$ modality and as a basis for meta-languages that include $\Box$ as a type constructor [12, 88, 12, 71]. Nanevski et al. [71] extend an S4 term language to a contextual modal type theory (CMTT). From the perspective of this thesis, CMTT can be viewed as a *meta-language* version of our coeffect calculus.

CONTEXT IN CMTT AND COEFFECTS. Aside from the fact that coeffect calculi use comonads for *semantics* and CMTT embeds comonads (the $\Box$ modality) into the meta-language, there are two important differences.

Firstly, the *context* in CMTT is a set of variables required by a computation, which makes CMTT useful for meta-programming and staged computations. In coeffect calculi, the context requirements are formed by an abstract coeffect algebra, which is more general and can capture variable contexts, but also integers, two-point lattices, etc.

Secondly, CMTT uses different intuitive understanding of the comonad (type constructor) and the associated operations. In categorical semantics of coeffect calculi, the $C^r\tau$ constructor refers to a value of type τ *together* with additional context specified by the coeffect annotation r (e. g. a list of past values or additional resources).

In contrast in CMTT[2] the type $C^{\Psi}\tau$ models a value that *requires* the context Ψ in order to produce value τ. This also changes the interpretation of the two operations of a comonad:

$$\begin{aligned}&\mathsf{counit} : C^{\mathsf{use}}\alpha \to \alpha\\&\mathsf{cobind} : (C^{r}\alpha \to \beta) \to C^{r \circledast s}\alpha \to C^{s}\beta\end{aligned}$$

Two readings of the signatures are possible, but they give quite different meanings to the operations:

- *Coeffect interpretation.* The counit operation extracts a value and does not require any additional context; the cobind operation requires context $r \circledast s$, uses the part specified by r to evaluate the function, ending with a value β together with remaining context s.
- *CMTT interpretation.* The counit operation evaluates a computation that requires no additional context to obtain a α value; given a function that turns a computation requiring context r into a value β, the cobind operation can turn a computation that requires context $r \circledast s$ into a computation that requires just s and contains β (a part of the context requirements is eliminated by the function).

Although the different reading does not affect formal properties of the systems, it is important to understand the difference when discussing the two systems as they provide a different intuition.

The sketch of a coeffect meta-language in the following section attempts to bridge the gap between coeffects and CMTT. Just like CMTT, it embeds comonads as language constructs, but it annotates them with a (flat) coeffect algebra, thus it generalizes CMTT which tracks only sets of variables. Future work on the coeffect meta-language would thus be an interesting development for both coeffect systems and CMTT.

2.3.2 *Coeffect meta-language*

The coeffect meta-language could be designed using both flat and structural indexed comonads. For simplicity, this section only discusses the flat variant. The syntax of types and terms of the language includes the type constructor $C^{r}\tau$ and four additional language constructs:

$$\begin{aligned}\tau \;&::=\; \alpha \mid \tau_1 \to \tau_2 \mid C^{r}\tau\\ e \;&::=\; v \mid \lambda x.e \mid e_1\ e_2 \mid\ !e\\ &\quad\mid\; \texttt{let box}\ x = e_1\ \texttt{in}\ e_2\\ &\quad\mid\; \texttt{split}\ e_1\ \texttt{into}\ x, y\ \texttt{in}\ e_2\\ &\quad\mid\; \texttt{merge}\ e_1, e_2\ \texttt{into}\ x\ \texttt{in}\ e_2\end{aligned}$$

The $!e$ and `let box` constructs correspond to the counit and cobind operation of the comonad. To define meta-language for flat indexed comonads, we also include `split` and `merge` that embed the corresponding operations.

TYPES FOR COEFFECT META-LANGUAGE. The type system for the language is shown in Figure 7. The first part shows the usual typing rules for simply-typed λ-calculus. For simplicity, we omit typing rules for pairs, but those need to be present as the `merge` operation works on tuples.

The second part of the typing rules is more interesting. The (*counit*) operation extracts a value from a comonadic context and corresponds to variable

2 To avoid using different notations, we write $C^{\Psi}\tau$ instead of the original $[\Psi]\tau$

a.) Typing rules for the simply typed λ-calculus

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

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

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

b.) Additional typing rules arising from *flat indexed comonads*

$$(cobind)\ \frac{\Gamma \vdash e_1 : C^{r \circledast s}\tau_1 \qquad \Gamma, x : C^{r}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let box}\ x = e_1\ \texttt{in}\ e_2 : C^{s}\tau_2}$$

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

$$(split)\ \frac{\Gamma \vdash e_1 : C^{r \oplus s}\tau_1 \qquad \Gamma, x : C^{r}\tau_1, y : C^{s}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{split}\ e_1\ \texttt{into}\ x, y\ \texttt{in}\ e_2 : \tau_2}$$

$$(merge)\ \frac{\Gamma \vdash e_1 : C^{r}\tau_1 \qquad \Gamma \vdash e_2 : C^{s}\tau_2 \qquad \Gamma, x : C^{r \wedge s}(\tau_1 \times \tau_2) \vdash e_2 : \tau_3}{\Gamma \vdash \texttt{merge}\ e_1, e_2\ \texttt{into}\ x\ \texttt{in}\ e_2 : \tau_3}$$

Figure 7: Type system for the (flat) coeffect meta-language

access in coeffect calculi. The `let box` construct (*cobind*) takes an input e_1 with context $r \circledast s$ and a computation that turns a variable x with a context r into a value τ_2. The result is a computation that produces a τ_2 value with the remaining context specified by s. Note that the expressions e_2 and e_1 correspond to the first and second arguments of the cobind operation. The keyword `let box` is chosen following CMTT[3].

The `split` and `merge` constructs follow a similar pattern. They both apply some transformation on one or two values in a context and then add the new value as a fresh variable to the variable context. We omit subcoeffecting, but it could be easily added following the method used elsewhere.

2.3.3 *Embedding flat coeffect calculus*

The *meta-language* approach of embedding comonads in a language is more general than the *semantics* approach. This thesis focuses on a narrower use that better guides the design of a type system for context-aware programming languages.

However, it is worth demonstrating that the (flat) coeffect calculus can be embedded in the meta-language described in the previous section. This may be desirable, e. g. when using the meta-language for reasoning about context-aware computations. We briefly consider the embedding as it illuminates the relationship between coeffects and CMTT (unfortunately it is not possible to embed coeffect calculi in CMTT because of the more general annotations structure).

3 The rule is similar to the letbox rule for ICML [71, p. 14], although it differs because of our generalization of comonads where bind composes coeffect annotations rather than requiring the same annotation everywhere.

$$[\![\Gamma\,@\,\mathsf{use} \vdash x_i : \tau_i]\!]_\nu = \pi_i(!\nu)$$

$$[\![\Gamma\,@\,r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2]\!]_\nu =$$
$$\quad \mathtt{split}\ \nu\ \mathtt{into}\ \nu_s, \nu_{rt}\ \mathtt{in}$$
$$\quad [\![\Gamma\,@\,s \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]_{\nu_s}\ (\mathtt{let\ box}\ \nu_r = \nu_{rt}\ \mathtt{in}\ [\![\Gamma\,@\,r \vdash e_2 : \tau_1]\!]_{\nu_r})$$

$$[\![\Gamma\,@\,r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!]_\nu = \lambda x.$$
$$\quad \mathtt{merge}\ \nu, x\ \mathtt{into}\ \nu_{rs}\ \mathtt{in}\ [\![(\Gamma, x : \tau_1)\,@\,r \wedge s \vdash e : \tau_2]\!]_{\nu_{rs}}$$

Figure 8: Embedding flat coeffect calculus in coeffect meta-language

Given a typing judgement $\Gamma\,@\,r \vdash e : \tau$ in the flat coeffect calculus, we define $[\![\Gamma\,@\,r \vdash e : \tau]\!]_\nu$ as its embedding in the coeffect meta-language. Note that the translation is indexed by ν, which is a name of variable used to represent the entire variable context of the source language. The translation is defined in Figure 8. The embedding resembles the semantics discussed in Section **??**. This is not surprising as the meta-language directly mirrors the operations of an indexed comonad.

2.4 FURTHER WORK

Most of the further work has been discussed throughout the thesis, so this section serves mainly as an overview. We follow the same structure as when discussing pathways to coeffects (Chapter **??**). We consider further work related to syntactic approach to coeffect systems, categorical semantics of coeffects, and connections with substructural type systems and bunched logics.

SYNTACTIC COEFFECT SYSTEMS. The coeffect calculi presented in this thesis are based on a simple λ-calculus, comprising variables, lambda abstraction, application and let-binding. This lets us focus on fundamental properties of coeffect systems (and explain how coeffect system differ from better-known effect systems), but it is hardly sufficient for a realistic programming language.

Further work is to extend the coeffect systems presented here to a full programming language. A useful reference is the work of Nielson and Nielson [72] who consider a similar development for effect systems, adding conditionals, recursion and polymorphism.

Coeffect annotations in context-aware programming languages can contain significant amount of information. Thus, the programming language also needs a form of type inference that can propagate such information. This has been done for effect systems [63]. For *flat coeffects*, type inference is complicated by the non-determinism in the typing rule for lambda abstraction, but the problem is simpler for *structural coeffects*. In addition to the *algebra providers* (discussed in Section **??**), it would also be worth developing a bidirectional type system [32] for coeffects, where *some* annotations are required, but most types and coeffects are reconstructed automatically.

LANGUAGE SEMANTICS. As discussed in Section **??**, comonads can be used to define the semantics of a programming language in two ways. The first, "language semantics", approach is to use a single comonad to describe the semantics of a specific context-aware property. The second, "meta-

language" approach is to extend the language with explicit constructs representing the operations of the underlying comonad.

The categorical semantics in this thesis used the "language semantics" style, but mainly as a guide for the development of the type systems. Further work includes more precise treatment of the categorical structure of indexed comonads. This has partly been done for flat coeffects by Orchard [76]. A joint work with Orchard [85] shows the first steps for structural systems. As discussed in Section **??**, further work is to develop a similar categorical model for the unified coeffect calculus, possibly using the categorical notion of containers [3].

We briefly outlined a calculus based on the "meta-language" approach in Section **??**. Developing this technique further could unify coeffect systems with the work on Contextual Modal Type-Theory (CMTT) [71] and allow other interesting applications of coeffects such as meta-programming [71] and distributed computing with explicit modalities [69].

SUBSTRUCTURAL AND BUNCHED LOGICS. In the *flat* and *structural* coeffect calculi, we attach annotations to the *entire context* and to *individual variables*, respectively. We later unified the two systems in Section **??**.

An intriguing question is whether coeffects can generalize *bunched typing* [74], which uses a tree-like structure of variable contexts (also discussed in Section **??**). The definition of the *unified coeffect calculus* is likely not expressive enough – bunched typing requires tree-like variable context, while we use a vector. Furthermore, bunched typing annotates internal nodes of the tree (sub-contexts are combined using ";" or using ",") rather than just variables. Finding a simple coeffect system that is capable of capturing bunched typing is thus an interesting further work. Indeed, this could lead to numerous uses of coeffects as bunched typing is the basis for widely-used separation logic [75].

APPLICATIONS. The last direction for further development is developing practical programming languages or libraries based on the theory presented in this thesis. We believe that the most useful approach is to extend programming languages with a reusable coeffect system and allow library developers define their own coeffect algebras. This option has been discussed in Section **??**. For languages like Haskell, this can be done through a light-weight syntactic sugar akin to the "do" notation.

Another option is to use coeffects just as the underlying theory of a single-purpose programming language feature. When discussing why context-aware programming matters (Section **??**), we covered a number of concrete problems that developers are facing today. Each of those would presents an important further work. The most important practical challenges are cross-compilation (often solved using complex #if pre-processor rules) and tracking of provenance and security properties.

2.5 SUMMARY

In this chapter, we completed our quest of presenting a *unified* system for tracking contextual properties and then we discussed two important topics of future and related work that require in-depth discussion.

The key technical contribution of this chapter is the *unified coeffect system* (Section 2.1), which unifies the flat and structural system presented in the previous two chapters. To achieve this, we introduced *coeffect container*,

which determines how are coeffect annotations attached to variable contexts. We then discussed two instances of the structure that capture flat and structural properties.

In the rest of the chapter, we discussed two topics of future and related work. First (Section 2.2), we considered pathways to practical implementations of coeffect systems, including a discussion of a novel lattice-based coeffect system, which is simpler, but cannot capture all our motivating examples. Finally, we discussed how our work relates to meta-language based on comonads (Section 2.3). We present a *coeffect meta-language* that follows similar style to CMTT, but is based on indexed comonads. Next, we discussed how to embed flat coeffect systems in this coeffect meta-language, which elucidates the relationship between our work and CMTT.

3 CONCLUSIONS

Some of the most fundamental academic work is not the one solving hard research problems, but the one that changes how we understand the world. Some philosophers argue that *language* is the key for understanding how we think, while in science the dominant thinking is determined by *paradigms* [58] or *research programmes* [59]. In a way, programming languages play a similar role for computer science and software development.

This thesis aims to change how developers and programming language designers think about *context* or *execution environments* of programs. Such context or execution environment has numerous forms – mobile applications access network, GPS location or user's personal data; journalists obtain information published on the web or through open government data initiatives; in dataflow programming, we have access to past values of expressions. This thesis aims to change our understanding of *context* so that the above examples are viewed uniformly through a single programming language abstraction, which we call *coeffects*, rather than as disjoint cases.

In this chapter, we give a brief overview of the technical contributions of this thesis (Section 3.1). The thesis looks at two kinds of context-dependence, identifies common patterns and captures those using two *coeffect calculi*. It gives formal semantics using categorical foundations and uses it as a basis for prototype implementation. Finally, Section 3.2 concludes the thesis.

3.1 OVERVIEW OF CONTRIBUTIONS

As observed in Chapter **??**, modern computer programs run in rich and diverse environments or *contexts*. The richness means that environments provides additional resources and capabilities that may be accessed by the program. The diversity means that programs often need to run in multiple different environments, such as mobile phones, servers, web browsers or even on the GPU. In this thesis, we present the foundations for building programming languages that simplify writing software for such rich and diverse environments.

NOTIONS OF CONTEXT. In λ-calculus, the term *context* is usually refers to the free-variable context. However, other programming language features are also related to context or program's execution environment. In Chapter **??**, we revisit many of such features – including resource tracking in distributed computing, cross-platform development, dataflow programming and liveness analysis, but also Haskell's implicit parameters.

The main contribution of the chapter is that it presents the disjoint language features in a unified way. We show type systems and semantics for many of the languages, illuminating the fact that they are closely related.

Considering applications is one way of approaching the theory of coeffects introduced in this thesis. Other pathways to coeffects are discussed in Chapter **??**, which looks at theoretical developments leading to coeffects, including the work on effect systems, comonadic semantics and linear logics.

FLAT COEFFECT CALCULUS. The applications discussed in Chapter **??** fall into two categories. In the first category (Section **??**), the additional contextual information are *whole-context* properties. They either capture properties of the execution environment or affect the whole free-variable context.

In Chapter **??**, we develop a *flat coeffect calculus* which gives us an unified way of tracking *whole-context* properties. The calculus is parameterized by a *flat coeffect algebra* that captures the algebraic properties of contextual information. Concrete instances of flat coeffects include Haskell's implicit parameters, whole-context liveness and whole-context dataflow.

We define a type system for the calculus (Section **??**), discuss how to resolve the ambiguity inherent in context-aware programming for sample languages we consider (Section **??**) and study equational properties of the calculus (Section **??**). In the flat coeffect calculus, β-reduction and η-expansion do not generally preserve the type of an expression, but we identify two conditions when this is the case.

STRUCTURAL COEFFECT CALCULUS. The second category of context-aware systems discussed in Section **??** captures *per-variable* contextual properties. The systems discussed here resemble substructural logics, but rather than *restricting* variable use, they *track* how the variables are used.

We unify systems with *per-variable* contextual properties in Chapter **??**, which describes our *structural coeffect calculus* (Section **??**). Similarly to the flat variant, the calculus is parameterized by a *structural coeffect algebra*. Concrete instances of the calculus track bounded variable use (i. e. how many times is a variable accessed), dataflow properties (how many past values are needed) and liveness (i. e. can variable be accessed).

The structural coeffect calculus has desirable equational properties (Section **??**). In particular, type preservation for β-reduction and η-expansion holds for all instances of the structural coeffect calculus. This follows from the fact that structural coeffect associates contextual requirements with individual variables and preserves the connection by including explicit structural rules (weakening, exchange and contraction).

SEMANTICS AND SAFETY. Both of the coeffect calculi are parameterized and can be instantiated to obtain a language with a type system that tracks concrete notion of context dependence. The coeffect calculus can be seen as a framework for building concrete domain-specific context-aware programming languages. In addition to the type system, the framework also provides a way for defining the semantics of concrete domain-specific context-aware languages, guides their implementation and simplifies safety proofs.

This is done using a *comonadically-inspired* semantics. We generalize the category-theoretical dual of monad to *indexed comonad* and use it to define the semantics of context-aware programs for both the flat coeffect calculus (Chapter **??**) and the structural coeffect calculus (Section **??**). Due to the ambiguity inherent in contextual properties, we define the semantics over a typing derivation, but we also give a domain-specific algorithm for choosing a unique typing derivation for each of our sample languages (Section **??** and Section **??**).

We use comonads in a syntactic way, following the example of Haskell's use of monads and treat it as a *translation* from source context-aware programming language to a simple target functional language. The target language includes uninterpreted comonadically-inspired primitives whose concrete reduction rules need to be provided for each concrete context-aware

deomain-specific language. We give concrete definitions and prove type safety for three sample context-aware languages (Section **??** and Section **??**). Together with well-typedness of our translation, these guarantee that "well-typed context-aware programs do not go wrong".

INTERACTIVE ESSAY. A part of the thesis is an interactive web-based essay (http://tomasp.net/coeffects), which implements three simple context-aware languages. In addition to more traditional prototype language implementations, the essay provides a number of novel features. As usual, it lets users write and evaluate sample programs, but it also lets them explore the details of the coeffect theory – this includes an interactive way of exploring typing derivations and program semantics. The essay also provides a number of case studies that highlight interesting aspects of the theory.

The implementation (Chapter 1) serves two purposes. First, it links together all parts of the theory developed in this thesis (Section 1.2). It uses the two coeffect calculi to build a language framework that is then instantiated to implement three concrete context-aware languages. The framework provides the type system and translation (based on the comonadically-inspired semantics) that is used for program execution. This shows the practical benefits of using the theory of coeffects as a basis of real-world programming languages.

Second, the implementation uses novel kind of medium (Section 1.3) to make our work *accessible* and *explorable*. Building a production-ready programming language is outside the scope of the thesis, so instead, our goal is to explain the theory and inspire authors of real-world programming languages to include support for context-aware programming, ideally using coeffects as a sound foundation. For this reason, we make the implementation accessible over web without requiring installation of specialized software and we make it explorable to encourage active reading.

3.2 SUMMARY

We believe that understanding what programs *require* from the world is equally important as how programs *affect* the world.

The latter has been uniformly captured by effect systems and monads. Those provide not just technical tools for defining semantics and designing type systems, but they also *shape* our thinking – they let us view seemingly unrelated programming language features (exceptions, state, I/O) as instances of the same concept and thus reduce the number of distinct language features that developers need to understand.

This thesis aims to provide a similar unifying theory and tools for capturing context-dependence in programming languages. We showed that programming language features (liveness, dataflow, implicit parameters, etc.) that were previously treated separately can be captured by a common framework developed in this thesis. The main technical contribution of this thesis is that it provides the necessary tools for programming language designers – including parameterized type systems, categorical semantics based on indexed comonads and equational theory.

If there is a one thing that the reader should remember from this thesis, it is the fact that there is a unified notion of *context*, capturing many common scenarios in programming.

BIBLIOGRAPHY

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

[2] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, pages 23–38. Springer, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. In *Proceedings of the 11th international conference on Database programming languages*, DBPL'07, pages 138–152, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 957–964. ACM, 2009.

[22] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *Proceedings of ICFP*, ICFP '13, pages 403–416, 2013.

[23] J. Clarke. *SQL Injection Attacks and Defense*. Syngress, 2009.

[24] J.-L. Colaço and M. Pouzet. Type-based initialization analysis of a synchronous dataflow language. *Int. J. Softw. Tools Technol. Transf.*, 6(3):245–255, Aug. 2004.

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

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

[27] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999.

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

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

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

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

[32] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 429–442. ACM, 2013.

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

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

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

[36] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, 1991.

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

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

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

[40] G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. *ACM SIGPLAN Notices*, 46(12):13–22, 2012.

[41] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical computer science*, 97(1):1–66, 1992.

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

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[44] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Eng.*, 34(1):65–81, Jan. 2008.

[45] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

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

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

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

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

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

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

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

[53] S.-y. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645, New York, NY, USA, 2014. ACM.

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

[55] A. J. Kennedy. Relational parametricity and units of measure. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 442–455. ACM, 1997.

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

[57] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[58] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago Press, 1970.

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

[60] D. Leijen and E. Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[61] J. R. Lewis, M. B. Shields, E. Meijert, and J. Launchbury. Implicit parameters: dynamic scoping with static types. In *Proceedings of POPL*, POPL '00, 2000.

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

[63] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

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

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

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

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

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

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

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

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

[72] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, pages 114–136. Springer, 1999.

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

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

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

[76] D. Orchard. Programming contextual computations.

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

[78] D. Orchard and A. Mycroft. Efficient and correct stencil computation via pattern matching and static typing. In *Proceedings of DSL 2011*, arXiv preprint arXiv:1109.0777, 2011.

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

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

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

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

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

[84] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: unified static analysis of context-dependence. In *Proceedings of International Conference on Automata, Languages, and Programming - Volume Part II*, ICALP 2013.

[85] T. Petricek, D. Orchard, and A. Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 123–135, 2014.

[86] T. Petricek and D. Syme. The f# computation expression zoo. In *Proceedings of Practical Aspects of Declarative Languages*, PADL 2014.

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

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

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

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

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

[92] C. W. Probst, C. Hankin, and R. R. Hansen, editors. *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*. Springer, 2016.

[93] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 13–24, 2008.

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

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

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

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

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

[99] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *J. Funct. Program.*, 17(4-5):547–612, July 2007.

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

[101] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *ACM SIGPLAN Notices*, volume 40, pages 183–194. ACM, 2005.

[102] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ml. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 15–27, New York, NY, USA, 2011. ACM.

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

[104] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming*, DDFP '13, pages 1–4, 2013.

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

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

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

[108] R. Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

[109] The F# Software Foundation. F#. See http://fsharp.org, 2014.

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

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

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

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

[114] T. Uustalu and V. Vene. The essence of dataflow programming. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 2–18, Berlin, Heidelberg, 2005. Springer-Verlag.

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

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

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

[118] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

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

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

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

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

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

[124] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM, 1988.

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

[126] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

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

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

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

[130] H. Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(02):215–286, 2007.

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