CONTEXT-AWARE PROGRAMMING LANGUAGES

TOMAS PETRICEK

Computer Laboratory
University of Cambridge

2014

ABSTRACT

The development of programming languages needs to reflect important changes in the industry. In recent years, this included e. g. the development of parallel programming models (in reaction to the multi-core revolution). This thesis is a response to another such revolution – the diversification of devices and systems where programs run.

We develop the foundations for statically typed functional languages that understand the *context* or environment in which programs execute. Such context includes different platforms (and their versions) in cross-platform applications, 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 context (tracking variable usage in static analyses) or past values in stream-based data-flow programming.

The thesis presents three *coeffect* calculi that capture different notions of context-awareness: *flat* calculus capturing contextual properties of the execution environment, *structural* calculus capturing contextual properties related to variable usage and *meta-language* calculus which allows reasoning about multiple notions of context-dependence in a single language and provides pathway to embedding the other two calculi in existing languages.

Although the focus of this thesis is on the syntactical properties of the presented systems, we also discuss their category-theoretical motivation. We introduce the notion of an *indexed* comonad (the dual of monads) and show how they provide semantics of the presented three calculi.

CONTENTS

1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

Many advances in programming language design are driven by some practical motivations. Sometimes, the practical motivations are easy to see – for example, when they come from an external change such as the rise of multi-core processors. Sometimes, discovering the practical motivations is a difficult task – perhaps because we are so used to a certain way of doing things that we do not even *see* the flaws of our approach.

Before exploring the motivations leading to this thesis, we briefly consider two recent practical concerns that led to the development of new programming languages. This helps to explain why context-aware programming is of importance. The examples are by no means representative, but they illustrate different kinds of motivations well.

PARALLEL PROGRAMMING. The rise of multi-core CPUs is a clear example of an external development influencing programming language research. As multi-core and multi-processor systems became de-facto standard, languages had to provide better abstractions for parallel programming. This led to the industrial popularity of *immutable* data structures (and functional programming in general), software transactional memory [5], data-parallelism and also asynchronous computing [12].

In this case, the motivation is easy to see – writing multi-core programs using earlier abstractions, such as threads and locks, is difficult and error-prone. At the same time, multi-core CPUs become the standard very quickly and so the lack of good language abstractions was apparent.

DATA ACCESS. Accessing data is an example of a more subtle challenge. Initiatives like open government data[1] certainly make more data available. However, to access the data, one has to parse CSV and Excel files, issue SQL or SPARQL queries (to query database and the semantic web, respectively).

Technologies like LINQ [8] make querying data significantly easier. But perhaps because accessing data became important more gradually, it was not easy to see that inline SQL is a poor solution *before* better approaches were developed.

This is even more the case for *type providers* – a recent feature in F# that integrates external data sources directly into the type system of the language and thus makes data explorable directly from the source code editor (through features such as auto-completion on object members). It is not easy to see the limitations of standard techniques (using HTTP requests to query REST services or parsing CSV files and using string-based lookup) until one sees how type providers change the data-scientist's workflow[2].

CONTEXT-AWARE PROGRAMMING. In this chapter, we argue that the next important practical challenge for programming language designers is designing languages that are better at working with (and understanding) the *context in which programs are executed*.

1 In the UK, the open government data portal is available at: `http://data.gov.uk/`
2 This is difficult to explain in writing and so the reader is encouraged to watch a video showing type providers for the WorldBank and CSV data sources [10].

This challenge is of the kind that is not easy to see – perhaps because we are so used to doing things in certain ways that we cannot see their flaws. In this chapter, we aim to uncover such flaws – we look at a number of basic programs that rely on contextual information, we explain why they are inappropriate and then we briefly outline how this thesis remedies the situation.

Putting deeper philosophical questions about the nature of scientific progress aside, the goal of programming language research is generally to design languages that provide more *appropriate abstractions* for capturing common problems, are *simple* and more *unified*. These are exactly the aims that we follow in this thesis. In this chapter, we explain what the common problems are. In Chapter 4 and Chapter 5, we develop two simple calculi to understand and capture the structure of the problems and, finally, Chapter 6 unifies the two abstractions.

1.1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

The phrase *context in which programs are executed* sounds rather abstract and generic. What notions of *context* can be identified in modern software systems? Different environments provide different resources (e. g. database or GPS sensor), environments are increasingly diverse (e. g. multiple versions of different mobile platforms). Web applications are split between client, server and mobile components; mobile applications must be aware of the physical environment while the "internet of things" makes the environments even more heterogeneous. At the same time, applications access rich data sources and need to be aware of security policies and provenance information from the environment.

Writing such context-aware (or environment-aware) applications is a fundamental problem of modern software engineering. The state of the art relies on ad-hoc approaches – using hand-written conditions or pre-processors for conditional compilation. Common problems that developers face include:

- **System capabilities.** When writing code that is cross-compiled to multiple targets (e.g. SQL [8], OpenCL or JavaScript [7]) a part of the compilation (generating the SQL query) often occurs at runtime and developers have no guarantee that it will succeed until the program is executed.

- **Platform versions.** When developing cross-platform applications, different platforms (and different versions of the same platform) provide different API functions. Writing a cross-platform code usually relies on (fragile) conditional compilation or (equally fragile) dynamic loading.

- **Security and provenance.** When working with data (be it sensitive database or social network data), we have permissions to access only some of the data and we may want to track *provenance* information. However, this is not checked – if a program attempts to access unavailable data, the access will be refused at run-time.

- **Resources & data availability.** When creating a mobile application, the program may (or may not) be granted access to device capabilities such as GPS sensor, social updates or battery status. We would like to know which of the capabilities are required and which are optional (i. e. enhance the user experience, but there is a fallback strategy).

```
for header, value in header do
    match header with
    | "accept" → req.Accept ← value
#if FX_NO_WEBREQUEST_USERAGENT
    | "user-agent" → req.UserAgent ← value
#else
    | "user-agent" → req.Headers.[HttpHeader.UserAgent] ← value
#endif
#if FX_NO_WEBREQUEST_REFERER
    | "referer" → req.Referer ← value
#else
    | "user-agent" → req.Headers.[HttpHeader.Referer] ← value
#endif
    | other → req.Headers.[other] ← value
```

Figure 1: Conditional compilation in the HTTP module of the F# Data library

Equally, on the server-side, we might have access to different database tables and other information sources.

Most developers do not perceive the above as programming language flaws – they are simply common programming problems (at most somewhat annoying and tedious) that had to be solved. However, this is because we do not realize that a suitable language extension could make the above problems significantly easier to solve. As the number of distinct contexts and their diversity increases, these problems will become even more commonplace.

The following sub-sections explore four examples in more details. The examples are chosen to demonstrate two distinct forms of contexts that are studied in this thesis.

1.1.1 *Context awareness #1: Platform versioning*

The diversity across devices means that developers need to target an increasing number of platforms and possibly also multiple versions of each platform. For Android, there is a number called API level [3] which "uniquely identifies the framework API revision offered by a version of the Android platform". Most changes in the libraries (but not all) are additive.

Equally, in the .NET ecosystem, there are multiple versions of the .NET runtime, mobile and portable versions of the framework etc. The differences may be subtle – for example, some members are omitted to make the mobile version of the library smaller, some functionality is not available at all, but naming can also vary between versions.

For example, the Figure 1 shows an excerpt from the Http module in the F# Data library[3]. The example uses conditional compilation to target multiple versions of the .NET framework. Such code is difficult to write – to see whether a change is correct, it had to be recompiled for all combinations of pre-processor flags – and maintaining the code is equally hard. The above example could be refactored and the .NET API could be cleaner, but the

3 The file version shown here is available at: https://github.com/fsharp/FSharp.Data/blob/b4c58f4015a63bb9f8bb4449ab93853b90f93790/src/Net/Http.fs

fundamental issue remains. If the language does not understand the context (here, the different platforms and platform versions), it cannot provide any static guarantees about the code.

As an alternative to conditional compilation, developers can use dynamic loading. For example, on Android, programs can access API from higher level platform dynamically using techniques like reflection and writing wrappers. This is even more error prone. As noted in an article introducing the technique[4]: *"Remember the mantra: if you haven't tried it, it doesn't work"*. Again, it would be reasonable to expect that statically-typed languages could provide a better solution.

1.1.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use meta-programming techniques (such as LINQ [8] or F# quotations [11]) to translate code written in a subset of a host language to some other target language, such as SQL, OpenCL or JavaScript. This is an important technique, because it lets developers targets multiple heterogeneous runtimes that have limited execution capabilities.

For example, consider the following LINQ query written in C# that queries a database and selects product names where the first upper case letter is "C":

```
var db = new NorthwindDataContext();

from p in db.Products
where p.ProductName.First(c ⇒ Char.IsUpper(c)) == "C"
select p.ProductName;
```

This appears as a perfectly valid code ant the C# compiler accepts it. However, when the program is executed, it fails with the following error:

> Unhandled Exception: `System.NotSupportedException`: Sequence operators not supported for type `System.String`.

The problem is that LINQ can only translate a *subset* of normal C# code. The above snippet uses First method to iterate over characters of a string, which is not supported. This is not a technical limitation of LINQ, but a fundamental problem of the approach.

When cross-compiling to a limited environment, we cannot always support the full source language. The example with LINQ and SQL demonstrates the importance of this problem. As of March 2014, Google search returns 11800 results for the message above and even more (44100 results) for a LINQ error message *"Method X has no supported translation to SQL"* caused by a similar limitation.

1.1.3 *Context awareness #3: Confidentiality and provenance*

The previous two examples were related to the non-existence of some library functions in another environment. Another common factor was that they were related to the execution context of the whole program or a scope. However, contextual properties can be also related to specific variables.

4 Retrieved from: `http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html`

For example, consider the following code sample that accesses database by building a SQL query using string concatenation:

```
let query = sprintf "SELECT * FROM Products WHERE Name='%s'" name
let cmd = new SqlCommand(query)
let reader = cmd.ExecuteReader()
```

The code compiles without error, but it contains a major security flaw called *SQL injection* (an attacker could enter `"'; DROP TABLE Products --"` as their name and delete the database table with products). For this reason, most libraries discourage building SQL commands by string concatenation, but there are still many systems that do so.

The example demonstrates a more general property. Sometimes, it is desirable to track additional meta-data about variables that are in some ways special. Such meta-data can determine how the variables can be used. Here, name comes from the user input. This *provenance* information should be propagated to query. The SqlCommand object should then require arguments that can not directly contain user input (in an unchecked form). Such marking of values (but at run-time) is also called tainting [4].

Similarly, if we had password or creditCard variables in a client/server web application, these should be annotated as sensitive and it should not be possible to send their values over an unsecured network connection.

In another context, when working with data (e. g. in data journalism), it would be desirable to track meta-data about the quality and the source of the data. For example, is the source trustworthy? Is the data up-to-date? Such meta-data could propagate to the result and tell us important information about the calculated results.

1.1.4 *Context-awareness #4: Checking array access patterns*

The last example leaves the topic of cross-platform and distributed computing. We focus on checking how arrays are accessed. This is a simpler version of the data-flow programming examples used later in the thesis.

Consider a simple programming language with arrays where n^{th} element of an array arr is accessed using arr[n]. Furthermore, we focus on performing local transformations and we assume that the keyword **cursor** returns the *current* location in the array.

The following example implements a simple one-dimensional cellular automata, reading from the input array and writing to output:

```
let sum = input[cursor − 1] + input[cursor] + input[cursor + 1]
if sum = 2 || (sum = 1 && input[cursor − 1] = 0)
then output[cursor] ← 1 else output[cursor] ← 0
```

In this example, we use the term *context* to refer to the values in the array around the current location provided by **cursor**. The interesting question is, how much of the context (i. e. how far in the array) does the program access.

This is contextual information attached to individual (array) variables. In the above example, we want to track that input is accessed in the range $\langle -1, 1\rangle$ while output is accessed in the range $\langle 0, 0\rangle$. When calculating the ranges, we need to be able to compose ranges $\langle -1, -1$, $\langle 0, 0\rangle$ and $\langle 1, 1\rangle$ (based on the accesses on the first line).

The information about access patterns can be used to efficiently compile the computation (as we know which sub-range of the array might be accessed) and it also allows better handling of boundaries. For example, wraparound behaviour we could pad the input with a known number of elements from the other side of the array.

1.2 TOWARDS CONTEXT-AWARE LANGUAGES

The four examples presented in the previous section cover different notions of *context*. The context can be viewed as execution environment, capabilities provided by the environment or input and meta-data about the input.

The different notions of context can be broadly classified into two categories – those that speak about the environment and those that speak about individual inputs (variables). In this thesis, we refer to them as *flat coeffects* and *structural coeffects*, respectively:

- **Flat coeffects** represent additional data, resources and meta-data that are available in the execution environment (regardless of how they are accessed in a program). Examples include resources such as GPS sensors and battery status (on a phone), databases (on the server), or framework version.
- **Structural coeffects** capture additional meta-data related to inputs. This can include provenance (source of the input value), usage information (how often is the value accessed and in what ways) or security information (whether it contain sensitive data or not).

This thesis follows the tradition of statically typed programming languages. As such, we attempt to capture such contextual information in the type system of context-aware programming languages. The type system should provide both safety guarantees (as in the first three examples) and also static analysis useful for optimization (as in the last example).

Although the main focus of this thesis is on the underlying theory of *coeffects* and on their structure, the following section briefly demonstrates the features that a practical context-aware language, based on the theory of coeffects, can provide.

1.2.1 *Context-aware language in action*

As an example, consider a news reader consisting of server-side (which stores the news in a database) and a number of clients applications for popular platforms (Android, Windows Phone, etc.). A simplified code excerpt that might appear somewhere in the implementation is shown in Figure 2.

We assume that the language supports cross-compilation and splits the single program into three components: one for the server-side and two for the client-side, for iPhone and Windows platforms, respectively. The cross-compilation could be done in a way similar to Links [1], but we do not require explicit annotations specifying the target platform.

If we were writing the code using current main-stream technologies, we would have to create three completely separate components. The server-side would include the fetchNews function, which queries the database. The iPhone version would include fetchLocalNews, which gets the current GPS location and performs a call to the remote server and iPhoneMain, which constructs the user-interface. For Windows, we would also need fetchLocalNews, but this time with windowsMain. When using a language that can be compiled for all of the platforms, we would need a number of **#if** blocks to delimit the platform-specific parts.

```
let fetchNews(loc) =
  let cmd = sprintf "SELECT * FROM News WHERE Location='%s'" loc
  query(cmd, password)

let fetchLocalNews() =
  let loc = gpsLocation()
  remote fetchNews(loc)

let iPhoneMain() =
  createCocoaListing(fetchLocalNews)

let windowsMain() =
  createMetroListing(fetchLocalNews)
```

Figure 2: News reader implemented in a context-aware language

To support cross-compilation, the language needs to be context-aware. Each of the function has a number of context requirements. The fetchNews function needs to have access to a database; fetchLocalNews needs access to a GPS sensor and to a network (to perform the remote call). However, it does not need a specific platform – it can work on both iPhone and Windows. The last two platform-specific functions inherit the requirements of fetchLocalNews and additionally also require a specific platform.

1.2.2 *Understanding context with types*

The approach advocated in this thesis is to track information about context requirements using the type system. To make this practical, the system needs to provide at least partial support for automatic type inference, as the information about context requirements makes the types more complex. An inspiring example might be the F# support for units of measure [6] – the user has to explicitly annotate constants, but the rest of the information is inferred automatically.

Furthermore, integrating contextual information into the type system can provide information for modern developer tools. For example, many editors for F# display inferred types when placing mouse pointer over an identifier. For fetchLocalNews, the tip could appear as follows:

fetchLocalNews

unit @ { gps, rpc } → (news list) async

Here, we use the notation $\tau_1 @ c \rightarrow \tau_2$ to denote a function that takes an input of type τ_1, produces a result of type τ_2 and has additional context requirements specified by c. In the above example, the annotation c is simply a set of required resources or capabilities. However, a more complex structure could be used as well, for example, including the Android API level.

The following summary shows the types of the functions from the code sample in Figure 2. These guide the code generation by specifying which function should be compiled for which of the platforms, but they also provide documentation for the developers:

```
password        :  string @ sensitive
fetchNews       :  location @ { database } → news list

gpsLocation     :  unit @ { gps } → location
fetchLocalNews  :  location @ { gps, rpc } → news list

iPhoneMain      :  unit @ { cocoa, gps, rpc } → unit
windowsMain     :  unit @ { windows, gps, rpc } → unit
```

As mentioned earlier, the concrete syntax used here is just for illustration. Furthermore, some information could even be mapped to other visual representations – for example, differently coloured backgrounds for platform-specific functions. The key point is that the type provides a number of useful information:

- The password variable is available in the context (we assume it has been declared earlier), but is marked as sensitive, which restricts how it can be used. In particular, we cannot return it as a result of a function that is called via a remote call (e. g. fetchNews) as that would leak sensitive data over an unsecured connection.

- The fetchNews function requires database access and so it can only run on the server-side (or on a thick client with local copy of the database, such as a desktop computer with an offline mode).

- The `gpsLocation` function accesses the GPS sensor and since we call it in from `fetchLocalNews`, this function also requires GPS (the requirement is propagated automatically).

- We can compile the program for two client-side platforms - the entry points are iPhoneMain and windowsMain and require Cocoa and Windows user-interface libraries, together with GPS and the ability to perform remote calls over the network.

The details of how the cross-compilation would work are out of the scope of this thesis. However, one can imagine that the compiler would take multiple sets of references (representing the different platforms), expose the *union* of the functions, but annotate each with the required platform. Then, it would produce multiple different binaries – here, one for the server-side (containing fetchNews), one for iPhone and one for Windows.

In this scenario, the main benefit of using an integrated context-aware language would be the ability to design appropriate abstractions using standard mechanisms of the language. For cross-compilation, we can structure code using functions, rather than relying on #**if** directives. Similarly, the splitting between client-side, server-side and shared code can be done using ordinary functions and modules – rather than having to split the application into separate independent libraries or projects.

The purpose of this section was to show that many modern programs rely on the context in which they execute in non-trivial ways. Thus designing context-aware languages is an important practical problem for language designers. The sample serves more as a motivation than as a technical background for this thesis. We explore more concrete examples of properties that can be tracked using the systems developed in this thesis in Chapter 3.

1.3 THEORY OF CONTEXT DEPENDENCE

The previous section introduced the idea of context-aware languages from the practical perspective. As already discussed, we approach the problem from the perspective of statically typed programming languages. This section outlines how can contextual information be integrated into the standard framework of static typing. This section is intended only as an informal overview and the related work is discussed in Chapter 2.

TYPE SYSTEMS. A type system is a form of static analysis that is usually specified by *typing judgements* such as $\Gamma \vdash e : \tau$. The judgement specifies that, given some variables described by the context Γ, the expression e has a type τ. The variable context Γ is necessary to determine the type of expressions. Consider an expression $x + y$. In many languages, including Java, C# and F#, the type could be int or float, depending on the types of the variables. For example, the following is a valid typing judgement in F#:

$$x : \mathsf{int},\ y : \mathsf{int} \vdash x + y : \mathsf{int}$$

This judgement assumes that the type of both x and y is int and so the result must also be int. The expression would also be typeable in a context $x : \mathsf{int},\ y : \mathsf{int}$, but not, for example, in a context where x has a type unit.

TRACKING EVALUATION EFFECTS. Type systems can be extended in numerous ways. The types can be more precise, for example, by specifying the range of an integer. However, it is also possible to track what program *does* when executed. In ML-like languages, the following is a valid judgement:

$$x : \mathsf{int} \vdash \mathsf{print}\ x : \mathsf{unit}$$

The judgement states that the expression print x has a type unit. This is correct, but it ignores the important fact that the expression has a *side-effect* and prints a number to the console. In purely functional languages, this would not be possible. For example, in Haskell, the type would be IO unit meaning that the result is a *computation* that performs I/O effects and then returns unit value.

Another option for tracking effects is to extend the judgement with additional information about the effects. The judgement in a language with effect system would look as follows:

$$x : \mathsf{int} \vdash \mathsf{print}\ x : \mathsf{unit}\ \&\ \{\ \mathsf{console}\ \}$$

Effect systems add *effect annotation* as another component of the typing judgement. In the above example, the return type is unit, but the effect annotation informs us that the expression also accesses console as part of the evaluation. To track such information, the compiler needs to understand the effects of primitive built-in functions – such as print.

The crucial part of type systems is dealing with different forms of composition. For example, assume we have a function read that reads from the console and a function send that sends data over the network. In that case, the type system should correctly infer that the effects of an expression $\mathsf{send}(\mathsf{read}())$ are $\{\mathsf{console}, \mathsf{network}\}$.

Effect systems are an established idea, but they are suitable only for tracking properties of a certain kind. They can be used for properties that describe how programs *affect* the environment. For context-aware languages, we instead need to track what programs *require* from the environment.

TRACKING CONTEXT REQUIREMENTS. The systems for tracking of context requirements developed in this thesis are inspired by the idea of effect systems. To demonstrate our approach, consider the following call from the sample program shown earlier – first using standard ML-like type system:

$$\mathsf{password} : \mathsf{string},\ \mathsf{cmd} : \mathsf{string} \vdash \mathsf{query}(\mathsf{cmd}, \mathsf{password}) : \mathsf{news\ list}$$

The expression queries a database and gets back a list of news values as the result. Recall from the earlier discussion that there are two contextual information that are desirable to track for this expression. First, the call to the query primitive requires *database access*. Second, the password argument needs to be marked as *sensitive value* to avoid sending it over an unsecure network connection. The *coeffect systems* developed in this thesis capture this information in the following way:

$$\begin{array}{l}(\mathsf{password} : \mathsf{string} \mathrel{@} \mathsf{sensitive}, \mathsf{cmd} : \mathsf{string}) \mathrel{@} \{\ \mathsf{database}\ \} \vdash \\ \qquad \mathsf{query}(\mathsf{cmd}, \mathsf{password}) : \mathsf{news\ list}\end{array}$$

Rather than attaching the annotation to the *resulting type*, we attach them to the variable context Γ. In other words, coeffect systems track more detailed information about the context, not just the available variables. In the above example, it tracks meta-data about the variables and annotates password as sensitive. Furthermore, it tracks requirements about the execution environment – for example, that the execution requires an access to database.

The example demonstrates the two kinds of coeffect systems outlined earlier. The tracking of *whole-context* information (such as environment requirements) is captured by the *flat coeffect calculus* developed in Chapter 4, while the tracking of *per-variable* information is captured by the *structural coeffect calculus* developed in Chapter 5.

As mentioned earlier, it is well-known fact that *effects* correspond to *monads* and languages such sa Haskell use monads to provide a limited form of effect system. An interesting observation made in this thesis is that *coeffects*, or systems for tracking contextual information, correspond to the category theoretical dual of monads called *comonads*. The details are explained when discussing the semantics of coeffects throughout the thesis.

1.4 THESIS OUTLINE

The key claim of this thesis is that programming languages need to provide better ways of capturing how programs rely on the context in which they execute. This chapter shows why this is an important problem. We looked at a number of properties related to context that are currently handled in ad-hoc and error-prone ways. Next, we considered the properties in a simplified, but realistic example of a client/server application for displaying local news.

Tracking of contextual properties may not be initially perceived as a major problem – perhaps because we are so used to write code in certain ways that prevent us from seeing the flaws. The purpose of this chapter was to uncover the flaws and convince the reader that there should be a better solution. Finding the foundations of such better solution is the goal of this thesis:

- In Chapter 2 we give an overview of related work. Most importantly, we show that the idea of context-aware computations can be naturally approached from a number of directions developed recently in theories of programming languages. Chapter 3 follows by showing practical motivation for coeffects – we look at a number of systems that can be captured using the systems developed later.

- In Chapter 2 and Chapter 3, we present the key novel contributions of this thesis. We develop the *flat* and *structural* calculi, show how they capture important contextual properties and develop their categorical semantics using a notion based on comonads. Chapter 6 links the two systems into a single formalism that is capable of capturing both flat and structural properties.
- Chapter 7 and Chapter 8

2 PATHWAYS TO COEFFECTS

There are many different directions from which the concept of *coeffects* can be approached and, indeed, discovered. In the previous chapter, we motivated it by practical applications, but coeffects also naturally arise as an extension to a number of programming language theories. Thanks to the Curry-Howard-Lambek correspondence, we can approach coeffects from the perspective of type theory, logic and also category theory. This chapter gives an overview of the most important directions.

We start by revisiting practical applications and existing language features that are related to coeffects (Section 3.1), then we look at coeffects as the dual of effect systems (Section 2.1) and extend the duality to category theory, looking at the categorical dual of monads known as *comonads* (Section 2.2). Finally we look at logically inspired type systems that are closely related to our structural coeffects (Section 2.3).

This chapter serves two purposes. Firstly, it provides a high-level overview of the related work, although technical details are often postponed until later. Secondly it recasts existing ideas in a way that naturally leads to the coeffect systems developed later in the thesis. For this reason, we are not always faithful to the referenced work – sometimes we focus on aspects that the authors consider unimportant or present the work differently than originally intended. The reason is to fulfil the second goal of the chapter. When we do so, this is explicitly said in the text.

2.1 THROUGH TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [2**?**], type and effect systems have been designed to track effectful operations performed by computations. Examples include tracking of reading and writing from and to memory locations [**?**], communication in message-passing systems [**?**] and atomicity in concurrent applications [**?**].

Type and effect systems are usually specified judgements of the form $\Gamma \vdash e : \alpha, \sigma$, meaning that the expression e has a type α in (free-variable) context Γ and additionally may have effects described by σ. Effect systems are typically added to a language that already supports effectful operations as a way of increasing the safety – the type and effect system provides stronger guarantees than a plain type system. Filinsky [**?**] refers to this approach as *descriptive*[1].

SIMPLE EFFECT SYSTEM The structure of a simple effect system is demonstrated in Figure 3. The example shows typing rules for a simply typed lambda calculus with an additional (effectful) operation $l \leftarrow e$ that writes the value of e to a mutable location l. The type of locations ($\mathsf{ref}_\rho\ \alpha$) is annotated with a *memory region* ρ of the location l. The effects tracked by the type and effect system over-approximate the actual effects and memory regions provide a convenient way to build such over-approximation. The effects are

1 In contrast to *prescriptive* effect systems that implement computational effects in a pure language – such as monads in Haskell

$$\text{(var)}\ \frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha, \emptyset} \qquad \text{(write)}\ \frac{\Gamma \vdash e : \alpha, \sigma \qquad l : \mathsf{ref}_\rho\ \alpha \in \Gamma}{\Gamma \vdash l \leftarrow e : \mathsf{unit}, \sigma \cup \{\mathsf{write}(\rho)\}}$$

$$\text{(fun)}\ \frac{\Gamma, x : \alpha_1 \vdash e : \beta, \sigma}{\Gamma \vdash \lambda x.e : \alpha \xrightarrow{\sigma} \beta, \emptyset} \qquad \text{(app)}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \alpha \xrightarrow{\sigma_1} \beta, \sigma_2 \\ \Gamma \vdash e_2 : \alpha, \sigma_3\end{array}}{\Gamma \vdash e_1\ e_2 : \beta, \sigma_1 \cup \sigma_2 \cup \sigma_3}$$

Figure 3: Simple effect system

$$\text{(var)}\ \frac{x : \alpha \in \Gamma}{\Gamma @ \emptyset \vdash x : \alpha} \qquad \text{(access)}\ \frac{\Gamma @ \sigma \vdash e : \mathsf{res}_\rho\ \alpha}{\Gamma @ \sigma_1 \cup \{\mathsf{access}(\rho)\} \vdash \textbf{access}\ e : \alpha}$$

$$\text{(fun)}\ \frac{\Gamma, x : \alpha @ \sigma_1 \cup \sigma_2 \vdash e : \beta}{\Gamma @ \sigma_1 \vdash \lambda x.e : \alpha \xrightarrow{\sigma_2} \beta} \qquad \text{(app)}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \alpha \xrightarrow{\sigma_1} \beta, \sigma_2 \\ \Gamma \vdash e_2 : \alpha, \sigma_3\end{array}}{\Gamma \vdash e_1\ e_2 : \beta, \sigma_1 \cup \sigma_2 \cup \sigma_3}$$

Figure 4: Simple effect system

represented as a set of effectful actions that an expression may perform and the effectful action (*write*) adds a primitive effect $\mathsf{write}(\rho)$.

The remaining rules are shared by a majority of effect systems. Variable access (*var*) has no effects, application (*app*) combines the effects of both expressions, together with the latent effects of the function to be applied. Finally, lambda abstraction (*fun*) is a pure computation that turns the *actual* effects of the body into *latent* effects of the created function.

SIMPLE COEFFECT SYSTEM When writing the judgements of coeffect systems, we want to emphasize the fact that coeffect systems talk about *context* rather than *results*. For this reason, we write the judgements in the form $\Gamma @ \sigma \vdash e : \alpha$, associating the additional information with the context (left-hand side) of the judgement rather than with the result (right-hand side) as in $\Gamma \vdash e : \alpha, \sigma$. This change alone would not be very interesting – we simply used different syntax to write a predicate with four arguments. As already mentioned, the key difference follows from the lambda abstraction rule.

The language in Figure 4 extends simple lambda calculus with resources and with a construct **access** e that obtains the resource specified by the expression e. Most of the typing rules correspond to those of effect systems. Variable access (*var*) has no context requirements, application (*app*) combines context requirements of the two sub-expressions and latent context-requirements of the function.

The (*fun*) rule is different – the resources requirements of the body $\sigma_1 \cup \sigma_2$ are split between the *immediate context-requirements* associated with the current context $\Gamma @ \sigma_1$ and the *latent context-requirements* of the function.

As demonstrated by examples in the Chapter **??**, this means that the resource can be captured when a function is declared (e.g. when it is constructed on the server-side where database access is available), or when a function is called (e.g. when a function created on server-side requires access to current time-zone, it can use the resource available on the client-side).

2.2 THROUGH LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [9] showed that effects (including partiality, exceptions, non-determinism and I/O) can be modelled uisng the category theoretic notion of *monad*.

When using monads, we distinguish effect-free values α from programs, or computations $M\alpha$. The *monad* M abstracts the *notion of computation* and provides a way of constructing and composing effectful computations:

Definition 1. *A* monad *over a category* $\mathcal{C}$ *is a triple* $(M, \mathsf{unit}, \mathsf{bind})$ *where:*

- M *is a mapping on objects (types)* $M : \mathcal{C} \to \mathcal{C}$
- unit *is a mapping* $\alpha \to M\alpha$
- bind *is a mapping* $(\alpha \to M\beta) \to (M\alpha \to M\beta)$

such that, for all $f : \alpha \to M\beta, g : \beta \to M\gamma$*:*

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

Without providing much details, we note that well known examples of monads include the partiality monad ($M\alpha = \alpha + \bot$) also corresponding to the Maybe type in Haskell, list monad ($M\alpha = \mu\gamma.1 + (\alpha \times \gamma)$) and other. In programming language semantics, monads can be used in two distinct ways.

2.2.1 *Effectful languages and meta-languages*

Moggi uses monads to define two formal systems. In the first formal system, a monad is used to model the *language* itself. This means that the semantics of a language is given in terms of a one specific monad and the semantics can be used to reason about programs in that language. To quote *"When reasoning about programs one has only one monad, because the programming language is fixed, and the main aim is to prove properties of programs"* [9, p. 5].

In the second formal system, monads are added to the programming language as type constructors, together with additional constructs corresponding to monadic bind and unit. A single program can use multiple monads, but the key benefit is the ability to reason about multiple languages. To quote *"When reasoning about programming languages one has different monads, one for each programming language, and the main aim is to study how they relate to each other"* [9, p. 5].

In this thesis, we generally follow the first approach – this means that we work with an existing programming language (without needing to add additional constructs corresponding to the primitives of our semantics). To explain the difference in greater detail, the following two sections show a minimal example of both formal systems. We follow Moggi and start with language where judgements have the form $x : \alpha \vdash e : \beta$ with exactly one variable[2].

LANGUAGE SEMANTICS When using monads to provide semantics of a language, we do not need to extend the language in any way – we assume

2 This simplifies the examples as we do not need *strong* monad, but that is an orthogonal issue to the distinction between language semantics and meta-language.

that the language already contains the effectful primitives (such as the assignment operator $x \leftarrow e$ or other). A judgement of the form $x : \alpha \vdash e : \beta$ is interpreted as a morphism $\alpha \to M\beta$, meaning that any expression is interpreted as an effectful computation. The semantics of variable access (x) and the application of a primitive function f is interpreted as follows:

$$\begin{aligned} [\![x : \alpha \vdash x : \alpha]\!] &= \mathsf{unit}_M \\ [\![x : \alpha \vdash f\ e : \gamma]\!] &= (\mathsf{bind}_M\ f) \circ [\![e]\!] \end{aligned}$$

Variable access is an effect-free computation, that returns the value of the variable, wrapped using unit_M. In the second rule, we assume that e is an expression using the variable x and producing a value of type β and that f is a (primitive) function $\beta \to M\gamma$. The semantics lifts the function f using bind_M to a function $M\beta \to M\gamma$ which is compatible with the interpretation of the expression e.

META-LANGUAGE INTERPRETATION When designing meta-language based on monads, we need to extend the lambda calculus with additional type(s) and expressions that correspond to monadic primitives:

$$\begin{aligned} \alpha, \beta, \gamma &:= \tau \mid \alpha \to \beta \mid M\alpha \\ e &:= x \mid f\ e \mid \mathbf{return}_M\ e \mid \mathbf{let}_M\ x \Leftarrow e_1\ \mathbf{in}\ e_2 \end{aligned}$$

The types consist of primitive type (τ), function type and a type constructor that represents monadic computations. This means that the expressions in the language can create both effect-free values, such as α and computations $M\alpha$. The additional expression $\mathbf{return}_M$ is used to create a monadic computation (with no actual effects) from a value and $\mathbf{let}_M$ is used to sequence effectful computations. In the semantics, monads are not needed to interpret variable access and application, they are only used in the semantics of additional (monadic) constructs:

$$\begin{aligned} [\![x : \alpha \vdash x : \alpha]\!] &= \mathsf{id} \\ [\![x : \alpha \vdash f\ e : \beta]\!] &= f \circ [\![e]\!] \\ [\![x : \alpha \vdash \mathbf{return}_M\ e : M\beta]\!] &= \mathsf{unit}_M \circ [\![e]\!] \\ [\![x : \alpha \vdash \mathbf{let}_M\ y \Leftarrow e_1\ \mathbf{in}\ e_2 : M\beta]\!] &= \mathsf{bind}_M\ [\![e_2]\!] \circ [\![e_1]\!] \end{aligned}$$

In this system, the interpretation of variable access becomes a simple identity function and application is just composition. Monadic computations are constructed explicitly using $\mathbf{return}_M$ (interpreted as unit_M) and they are also sequenced explicitly using the $\mathbf{let}_M$ construct. As noted by Moggi, the first formal system can be easily translated to the latter by inserting appropriate monadic constructs.

Moggi regards the meta-language system as more fundamental, because *"its models are more general"*. Indeed, this is a valid and reasonable perspective. Yet, we follow the first style, precisely because it is *less general* – our aim is to develop concrete context-aware programming languages (together with their type theory and semantics) rather than to build a general framework for reasoning about languages with context-dependent properties.

2.2.2 *Marriage of effects and monads*

The work on effect systems and monads both tackle the same problem – representing and tracking of computational effects. The two lines of research have been joined by Wadler and Thiemann [15]. This requires extending the categorical structure. A monadic computation $\alpha \to M\beta$ means that the computation has *some* effects while the judgement $\Gamma \vdash e : \alpha, \sigma$ specifies *what* effects the computation has.

To solve this mismatch, Wadler and Thiemann use a *family* of monads $M^{\sigma}\alpha$ with an annotation that specifies the effects that may be performed by the computation. In their system, an effectful function $\alpha \xrightarrow{\sigma} \beta$ is modelled as a pure function returning monadic computation $\alpha \to M^{\sigma}\beta$. Similarly, the semantics of a judgement $x : \alpha \vdash e : \beta, \sigma$ can be given as a function $\alpha \to M^{\sigma}\beta$. The precise nature of the family of monads has been later called *indexed monads* (e.g. by Tate [13]) and further developed by Atkey [**?**] in his work on *parameterized monads*.

THESIS PERSPECTIVE The key takeaway for this thesis from the outlined line of research is that, if we want to develop a language with type system that captures context-dependent properties of programs more precisely, the semantics of the language also needs to be a more fine-grained structure (akin to indexed monads). While monads have been used to model effects, an existing research links context-dependence with *comonads* – the categorical dual of monads.

2.2.3 *Context-dependent languages and meta-languages*

The theoretical parts of this thesis extend the work of Uustalu and Vene who use comonads to give the semantics of data-flow computations [**?**] and more generally, notions of *context-dependent computations* [14]. The computations discussed in the latter work include streams, arrays and containers – this is a more diverse set of examples, but they all mostly represent forms of collections. Ahman et al. [**?**] discuss the relation between comonads and *containers* in more details.

The utility of comonads has been explored by a number of authors before. Brookes and Geva [**?**] use *computational* comonads for intensional semantics[3]. In functional programming, Kieburtz [**?**] proposed to use comonads for stream programming, but also handling of I/O and interoperability.

Biermann and de Paiva used comonads to model the necessity modality $\Box$ in intuitionistic modal S4 [**?**], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [**?**]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [**?**], which again shows the importance of comonads for *context-dependent* computations.

While Uustalu and Vene use comonads to define the *language semantics* (the first style of Moggi), Nanevski, Pfenning and Davies use comonads as part of meta-language, in the form of $\Box$ modality, to reason about context-dependent computations (the second style of Moggi). Before looking at the details, we use the following definition of comonad:

Definition 2. *A* comonad *over a category* $\mathcal{C}$ *is a triple* $(C, \mathsf{counit}, \mathsf{cobind})$ *where:*

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

3 The structure of computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [**?**].

such that, for all $f : \alpha \to M\beta, g : \beta \to M\gamma$:

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

The definition is similar to monad with "reversed arrows". Intuitively, the counit operation extracts a value α from a value that carries additional context $C\alpha$. The cobind operation turns a context-dependent function $C\alpha \to \beta$ into a function that takes a value with context, applies the context-dependent function to value(s) in the context and then propagates the context. The next section makes this intuitive definition more concrete. More detailed discussion about comonads can be found in Orchard's PhD thesis [**?**].

LANGUAGE SEMANTICS To demonstrate the approach of Uustalu and Vene, we consider the non-empty list comonad $C\alpha = \mu\gamma.\alpha + (\alpha \times \gamma)$. A value of the type is either the last element α or an element followed by another non-empty list $\alpha \times \gamma$. Note that the list must be non-empty – otherwise counit would not be a complete function (it would be undefined on empty list). In the following, we write $(l_1, \ldots, l_n)$ for a list of n elements:

$$
\begin{aligned}
\mathsf{counit}\ (l_1, \ldots, l_n) &= l_1\\
\mathsf{cobind}\ f\ (l_1, \ldots, l_n) &= (f(l_1, \ldots, l_n), f(l_2, \ldots, l_n), \ldots, f(l_n))
\end{aligned}
$$

The counit operation returns the current (first) element of the (non-empty) list. The cobind operation creates a new list by applying the context-dependent function f to the entire list, to the suffix of the list, to the suffix of the suffix and so on.

In causal data-flow, we can interpret the list as a list consisting of past values, with the current value in the head. Then, the cobind operation calculates the current value of the output based on the current and all past values of the input; the second element is calculated based on all past values and the last element is calculated based just on the initial input (l_n). In addition to the operations of comonad, the model also uses some operations that are specific to causal data-flow:

$$
\mathsf{prev}\ (l_1, \ldots, l_n) = (l_2, \ldots, l_n)
$$

The operation drops the first element from the list. In the data-flow interpretation, this means that it returns the previous state of a value.

Now, consider a simple data-flow language with single-variable contexts, variables, primitive built-in functions and a construct **prev** e that returns the previous value of the computation e. We omit the typing rules, but they are simple – assuming e has a type α, the expression **prev** e has also type α. The fact that the language models data-flow and values are lists (of past values) is a matter of semantics, which is defined as follows:

$$
\begin{aligned}
[\![x : \alpha \vdash x : \alpha]\!] &= \mathsf{counit}_C\\
[\![x : \alpha \vdash f\ e : \gamma]\!] &= f \circ (\mathsf{cobind}_C\ [\![e]\!])\\
[\![x : \alpha \vdash \mathbf{prev}\ e : \gamma]\!] &= \mathsf{prev} \circ (\mathsf{cobind}_C\ [\![e]\!])
\end{aligned}
$$

The semantics follows that of effectful computations using monads. A variable access is interpreted using counit_C (obtain the value and ignore additional available context); composition uses cobind_C to propagate the context to the function f and **prev** is interpreted using the primitive prev (which takes a list and returns a list).

$$(\text{eval})\frac{\Gamma \vdash e : C^{\emptyset}\alpha}{\Gamma \vdash !e : \alpha} \qquad (\text{letbox})\frac{\Gamma \vdash e_1 : C^{\Phi,\Psi}\alpha \qquad \Gamma, x : C^{\Phi}\alpha \vdash e_2 : \beta}{\Gamma \vdash \textbf{let box } x = e_1 \textbf{ in } e_2 : C^{\Psi}\beta}$$

Figure 5: Typing for a comonadic language with contextual staged computations

For example, the judgement $x : \alpha \vdash \textbf{prev} \ (\textbf{prev} \ x) : \alpha$ represents an expression that expects context with variable x and returns a stream of values before the previous one. The semantics of the term expresses this behaviour: $(\mathsf{prev} \circ \mathsf{prev} \circ (\mathsf{cobind}_C \ \mathsf{counit}_C))$. Note that the first operation is simply an identity function thanks to the comonad laws discussed earlier.

In the outline presented here, we ignored lambda abstraction. Similarly to monadic semantics, where lambda abstraction requires *strong* monad, the comonadic semantics also requires additional structure called *symmetric (semi)monoidal* comonads. This structure is responsible for the splitting of context-requirements in lambda abstraction. We return to this topic when discussing flat coeffect system later in the thesis.

META-LANGUAGE INTERPRETATION To briefly demonstrate the approach that employs comonads as part of a meta-language, we look at an example inspired by the work of Pfenning, Davies and Nanevski et al. We do not attempt to provide precise overview of their work. The main purpose of our discussion is to provide a different intuition behind comonads, and to give an example of a language that includes comonad as a type constructor, together with language primitives corresponding to comonadic operations[4].

In languages inspired by modal logics, types can have the form $\Box\alpha$. In the work of Pfenning and Davies, this means a term that is provable with no assumptions. In distributed programming language ML5, Murphy et al. [? ?] use the $\Box\alpha$ type to mean *mobile code*, that is code that can be evaluated at any node of a distributed system (the evaluation corresponds to the axiom $\Box\alpha \rightarrow \alpha$). Finally, Davies and Pfenning [?] consider staged computations and interpret $\Box\alpha$ as a type of (unevaluated) expressions of type α.

In Contextual Modal Type Theory, the modality $\Box$ is further annotated. To keep the syntax consistent with earlier examples, we use $C^{\Psi}\alpha$ for a type $\Box\alpha$ with an annotation Ψ. The type is a comonadic counterpart to the *indexed monads* used by Wadler and Thiemann when linking monads and effect systems and, indeed, it gives rise to a language that tracks context-dependence of computations in a type system.

In staged computation, the type $C^{\Psi}\alpha$ represents an expression that requires the context Ψ (i.e. the expression is an open term that requires variables Ψ). The Figure 5 shows two typing rules for such language. The rules directly correspond to the two operations of a comonad and can be interpreted as follows:

- (*eval*) corresponds to $\mathsf{counit} : C^{\emptyset}\alpha \rightarrow \alpha$. It means that we can evaluate a closed (unevaluated) term and obtain a value. Note that the rule requires a specific context annotation. It is not possible to evaluate an open term.
- (*letbox*) corresponds to $\mathsf{cobind} : (C^{\Psi}\alpha \rightarrow \beta) \rightarrow C^{\Psi,\Phi}\alpha \rightarrow C^{\Phi}\beta$. It means that given a term which requires variable context Ψ, Φ (expres-

4 In fact, Pfenning and Davies [? ?] never mention comonads explicitly. This is done in later work by Gabbay et al. [?], but the connection between the language and comonads is not as direct as in case of monadic or comonadic semantics covered in the last few pages.

sion e_1) and a function that turns a term needing Ψ into an evaluated value (expression e_2), we can construct a term that requires just Φ.

The fact that the (*eval*) rule requires a specific context is an interesting relaxation from ordinary comonads where counit needs to be defined for all values. Here, the indexed counit operation needs to be defined only on values annotated with $\emptyset$.

The annotated cobind operation that corresponds to (*letbox*) is in details introduced in Chapter X. An interesting aspect is that it propagates the context-requirements "backwards". The input expression (second parameter) requires a combination of contexts that are required by the two components – those required by the input of the function (first argument) and those required by the resulting expression (result). This is another key aspect that distinguishes coeffects from effect systems.

THESIS PERSPECTIVE As mentioned earlier, we are interested in designing context-dependent languages and so we use comonads as *language semantics*. Uustalu and Vene present a semantics of context-dependent computations in terms of comonads. We provide the rest of the story known from the marriage of monads and effects. We develop coeffect calculus with a type system that tracks the context requirements more precisely (by annotating the types) and we add indexing to comonads and link the two by giving a formal semantics.

The *meta-language* approach of Pfenning, Davies and Nanevski et al. is closely related to our work. Most importantly, Contextual Modal Type Theory (CMTT) uses indexed $\Box$ modality which seems to correspond to indexed comonads (in a similar way in which effect systems correspond to indexed monads). The relation between CMTT and comonads has been suggested by Gabbay et al. [**?**], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our let box typing rule from Figure 5 is not a primitive of CMTT and would correspond to $\mathsf{box}(\Psi, \mathsf{letbox}(e_1, x, e_2))$. Nevertheless, the indexing in CMTT provides a useful hint for adding indexing to the work of Uustalu and Vene.

2.3 THROUGH SUB-STRUCTURAL AND BUNCHED LOGICS

In the coeffect system for tracking resource usage outlined earlier, we associated additional contextual information (set of available resources) with the variable context of the typing judgement: $\Gamma @ \sigma \vdash e : \alpha$. In other words, our work focuses on "what is happening on the left hand side of $\vdash$".

In the case of resources, the additional information about the context are simply added to the variable context (as a products), but we will later look at contextual properties that affect how variables are represented. More importantly, *structural coeffects* link additional information to individual variables in the context, rather than the context as a whole.

In this section, we look at type systems that reconsider Γ in a number of ways. First of all, sub-structural type systems [**?**] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [**?**] can guarantee that variable is used exactly once. This has interesting implications for memory management and I/O.

In bunched typing developed by O'Hearn [**?**], the variable context is a tree formed by multiple different constructors (e.g. one that allows sharing and one that does not). Most importantly, bunched typing has contributed

$$\text{(exchange)}\frac{\Gamma, x : \alpha, y : \beta \vdash e : \gamma}{\Gamma, y : \beta, x : \alpha \vdash e : \gamma} \qquad \text{(weakening)}\frac{\Gamma, \Delta \vdash e : \gamma}{\Gamma, x : \alpha, \Delta \vdash e : \gamma}$$

$$\text{(contraction)}\frac{\Gamma, x : \alpha, y : \alpha, \Delta \vdash e : \gamma}{\Gamma, x : \alpha, \Delta \vdash e[y \leftarrow x] : \gamma}$$

Figure 6: Exchange, weakening and contraction typing rules

to the development of separation logic [**?**] (starting a fruitful line of research in software verification), but it is also interesting on its own.

SUB-STRUCTURAL TYPE SYSTEMS Traditionally, Γ is viewed as a set of assumptions and typing rules admit (or explicitly include) three operations that manipulate the variable contexts which are shown in Figure 6. The (*exchange*) rule allows us to reorder variables (which is implicit, when assumptions are treated as set); (*weakening*) makes it possible to discard an assumption – this has the implication that a variable may be declared but never used. Finally, (*contraction*) makes it possible to use a single variable multiple times (by joining multiple variables into a single one using substitution).

In sub-structural type systems, the assumptions are typically treated as a list. As a result, they have to be manipulated explicitly. Different systems allow different subset of the rules. For example, *affine* systems allows exchange and weakening, leading to a system where variable may be used at most once; in *linear* systems, only exchange is permitted and so every variable has to be used exactly once.

When tracking context-dependent properties associated with individual variables, we need to be more explicit in how variables are used. Sub-structural type systems provide a way to do this. Even when we allow all three operations, we can track which variables are used and how (and use that to track additional contextual information about variables).

BUNCHED TYPE SYSTEMS Bunched typing makes one more refinement to how Γ is treated. Rather than having a list of assumptions, the context becomes a tree that contains variable typings (or special identity values) in the leaves and has multiple different types of nodes. The context can be defined, for example, as follows:

$$\Gamma, \Delta, \Sigma := x : \alpha \mid I \mid \Gamma, \Gamma \mid 1 \mid \Gamma; \Gamma$$

The values I and 1 represent two kinds of "empty" contexts. More interestingly, non-empty variable contexts may be constructed using two distinct constructors – Γ, Γ and $\Gamma; \Gamma$ – that have different properties. In particular, weakening and contraction is only allowed for the ; constructor, while exchange is allowed for both.

The structural rules for bunched typing are shown in Figure 7. The syntax $\Gamma(\Delta)$ is used to mean an assumption tree that contains Δ as a sub-tree and so, for example, (*exchange1*) can switch the order of contexts anywhere in the tree. The remaining rules are similar to the rules of linear logic.

One important note about bunched typing is that it requires a different interpretation. The omission of weakening and contraction in linear logic means that variable can be used exactly once. In bunched typing, variables may still be duplicated, but only using the ";" separator. The type system can be interpreted as specifying whether a variable may be shared between the

$$\text{(exchange1)}\frac{\Gamma(\Delta,\Sigma)\vdash e:\alpha}{\Gamma(\Sigma,\Delta)\vdash e:\alpha} \qquad \text{(weakening)}\frac{\Gamma(\Delta)\vdash e:\alpha}{\Gamma(\Delta;\Sigma)\vdash e:\alpha}$$

$$\text{(exchange2)}\frac{\Gamma(\Delta;\Sigma)\vdash e:\alpha}{\Gamma(\Sigma;\Delta)\vdash e:\alpha} \qquad \text{(contraction)}\frac{\Gamma(\Delta;\Sigma)\vdash e:\alpha}{\Gamma(\Delta)\vdash e[\Sigma\leftarrow\Delta]:\alpha}$$

Figure 7: Exchange, weakening and contraction rules for bunched typing

body of a function and the context where a function is declared. The system introduces two distinct function types $\alpha \to \beta$ and $\alpha \twoheadrightarrow \beta$ (corresponding to ";" and "," respectively). The key property is that only the first kind of functions can share variables with the context where a function is declared, while the second restricts such sharing. We do not attempt to give a detailed description here as it is not immediately to coeffects – for more information, refer to O'Hearn's introduction [**?**].

THESIS PERSPECTIVE Our work can be viewed as annotating bunches. Such annotations then specify additional information about the context – or, more specifically, about the sub-tree of the context. Although this is not the exact definition used in Chapter X, we could define contexts as follows:

$$\Gamma,\Delta,\Sigma := x:\alpha \mid 1 \mid \Gamma,\Gamma \mid \Gamma@\sigma$$

Now we can not only annotate an entire context with some information (as in the simple coeffect system for tracking resources that used judgements of a form $\Gamma@\sigma \vdash e : \alpha$). We can also annotate individual components. For example, a context containing variables x, y, z where only x is used could be written as $(x : \alpha\,@\,\mathsf{used}), ((y : \alpha, z : \alpha)\,@\,\mathsf{unused})$.

For the purpose of this introduction, we ignore important aspects such as how are nested annotations interpreted. The main goal is to show that coeffects can be easily viewed as an extension to the work on bunched logic. Aside from this principal connection, *structural coeffects* also use some of the proof techniques from the work on bunched logics, because they also use tree-like structure of variable contexts.

2.4 SUMMARY

This chapter presented four different pathways leading to the idea of coeffects. We also introduced the most important related work, although presenting related work was not the main goal of the chapter. The main goal was to show the idea of coeffects as a logical follow up to a number of research directions. For this reason, we highlighted only certain aspects of related work – the remaining aspects as well as important technical details are covered in later chapters.

The first pathway looks at applications and systems that involve notion of *context*. The two coeffect calculi we present aim to unify some of these systems. The second pathway follows as a dualization of well-known effect systems. However, this is not simply a syntactic transformation, because coeffect systems treat lambda abstraction differently. The third pathway follows by extending comonadic semantics of context-dependent computations with indexing and building a type system analogous to effect system from the "marriage of effects and monads". Finally, the fourth pathway starts with sub-structural type systems. Coeffect systems naturally arise by annotating bunches in bunched logics with additional information.

3 CONTEXT-AWARE APPLICATIONS

TODO

3.1 THROUGH APPLICATIONS

The general theme of this thesis is improving programming languages to better support writing *context-dependent* (or *context-aware*) computations. With current trends in the computing industry such as mobile and ubiquitous computing, this is becoming an important topic. In software engineering and programming community, a number of authors have addressed this problem from different perspectives. Hirschfeld et al. propose *Context-Oriented Programming* (COP) as a methodology [**?**], and the subject has also been addressed in mobile computations [**? ?**]. In programming languages, Costanza [**?**] develops a domain-specific LISP-like language ContextL and Bardram [**?**] proposes a Java framework for COP.

We approach the problem from a different perspective, building on the tradition of statically-typed functional programming languages and their theories. However, even in this field, there is a number of calculi or language features that can be viewed as context-dependent.

3.1.1 *Motivation for flat coeffects*

In a number of systems, the execution environment provides some additional data, resources or information about the execution context, but are independent of the variables used by the program. We look at implicit parameters and rebindable resources (that both provide additional identifiers that can be accessed similarly to variables, but follow different scoping rules), distributed programming, cross-compilation and data-flow.

IMPLICIT PARAMETERS In Haskell, implicit parameters [**?**] are a special kind of variables that may behave as dynamically scoped. This means, if a function uses parameter ?p, then the caller of the function must define ?p and set its value. Implicit parameters can be used to parameterise a computation (involving a chain of function calls) without passing parameters explicitly as additional arguments of all involved functions. A simple language with implicit parameters has an expression ?p to read a parameter value and an expression[1] **letdyn** $?p = e_1$ **in** e_2 that sets a parameter ?p to the value of e_1 and evaluates e_2 in a context containing ?p

An interesting question arises when we use implicit parameters in a nested function. The following function does some pre-processing and then returns a function that builds a formatted string based on two implicit parameters ?width and ?size:

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

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

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

In a purely dynamically scoped system, they would have to be defined when the user invokes the nested function. However, in Haskell, implicit parameters behave as a combination of lexical and dynamic scoping. This means that the nested function can capture the value of ?width and require just ?size In Haskell, this corresponds to the following type:

$$(\text{?width} :: \text{Int}) \Rightarrow \text{String} \rightarrow ((\text{?size} :: \text{Int}) \Rightarrow \text{String} \rightarrow \text{string})$$

As a result, the function can be called as follows:

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

This way of assigning type to format and calling it is not the only possible, though. We could also say that the outer function requires both of the implicit parameters and the result is a (pure) function with no context requirements. This interaction between implicit parameters and lambda abstraction demonstrates one of the key aspects of coeffects and will be discussed later. Implicit parameters will also sever as one of our examples in Chapter Y.

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

```
twoTimes :: Num α ⇒ α → α
twoTimes x = x + x
```

The constraint Num a on the function type arises from the use of the + operator. From the implementation perspective, the type class constraint means that the function takes a hidden parameter – a dictionary that provides the operation $+ :: \alpha \rightarrow \alpha \rightarrow \alpha$. Thus, the type $\text{Num}\ \alpha \Rightarrow \alpha \rightarrow \alpha$ can be viewed as $(\text{Num}_\alpha \times \alpha) \rightarrow \alpha$. Implicit parameters work in exactly the same way – they are passed around as hidden parameters.

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

REBINDABLE RESOURCES The need for parameters that do not strictly follow static scoping rules also arises in distributed computing. This problem has been addressed, for example, by Bierman et al. and Sewell et al. [**? ?**]. To quote the first work: *"Dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources."*

This situation arises when marshalling and transferring function values. A function may depend on a local resource (e.g. a database available only on the server) and also resources that are available on the target node (e.g. current time). In the following example, the construct **access** Res represents access to a re-bindable resource named Res:

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

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

The use of re-bindable resources creates a context requirement similar to the one arising from the use of implicit parameters. For function values, such context-requirements can be satisfied in different ways – resources must be available either at the declaration site (i.e. when a function is created) or at the call site (i.e. when a function is called).

DISTRIBUTED COMPUTING AND MULTI-TARGETTING An increasing number of programming languages is capable of running across multiple different platforms or execution environments. Functional programming languages that can be compiled to JavaScript (to target web and mobile clients) include, among others, F#, Haskell and OCaml [?].

Links [1], F# libraries [? ?], ML5 and QWeSST [? ?] and Hop [7] go further and allow a single source program to be compiled to multiple target runtimes. This posses additional challenges – it is necessary to track where each part of computation runs and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

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

```
let validateInput = λname →
  name ≠ """" && forall isLetter name

let displayProduct = λname →
  if validateInput name then displayProductPage name
  else displayErrorPage ()
```

The function validateInput can be compiled to both JavaScript (for client-side) and native code (for server-side). However, displayProduct uses other functionality (generating web pages) that is only available on the server-side, so it can only be compiled to native code.

In Links [1], functions can be annotated as client-side, server-side and database-side. F# WebTools [?] adds functions that support multiple targets (mixed-side). However, these are single-purpose language features and they are not extensible. For example, in modern mobile development it is also important to track minimal supported version of runtime[2].

2 Android Developer guide [?] demonstrates how difficult it is to solve the problem without language support.

Requirements on the execution environment can be viewed as contextual properties, but could be also presented as effects (use of some API required only in certain environment is a computational effect). We discuss the difference in Section X. Furthermore, the theoretical foundations of distributed languages like ML5 [?] suggest that a contextual treatment is more appropriate. We return to ML5 when discussing semantics in Section 2.2.3.

SAFE LOCKING In the previous examples, the context provides additional values or functions that may be accessed at runtime. However, it may also track *permissions* to perform some operation. This is done in the type system for safe locking of Flanagan and Abadi [?].

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

$$\begin{aligned}&\textbf{newlock}\ l : \rho\ \textbf{in}\\&\textbf{let}\ \mathsf{state}\ =\ \mathsf{ref}_\rho\ 10\ \textbf{in}\\&\textbf{sync}\ l\ (!\mathsf{state})\end{aligned}$$

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

The type system for safe locking associates the list of permission with the variable context. It uses judgements of a form $\Gamma, m \vdash e : \alpha$ specifying that an expression has a type in context Γ, given permissions (a list of locked regions) m. However, the treatment of lambda abstraction differs from the one for implicit parameters or rebindable resources. In the system for locking, code inside lambda function cannot use permissions from the scope where the function is declared. This is a necessary requirement – a lambda function created under a lock cannot access protected memory, because it will be executed later. We discuss how this restriction fits into our general coeffect framework in Section X.Y.

DATA-FLOW LANGUAGES The examples discussed so far are all – to some extent – similar. They attach additional information (implicit parameters, dictionaries) or restrictions (on execution environment) to the context where code evaluates. By *context*, we mean, most importantly, the values of variables and declarations that are in scope. The examples so far add more information to the context, but do not operate on the variable values.

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

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

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

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

The construct **prev** x returns a stream consisting of previous values of the stream x. The second value of **prev** x is first value of x (and the first value is undefined). The construct y **fby** x returns a stream whose first element is the first element of y and the remaining elements are values of x. Note that in Lucid, the constants such as false and 0 are constant streams. Formally, the construct are defined as follows (writing x_n for n-th element of a stream x):

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

When reading data-flow programs, we do not need to think about variables in terms of streams – we can see them as simple values. However, the operations **fby** and **prev** cannot operate on plain values – they require additional *context* which provides past values of variables (for **prev**) and information about the current location in the stream (for **fby**).

In this case, the context is not simply an additional (hidden) parameter. It completely changes how variables must be represented. We may want to capture various *contextual properties* of Lucid programs. For example, how many past elements need to be cached when we evaluate the stream.

To understand the nature of the context, we later look at the semantics of Lucid. This can be captured using a number of mathematical structures. Wadge [?] originally proposed to use monads, while Uustalu and Vene later used comonads [?].

3.1.2 *Motivation for structural coeffects*

We now turn our attention to system where additional contextual information are associated not with the context as a whole (or program scope), but with individual variables. We start by looking simple static analysis – variable *liveness*. Then we revisit data-flow computations and look at applications in security and software updating.

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

In this thesis, we first use a restricted (and not practically useful) form of liveness analysis to introduce the theory of indexed comonads (Section X) and then use liveness analysis as one of the motivations for structural coeffects. Consider the following two simple functions:

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

In liveness analysis, we annotate the context with a value specifying whether the variables in scope are *live* or *dead*. If we associate just a single value with the entire context, then the liveness analysis is very limited – it can say that the context of the expression 42 in the first function is dead, because no variables are accessed.

A useful liveness analysis needs to consider individual variables. For example, in the body of the second function (value), two variables are in scope. The variable value is accessed and thus is *live*, but the variable x is dead.

Static analyses can be classified as either *forward* or *backward* (depending on how they propagate information) and as either *must* or *may* (depending on what properties they guarantee). Liveness is a *backward* analysis – this means that the requirements propagates from variables to their declaration sites. The distinction between *must* and *may* is apparent when we look at an example with conditionals:

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

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

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

DATA-FLOW LANGUAGES (REVISITED) When discussing data-flow languages in the previous section, we said that the context provides past values of variables. This can be viewed as a flat contextual property (the context needs to keep all past values), but we can also view it as a structural property. Consider the following example:

```
let offsetZip = 0 fby (left + prev right)
```

The value offsetZip adds values of left with previous values of right. To evaluate a current value of the stream, we need the current value of left and one past value of right.

As mentioned earlier, a static analysis for data-flow computations could calculate how many past values must be cached. This can be done as a *flat* coeffect analysis that produces just a single number for each function. However, we can design a more precise *structural* analysis and track the number of required elements for individual variables.

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

Statically typed systems that based on tainting have been use to prevent cross-site scripting attacks [**?**] and a well known attack known as SQL injection [4**?**]. In the latter chase, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables id and msg:

```
let name = query (""SELECT Name WHERE Id = "" + id) in
msg + name
```

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

In runtime checking, all (string) values need to be wrapped in an object that stores Boolean flag (for tainting) or more complex data (for provenance). In static checking, the information need to be associated with the variables in the variable context. We use tainting as a motivating example for *structural* coeffects in Section X.

security and core dependency calculus The checking of tainting is a special case of checking of the *non-interference* property in *secure information flow*. Here, the aim is to guarantee that sensitive information (such as credit card number) cannot be leaked to contexts with low secrecy (e.g. sent via an unsecured network channel). Volpano et al. [**?**] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [**?**] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the Flow-Caml [**?**] language. Finally, Russo et al. and Swamy et al. [**? ?**] show that the properties can be checked using a monadic library.

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

An important aspect of secure information flow is called *implicit flows*. Consider the following example which may assign a new value to z:

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

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

Abadi et al. realized that there is a number of analyses similar to secure information flow and proposed to unify them using a single model called Dependency Core Calculus (DCC) [**?**]. It captures other cases where some information about expression relies on properties of variables in the context where it executes. The DCC captures, for example, *binding time analysis* [**?**], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [**?**] that identifies parts of programs that contribute to the output of an expression.

3.1.3 *Beyond passive contexts*

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

However, there is a number of systems where the context may be changed – not just be evaluating certain code block in a different scope (e.g. by wrapping it in prev in data-flow), but also by calling a function that, for example, acquires new capabilities. While this thesis focuses on systems with passive context, we quickly look at the most important examples of the *active* variant.

CALCULUS OF CAPABILITIES Crary et al. [**?**] introduced the Calculus of Capabilities to provide a sound system with region-based memory management for low-level code that can be easily compiled to assembly language. They build on the work of Tofte and Talpin [**?**] who developed an *effect system* (discussed in Section 2.2.2) that uses lexically scoped *memory regions* to provide an efficient and controlled memory management.

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

```
let calculate = λinput →
  letrgn ρ in
  let x = ref_ρ input in !x
```

The memory region ρ is a part of the context, but only in the scope of the body of **letrgn**. It is only available to the last line which allocates a memory cell in the region and reads it (before the region is deallocated). There is no way to allocate a region inside a function and pass it back to the caller.

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

```
let calculate = λinput →
  letrgn ρ in
  let x = ref_ρ input in x
```

The example is almost identical to the previous one, except that it does not return the value of reference x. Instead, it returns the reference, which is located in a newly allocated region. Together with the value, the function returns a *capability* to access the region ρ.

This is where systems with active context differ. To type check such programs, we do not only need to know what context is required to call calculate. We also need to know what effects it has on the context when it evaluates and the current context meeds to be appropriately adjusted after a function call. We briefly consider this problem in Section X.

SOFTWARE UPDATING Dynamic software updating (DSU) [**? ?**] is the ability to update programs at runtime without stopping them. The Proteus system developed by Stoyle et al. [**?**] investigates what language support is needed to enable safe dynamic software updating in C-like languages. The system is based on the idea of capabilities.

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

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

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

THESIS PERSPECTIVE As demonstrated in this section, there is a huge number of systems and applications that exhibit a form of context-dependence. The range includes different static analyses (liveness, provenance), well-known programming language features (implicit parameters and type classes) as well as features not widely available (e.g. for distributed programming).

It is impossible to cover all of these topics in a single coherent thesis and so we focus on two key aspects:

- **Flat vs. structural.** We look at both flat coeffects (single value for entire context) and structural coeffects (single value per variable). We use liveness, implicit parameters and data-flow to introduce flat coeffects (Section X) and liveness, refined data-flow and tainting to talk about structural coeffects (Section Y).
- **Analysis vs. restriction.** Some of the discussed examples can be viewed as static analyses that obtain some information about programs (i.e. the number of required past values in data-flow). Other examples provide type system that rules out certain invalid programs (e.g. safe locking). We cover this topic when discussing *partial coeffects* in Section Z.
- **May vs. must analysis.** When discussing liveness, we observed that we can obtain two different analyses depending on how conditionals are treated. We discuss this topic in Section X.

Although we also looked at examples of *active* contextual computations (where developers can write functions that modify the context), we do not consider these applications, to keep the material presented in this thesis focused. We briefly discuss them as future work in Section X.

3.2 SUMMARY

TODO

4 FLAT COEFFECT LANGUAGE

TODO: Update the introduction & edit the whole chapter – currently, this is just a direct copy from the accepted ICALP paper on flat coeffects.

TODO: Other extensions – add more examples of indexed comonoads, consider recursion and fixed-point operator, possibly also consider partially defined operations, which lets us express type systems that rule out some computations (rather than just static analyses).

Understanding how programs affect their environment is a well studied area: *effect systems* [**?**] provide a static analysis of effects and *monads* [9] provide a unified semantics to different notions of effect. Wadler and Thiemann unify the two approaches [15], *indexing* a monad with effect information, and showing that the propagation of effects in an effect system matches the semantic propagation of effects in the monadic approach.

No such unified mechanism exists for tracking the context requirements. We use the term *coeffect* for such contextual program properties. Notions of context have been previously captured using comonads [14] (the dual of monads) and by languages derived from modal logic [**? ?**], but these approaches do not capture many useful examples which motivate our work. We build mainly on the former comonadic direction (§4.3) and discuss the modal logic approach later (§4.5).

We extend a simply typed lambda calculus with a coeffect system based on comonads, replicating the successful approach of effect systems and monads.

EXAMPLES OF COEFFECTS. We present three examples that do not fit the traditional approach of *effect systems* and have not been considered using the *modal logic* perspective, but can be captured as *coeffect systems* (§4.1) – the tracking of implicit dynamically-scoped parameters (or resources), analysis of variable liveness, and tracking the number of required past values in dataflow computations.

COEFFECT CALCULUS. Informed by the examples, we identify a general algebraic structure for coeffects. From this, we define a general *coeffect calculus* that unifies the motivating examples (§4.2) and discuss its syntactic properties (§4.4).

INDEXED COMONADS. Our categorical semantics (§4.3) extends the work of Uustalu and Vene [14]. By adding annotations, we generalize comonads to *indexed comonads*, which capture notions of computation not captured by ordinary comonads.

4.1 MOTIVATION

Effect systems, introduced by Gifford and Lucassen [2], track *effects* of computations, such as memory access or message-based communication [**?**]. Their approach augments typing judgments with effect information: $\Gamma \vdash e : \tau, F$. Wadler and Thiemann explain how this shapes effect analysis of lambda abstraction [15]:

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

In contrast to the static analysis of *effects*, the analysis of *context-dependence* does not match this pattern. In the systems we consider, lambda abstraction places requirements on both the *call-site* (latent requirements) and the *declaration-site* (immediate requirements), resulting in different syntactic properties. We informally discuss three examples first that demonstrate how contextual requirements propagate. Section (§4.2) then unifies these in a single calculus.

We write coeffect judgements $C^s\Gamma \vdash e : \tau$ where the coeffect annotation s associates context requirements with the free-variable context Γ. Function types have the form $C^s\tau_1 \rightarrow \tau_2$ associating *latent* coeffects s with the parameter. The $C^s\Gamma$ syntax and $C^s\tau$ types are a result of the indexed comonadic semantics (§4.3).

4.1.1 *Implicit parameters and resources.*

Implicit parameters [**?**] are *dynamically-scoped* variables. They can be used to parameterize a computation without propagating arguments explicitly through a chain of calls and are part of the context in which expressions evaluate. As correctly expected [**?**], they can be modelled by comonads. Rebindable resources in distributed computations follow a similar pattern, but we discuss implicit parameters for simplicity.

The following function prints a number using implicit parameters ?culture (determining the decimal mark) and ?format (the number of decimal places):

$$\lambda n.\mathsf{printNumber}\ n\ \mathsf{?culture}\ \mathsf{?format}$$

Figure 8 shows a type-and-coeffect system tracking the set of an expression's implicit parameters. For simplicity here, all implicit parameters have type ρ.

Context requirements are created in (*access*), while (*var*) requires no implicit parameters; (*app*) combines requirements of both sub-expressions as well as the latent requirements of the function. The (*abs*) rule is where the example differs from effect systems. Function bodies can access the union of the parameters (or resources) available at the declaration-site ($C^r\Gamma$) and at the call-site ($C^s\tau_1$). Two of the nine permissible judgements for the above example are:

$$\begin{aligned} C^{\emptyset}\Gamma &\vdash (\ldots) : C^{\{\mathsf{?culture},\mathsf{?format}\}}\mathsf{int} \rightarrow \mathsf{string} \\ C^{\{\mathsf{?culture},\mathsf{?format}\}}\Gamma &\vdash (\ldots) : C^{\{\mathsf{?format}\}}\mathsf{int} \rightarrow \mathsf{string} \end{aligned}$$

The coeffect system infers multiple, i.e. non-principal, coeffects for functions. Different judgments are desirable depending on how a function is used. In the first case, both parameters have to be provided by the caller. In the second, both are available at declaration-site, but ?format may be re-

$$(\textit{var})\ \frac{x : \tau \in \Gamma}{C^{\emptyset}\Gamma \vdash x : \tau} \qquad (\textit{app})\ \frac{C^{r}\Gamma \vdash e_1 : C^{t}\tau_1 \to \tau_2 \qquad C^{s}\Gamma \vdash e_2 : \tau_1}{C^{r \cup s \cup t}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\textit{access})\ \frac{}{C^{\{?a\}}\Gamma \vdash ?a : \rho} \qquad (\textit{abs})\ \frac{C^{r \cup s}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^{r}\Gamma \vdash \lambda x.e : C^{s}\tau_1 \to \tau_2}$$

Figure 8: Selected coeffect rules for implicit parameters

bound (precise meaning is provided by the monoidal structure on the product comonad in §4.3).

Implicit parameters can be captured by the *reader* monad, where parameters are associated with the function codomain $M^{\emptyset}(\mathsf{int} \to M^{\{?\mathsf{culture},?\mathsf{format}\}}\mathsf{string})$, modelling only the first case. Whilst the reader monad can be extended to model rebinding, the next example cannot be structured by *any* monad.

4.1.2 *Liveness analysis.*

Liveness analysis detects whether a free variable of an expression may be used (*live*) or whether it is definitely not needed (*dead*). A compiler can remove bindings to dead variables as the result is never used.

We start with a restricted analysis and briefly mention how to make it practical later (§4.5). The restricted form is interesting theoretically as it gives rise to the indexed Maybe comonad (§4.3), which is a basic but instructive example.

A coeffect system in Fig. 9 detects whether all variables are dead ($C^{D}\Gamma$) or whether at least one variable is live ($C^{L}\Gamma$). Variable access (*var*) is annotated with L and constant access with D. That is, if $c \in \mathbb{N}$ then $C^{D}\Gamma \vdash c : \mathsf{int}$. A dead context may be marked as live by letting $D \sqsubseteq L$ and adding subcoeffecting (§4.2).

The (*app*) rule is best understood by discussing its semantics. Consider first *sequential composition* of (semantic) functions g, f annotated with r, s. The argument of $g \circ f$ is live only when arguments of both f and g are live. The coeffect semantics captures the additional behaviour that f is not evaluated when g ignores its input (regardless of the evaluation order of the underlying language). We write $r \sqcap s$ for a conjunction (returning L iff $r = s = L$). Secondly, a *pointwise composition* passes the same argument to g and h. The parameter is live if either the parameter of g or h is live ($r \sqcup s$). Application combines the two operations, so the context Γ is live if it is needed by e_1 *or* by the function value *and* by e_2.

An (*abs*) rule (not shown) compatible with the structure in Fig. 8 combines the context annotations using $\sqcap$. Thus, if the body uses some variables, both the function argument and the context of the declaration-site are marked as live.

Liveness cannot be modelled using monads as $\tau_1 \to M^{r}\tau_2$. In call-by-value languages, the argument τ_1 is always evaluated. Using indexed comonads (§4.3), we model liveness as $C^{r}\tau_1 \to \tau_2$ where C^{r} is the parametric type $\mathsf{Maybe}\ \tau = \tau + 1$ (which contains a value τ when $r = L$ and does not contain value when $r = D$).

4.1.3 *Efficient dataflow.*

Dataflow languages (e. g. [**?**]) declaratively describe computations over streams. In *causal* data flow, program may access past values – in this setting, a func-

$$(\textit{var})\ \frac{x : \tau \in \Gamma}{C^{L}\Gamma \vdash x : \tau} \qquad (\textit{app})\ \frac{C^{s}\Gamma \vdash e_2 : \tau_1 \qquad C^{r}\Gamma \vdash e_1 : C^{t}\tau_1 \to \tau_2}{C^{r \sqcup (s \sqcap t)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

Figure 9: Selected coeffect rules for liveness analysis

$$(var)\ \frac{x:\tau \in \Gamma}{C^{0}\Gamma \vdash x:\tau} \qquad (app)\ \frac{C^{m}\Gamma \vdash e_1 : C^{p}\tau_1 \rightarrow \tau_2 \qquad C^{n}\Gamma \vdash e_2 : \tau_1}{C^{max(m,n+p)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(prev)\ \frac{C^{n}\Gamma \vdash e:\tau}{C^{n+1}\Gamma \vdash \textbf{prev}\ e:\tau} \qquad (abs)\ \frac{C^{min(m,n)}(\Gamma, x:\tau_1) \vdash e:\tau_2}{C^{m}\Gamma \vdash \lambda x.e : C^{n}\tau_1 \rightarrow \tau_2}$$

Figure 10: Selected coeffect rules for causal data flow

tion $\tau_1 \rightarrow \tau_2$ becomes a function from a list of historical values $[\tau_1] \rightarrow \tau_2$. A coeffect system here tracks how many past values to cache.

Figure 10 annotates contexts with an integer specifying the maximum number of required past values. The current value is always present, so (*var*) is annotated with 0. The expression **prev** e gets the previous value of stream e and requires one additional past value (*prev*); e. g. **prev** (**prev** e) requires 2 past values.

The (*app*) rule follows the same intuition as for liveness. Sequential composition adds the tags (the first function needs $n+p$ past values to produce p past inputs for the second function); passing the context to two subcomputations requires the maximum number of the elements required by the two subcomputations. The (*abs*) rule for data-flow needs a distinct operator – *min* – therefore, the declaration-site and call-site must each provide at least the number of past values required by the function body (the body may use variables coming from the declaration-site as well as the argument).

The soundness follows from our categorical model (§4.3). Uustalu and Vene [14] model causal dataflow computations using a non-empty list comonad $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$. However, such model leads to (inefficient) unbounded lists of past elements. The above static analysis provides an approximation of the number of required past elements and so we use just fixed-length lists.

4.2 GENERALIZED COEFFECT CALCULUS

The previous three examples exhibit a number of commonalities. We capture these in the *coeffect calculus*. We do not overly restrict the calculus to make it open for notions of context-dependent computations not discussed above.

The syntax of our calculus is that of the simply-typed lambda calculus (where v ranges over variables, T over base types, and r over coeffect annotations):

$$e ::= v \mid \lambda v.e \mid e_1\ e_2 \qquad\qquad \tau ::= T \mid \tau_1 \rightarrow \tau_2 \mid C^{r}\tau$$

The type $C^{r}\tau$ captures values of type τ in a context specified by the annotation r. This type appears only on the left-hand side of a function arrow $C^{r}\tau_1 \rightarrow \tau_2$. In the semantics, C^{r} corresponds to some data type (e. g. list or Maybe). Extensions such as explicit *let*-binding are discussed later (§4.4).

The coeffect tags r, that were demonstrated in the previous section, can be generalized to a structure with three binary operators and a particular element.

Definition 3. *A* coeffect algebra $(S, \oplus, \vee, \wedge, e)$ *is a set* S *with an element* $e \in S$*, a semi-lattice* $(S, \vee)$*, a monoid* $(S, \oplus, e)$*, and a binary* $\wedge$*. That is,* $\forall r, s, t \in S$*:*

$$r \oplus (s \oplus t) = (r \oplus s) \oplus t \qquad e \oplus r = r = r \oplus e \qquad \text{(monoid)}$$

$$r \vee s = s \vee r \qquad r \vee (s \vee t) = (r \vee s) \vee t \qquad r \vee r = r \qquad \text{(semi-lattice)}$$

The generalized coeffect calculus captures the three motivating examples (§4.1), where some operators of the coeffect algebra may coincide.

$$(var)\ \frac{x : \tau \in \Gamma}{C^{e}\Gamma \vdash x : \tau} \qquad (app)\ \frac{C^{r}\Gamma \vdash e_1 : C^{s}\tau_1 \rightarrow \tau_2 \qquad C^{t}\Gamma \vdash e_2 : \tau_1}{C^{r \vee (s \oplus t)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(sub)\ \frac{C^{s}\Gamma \vdash e : \tau}{C^{r}\Gamma \vdash e : \tau}\ (s \leqslant r) \qquad (abs)\ \frac{C^{r \wedge s}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^{r}\Gamma \vdash \lambda x.e : C^{s}\tau_1 \rightarrow \tau_2}$$

Figure 11: Type and coeffect system for the coeffect calculus

The $\oplus$ operator represents *sequential* composition; guided by the categorical model (§4.3), we require it to form a monoid with e. The operator $\vee$ corresponds to merging of context-requirements in *pointwise composition* and the semi-lattice $(S, \vee)$ defines a partial order: $r \leqslant s$ when $r \vee s = s$. This ordering implies a sub-coeffecting rule. The coeffect e is often the top or bottom of the lattice.

The $\wedge$ operator corresponds to splitting requirements of a function body. The operator is unrestricted in the general system. A number of additional laws holds for *some* coeffects systems, e. g. semi-lattice structure of $\wedge$ and a form of distributivity. Quite possibly, they should hold for all coeffect systems. We start with as few laws as possible so as not to limit possible uses of the calculus. We consider constrained variants that provide useful syntactic properties later (§4.4).

IMPLICIT PARAMETERS use sets of names $S = \mathcal{P}(\mathsf{Id})$ as tags with union $\cup$ for all three operators. Variable access is annotated with $e = \emptyset$ and $\leqslant$ is subset ordering.

LIVENESS uses a two point lattice $S = \{D, L\}$ where $D \sqsubseteq L$. Variables are annotated with the top element $e = L$ and constants with bottom D. The $\vee$ operation is $\sqcup$ (join) and $\wedge$ and $\oplus$ are both $\sqcap$ (meet).

DATAFLOW tags are natural numbers $S = \mathbb{N}$ and operations $\vee, \wedge$ and $\oplus$ correspond to *max*, *min* and $+$, respectively. Variable access is annotated with $e = 0$ and the order $\leqslant$ is the standard ordering of natural numbers.

4.2.1 *Coeffect typing rules.*

Figure 11 shows the rules of the coeffect calculus, given some coeffect algebra $(S, \oplus, \vee, \wedge, e)$. The context required by a variable access (*var*) is annotated with e. The sub-coeffecting rule (*sub*) allows the contextual requirements of an expression to be generalized.

The (*abs*) rule checks the body of the function in a context $r \wedge s$, which is a combination of the coeffects available in the context r where the function is defined and in a context provided by the caller of the function. Note that none of the judgements create a *value* of type $C^{r}\tau$. This type appears only immediately to the left of an arrow $C^{r}\tau_1 \rightarrow \tau_2$.

In function application (*app*), context requirements of both expressions and the function are combined as previously: the pointwise composition $\vee$ is used to combine the coeffects of the expression representing a function r and the coeffects of the argument, sequentially composed with the coeffects of the function $s \oplus t$.

For space reasons, we omit recursion. We note that this would require adding effect variables and extending the coeffect algebra with a fixed point operation.

4.3 COEFFECT SEMANTICS USING INDEXED COMONADS

The approach of *categorical semantics* interprets terms as morphisms in some category. For typed calculi, typing judgments $x_1 : \tau_1 \ldots x_n : \tau_n \vdash e : \tau$ are usually mapped to morphisms $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau]\!]$. Moggi showed the semantics of various effectful computations can be captured generally using the (*strong*) *monad* structure [9]. Dually, Uustalu and Vene showed that (*monoidal*) *comonads* capture various kinds of context-dependent computation [14].

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

INDEXED COMONADS. Uustalu and Vene's approach interprets well-typed terms as morphisms $C(\tau_1 \times \ldots \times \tau_n) \to \tau$, where C encodes contexts and has a comonad structure [14]. Indexed comonads comprise a *family* of object mappings C^r indexed by a coeffect r describing the contextual requirements satisfied by the encoded context. We interpret judgments $C^r(x_1 : \tau_1, \ldots, x_n : \tau_n) \vdash e : \tau$ as morphisms $C^r([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$.

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

Definition 4. *Given a monoid* $(S, \oplus, e)$ *with binary operator* $\oplus$ *and unit* e*, an* indexed comonad *over a category* $\mathcal{C}$ *comprises a family of object mappings* C^r *where for all* $r \in S$ *and* $A \in \text{obj}(\mathcal{C})$ *then* $C^r A \in \text{obj}(\mathcal{C})$ *and:*

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

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

$$\varepsilon \circ f^\dagger_{r,e} = f \qquad (\varepsilon)^\dagger_{e,r} = \text{id} \qquad (g \circ f^\dagger_{r,s})^\dagger_{(r \oplus s),t} = g^\dagger_{s,t} \circ f^\dagger_{r,(s \oplus t)}$$

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

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

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

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

EXAMPLE The *indexed partiality comonad* encodes free-variable contexts of a computation which are either *live* or *dead* (i. e., have *liveness* coeffects) with the monoid $(\{D, L\}, \sqcap, L)$, where $C^L A = A$ encodes live contexts and $C^D A = 1$ encodes dead contexts, where 1 is the unit type inhabited by a single value

$()$. The *counit* operation $\varepsilon : C^{\mathsf{L}}A \to A$ is defined $\varepsilon\ x = x$ and *coextend*, for all $f : C^{r}A \to B$, and thus $f^{\dagger}_{r,s} :\ C^{r \sqcap s}A \to C^{s}B$, is defined:

$$f^{\dagger}_{\mathsf{D},\mathsf{D}}x = () \qquad f^{\dagger}_{\mathsf{D},\mathsf{L}}x = f() \qquad f^{\dagger}_{\mathsf{L},\mathsf{D}}x = () \qquad f^{\dagger}_{\mathsf{L},\mathsf{L}}x = f\ x$$

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

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

4.3.1 *Monoidal indexed comonads.*

Indexed comonads provide a semantics to sequential composition, but additional structure is needed for the semantics of the full coeffect calculus. Uustalu and Vene [14] additionally require a (*lax semi-*) *monoidal comonad* structure, which provides a monoidal operation $\mathsf{m} : CA \times CB \to C(A \times B)$ for merging contexts (used in the semantics of abstraction).

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

- $\mathsf{m}_{r,s} : C^{r}A \times C^{s}B \to C^{(r \wedge s)}(A \times B)$ natural in A, B;
- $\mathsf{n}_{r,s} : C^{(r \vee s)}(A \times B) \to C^{r}A \times C^{s}B$ natural in A, B;

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

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

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

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

$$\begin{array}{lll} \mathsf{m}_{\mathsf{L},\mathsf{L}}(x,y) = (x,y) & \mathsf{n}_{\mathsf{D},\mathsf{D}}\ \ () \ \ = ((),()) & \mathsf{n}_{\mathsf{D},\mathsf{L}}(x,y) = ((),y) \\ \mathsf{m}_{r,s}\ \ (x,y) = () & \mathsf{n}_{\mathsf{L},\mathsf{D}}(x,y) = (x,()) & \mathsf{n}_{\mathsf{L},\mathsf{L}}(x,y) = (x,y) \end{array}$$

EXAMPLE Uustalu and Vene model causal dataflow computations using the non-empty list comonad $\mathsf{NEList}\ A = A \times (1 + \mathsf{NEList}\ A)$ [14]. Whilst this comonad implies a trivial indexed comonad, we define an indexed comonad with integer indices for the number of past values demanded of the context.

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

Figure 12: Categorical semantics for the coeffect calculus

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

4.3.2 *Categorical Semantics.*

Figure 20 shows the categorical semantics of the coeffect calculus using additional operations π_i for projection of the i^{th} element of a product, usual *curry* and *uncurry* operations, and $\Delta : A \to A \times A$ duplicating a value. While C^r is a family of object mappings, it is promoted to a family of functors with the derived morphism mapping $C^r(f) = (f \circ \varepsilon)^{\dagger}_{e,r}$.

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

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

4.4 SYNTAX-BASED EQUATIONAL THEORY

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

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

The call-by-name evaluation strategy modelled by β-reduction is not suitable for impure calculi therefore a restricted β rule, corresponding to call-by-value, is used, i. e. $(\lambda x.e_2)v \equiv e_2[x \leftarrow v]$. Such reduction can be encoded by a *let*-binding term, **let** $x = e_1$ **in** e_2, which corresponds to sequential composition of two computations, where the resulting pure value of e_1 is substituted into e_2 [**?** 9].

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

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

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

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

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

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

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

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

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

Proof. By induction over $\vdash$, using the laws (§4.2) and additional assumptions. □

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

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

Proof. A direct consequence of Lemma 1. □

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

However, the coeffect calculus also captures context-dependence in languages with more complex evaluation strategies than *call-by-name* reduction based on syntactic substitution. In particular, syntactic substitution does not provide a suitable evaluation for dataflow (because a substituted expression needs to capture the context of the original scope).

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

4.5 RELATED AND FURTHER WORK

This paper follows the approaches of effect systems [2**?** , 15] and categorical semantics based on monads and comonads [9, 14]. Syntactically, *coeffects* differ from *effects* in that they model systems where λ-abstraction may split contextual requirements between the declaration-site and call-site.

Our *indexed (monoidal) comonads* (§4.3) fill the gap between (non-indexed) *(monoidal) comonads* of Uustalu and Vene [14] and indexed monads of Atkey [**?**], Wadler and Thiemann [15]. Interestingly, *indexed* comonads are *more general* than comonads, capturing more notions of context-dependence (§4.1).

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

Pfenning et al. and Nanevski et al. derive term languages using the Curry-Howard correspondence [**? ? ?**], building a *metalanguage* (akin to Moggi's monadic metalanguage [9]) that includes $\Box$ as a type constructor. For example, in [**?**], the modal type $\Box\tau$ represents closed terms. In contrast, the *semantic* approach uses monads or comonads *only* as a semantics. This has been employed by Uustalu and Vene and (again) Moggi [9, 14]. We follow the semantic approach.

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

The work on CMTT suggests two extensions to coeffects. The first is developing the logical foundations. We briefly considered special cases of our system that permits local soundness in §4.4 and local completeness can be treated similarly. The second problem is developing the coeffects *metalanguage*. The use of coeffect algebras would provide an additional flexibility over CMTT, allowing a wider range of applications.

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

STRUCTURAL COEFFECTS. To make the liveness analysis practical, we need to associate information with individual variables (rather than the entire context). We can generalize the calculus from this paper by adding a product operation $\times$ to the coeffect algebra. A variable context $x : \tau_1, y : \tau_2, z : \tau_3$ is then annotated with $r \times s \times t$ where each component of the tag corresponds to a single variable. The system then needs to be extended with structural rules such as:

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

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

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

4.6 CONCLUSIONS

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

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

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

5 STRUCTURAL COEFFECT LANGUAGE

The *flat coeffect system* presented in the previous sections has a number of uses, but often we need to track context-dependence in a more fine-grained way. To track neededness or security, we need to associate information with individual *variables* of the context.

5.1 INTRODUCTION

5.1.1 *Motivation: Tracking array accesses*

Similarly to the flat version, the *structural coeffect calculus* works with contexts and functions annotated with a coeffectt tags, written $C^r\Gamma$ and $C^r\tau_1 \rightarrow \tau_2$, respectively, but we use richer tag structure.

As an example, consider a language that allows us to get a value of a variable (representing some changing data-source) x versions back using the syntax $\mathsf{a}_{[x]}$. To track information about individual variables, we use a product-like operation $\times$ on tags to mirrors the product structure of variables. For example:

$$C^{5\times 10}(\mathsf{a} : \mathsf{stream}, \mathsf{b} : \mathsf{stream}) \vdash \mathsf{a}_{[5]} + \mathsf{b}_{[10]} : \mathsf{nat}$$

The coeffect tag 5×10 corresponds to the free-variable context a, b, denoting that we need at most 5 and 10 past values of a and b. If we substitute c for both a and b, we need another operation to combine multiple tags associated with a single variable:

$$C^{5\vee 10}(\mathsf{c} : \mathsf{stream}) \vdash \mathsf{c}_{[5]} + \mathsf{c}_{[10]} : \mathsf{nat}$$

In this example, the operation $\vee$ would be the *max* function and so $5 \vee 10 = 10$. Before looking at the formal definition, consider the typing of let bindings:

$$\textbf{let } \mathsf{c} = \textbf{if } \mathsf{test}() \textbf{ then } \mathsf{a} \textbf{ else } \mathsf{b}$$
$$\mathsf{a}_{[15]} + \mathsf{c}_{[10]}$$

The expression has free variables a and b (we ignore test, which is not a data source). It defines c, which may be assigned either a or b. The variable a may be used directly (second line) or indirectly via c.

The expression assigned to c uses variables a and b, so its typing context is $C^{0\times 0}(\mathsf{a}, \mathsf{b})$. The value 0 is the unit of $\vee$ and it denotes empty coeffect. The typing context of the body is $C^{15\times 10}(\mathsf{a}, \mathsf{c})$.

To combine the tags, we take the coeffect associated with c and apply it to the tags of the context in which c was defined using the $\vee$ operation. This is then combined with the remaining tags from the body yielding the overall context: $C^{15\times(10\vee(0\times 0))}(\mathsf{a}, (\mathsf{a}, \mathsf{b}))$. Using a simple normalization mechanism (described later), this can be further reduced to $C^{(15\vee 10)\times 10}(\mathsf{a}, \mathsf{b})$. This gives us the required information – we need at most $max(15, 10)$ past values of a and at most 10 past values of b.

5.1.2 *Structural coeffect tags*

In the previous section, the $\vee$ operation behaves similarly to the flat $\vee$ operation. However, the type system does not require some of the semilattice properties, because some uses are replaced with the $\times$ operation. We do not require any properties about the $\times$ operation. For example, in the previous example cannot be commutative (since a tag 15×10 has different meaning than 10×15). However, we relate the operations using distributivity laws to allow normalization that was hinted above.

Definition 5. *A* structural coeffect tag structure $(S, \times, \vee, 0, 1)$ *is a tuple where* $(S, \vee)$ *is a lattice-like structure with unit* 0*. The additional structure is formed by a binary operation* $\times$ *and element* $1 \in S$ *such that for all* $r, s, t \in S$*, the following equalities hold:*

$$\begin{aligned} r \vee (s \vee t) &= (r \vee s) \vee t && \text{(associativity)}\\ r \vee s &= s \vee r && \text{(commutativity)}\\ r \vee 0 &= r && \text{(lower bound)}\\ r \vee (s \times t) &= (r \vee s) \times (r \vee t) && \text{(distributivity)}\\ 1 \vee r &= 1 && \text{(upper bound)} \end{aligned}$$

The tag 0 represents that no coeffect is associated with a variable (i.e when a variable is always accessed using standard variable access). The tag 1 is used to annotate empty variable context. For example, the context of an expression $\lambda x.x$ is empty and it needs to carry an annotation. We explain exactly how this works when we introduce the type system. The fact that 1 is the upper bound means that combining it with other coeffect annotations does not affect it and so empty contexts cannot carry any information.

The structure generalizes the *flat coeffect tag structure* introduced in Section **??**, but it additionally requires a special element 1 representing the upper bound. Given a flat coeffect structure, we can construct a structural coeffect structure (but not the other way round). For certain structures, the element 1 may be already present, but in general, it can be added as a new element. This construction will be important in Section **??**, where we show that λ_{Cs} calculus generalizes λ_{Cf}.

Lemma 2. *A flat coeffect tag structure* $(S, \vee, 0)$ *implies a structural coeffect tag structure.*

Proof. Take $1 \notin S$, then $(S \cup \{1\}, \vee, \vee, 0, 1)$ is a structural coeffect tag structure. From the properties of flat coeffect tag structure, we get that the $\vee$ operation is associative, commutative and 0 is the unit with respect to $\vee$.

To prove the distributivity, we need to show $r \vee (s \vee t) = (r \vee s) \vee (r \vee t)$. This easily follows from commutativity, associativity and idempotence of the flat coeffect tag structure. □

5.1.3 *Structural coeffect type system*

The simply typed system for λ_{Cs} uses the same syntax of types as the system for λ_{Cf}. The rules are of a form $C^r\Gamma \vdash e : \tau$ where r is a tag provided by a *structural coeffect tag structure* $(S, \times, \vee, 0)$.

The system differs from λ_{Cf} in a significant aspect. It contains explicit structural rules that manipulate with the context. Such rules allow reordering, duplicating and other manipulations with variable context. Such rules are known from affine or linear type systems where they are removed to obtain more restrictive system. In our system, the rules are present, but they manipulate the variable structure Γ as well as the associated tag structure r.

As in linear and affine systems, the variable context Γ in our system is not a simple set. Instead, we use the following tree-like structure (which is more similar to bunched types than to linear or affine systems):

$$\Gamma ::= () \mid (x : \tau) \mid (\Gamma, \Gamma)$$

The syntax $()$ represents an empty context, so the structure defines a binary trees where leaves are either variables or empty. Contexts such as $C^{1\times(r\times 1)}((), (x, ()))$ contain unnecessary number of empty contexts $()$. However, we need to construct them temporarily, because certain rules require splitting a context and, by our definition, the context $(x : \tau)$ is not splitable.

The typing rules of the system are shown in Figure **??**. Many of the structural rules are expressed in terms of a helper judgement $C^r\Gamma \Rrightarrow_c C^r\Gamma$.

STANDARD RULES. Variable access (λ_{Cs} -Tvar) annotates the corresponding variable with an empty coeffect 0. The λ_{Cs} -Tfun rule assumes that the context of the body can be split into the variable of the function and other (potentially empty) context and it attaches the coeffect associated with the function variable to the resulting function type $C^s\tau_1 \rightarrow \tau_2$.

The λ_{Cs} -Tapp rule combines coeffects s, t, r associated with the function-returning expression, argument and the function type respectively. The result of evaluating the argument in the context t is passed to the function that requires context r, so the variables used in the context Γ_2 are annotated with the combination of coeffects $r \vee t$. The variable context required to evaluate the function value is independent and so it is annotated just with coeffects s. Finally, the λ_{Cs} -Tlet rule is derived from let binding and application, but we show it separately to aid the understanding.

STRUCTURAL RULES. The remaining rules are not syntax-directed. They are embedded using the λ_{Cs} -Tctx rule and expressed using a helper judgement $C^{r_1}\Gamma_1 \Rrightarrow_c C^{r_2}\Gamma_2$ that says that the context on the left-hand side can be transformed to the context on the right-hand side. The transformation can be applied to any part of the context, which is captured using the λ_{Cs} -Tnest rule (it is sufficient to apply the transformation on the left part of the context; the right part can be transformed using λ_{Cs} -Texch).

The λ_{Cs} -Tempty rule allows attaching empty context to any existing context. The rule is needed to type-check lambda abstractions that do not capture any outer variable. The λ_{Cs} -Tweak rule is similar, but it represents *weakening* where an unused variable is added. The associated coeffect tag does not associate context information with the variable. This is needed to type-check lambda abstraction that does not use the argument.

The λ_{Cs} -Tcontr rule is a limited form of contraction. When a variable appears repeatedly, it can be reduced to a single occurrence. The rule is needed to satisfy the side condition of λ_{Cs} -Tfun when the body uses the argument repeatedly. The two associativity rules together with λ_{Cs} -Texch provide ways to rearrange variables. Finally, λ_{Cs} -Tsub represents sub-coeffecting – in λ_{Cs} the rule operates on coeffects of individual variables.

5.1.4 *Properties of reductions*

Similarly to the flat version, the λ_{Cs} calculus is defined abstractly. We cannot define its operational meaning, because that will differ for every con-

TYPING RULES $C^r\Gamma \vdash e : \alpha$

$$(var)\ \frac{}{C^e(v : \alpha) \vdash v : \alpha}$$

$$(const)\ \frac{c : \alpha \in \Phi}{C^1() \vdash c : \alpha}$$

$$(fun)\ \frac{C^{r \times s}(\Gamma, v : \alpha) \vdash e : \beta \qquad v \notin \Gamma}{C^r\Gamma \vdash \lambda v.e : C^s\alpha \to \beta}$$

$$(app)\ \frac{C^s\Gamma_1 \vdash e_1 : C^r\alpha \to \beta \qquad C^t\Gamma_2 \vdash e_2 : \alpha}{C^{s \times (r \oplus t)}(\Gamma_1, \Gamma_2) \vdash e_1\ e_2 : \beta}$$

$$(let)\ \frac{C^{r \times s}(\Gamma_1, v : \alpha) \vdash e_1 : \beta \qquad C^t\Gamma_2 \vdash e_2 : \alpha}{C^{r \times (s \oplus t)}(\Gamma_1, \Gamma_2) \vdash \textbf{let}\ x = e_2\ \textbf{in}\ e_1 : \beta}$$

$$(ctx)\ \frac{C^r\Gamma \vdash e : \alpha \qquad C^{r'}\Gamma' \Rightarrow_c C^r\Gamma}{C^{r'}\Gamma' \vdash e : \alpha}$$

CONTEXT RULES $C^r\Gamma \Rightarrow_c C^r\Gamma$

$$(nest)\ \frac{C^{r'}\Gamma_1' \Rightarrow_c C^r\Gamma_1}{C^{r' \times s}(\Gamma_1', \Gamma_2) \Rightarrow_c C^{r \times s}(\Gamma_1, \Gamma_2)}$$

$$(nest)\ C^{r \times s}(\Gamma_1, \Gamma_2) \Rightarrow_c C^{s \times r}(\Gamma_2, \Gamma_1)$$

$$(empty)\ C^{r \times 1}(\Gamma, ()) \Rightarrow_c C^r\Gamma$$

$$(weak)\ C^{r \times 0}(\Gamma, x : \alpha) \Rightarrow_c C^r\Gamma$$

$$(contr)\ C^{r+s}(x : \alpha) \Rightarrow_c C^{r \times s}(x : \alpha, x : \alpha)$$

$$(assoc)\ C^{r \times (s \times t)}(\Gamma_1, (\Gamma_2, \Gamma_3)) \Rightarrow_c C^{(r \times s) \times t}((\Gamma_1, \Gamma_2), \Gamma_3)$$

$$(sub)\ C^r\Gamma \Rightarrow_c C^s\Gamma \quad (\text{when } s \leqslant r)$$

Figure 13: Type system for the structural coeffect language λ_{sc}

crete application. For example, when tracking array accesses, variables are interpreted as arrays and $\mathsf{a}_{[n]}$ denotes access to a specified element.

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

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

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

Lemma 3 (Definition of let binding). *If* $C^r\Gamma \vdash (\lambda x.e_2)\ e_1 : \tau_2$ *then* $C^r\Gamma \vdash$ **let** $x = e_1$ **in** $e_2 : \tau_2$.

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

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

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

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

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

Assuming we have a context with hole $C^r\Gamma \in \Delta[-]$, the hole filling operation $C^r\Gamma[r'|\Gamma']$ fills the hole in the variable context with Γ' and the corresponding coeffect tag hole with r'. The operation is defined in Figure 14. Using contexts with holes, we can now formulate the general substitution lemma for λ_{Cs}.

Lemma 4 (Substitution Lemma). *If* $C^r\Gamma[R|\nu : \tau'] \vdash e : \tau$ *and* $C^S\Gamma' \vdash e' : \tau'$ *then* $C^r\Gamma[R \vee S|\Gamma'] \vdash e[\nu \leftarrow e'] : \tau$.

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

Figure 14: The definition of hole filling operation for $\Delta[-]$

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

Theorem 2 (Subject reduction). *If* $C^{r}\Gamma \vdash e_1 : \tau$ *and* $e_1 \twoheadrightarrow_{\beta} e_2$ *then* $C^{r}\Gamma \vdash e_2 : \tau$.

Proof. Direct consequence of Lemma 5 (see Appendix **??**). □

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

Theorem 3 (Local soundness). *If* $C^{r}\Gamma \vdash (\lambda x.e_2)\ e_1 : \tau$ *then* $C^{r}\Gamma \vdash e_2[x \leftarrow e_1] : \tau$.

Proof. Special case of subject reduction (Theorem 2). □

Theorem 4 (Local completeness). *If* $C^{r}\Gamma \vdash f : C^{s}\tau_1 \to \tau_2$ *then* $C^{r}\Gamma \vdash \lambda x.fx : C^{s}\tau_1 \to \tau_2$.

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

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

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

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

5.2 SEMANTICS OF STRUCTURAL COEFFECTS

The semantics of structural coeffect calculus λ_{Cs} can be defined similarly to the semantics of λ_{Cf}. The most notable difference is that the structure of coeffect tag now mirrors the structure of the variable context. Thus an expression $C^{r\times s}(\Gamma_1, \Gamma_2) \vdash e : \tau$ is modelled as a function $C^{r\times s}(\Gamma_1 \hat{\times} \Gamma_2) \to \tau$.

$$[\![C^{r_1\times\ldots\times r_n}(x_1:\tau_1\times\ldots\times x_n:\tau_n)\vdash e:\tau]\!] : C^{r_1\times\ldots\times r_n}(\tau_1\times\ldots\times\tau_n)\to\tau$$

$$\begin{aligned}
[\![C^0\Gamma\vdash x_i:\tau_i]\!] &= \epsilon_0\\
[\![C^r\Gamma\vdash \lambda x.e : C^s\tau_1\to\tau_2]\!] &= \Lambda([\![C^{r\times s}(\Gamma,x:\tau_1)\vdash e:\tau_2]\!]\circ m_{r,s})\\
[\![C^{s\times(r\vee t)}(\Gamma_1,\Gamma_2)\vdash e_1\ e_2:\tau]\!] &= (\lambda(\gamma_1,\gamma_2)\to[\![C^s\Gamma_1\vdash e_1 : C^r\tau_1\to\tau_2]\!]\ \gamma_1\ ([\![C^t\Gamma_2\vdash e_2:\tau_1]\!]^{\dagger}_{t,r}\ \gamma_2))\circ n_{s,(r\vee t}\\
[\![C^{r'}\Gamma'\vdash e:\tau]\!] &= [\![C^r\Gamma\vdash e:\tau]\!]\circ[\![C^{r'}\Gamma'\Rightarrow_c C^r\Gamma]\!]
\end{aligned}$$

$$\begin{aligned}
[\![C^{r'\times s}(\Gamma_1',\Gamma_2)\Rightarrow_c C^{r\times s}(\Gamma_1,\Gamma_2)]\!] &= m_{r,s}\circ([\![C^{r'}\Gamma_1'\Rightarrow_c C^r\Gamma_1]\!]\times \mathsf{id})\ \circ n_{r',s}\\
[\![C^{r\times s}(\Gamma_1,\Gamma_2)\Rightarrow_c C^{s\times r}(\Gamma_2,\Gamma_1)]\!] &= m_{s,r}\circ\mathsf{swap}\circ n_{r,s}\\
[\![C^{r\times 1}(\Gamma,())\Rightarrow_c C^r\Gamma]\!] &= \mathsf{fst}\circ n_{r,1}\\
[\![C^{r\times 0}(\Gamma,x:\tau)\Rightarrow_c C^r\Gamma]\!] &= \mathsf{fst}\circ n_{r,0}\\
[\![C^{r\vee s}(x:\tau)\Rightarrow_c C^{r\times s}(x:\tau,x:\tau)]\!] &= m_{r,s}\circ\Delta_{r,s}\\
[\![C^{r\times(s\times t)}(\Gamma_1,(\Gamma_2,\Gamma_3))\Rightarrow_c C^{(r\times s)\times t}((\Gamma_1,\Gamma_2),\Gamma_3)]\!] &= m_{r\times s,t}\circ(m_{r,s}\times\mathsf{id})\circ\mathsf{assoc}_1\circ(\mathsf{id}\times n_{s,t})\circ n_{r,s\times t}\\
[\![C^r\Gamma\Rightarrow_c C^s\Gamma]\!] &= \iota_{r,s}
\end{aligned}$$

Figure 15: Categorical semantics for λ_{Cs}

As discussed in 5.1.3, the variable context Γ in structural coeffect system is not a simple finite product, but instead a binary tree. To model this, we do not use ordinary products in the domain of the semantic function, but instead use a special constructor $\hat{\times}$. This way, we can guarantee that the variable structure corresponds to the tag structure.

5.2.1 *Structural tagged comonads*

To model composition of functions, we reuse the definition of *tagged comonads* from Section **??** without any change. This means that composing morphisms $T^r\tau_1\to\tau_2$ with $T^s\tau_2\to\tau_3$ still gives us a morphism $T^{r\vee s}\tau_1\to\tau_3$ and we use the $\vee$ operation to combine the context-requirements.

However, functions that do not exist in context have only a single input variable (with a single corresponding tag). To model complex variable contexts, we need two additional operations that allow manipulation with the variable context. Similarly to the model of λ_{Cf}, we also require operations that model duplication and sub-coeffecting:

Definition 6 (Structural tagged comonad)**.** *Given a structural coeffect tag structure* $(S,\times,\vee,0,1)$ *a* structural tagged comonad *is a tagged comonad over* $(S,\vee,0)$ *comprising of* T^r, ϵ_0 *and* $(-)^{\dagger}_{r,s}$ *together with a mapping* $-\hat{\times}-$ *from a pair of objects* $\mathsf{obj}(\mathcal{C})\times\mathsf{obj}(\mathcal{C})$ *to an object* $\mathsf{obj}(\mathcal{C})$ *and families of mappings:*

$$\begin{aligned}
m_{r,s} &: T^rA\times T^sB\to T^{(r\times s)}(A\hat{\times}B)\\
n_{r,s} &: T^{(r\times s)}(A\hat{\times}B)\to T^rA\times T^sN
\end{aligned}$$

And with a family of mappings $\iota_{r,s} : T^rA\to T^sA$ *for all* $r,s\in S$ *such that* $r\vee s=r$.

The family of mappings $\iota_{r,s}$ is the same as for *flat* coeffects and it can still be used to define a family of mappings that represents *duplicating* of variables while splitting the additional coeffect tags:

$$\Delta_{r,s} : \mathsf{T}^{(r \vee s)} A \to \mathsf{T}^r A \times \mathsf{T}^s A$$
$$\Delta_{r,s}(\gamma) = (\iota_{(r \vee s),r}\ \gamma, \iota_{(r \vee s),s}\ \gamma)$$

The type of the $\mathsf{m}_{r,s}$ operation looks similar to the one used for *flat* coeffects, but with two differences. Firstly, it combines tags using $\times$ instead of $\vee$, which corresponds to the fact that the variable context now consists of two parts (a tree node). Secondly, to model the tree node, the resulting context is modelled as $A \hat{\times} B$ (instead of $A \times B$ as previously).

To model structural coeffects, we also need $\mathsf{n}_{r,s}$, which serves as the dual of $\mathsf{m}_{r,s}$. It represents *splitting* of context containing multiple variables. The operation was not needed for λ_{Cf}, because there *splitting* could be defined in terms of *duplication* provided by $\Delta_{r,s}$. For λ_{Cs}, the situation is different. The $\mathsf{n}_{r,s}$ operation takes a context annotated with $r \times s$ that carries $A \hat{\times} B$.

Examples of *structural tagged comonads* are shown in Section 5.3.2. Before looking at them, we finish our discussion of categorical semantics.

CATEGORICAL NOTES. The mapping T^r can be extended to an endofunctor $\hat{\mathsf{T}}^r$ in the same way as in Section **??**. However, we still cannot freely manipulate the variables in the context. Given a context modelled as $\mathsf{T}^{r \times s}(A \hat{\times} B)$, we can lift a morphism f to $\hat{\mathsf{T}}^{r \times s}(f)$, but we cannot manipulate the variables, because $A \hat{\times} B$ is not a product and does not have projections π_i.

This also explains why n cannot be defined in terms of Δ. Even if we could apply $\Delta_{r,s}$ on the input (if the tag $r \times s$ coincided with tag $r \vee s$) we would still not be able to obtain $\mathsf{T}^r A$ from $\mathsf{T}^r(A \hat{\times} B)$.

This restriction is intentional – at the semantic level, it prevents manipulations with the context that would break the correspondence between tag structure and the product structure.

5.2.2 *Categorical semantics*

The categorical semantics of $\lambda_{Cs}$is shown in Figure 15. It uses the *structural tagged comonad* structure introduced in the previous section, together with the helper operation $\Delta_{r,s}$ and the following simple helper operations:

$$\begin{aligned} \mathsf{assoc} &= \lambda(\delta_r, (\delta_s, \delta_t)) \to ((\delta_r, \delta_s), \delta_t) \\ \mathsf{swap} &= \lambda(\gamma_1, \gamma_2) \to (\gamma_2, \gamma_1) \\ f \times g &= \lambda(x, y) \to (f\ x, g\ y) \end{aligned}$$

When compared with the semantics of λ_{Cf}(Figure **??**), there is a number of notable differences. Firstly, the rule λ_{Cs} -Svar is now interpreted as ϵ_0 without the need for projection π_i. When accessing a variable, the context contains only the accessed variable. The λ_{Cs} -Sfun rule has the same structure – the only difference is that we use the $\times$ operator for combining context tags instead of $\vee$ (which is a result of the change of type signature in $\mathsf{m}_{r,s}$).

The rule λ_{Cs} -Sapp now uses the operation $\mathsf{n}_{s,(r \vee t)}$ instead of $\Delta_{s,(r \vee t)}$, which means that it splits the context instead of duplicating it. This makes the system more structural – the expressions use disjunctive parts of the context – and also explains why the composed coeffect tag is $s \times (r \vee t)$.

The only rule from λ_{Cf} that was not syntax-directed (λ_{Cf} -Ssub) is now generalized to a number of non-syntax-directed rules λ_{Cs} -SC that perform various manipulations with the context. The semantics of $[\![C^{r_1}\Gamma_1 \Rightarrow_c C^{r_2}\Gamma_2]\!]$ is a function that, when given a context $C^{r_1}\Gamma_1$ produces a new context $C^{r_2}\Gamma_2$. The semantics in λ_{Cs} -Sctx then takes a context, converts it to a new context which is compatible with the original expression e. The context manipulation rules work as follows:

- The λ_{Cs} -SCnest and λ_{Cs} -SCexch rules use $n_{r,s}$ to split the context into a product of contexts, then perform some operation with the contexts – transform one and swap them, respectively. Finally, they re-construct a single context using $m_{r,s}$.
- The λ_{Cs} -SCempty and λ_{Cs} -SCweak rules have the same semantics. They both split the context and discard one part (containing either an unused variable or an empty context).
- If we interpreted λ_{Cs} -SCcontr by applying functor $T^{r \vee s}$ to a function that duplicates a variable, the resulting context would be $C^{r\vee s}(x : \tau, x : \tau)$, which would break the correspondence between coeffect tag and context variable structure. However, that interpretation would be incorrect, because we use $\hat{\times}$ instead of normal product for variable contexts. As a result, the rule has to be interpreted as a composition of $\Delta_{r,s}$ and $m_{r,s}$, which also turns a tag $r \vee s$ into $r \times s$.
- The λ_{Cs} -SCassoc rule is similar to λ_{Cs} -SCexch in the sense that it de-constructs the context, manipulates it (using assoc) and then re-constructs it.
- Finally, the λ_{Cs} -SCsub rule interprets sub-coeffecting on the context associated with a single variable using the primitive natural transformation $\iota_{r,s}$.

ALTERNATIVE: SEPARATE VARIABLES. As an alternative, we could model an expression by attaching the context separately to individual variables. This an expression $C^{r\times s}(\Gamma_1, \Gamma_2) \vdash e : \tau$ would be modelled as $C^r\Gamma_1 \times C^s\Gamma_2 \to \tau$. However, this approach largely complicates the definition of application (where tag of all variables in a context is affected). Moreover, it makes it impossible to express λ_{Cf} in terms of λ_{Cs} as discussed in Section **??**.

ALTERNATIVE: WITHOUT SUB-COEFFECTING. The semantics presented above uses the natural transformation $\iota_{r,s}$, which represents sub-coeffecting, to define the duplication operation $\Delta_{r,s}$. However, structural coeffect calculus λ_{Cs}does not require sub-coeffecting in the same way as flat λ_{Cf}(where it is required for subject reduction).

This means that it is possible to define a variant of the system that does not have the λ_{Cs} -Tsub typing rule. Then the semantics does not need the $\iota_{r,s}$ transformation, but instead, the following natural transformation has to be provided:

$$\Delta_{r,s} : T^{(r\vee s)}A \to T^r A \times T^s A$$

This variant of the system could be used to define a system that ensures that all provided context is used and is not over-approximated. This difference is similar to the difference between affine type systems (where a variable can be used at most once) and linear type systems (where a variable has to be used exactly once).

5.3 EXAMPLES OF STRUCTURAL COEFFECTS

5.3.1 *Example: Liveness analysis*

5.3.2 *Example: Data-flow (revisited)*

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

5.4 CONCLUSIONS

TODO: (...)

6 UNIFIED COEFFECT LANGUAGE

6.1 INTRODUCTION

Context is important for defining meaning– not just in natural languages, but also in logics and programming languages. The standard notion of context in programming is an environment providing values for free variables. An open term with free variables is context dependent – its meaning depends on the free-variable context. The simply-typed λ-calculus famously analyses such context usage. Other systems go further. For example, bounded linear logic tracks the number of times a variable is used [**?**].

In software engineering, "context" provides more than just free-variable values. For example, in a distributed system, the context provides different resources that may be available on different devices (e. g., a database on a server or a GPS sensor on a phone).

In this paper we develop a calculus for capturing various notions of context in programming. A key feature and contribution of the calculus is its *coeffect system* which provides a static analysis for contextual properties (coeffects). The system follows the style of type and effect systems, but captures a different class of properties. Another key contribution of the calculus is its semantics which can be smoothly instantiated for specific notions of context.

Coeffect systems were previously introduced as a generic analysis of context dependence which can be instantiated for various notions of context [**?**]. However, the formalization was restricted to tracking a class of *whole-context* properties where a term has just one coeffect. This limited the applications and precision of any analysis. For example, a whole-context liveness analysis marks the free-variable context as live (some variable may be used) or dead (no variable is used), but it cannot record liveness *per-variable*.

We develop a more general system which captures both *per-variable* coeffects, which we call *structural*, and *whole-context* coeffects, which we call *flat*, and more. Our key contributions are:

- We present a general coeffect type system (Section 6.3) and demonstrate two concrete uses – *flat* coeffect systems that tracks whole-context information and *structural* coeffect systems which tracks fine-grained per-variable information.

- We show practical examples, instantiating the calculus for structural systems capturing variable usage based on bounded linear logic, dataflow caching, and precise liveness analysis. We also instantiate the calculus to flat systems, building on and extending previous examples [**?**].

- We discuss the syntactic properties of the *structural* coeffect system (Section 6.4). It satisfies type preservation under both β-reduction and η-expansion, allowing its use with both call-by-name and call-by-value languages. This important property distinguishes it from both effect systems and flat coeffects.

- We provide a denotational semantics, resisting and extending the notion of *indexed comonads* to the structural setting (Section 6.5). We prove soundness by showing the correspondence between syntactic and semantic properties of coeffect systems.

Coeffects can be approached from multiple directions (Section 6.2.5) including syntactic (effect systems), semantic, and proof-theoretic. We emphasize the syntactic view, though we also outline a categorical semantics and note the interesting technical details.

6.2 WHY COEFFECTS MATTER

Coeffects are way to describe notions of context in programming that keep turning up. To illustrate this, we overview three systems tracking contextual properties that motivate our general coeffect system. Two systems track per-variable properties (bounded linear logic and dataflow) and one tracks whole-context properties (implicit parameters). We start with some background and finish with a brief overview of the literature leading to coeffects.

6.2.1 *Background, scalars and vectors*

The λ-calculus is asymmetric– it maps a context with *multiple* variables to a *single* result. An expression with n free variables of types σ_i can be modelled by a function $\sigma_1 \times \ldots \times \sigma_n \to \tau$ with a product on the left, but a single value on the right. Effect systems attach effect annotations to the result τ. In coeffect systems, we attach coeffects to the context $\sigma_1 \times \ldots \times \sigma_n$ and we often (but not always) have one coeffect per each variable. We call the overall coeffect a *vector* consisting of *scalar* coeffects. This asymmetry explains why coeffect systems are not trivially dual to effect systems.

It is useful to clarify how vectors are used in this paper. Suppose we have a set $\mathcal{C}$ of *scalars* such as $r_1, \ldots, r_n \in \mathcal{C}$. A vector R over $\mathcal{C}$ is a tuple $\langle r_1, \ldots, r_n \rangle$ of scalars. We use letters like R, S, T for vectors and r, s, t for scalars.[1] We also say that a *shape* of a vector $[R]$ (or more generally any container) is the set of *positions* in a vector. So, a vector of length n has shape $\{1, 2, \ldots, n\}$.

Just as in scalar-vector multiplication, we lift any binary operation on scalars into a scalar-vector one: $s \bullet R = \langle s \bullet r_1, \ldots, s \bullet r_n \rangle$. Given two vectors R, S of the same shape, containing partially ordered scalars, we write $R \leqslant S$ for the pointwise extension of $\leqslant$ on scalars. Finally, the associative operation $\times$ concatenates vectors.

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

6.2.2 *Bounded reuse*

Bounded linear logic [**?**] restricts well-typed terms to polynomial-time algorithms. This is done by limiting the number of times a value (proposition) can be used. An assumption $!_k A$ means that a variable can be used at most k times. We attach annotations to the whole context rather than individual assumptions and so a context $!_{k_1} A_1, ..., !_{k_n} A_n$ is written as $\tau_1, ..., \tau_n @ \langle k_1, ..., k_n \rangle$. This difference is further explained in Section 6.6.

1 For better readability, the paper distinguishes different structures using colours. However ignoring the colour does not introduce any ambiguity.

$$(\textit{var})\ \frac{}{x\!:\!\tau @ \langle 1\rangle \vdash x : \tau} \qquad (\textit{weak})\ \frac{\Gamma @ R \vdash e : \tau}{\Gamma, x\!:\!\sigma @ R \times \langle 0\rangle \vdash e : \tau}$$

$$(\textit{sub})\ \frac{\Gamma @ R \vdash e : \tau}{\Gamma @ R' \vdash e : \tau}\ (R \leqslant R') \qquad (\textit{abs})\ \frac{\Gamma, x\!:\!\sigma @ R \times \langle s\rangle \vdash e : \tau}{\Gamma @ R \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$$

$$(\textit{app})\ \frac{\Gamma_1 @ R \vdash e_1 : \sigma \xrightarrow{t} \tau \quad \Gamma_2 @ S \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ R \times (t * S) \vdash e_1\ e_2 : \tau}$$

$$(\textit{contr})\ \frac{\Gamma_1, y\!:\!\sigma, z\!:\!\sigma, \Gamma_2 @ R \times \langle s, t\rangle \times Q \vdash e : \tau}{\Gamma_1, x\!:\!\sigma, \Gamma_2 @ R \times \langle s + t\rangle \times Q \vdash e[z, y \leftarrow x] : \tau}$$

$$(\textit{exch})\ \frac{\Gamma_1, x\!:\!\sigma', y\!:\!\sigma, \Gamma_2 @ R \times \langle s, t\rangle \times Q \vdash e : \tau}{\Gamma_1, y\!:\!\sigma, x\!:\!\sigma', \Gamma_2 @ R \times \langle t, s\rangle \times Q \vdash e : \tau}$$

Figure 16: type and coeffect system for bounded reuse

Bounded linear logic includes explicit weakening and contraction rules that affect the multiplicity. Following the original logical style (but with our notation), these are written as:

$$\frac{\Gamma @ R \vdash \tau}{\Gamma, \sigma @ R \times \langle 0\rangle \vdash \tau} \qquad \frac{\Gamma_1, \sigma, \sigma, \Gamma_2 @ R \times \langle s, t\rangle \times Q \vdash \tau}{\Gamma_1, \sigma, \Gamma_2 @ R \times \langle s + t\rangle \times Q \vdash \tau}$$

The context $\Gamma @ R$ includes a *coeffect annotation* R which is a vector $\langle r_1, \ldots, r_n\rangle$ of the same length as Γ (a side-condition omitted for brevity). In weakening (left), unused propositions are annotated with 0 (no uses), while in contraction (right), multiple occurrences of a proposition are joined by adding the number of uses.

BOUNDED LINEAR COEFFECTS. The system in Figure 16 extends the outlined idea into a simple calculus. Variable access (*var*) has a singleton context with a singleton coeffect vector $\langle 1\rangle$. Weakening (*weak*) extends the free-variable context with an unused variable and the coeffect with an associated scalar 0. Explicit contraction (*contr*) and exchange (*exch*) rules manipulate variables in the context and modify the annotations accordingly – adding the number of uses in contraction and switching vector elements in exchange.

For abstraction (*abs*), we know the number of uses of the parameter variable x and attach it to the function type $\sigma \xrightarrow{s} \tau$ as a *latent* coeffect. The remaining variables in Γ are annotated with the remaining coeffect vector R, specifying *immediate* coeffects.

Application (*app*) describes call-by-name evaluation. Applying a function that uses its parameter t-times to an argument that uses variables in Γ_2 S-times means that, in total, the variables in Γ_2 will be used $(t * S)$-times. Recall that $t * S$ is a scalar multiplication of a vector. Meanwhile, the variables in Γ_1 are used just R-times when reducing the expression e_1 to a function value.

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

EXAMPLE. To demonstrate, consider a term $(\lambda v.x + v + v)\ (x + y)$. According to the call-by-name intuition, the variable x is used three times – once directly inside the function and twice via the variable v after substitution.

Similarly, y is used twice. Assuming a judgment for the function body, abstraction yields:

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

To avoid name clashes, we α-rename x to x' and later join x and x' using contraction. Assuming $(x' + y)$ is checked in a context that marks x' and y as used once, the application rule yields a judgment that is simplified as follows:

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

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

It is worth pointing out that reduction by substitution yields $x + (x + y) + (x + y)$ which has the same coeffect as the original. We return to evaluation strategies in Section 6.4, and show that structural coeffect systems preserve types and coeffects under β-reduction.

6.2.3 *Dataflow and data access*

Dataflow languages such as Lucid [**?**] describe computations over *streams*. An expression is re-evaluated when new inputs are available (push) or when more output is demanded (pull). In causal dataflow, programs can access past values of a stream. We consider a language where **prev** e returns the previous value of e, where **prev** (**prev** e) therefore returns the second past value.

An implementation of causal dataflow may cache past values of variables as an optimisation. The question is, how many past values should be cached? This can be approximated by a coeffect system.

DATAFLOW COEFFECTS. The coeffect system for dataflow is similar to the one for bounded reuse in that it tracks a vector of numbers R as part of the context $\Gamma @ R$. Here, coeffects represent the maximal number of past values (*causality depth*) required for a variable.

Weakening, exchange, abstraction and sub-coeffecting are the same as in bounded linear coeffects, but the remaining rules differ. In Figure 17, accessed variables (*var*) are annotated with 0 meaning that no past value is required (only the current one). The (*prev*) rule crates caching requirements – it increments the number of required values for all variables used in e using scalar-vector addition.

Application and contraction have the same structure as before, but use different operators. If two variables are contracting, requiring s and t past values, then overall we need at most $\mathsf{max}(s, t)$ past values (*contr*). That is, two caches are combined with the maximum of the two requirements, which satisfy the smaller requirements.

In (*app*), the function requires t past values of its parameter. This means t past values of e_2 are needed which in turn requires S past values of its free variables Γ_2. Thus, we need $t + S$ past values of Γ_2 to perform the call (e. g., we need $1 + S$ values to get 1 past value of the input σ, $2 + S$ values to get 2 past values of σ, *etc.*).

$$(\text{contr})\ \frac{\Gamma_1, y{:}\tau, z{:}\tau, \Gamma_2 @ R \times \langle s, t\rangle \times Q \vdash e : \tau}{\Gamma_1, x{:}\tau, \Gamma_2 @ R \times \langle \mathsf{max}(s,t)\rangle \times Q \vdash e[y, z \leftarrow x] : \tau}$$

$$(\text{app})\ \frac{\Gamma_1 @ R \vdash e_1 : \sigma \xrightarrow{t} \tau \qquad \Gamma_2 @ S \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ R \times (t + S) \vdash e_1\ e_2 : \tau}$$

$$(\text{var})\ \frac{}{x{:}\tau @ \langle 0\rangle \vdash x : \tau} \qquad (\text{prev})\ \frac{\Gamma @ R \vdash e : \tau}{\Gamma @ 1 + R \vdash \textbf{prev}\ e : \tau}$$

Figure 17: type and coeffect system for dataflow caching

EXAMPLE. As an example, consider a function $\lambda x.\textbf{prev}\ (y + x)$ applied to an argument $\textbf{prev}\ (\textbf{prev}\ y)$. The body of the function accesses the past value of two variables, one free and one bound:

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

The expression always requires the previous value of y and adds it to a previous value of the parameter x. Evaluating the value of the argument $\textbf{prev}\ (\textbf{prev}\ y)$ requires two past values of y and so the overall requirement is 3 past values:

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

The derivation uses (*app*) to get requirements $\langle 1, 3\rangle$ and then (*contr*) to take the maximum, showing three past values are sufficient. Reducing the expression by substitution we get $\textbf{prev}\ (y + (\textbf{prev}\ (\textbf{prev}\ y)))$. Semantically, this performs stream lookups $y[1] + y[3]$ where the indices are the number of enclosing **prev**s.

We previously used dataflow as an example of coeffects [?], but tracked caching requirements on the whole context. The system outlined here is more powerful and practically useful, with finer-grained coeffects tracking caching requirements per-variable.

6.2.4 *Implicit parameters*

As our third example, we revisit Haskell implicit parameters [?] used in our earlier coeffect work [?]. Implicit parameters are variables that mix aspects of dynamic and lexical scoping. Implicit parameters are a distinct syntactic category to variables and we write them as $?p$. For simplicity, we omit *let*-binding for implicit parameters and focus just on tracking requirements.

IMPLICIT PARAMETERS COEFFECTS. Implicit parameters are a whole-context coeffect not linked to ordinary variables. We keep track of sets of implicit parameters that are required by an expression (and their types). For example $\Gamma @ \{?p_1 : \tau_1, \ldots, ?p_n : \tau_n\}$ means that a context provides ordinary variables Γ and values for implicit parameters $?p_i$. Unlike in the previous examples, we no longer need to distinguish between coeffects attached to variables (scalars) and coeffects attached to contexts (vectors), so we write r, s, t for both.

Despite the differences, the type system in Figure 18 follows the same structure as the earlier two examples. Context requirements are created

$$(\ \text{exch})\ \frac{\Gamma_1, x:\tau, y:\sigma, \Gamma_2 @ r \cup s \cup t \cup q \vdash e : \tau}{\Gamma_1, y:\sigma, x:\tau, \Gamma_2 @ r \cup t \cup s \cup q \vdash e : \tau}$$

$$(\ \text{app})\ \frac{\Gamma_1 @ r \vdash e_1 : \sigma \xrightarrow{t} \tau \qquad \Gamma_2 @ s \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ r \cup t \cup s \vdash e_1\ e_2 : \tau}$$

$$(\ \text{param})\ \frac{}{() @ \{?p : \tau\} \vdash ?p : \tau} \qquad (\ \text{abs})\ \frac{\Gamma, x:\sigma @ r \cup s \vdash e : \tau}{\Gamma @ r \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$$

Figure 18: Type and coeffect system for implicit parameters

when accessing an implicit parameter (*param*) (a system-specific rule). Structural rules (exchange, weaken, contract) do not affect the coeffects. For example parameters are reordered in (*exch*), but this has no effect as set union $\cup$ is commutative.

In abstraction and application, the structural $\times$operator (previously vector concatenation) becomes $\cup$. Sets of implicit parameters are not associated to individual variables and so they are unioned. The (*app*) rule uses $\cup$ to combine the implicit parameters required by the function with the requirements of the argument too.

We call this a *flat* coeffect system since coeffects have only one shape (there is no scalar/vector distinction). Other flat coeffect systems may use a richer structure [**?**]. In particular, the operations used in abstraction and application may differ (to accommodate over-approximation). We return to this in Section 6.3.4.

EXAMPLE. Unlike structural coeffect systems, flat systems do not necessarily have principal coeffects. This arises from the (*abs*) rule which can freely split requirements between the function type and the declaring context. Consider a function $\lambda().?p_1 + ?p_2$. There are nine possible type and coeffect derivations, two of which are:

$$\emptyset @ \{\} \vdash (\ldots) : \mathsf{unit} \xrightarrow{\{?p_1 : \mathbb{Z}, ?p_2 : \mathbb{Z}\}} \mathbb{Z}$$
$$\emptyset @ \{?p_1 : \mathbb{Z}\} \vdash (\ldots) : \mathsf{unit} \xrightarrow{\{?p_2 : \mathbb{Z}\}} \mathbb{Z}$$

In the first case, both parameters are dynamically scoped and have to be provided by the caller. In the second case, the parameter $?p_1$ is available in the declaring scope and so it is (lexically) captured.

Although structural coeffects have more desirable syntactic properties, we aim to capture this non-principality too as it is practically useful – Haskell's implicit parameters use it and it can be used to model resource rebinding in distributed systems such as [**?**]).

6.2.5 *Pathways to coeffects*

This paper mainly follows work on effect systems and their link to categorical semantics. We briefly review this and other directions leading to coeffects. An eager reader can return to this section later.

EFFECT SYSTEMS. Effect systems [**?**] track effectful operations of computations such as memory access or lock usage [**?**]. They are written as judgments $\Gamma \vdash e : \tau\ \&\ \rho$ associating effects ρ with the result. Effect systems capture *output effects* where, as Tate puts it, *"all computations with [an] effect* *can be thunked as pure computations for a domain-specific notion of purity."* [13]. This thunking is typically a λ-abstraction. Given an effectful expression e, the function $\lambda x.e$ is an effect-free value that delays all effects:

$$\text{(abs)}\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \ \&\ \rho}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\rho} \tau_2 \ \&\ \emptyset}$$

Coeffects do not follow this pattern. In contrast to effect systems, context requirements cannot be easily "thunked" as pure values. Lambda abstraction can split context requirements between *immediate* and *latent* requirements. This is akin to how lambda abstraction splits a free-variable context into the bound parameter (call site) and the remaining free variables (declaration site).

CATEGORICAL SEMANTICS. Moggi models effectful computations as functions of type $\tau_1 \to M\tau_2$ where M is monad providing composition of effectful computations [?]. Wadler and Thiemann [?] link effect systems with monads using annotated monads $\tau_1 \to M^{\rho}\tau_2$ whose semantics has been provided by Katsumata [?].

Context-dependent computations require a different model. Uustalu and Vene [?] use functions $C\tau_1 \to \tau_2$ where C is a *comonad*. Our earlier work [?] used indexed comonads with denotations $C^{r}\tau_1 \to \tau_2$ adding annotations akin to Wadler and Thiemann. In Section 6.5 we extend indexed comonads to capture the general coeffect systems of this paper, in the style of Katsumata.

LANGUAGE AND META-LANGUAGE. Moggi uses monads in two systems [?]. In the first system, a monad is used to model an effectful language itself – the semantics of a language uses a specific monad. In the second system, monads are added as type constructors, together with syntax corresponding to *unit* and *bind* operations.

Looking at context, Uustalu and Vene [?] follow the first approach (using a concrete comonad to model dataflow). Contextual-Modal Type Theory (CMTT) of Nanevski et al. [?] follows the latter approach, adding a comonad via the $\Box$ modality of modal S4 to the language. We focus on concrete languages using the first approach. A "coeffect meta-language" is an interesting future work.

SUB-STRUCTURAL AND BUNCHED TYPES. Sub-structural type systems restrict how a context is used. This is achieved by removing some of the structural typing rules (weakening, contraction, exchange). As the bounded linear logic example (Section 6.2.2) shows, our system can be viewed as a generalization.

6.3 THE COEFFECT CALCULUS

The three calculi shown in the previous section track two kinds of contextual properties: bounded reuse and dataflow are structural (per-variable) systems, and implicit parameters and our earlier coeffect systems [?] are flat (whole-context) systems. This section presents our primary contribution: the general coeffect calculus.

The calculus is parameterised by an algebraic structure of coeffects. To capture both structural and flat systems, coeffect annotations are indexed by a *shape*. In flat systems, the shape is a singleton set and so annotations are *scalar* values. Structural systems use shapes matching the number of variables in a free-variable context and so annotations are *vectors*. However, the coeffect calculus could also use shapes describing trees and other structures.

6.3.1 *Understanding coeffects: syntax and semantics*

The coeffect calculus provides both an analysis of context dependence (its coeffect system) and a semantics for context (see Section 6.5). These two features of the calculus provide different perspectives on coeffect annotations R in a judgment $\Gamma @ R \vdash e : \tau$:

- Syntactically, coeffects model *contextual requirements* and may be over-approximated, such that more capabilities are required than necessary at runtime.

- Semantically, coeffects model *contextual capabilities* and behave like containers of capabilities, such that the semantics may throw away capabilities that will not be needed.

Thus there are two dual ways to understand coeffect annotations. Each perspective implies an alternate reading of the typing rules.

- As *contextual requirements*, the rules should be read top-down. The requirements of multiple sub-terms are *merged* and the requirements of function body are *split* between immediate (declaration-site) and latent (call-site) coeffects.

- As *contextual capabilities*, the rules should be read bottom-up. The capabilities provided to a larger term are *split* between sub-terms; for functions, the capabilities of declaration-site and call-site are *merged* and passed to the body.

The reason for this asymmetry follows from the fact that context appears in a *negative position* in the model. In Section 6.5, the denotation of a judgment is a function of the form $D_R[\![\Gamma]\!] \rightarrow [\![\tau]\!]$ where $D_R[\![\Gamma]\!]$ encodes the contextual capabilities used to evaluate a term. Similarly functions have models of the form $D_s[\![\sigma]\!] \rightarrow [\![\tau]\!]$ with additional contextual capabilities attached to the input.

6.3.2 *Structure of coeffects*

We describe the algebraic structure of coeffects in three steps. First, we define a *coeffect scalar* structure which defines the basic building blocks of coeffect information; then we define *coeffect shapes* which determines how coeffect scalar values are related to the free-variable context. Finally, we define the *coeffect algebra* which consists of shape-indexed coeffect scalar values.

For example, in bounded reuse the coeffect scalar structure comprised natural numbers $\mathbb{N}$ with $+$ and $*$ operators. The shape for bounded reuse is the length of the free-variable context and so the coeffect annotation is a vector of matching length. Finally, the coeffect algebra specifies how vectors are concatenated and split in abstraction and application.

In the coeffect system of the calculus, contexts are annotated with shape-indexed coeffects (e. g., vectors) as in $\Gamma @ R \vdash e : \tau$. However, functions take just a single input parameter and so are annotated with scalar coeffect values as in $\sigma \xrightarrow{r} \tau$.

COEFFECT SCALAR. Coeffect scalar structures are equipped with two operations. In bounded reuse, those were $*$ for sequencing (in function application) and $+$ for context sharing (in contraction). Additional structure is needed for variable access and sub-coeffecting.

Definition 7. 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. The following distributivity axioms are required:*

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

The operation $\circledast$ must form a monoid with use to guarantee an underlying category in the semantics (Section 6.5). It models sequential composition with variable access (use) as the identity. The other element (ign) is used for variables that are not accessed. The operation $\oplus$ combines coeffects for contexts used in multiple places (contraction). The notation is inspired by the bounded reuse example, which uses coeffect scalar structure $(\mathbb{N}, *, +, 1, 0, \leqslant)$, but be aware that $\circledast$ and $\oplus$ do not always mean $*$ and $+$.

The context annotations can be viewed as *containers* of scalar coeffects. For structural coeffects, the container is a vector, while for flat coeffects, it is a trivial singleton container. We take inspiration from the work of Abbott *et al.* [**?**] which describes containers in terms of *shapes* and a set of *positions* in each shape.

COEFFECT SHAPES. The coeffect system is parameterised by a set of shapes $\mathcal{S}$. A coeffect annotation is indexed by a shape $s \in \mathcal{S}$ calculated from the shape of the free-variable vector. The correspondence is not necessarily bijective. For example, flat coeffect systems have just a single shape $\mathcal{S} = \{*\}$.

In the coeffect judgment $\Gamma @ R \vdash e : \tau$, the coeffect annotation R is drawn from the set of coeffect scalars $\mathcal{C}$ indexed by the shape of Γ. We write $s = [\Gamma]$ for the shape corresponding to Γ. We define shapes by a *set* of positions and so we can write $R \in s \to \mathcal{C}$ as a mapping from positions (defined by the shape) to scalar coeffects. We usually write this as the exponent $R \in \mathcal{C}^s$.

The set of shapes is equipped with an operation that combines shapes (when we combine variable contexts), an operation that computes shape from the free-variable contexts, and two special shapes in $\mathcal{S}$ representing empty context and singleton context.

Definition 8. *A* coeffect shape *structure* $(\mathcal{S}, [-], \diamond, \hat{0}, \hat{1})$ *comprises a set* $\mathcal{S}$ *with a binary operation* $\diamond$ *on* $\mathcal{S}$ *for shape composition, a mapping from contexts to shapes* $[\Gamma] \in \mathcal{S}$*, and elements* $\hat{0}, \hat{1} \in \mathcal{S}$ *such that* $(\mathcal{S}, \diamond, \hat{0})$ *is a monoid. The elements* $\hat{0}$ *and* $\hat{1}$ *represent the shapes of empty and singleton free-variable contexts respectively.*

As said earlier, we use two kinds of shape structures that describe the shape of vectors and the shape of trivial singleton container:

- Structural coeffect shape is defined as $(\mathbb{N}, |-|, +, 0, 1)$. We treat numbers as sets $0 = \{\}, 1 = \{\emptyset\}, 2 = \{\emptyset, 1\}, 3 = \{\emptyset, 1, 2\} \ldots$ (so that a number is a set of positions). 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* $R \in \mathcal{C}^n$ and assigns a coeffect scalar $R(i) \in \mathcal{C}$ for each variable x_i in the context.

- Flat coeffect shape is defined as $(\{\star\}, \mathsf{const}\ \star, \diamond, \star, \star)$ where $\star \diamond \star = \star$ and $\star = \{\emptyset\}$. 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.

Using a shape with *no* positions reduces our system to the simply-typed λ-calculus with no context annotations. Trees could be used to build a system akin to bunched typing [**?**].

COEFFECT ALGEBRA. The coeffect calculus annotates judgments with shape-indexed (or, *shaped*) coeffects. The *coeffect algebra* structure combines a coeffect scalar and coeffect shape structures to define shaped coeffects and operations for combining these. In Section 6.2, shaped coeffects were combined by the tensor $\times$ in structural examples and $\cup$ in the implicit parameters example. To capture the examples so far and those described previously [**?**], we distinguish two operators for combining shaped coeffects.

Definition 9. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a coeffect shape* $(\mathcal{S}, [-], \diamond, \hat{0}, \hat{1})$ *a* coeffect algebra *extends the two structures with* $(\overline{\times}, \underline{\times}, \bot)$ *where* $\bot \in \mathcal{C}^{\hat{0}}$ *is a coeffect annotation for the empty context and* $\overline{\times}, \underline{\times}$ *are families of operations that combine coeffect annotations indexed by shapes. That is* $\forall n, m \in \mathcal{S}$*:*

$$\underline{\times}_{m,n}, \overline{\times}_{m,n} \ :\ \mathcal{C}^{m} \times \mathcal{C}^{n} \to \mathcal{C}^{m \diamond n}$$

A coeffect algebra induces the following two additional operations:

$$\begin{array}{lcl} \langle - \rangle & : & \mathcal{C} \to \mathcal{C}^{\hat{1}} \\ \langle x \rangle & = & \lambda \hat{1}.x \end{array} \qquad \begin{array}{lcl} \circledast_{m} & : & \mathcal{C} \times \mathcal{C}^{m} \to \mathcal{C}^{m} \\ r \circledast S & = & \lambda s.r \circledast (S(s)) \end{array}$$

$\langle - \rangle$ lifts a scalar coeffect to a shaped coeffect indexed by the singleton context shape. The $\circledast_m$ operation is a left multiplication of a vector by scalar. As we always use lower-case for scalars and upper-case for vectors, using the same symbol is not ambiguous. We also tend to omit the subscript m and write just $\circledast$.

The operators $\underline{\times}$ and $\overline{\times}$ combine shaped coeffects associated with two contexts. For example, assume we have Γ_1 and Γ_2 with coeffects $R \in \mathcal{C}^{m}$ and $S \in \mathcal{C}^{n}$. 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 $R \overline{\times} S, R \underline{\times} S \in \mathcal{C}^{m \diamond n}$ are indexed by that shape.

For structural coeffect systems such as bounded reuse, both $\underline{\times}$ and $\overline{\times}$ are just the tensor product $\times$ of vectors. However, we need to distinguish them for flat coeffect systems discussed later.

The difference is explained by the semantics (Section 6.5), where $R \overline{\times} S$ 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); $R \underline{\times} S$ 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 $\overline{\times}$ in the rule *assumptions* and $\underline{\times}$ in *conclusions*. For now, it suffices to use the bounded reuse intuition and read the operations as tensor products.

$$\boxed{\Gamma @ R \vdash e : \tau}$$

$$(\quad \text{const})\ \frac{}{()@\bot \vdash c : \iota} \qquad (\quad \text{var})\ \frac{}{(x:\tau)@\langle \mathsf{use} \rangle \vdash x : \tau}$$

$$(\quad \text{abs})\ \frac{\Gamma, x : \sigma @ R \underline{\times} \langle s \rangle \vdash e : \tau}{\Gamma @ R \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$$

$$(\quad \text{app})\ \frac{\Gamma_1 @ R \vdash e_1 : \tau \xrightarrow{t} \sigma \qquad \Gamma_2 @ S \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 @ R \underline{\times} (t \circledast S) \vdash e_1\ e_2 : \sigma}$$

$$(\quad \text{let})\ \frac{\Gamma_1 @ S \vdash e_1 : \sigma \qquad \Gamma_2, x : \sigma @ R \underline{\times} \langle t \rangle \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 @ R \underline{\times} (t \circledast S) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

$$(\quad \text{ctx})\ \frac{\Gamma @ R \vdash e : \tau \qquad \Gamma' @ R' \rightsquigarrow \Gamma @ R, \theta}{\Gamma' @ R' \vdash \theta e : \tau}$$

$$\boxed{\Gamma' @ R' \rightsquigarrow \Gamma @ R, \theta}$$

$$(\quad \text{weak}) \qquad \Gamma, x : \tau @ R \underline{\times} \langle \mathsf{ign} \rangle \rightsquigarrow \Gamma @ R, \emptyset$$

$$(\quad \text{exch}) \qquad \begin{array}{c} \Gamma_1, y : \sigma, x : \tau, \Gamma_2 @ R \underline{\times} \langle t \rangle \underline{\times} \langle s \rangle \underline{\times} Q \rightsquigarrow \\ \Gamma_1, x : \tau, y : \sigma, \Gamma_2 @ R \overline{\times} \langle s \rangle \overline{\times} \langle t \rangle \overline{\times} Q, \emptyset \end{array}$$

$$(\quad \text{contr}) \qquad \begin{array}{c} \Gamma_1, x : \tau, \Gamma_2 @ R \underline{\times} \langle s \oplus t \rangle \underline{\times} Q \rightsquigarrow \\ \Gamma_1, y : \tau, z : \tau, \Gamma_2 @ R \overline{\times} \langle s \rangle \overline{\times} \langle t \rangle \overline{\times} Q, [y, z \mapsto x] \end{array}$$

$$(\quad \text{sub}) \qquad \begin{array}{c} \Gamma_1, x : \tau, \Gamma_2 @ R \underline{\times} \langle s' \rangle \underline{\times} T \rightsquigarrow \\ \Gamma_1, x : \tau, \Gamma_2 @ R \overline{\times} \langle s \rangle \overline{\times} T, \emptyset \end{array} \qquad (s \leqslant s')$$

Figure 19: The general coeffect calculus

6.3.3 *General coeffect type system*

In the previous section, we developed an algebraic structure capable of capturing different concrete context-dependent properties discussed in Section 6.2. Now, we use the structure to define the general coeffect calculus in Figure 19.

Coeffect annotations on free-variable contexts are shape-indexed coeffects $R, S, T \in \mathcal{C}^{S}$ and function types are annotated with coeffects scalars $r, s, t \in \mathcal{C}$. The rules manipulate coeffect annotations using the operations provided by coeffect algebra $(\overline{\times}, \underline{\times}, \bot)$ and the derived constructs $\langle - \rangle$ and $\circledast$. 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 structural rules. To make the system easier to follow, structural rules are expressed using a separate judgment.

TYPING RULES. Constants (*const*) and variable access (*var*) annotate the context with special values. Empty unused context is annotated with $\bot \in \mathcal{C}^{\hat{0}}$, while singleton context is annotated with $\langle \mathsf{use} \rangle \in \mathcal{C}^{\hat{1}}$. Note that the shapes $\hat{0}, \hat{1}$ match the shape of the variable contexts.

Lambda abstraction *splits* the context requirements using $\overline{\times}$ into a coeffect R and a coeffect $\langle s \rangle$ of a shape $\hat{1}$ (semantically, it *merges* capabilities provided by the declaration-site and call-site contexts). In structural systems such as bounded reuse, $\underline{\times}$ is not symmetric and so this gives us a coeffect associated with the bound variable.

The (*app*) rule follows the patterns seen earlier – it uses the scalar-vector multiplication ($t \circledast S$) on coeffects associated with Γ_2. Using the syntactic reading, it then *merges* context requirements for Γ_1 and Γ_2. In the dual semantic reading, it *splits* the provided context into two parts passed to the sub-expressions.

The typing of let-binding (*let*) corresponds to the typing of an expression $(\lambda x.e_2)\ e_1$. Syntactically, the context requirements are first split using $\overline{\times}$ and then re-combined using $\underline{\times}$.

STRUCTURAL RULES. The coeffect-annotated context can be transformed using structural rules that are not syntax-directed. These are captured by (*ctx*), which uses a helper judgment representing context transformations $\Gamma'@R' \rightsquigarrow \Gamma@R, \theta$. The rule models that a context used in the rule conclusion $\Gamma'@R'$ can be transformed to a context required by the assumptions $\Gamma@R$ (using the semantic bottom-up reading). In the rule, θ is a variable substitution generated by the transformation, which is used in the (*contr*) rule.

Exchange and contraction decompose and reconstruct coeffect annotations using $\overline{\times}_{m,n}$ (in assumption) and $\underline{\times}_{m,n}$ (in conclusion). The shape subscripts are omitted, but we require the shapes to match using $m = [\Gamma_1]$ and $n = [\Gamma_2]$.

The (*weak*) rule drops an ignored variable annotated with $\langle \mathsf{ign} \rangle$ (compare with (*var*) annotated using $\langle \mathsf{use} \rangle$). The (*exch*) rule switches the values while (*contr*) combines them using $\oplus$ to represent sharing of the context. Finally, (*sub*) represents sub-coeffecting that can be applied (point-wise) to any individual coeffect.

6.3.4 *Structural coeffects*

The coeffect system uses a general notion of context shape, but it has been designed with structural and flat systems in mind. The structural system is new in this paper and so we look at it first.

Recall the coeffect shapes that characterise structural systems: the shape is formed by natural numbers (with addition) modelling the number of variables in the context. The coeffect algebra is therefore formed by the free monoid (vectors) over a coeffect scalar. This means that the system keeps a vector of basic coeffect annotations – one for each variable. An empty context (e. g., in the (*const*) rule) is annotated with an empty vector.

Definition 10. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *a* structural coeffect system *has:*

- *Coeffect shape* $(\mathbb{N}, |-|, +, 0, 1)$ *formed by natural numbers*
- *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* $\times : \mathcal{C}^n \times \mathcal{C}^m \to \mathcal{C}^{n+m}$ *appends vectors (lists) and* $\epsilon : \mathcal{C}^0$ *represents empty vectors (lists)*

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

EXAMPLES. Defining a concrete structural coeffect system is easy, we just provide the coeffect scalar structure and the rest is free.

- To recreate the system for bounded reuse, we use coeffect scalars formed by $(\mathbb{N}, *, +, 1, 0, \leqslant)$. As in the system of Figure 16, *used* variables are annotated with 1 and *unused* with 0. Contraction adds the number of uses via $+$ and application (sequencing) multiplies the uses.
- *Dataflow* uses natural numbers (of past values), but differently: $(\mathbb{N}, +, \mathsf{max}, 0, 0, \leqslant)$. Variables are initially annotated with 0 (and can be incremented using the **prev** keyword). Annotations of a shared variable are combined by taking maximum (of past values needed) and sequencing uses $+$.
- Another use of the system is to track *variable liveness*. The annotations are formed by $\mathcal{C} = \{\mathsf{D}, \mathsf{L}\}$ where L represents a *live* (used) variable and D represents a *dead* (unused) variable. The scalars are given as $(\mathcal{C}, \sqcap, \sqcup, \mathsf{L}, \mathsf{D}, \sqsubseteq)$.

  In sequential composition $(\sqcap)$, a variable is live only if it is required by both of the computations $(\mathsf{L} \sqcap \mathsf{L} = \mathsf{L})$, otherwise it is marked as dead (D). A computation is not evaluated if its result is not needed. A shared variable $(\sqcup)$ is live if either of the uses is live $(\mathsf{D} \sqcup \mathsf{D} = \mathsf{D}$, otherwise $\mathsf{L})$.

Structural liveness is a practically useful, precise version of an example from our earlier work, which was a flat system overapproximating liveness to the entire context [?].

6.3.5 *Flat coeffects*

The same general coeffect system can be used to define systems that track whole-context coeffects as in the implicit parameters example (Section 6.2.4). Flat coeffect systems are characterised by a singleton set of shapes, such as $\{\star\}$. In this setting, the context annotations $\mathcal{C}^\star$ are equivalent to coeffect scalars $\mathcal{C}$.

In addition to the coeffect scalar structure, we also need to define $\ltimes$ and $\rtimes$. Our examples of flat coeffects use $\oplus$ (merging of scalar coeffects) for $\ltimes$ (merging of shaped coeffect annotations). However, the $\rtimes$ operation needs to be provided explicitly.

Definition 11. *Given a coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and a binary* $\wedge : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *such that* $(r \wedge s) \leqslant (r \oplus s)$, *we define:*

- *Flat coeffect shape* $(\{\star\}, \mathsf{const}\,\star, \diamond, \star, \star)$ *where* $\star \diamond \star = \star$
- *Flat coeffect algebra* $(\wedge, \oplus, \mathsf{ign})$, *i. e., the* $\ltimes = \oplus$ *and* $\perp = \mathsf{ign}$ *with the additional binary operation* $\rtimes = \wedge$.

The requirement $(r \wedge s) \leqslant (r \oplus s)$ guarantees exchange and contraction preserve the coeffect of the assumption in the conclusion. Thus, flat coeffect calculi do not require substructural-style rules.

EXAMPLES. Implicit parameters are the prime example of a flat coeffect system, but other examples include rebindable resources [?] and Haskell type classes [?].

In the *implicit parameters* system (Section 6.2.4), the coeffect scalars are sets of names with types $\mathcal{C} = \mathcal{P}(\mathsf{Name} \times \mathsf{Types})$. Variables are always annotated with $\emptyset$ and coeffects are combined or split using set union $\cup$. Thus the system is given by coeffect scalar structure $(\mathcal{P}(\mathsf{Name} \times \mathsf{Types}), \cup, \cup, \emptyset, \emptyset, \subseteq)$ with $\wedge = \cup$.

Remark 1. *We previously described flat systems for* liveness *and* dataflow *[?]. Turning a structural system to flat requires finding $\wedge$ that underapproximates the capabilities of combined contexts. For dataflow, this is given by the* min *function as* $\mathsf{min}(r, s) \leqslant \mathsf{max}(r, s)$.

In flat dataflow*, we annotate the entire context with the maximal number of past elements required overall. We use the same coeffect scalars $(\mathbb{N}, +, \mathsf{max}, 0, 0, \leqslant)$ as in the structural version, but with $\wedge = \mathsf{min}$. Abstraction (which is the only rule using $\wedge$) becomes:*

$$(\;\;abs)\;\frac{\Gamma, x : \sigma @ \mathsf{min}(r, s) \vdash e : \tau}{\Gamma @ r \vdash \lambda x.e : \sigma \xrightarrow{s} \tau}$$

Both the declaration-site and the call-site need to provide at least the number of past values required by the body. The overapproximation means that both r and s can be greater than actually required. For dataflow, we could annotate both contexts with the same coeffect, but that would require treating $\times$ as a partial function.

6.4 EQUATIONAL THEORY

Each of the concrete coeffect systems discussed in this paper has a different notion of context-dependence, much like various effectful languages have different notions of effects (such as state or exceptions). However, there are equational properties that hold for all (or some) of the systems we consider.

The equational theory in this section illuminates the axioms of coeffect algebra and the semantics of the calculus. We discuss syntactic substitution as it can form the basis for reduction in a concrete operational semantics. We consider structural and flat systems separately. This provides better insight into how the two systems work and differ. In particular, call-by-name evaluation is *coeffect preserving* for all structural, but only some flat systems.

The properties and proofs in this section are syntactic. In Section 6.5.5 we show that our denotational model of the coeffect calculus is sound with respect to the equational theory here.

We use standard syntactic substitution written as $e_1[x \leftarrow e_2]$, β-reduction and η-expansion, written as $\rightsquigarrow_\beta$ and $\rightsquigarrow_\eta$. Equality of terms e_1 and e_2, defined by a relation $\equiv$, requires the equality of their contexts, types, and coeffects, written $\Gamma @ R \vdash e_1 \equiv e_2 : \tau$.

6.4.1 *Structural coeffect systems*

For structural coeffect systems, recall that coeffects are vectors with $\times = \times = \times$ (vector concatenation) and $\perp = \langle\rangle$ (the empty vector), thus coeffect annotations comprise the *free monoid* i. e., lists over the coeffect scalars (although we continue using the vector terminology). We first show substitution:

Lemma 5 (Substitution lemma)**.** *In a structural coeffect calculus with a coeffect scalar structure* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$:

$$\Gamma @ S \vdash e_s : \sigma \;\;\wedge\;\; \Gamma_1, x : \sigma, \Gamma_2 @ R_1 \times \langle r \rangle \times R_2 \vdash e_r : \tau$$
$$\Rightarrow\; \Gamma_1, \Gamma, \Gamma_2 @ R_1 \times (r \circledast S) \times R_2 \vdash e_r[x \leftarrow e_s] : \tau$$

Proof. By induction over the e_2 derivation using the free monoid structure $(\mathcal{C}, \times, \langle\rangle)$ and coeffect scalar axioms (full proof [?]). ☐

Because of the vector (free monoid) structure, coeffects R_1, R_2, and $\langle r \rangle$ for the receiving term e_r are uniquely associated with Γ_1, Γ_2, and x respectively. Therefore, substituting e_s (which has coeffects S) for x introduces the context dependencies specified by S which are composed with the requirements r on x. Using the substitution lemma, we can demonstrate β-equality:

$$\dfrac{\dfrac{\Gamma_1, x : \sigma @ R \times \langle r \rangle \vdash e_1 : \tau}{\Gamma_1 @ R \vdash \lambda x.e_1 : \sigma \xrightarrow{r} \tau} \qquad \Gamma_2 @ S \vdash e_2 : \sigma}{\Gamma_1, \Gamma_2 @ R \times (r \circledast S) \vdash (\lambda x.e_1) e_2 \equiv e_1[x \leftarrow e_2] : \tau}$$

As a result, β-reduction preserves the type and coeffects of a term. This gives the following subject reduction property:

Theorem 5 (Subject reduction)**.** *In a structural coeffect calculus, if* $\Gamma @ R \vdash e : \tau$ *and* $e \rightsquigarrow_\beta e'$ *then* $\Gamma @ R \vdash e' : \tau$*.*

Proof. Following from Lemma 5 and β-equality. □

Structural coeffect systems also exhibit η-equality, therefore satisfying both *local soundness* and *local completeness* conditions set by Pfenning and Davies [**?**]. This means that abstraction does not introduce too much, and application does not eliminate too much.

$$\dfrac{\dfrac{\Gamma @ R \vdash e : \sigma \xrightarrow{s} \tau \qquad x : \sigma @ \langle \mathsf{use} \rangle \vdash x : \sigma}{\Gamma, x : \sigma @ R \times (s \circledast \langle \mathsf{use} \rangle) \vdash e\,x : \tau}}{\Gamma @ R \vdash \lambda x.e\,x \equiv e : \sigma \xrightarrow{s} \tau}$$

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

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

6.4.2 *Flat coeffect systems*

The equational theory for flat coeffect systems is somewhat similar to effect systems where (co)effects are not linked to individual variables. In effectful languages, substituting an effectful computation for y in $\lambda x.y$ changes the latent effect associated with the function.

Similarly, for some of the flat coeffect systems, substituting a context-dependent computation for y in $\lambda x.y$ adds latent context requirements to the function type. However, this is not the case for *all* flat coeffect systems – for example, call-by-name reduction preserves types and coeffects for the implicit parameters system (which makes it a suitable model for Haskell). For other systems, we first briefly consider call-by-value reduction.

WHAT IS A VALUE? The notion of *value* in coeffect systems differs from the usual syntactic understanding. As discussed earlier, a function $(\lambda x.e)$ is not necessarily a value in coeffect calculi, because it may not delay all context requirements of e. Thus we say that e is a value if it has no immediate context requirements.

Definition 12. *An expression* e *is a* value, *written as* $val(e)$ *if* $\Gamma@\mathsf{VAL} \vdash e : \tau$ *where* $\mathsf{VAL} : \mathcal{C}^{[\Gamma]}$ *is a coeffect indexed by the shape of* Γ *that always returns* use. *That is* $\mathsf{VAL} = \lambda n.\mathsf{use}$.

CALL-BY-VALUE. In call-by-value, a pure *value* is substituted for a variable. The right-hand side of an application is evaluated to a value before the β-reduction, but the discharging of coeffects prior to substitution is different for each concrete coeffect system.

Recall that a flat coeffect system consists of coeffect scalars $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ together with a binary operation $\wedge$ on $\mathcal{C}$. The coeffect algebra is then defined as $(\wedge, \oplus, \mathsf{ign})$ where $\wedge$ and $\oplus$ represent splitting and merging of context-requirements, respectively.

Lemma 6 (Call-by-value substitution)**.** *In a flat coeffect calculus with coeffect scalars* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and the* $\wedge$ *operator:*

$$\begin{aligned}&\Gamma@\mathsf{VAL} \vdash e_s : \sigma \quad \wedge \quad \Gamma_1, x : \sigma, \Gamma_2@r \vdash e_r : \tau \\ &\quad \Rightarrow \quad \Gamma_1, \Gamma, \Gamma_2@r \vdash e_r[x \leftarrow e_s] : \tau\end{aligned}$$

Proof. By induction over the type derivation, using the fact that both x and e_s are annotated with use. □

Lemma 6 holds for all flat coeffect systems, but it is weak. To use it, the operational semantics must provide a way of partially evaluating a term with requirements $\Gamma@r$ to a value. Assuming $\rightsquigarrow_{\mathsf{cbv}}$ is call-by-value reduction using the above definition of value:

Theorem 6 (Call-by-value reduction)**.** *In a flat coeffect system, if* $\Gamma@r \vdash e : \tau$ *and* $e \rightsquigarrow_{\mathsf{cbv}} e'$ *then* $\Gamma@r \vdash e' : \tau$.

Proof. A direct consequence of Lemma 6, using the flat coeffect system requirement $(r \wedge s) \leqslant (r \oplus s)$ to prove β-equality. □

CALL-BY-NAME. The call-by-name strategy reduces a term $(\lambda x.e_1)\ e_2$ where both sub-expressions may have contextual requirements. Assume that $\Gamma@s \vdash e_2 : \tau_2$ and $\Gamma_1, x : \tau_1@r \vdash e_1 : \tau_1$.

We call a flat coeffect algebra *top-pointed* if use is the greatest (top) coeffect scalar $\mathcal{C}$ and *bottom-pointed* if it is the smallest (bottom) coeffect scalar. Liveness analysis is an example of top-pointed coeffects as variables are annotated with L and $\mathsf{D} \leqslant \mathsf{L}$.

Lemma 7 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and the* $\wedge$ *operator:*

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

Proof. Using sub-coeffecting ($s \leqslant \mathsf{use}$) and Lemma 6. □

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

In a bottom pointed coeffect system, substituting e for x increases the context requirements. However, if the system satisfies the strong condition that $\wedge = \circledast = \oplus$ then the context requirements arising from the substitution can be associated with the context Γ. As a result, substitution does not break soundness as in effect systems. The requirement $\wedge = \circledast = \oplus$ holds for our implicit parameters example (all three operators are set union) and allows the following substitution lemma:

Lemma 8 (Bottom-pointed substitution). *In a bottom-pointed flat coeffect calculus with* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *and the* $\wedge$ *operator where* $\wedge = \circledast = \oplus$ *is idempotent and commutative:*

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

Proof. By induction over $\vdash$, using the idempotent, commutative monoid structure to keep s with the free-variable context. □

The structural system is precise enough to keep the coeffects associated with a concrete variable. The flat variant described here is flexible enough to let us always re-associate new context requirements with the free-variable context.

The two substitution lemmas show that the call-by-name evaluation strategy can be used for certain coeffect calculi, including liveness and implicit parameters. Assuming $\rightsquigarrow_{\mathsf{cbn}}$ is the standard call-by-name reduction, the following theorem holds:

Theorem 7 (Call-by-name reduction). *In a coeffect system that satisfies the conditions for Lemma 7 or Lemma 8, if* $\Gamma@r \vdash e : \tau$ *and* $e \rightarrow_{\mathsf{cbn}} e'$ *then* $\Gamma@r \vdash e' : \tau$.

Proof. A direct consequence of Lemma 7 or Lemma 8. □

6.5 SEMANTICS

Coeffects provide a unified notion of context-dependence. In the previous sections, we used this to define a unified coeffect calculus. We now define a unified (categorical) semantics for the coeffect calculus. The semantics can be instantiated for different notions of context dependence and thus can model a wide range of context-aware languages (both for flat and structural systems).

We relate the semantics to the equational theory and show that it is sound with respect to term equality. For a variant of the the flat system, a similar result has already been shown in the second author's dissertation [**?**]. The semantics is introduced in pieces:

- Section 6.5.1 describes the range and domain (signature) of the interpretation $[\![-]\!]$, gives the interpretations for types and free-variable contexts (in flat and structural systems), and defines the signature of functors D which encode contexts.

- The first part of the semantics (Section 6.5.2) defines *sequential composition* of context-dependent computations via *indexed comonads* (introduced briefly in our previous work [**?**]) and the *indexed structural comonad* structure (new here).

- More structure is needed for application and abstraction. Section 6.5.3 defines indexed monoidal operations for splitting and merging contexts. Concrete structures are given throughout for the semantics of the structural bounded reuse and flat implicit parameter systems.

- Section 6.5.4 puts the pieces together, defining the semantics of the coeffect calculus along with structural rules. The semantics is illustrated by executing an example bounded-reuse program in the semantics (Example 8).

- Section 6.5.5 shows our semantics sound with respect to the syntactic equational theory of Section 6.4. This uses the derivation of categorical structures for the semantics as *lax homomorphisms* between structure on the coeffect category $\mathbb{I}$ and the base $\mathbb{C}$.

In this section, $\mathbb{C}, \mathbb{D}, \mathbb{I}$ range over categories. The objects of a category $\mathbb{C}$ are written $obj(\mathbb{C})$. The category of functors between $\mathbb{C}$ and $\mathbb{D}$ is written $[\mathbb{C}, \mathbb{D}]$. Exponential objects, representing function types in our model, are written in two ways, either B^A or $A \Rightarrow B$.

6.5.1 *Interpreting contexts and judgments*

The semantics is parameterised by a coeffect algebra, with scalar coeffects $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$, coeffect shape $(\mathcal{S}, [-], \diamond, \hat{0}, \hat{1})$, and $(\bar{\times}, \underline{\times}, \perp)$. An interpretation $[\![-]\!]$ is given to types, free-variable contexts, and type and coeffect judgments, with a base Cartesian-closed category $\mathbb{C}$ for denotations and a category $\mathbb{I}$ of scalar coeffects, where $obj(\mathbb{I}) = \mathcal{C}$. Since $\mathbb{C}$ is Cartesian-closed, we use the λ-calculus as the syntax for giving concrete definitions.

The interpretation $[\![-]\!]$ is parameterised by categorical structures which model a particular notion of context. The interpretation of free-variable contexts depends on shape, for which we give concrete definitions for flat and structural shapes.

INTERPRETING JUDGMENTS. type and coeffect judgments are interpreted (given denotations) as morphisms in $\mathbb{C}$, of the form:

$$[\![\Gamma @ R \vdash e : \tau]\!] : D_R^{[\Gamma]} [\![\Gamma]\!] \to [\![\tau]\!]$$

The interpretation is a morphism from an interpretation of the context Γ to the interpretation of the result. The functor $D_R^{[\Gamma]}$ over the context encodes the semantic notion of context and is indexed by the free-variable context shape $[\Gamma]$ and coeffect annotation R.

The structure D can be thought of as a coproduct of functors D^n for every possible shape $n \in \mathcal{S}$ of free-variable context:

$$D : \Sigma_{n:\mathcal{S}}.D^n \quad \text{where} \quad D^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]$$

For a fixed context shape n the functor $D^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]$ maps an n-indexed coeffect (think positions) 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.

TYPES. Types are interpreted as objects of $\mathbb{C}$, that is $[\![\tau]\!] : obj(\mathbb{C})$ where function types have the interpretation as exponents:

$$[\![\sigma \xrightarrow{r} \tau]\!] = D_r^{\hat{1}} [\![\sigma]\!] \Rightarrow [\![\tau]\!]$$

The parameter of a function is wrapped by a functor $D_r^{\hat{1}}$ that defines a context with singleton shape $\hat{1}$, matching the single value that it contains. This interpretation is shared by all coeffect calculi.

FREE-VARIABLE CONTEXTS. As described above, free-variable contexts Γ are given an interpretation as objects in $\mathbb{C}^{[\Gamma]}$. Thus, the interpretation of contexts is shape dependent.

We define $[\![-]\!]$ on free-variable context for structural and flat systems. For flat systems, there is only a single shape, so the interpretation is a product type inside the Cartesian closed category $\mathbb{C}$. For structural systems, the shape matches the number of variables and so the model is a value in the product category $\mathbb{C} \times \ldots \times \mathbb{C}$.

flat coeffects. Recall that $\mathcal{S} = \{\star\}$ and $[\Gamma] = \star$. Since the set of positions $\star$ is a singleton, then $\mathbb{C}^\star$ is isomorphic to $\mathbb{C}$. Therefore $[\![\Gamma]\!] : obj(\mathbb{C})$, which is defined as:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$$

Denotations of typing judgments in a flat coeffect system are thus of the form (where $R \in \mathbb{I}$):

$$[\![x_1 : \tau_1, .., x_n : \tau_n @ R \vdash e : \tau]\!] : D^\star_R([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$$

structural coeffects. Recall that $\mathcal{S} = \mathbb{N}$ and $[\Gamma] = |\Gamma|$ (number of free variables), thus $[\![\Gamma]\!] : obj(\mathbb{C}^{|\Gamma|})$. This is defined similarly to the above, but instead of using products in $\mathbb{C}$, we use the product of categories. Thus, denotations have the form:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n @ R \vdash e : \tau]\!] : D^n_R([\![\tau_1]\!], \ldots, [\![\tau_n]\!]) \to [\![\tau]\!]$$

where $|R| = n$ and we use commas (instead of $\times$) to denote the product of categories. This means that $D^n : \mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}]$ is a functor between an n-length vector of coeffects indices and an *n*-ary endofunctor. As mentioned, the key difference between the flat and structural interpretations of free-variable contexts is that flat uses products of objects in $\mathbb{C}$ and the structural uses products of $\mathbb{C}$ in the category of categories.

Example 1 (Bounded reuse). *Recall bounded reuse has coeffect scalars $\mathcal{C} = \mathbb{N}$ and shapes $\mathcal{S} = \mathbb{N}$. We model contexts by replicating the value of each variable so there is a value for each use. This matches the model used by Girard* et al. *[?]. Contexts are described by $\mathsf{B} : \Sigma_{n:\mathbb{N}}.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$, where for $R = \langle r_1, \ldots, r_n \rangle$:*

$$\mathsf{B}^n_R(A_1, \ldots, A_n) = A_1^{r_1} \times \ldots \times A_n^{r_n}$$
$$\mathsf{B}^n_R(f_1, \ldots, f_n) = \lambda\langle a_1, \ldots, a_n \rangle.\langle (f_1 \circ a_1), \ldots, (f_n \circ a_n) \rangle$$

Thus each object in the free-variable context A_i is indexed by its associated coeffect r_i. For the morphism mapping, $f_i : A_i \to B_i$ and $a_i : {A_i}^{r_i}$ thus $(f_i \circ a_i) : {B_i}^{r_i}$. The exponent ${A_i}^{r_i}$ can be read as a product of r_i copies of A_i, e.g.:

$$\mathsf{B}^3_{1,0,2}(A, B, C) = A^1 \times B^0 \times C^2 = (A) \times 1 \times (C \times C)$$

Example 2 (Implicit paramters). *Recall the implicit parameter calculus with scalar coeffects as sets of names paired with types $\mathcal{C} = \mathcal{P}(\mathsf{Name} \times \mathsf{Types})$ and flat shape with singleton $\mathcal{S} = \{\star\}$.*

Its contexts are defined by $\mathsf{I}^\star : \Sigma_{n:\{\star\}}.(\mathbb{I}^n \to [\mathbf{Set}^n, \mathbf{Set}])$, which is equivalent to $\mathbb{I} \to [\mathbf{Set}, \mathbf{Set}]$) and defined as follows:

$$\mathsf{I}^\star_R A = A \times [\![R]\!] \qquad \mathsf{I}^\star_R f = \lambda(a, r).(f\ a, r)$$

The interpretation $[\![R]\!]$ maps a set of variable-type pairs to an object representing a set of variables-values pairs in **Set***.*

6.5.2 *Sequential composition*

Following the usual categorical semantics approach, we require a notion of sequential composition for our denotations. We show first a special case for $D^{\hat{1}}$, where $\mathbb{I}^{\hat{1}} = \mathbb{I}$ and $\mathbb{C}^{\hat{1}} = \mathbb{C}$ in both flat and structural systems[2] and thus $D^{\hat{1}} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$. Composition of morphisms $f : D^{\hat{1}}_S A \to B$ and $g : D^{\hat{1}}_R B \to C$ is defined by an *indexed comonad* (which we previously introduced briefly [?]).

Definition 13. *An* indexed comonad *comprises a strict monoidal category* $(\mathbb{I}, \bullet, I)$ *and a functor* $F : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ *with two natural transformations (where we write* $(F\,R)\,A$ *as* $F_R A$*):*

$$(\delta_{X,Y})_A : F_{(X \bullet Y)}A \to F_X(F_Y A) \qquad (\varepsilon_I)_A : F_I A \to A$$

where δ *is called* comultiplication *and* ε *is called* counit. *We require indexed analogues of the usual comonad axioms* (cf. [?])*:*

$$\begin{tikzcd}
F_R \arrow[r, "\delta_{R,I}"] \arrow[d, "\delta_{I,R}"'] \arrow[dr, equal] \arrow[dr, phantom, "{[C2]}", near end, shift left=3] \arrow[dr, phantom, "{[C1]}", near start, shift right=3] & F_R F_I \arrow[d, "F_R \varepsilon_I"] \\
F_I F_R \arrow[r, "\varepsilon_I F_R"'] & F_R
\end{tikzcd}
\qquad
\begin{tikzcd}
F_{R \bullet S \bullet T} \arrow[r, "\delta_{R \bullet S, T}"] \arrow[d, "\delta_{R, S \bullet T}"'] \arrow[dr, phantom, "{[C3]}"] & F_{R \bullet S} F_T \arrow[d, "\delta_{R,S} F_T"] \\
F_R F_{S \bullet T} \arrow[r, "F_R \delta_{S,T}"'] & F_R F_S F_T
\end{tikzcd}$$

An indexed comonad $F : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ induces a notion of composition for all $f : F_S A \to B, g : F_R B \to C$:

$$g \hat{\circ} f = g \circ F_R f \circ \delta_{R,S} \ : \ F_{R \bullet S} A \to B$$

with the identity $\hat{id}_A = (\varepsilon_I)_A : F_I A \to A$ for all A. Thus indexed comonads induce a category which has the same objects as $\mathbb{C}$ and morphisms $\mathbb{C}_F(A, B) = \bigcup_{R \in \mathbb{I}} \mathbb{C}(F_R A, B)$. Note that an indexed comonad is not a family of (ordinary) comonads, because the identity is only defined for the functor F_I.

Therefore, if $D^{\hat{1}}$ is an indexed comonad, there is a notion of composition for denotations with a single coeffect index.

Example 3 (Bounded reuse). $B^{\hat{1}}_R$ *(Example 1) has an indexed comonad structure, where the monoid* $(\mathbb{N}, *, 1)$ *from the coeffect scalar for bounded reuse induces a monoidal category structure on* $\mathbb{I}$ *(with* $1 : \mathbb{I}$ *and the bifunctor* $* : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$*), with operations:*

$$\begin{aligned}
\varepsilon^{\hat{1}}_1 &= \lambda\langle a_1 \rangle . a_1 \\
\delta^{\hat{1}}_{R,S} &= \lambda\langle a_1 ..., a_{RS} \rangle . \\
&\qquad \langle\langle a_1 ..., a_S \rangle, \langle a_{S+1}, ..., a_{S+S} \rangle, ..., \langle a_{(R-1)S+1}, ..., a_{RS} \rangle\rangle
\end{aligned}$$

Indexed comonads essentially model single-variable contexts. The counit requires a single copy of the value from the context. The comultiplication splits R *times* S *copies of a value into* R *copies of a context where each context contains just* S *copies of the value.*

Remark 2. *A semantics for dataflow coeffects is similar to bounded reuse with* $D^n_R(A_1, \ldots, A_n) = (A_1 \times A_1{}^{R_1}) \times ... \times (A_n \times A_n{}^{R_n})$*, i. e., each free-variable has an extra value representing the "current" value. A dataflow indexed comonad is similar to the above but with additive rather than multiplicative behaviour.*

2 since $\hat{1} = 1$ in structural and $\hat{1} \cong 1$ in flat, i. e., $\star$ is isomorphic to 1

Example 4 (Implicit parameters). *For the coeffect scalar monoid* $(\mathcal{P}(\mathsf{Name} \times \mathsf{Types}), \cup, \emptyset)$ *of implicit parameters,* I^* *(Example 2) there is an indexed comonad structure, with operations:*

$$\varepsilon_{\emptyset}\,(a, \emptyset) \mapsto a \qquad\qquad \delta_{R,S}(a, \gamma) \mapsto ((a, \gamma|_S), \gamma|_R)$$

where $\gamma|_R = \{(x, v) \mid (x, v) \in \gamma, (x, t) \in R\}$ *filters incoming implicit parameters to those variable-value pairs where the variable is in the coeffect* R.

These two examples (which are new here) provide composition for context-dependent computations indexed by coeffects in a flat calculus. For structural coeffects, we need to compose morphisms which have more than a single coeffect annotation. For this, we introduced the notion of *structural indexed comonads*.

Definition 14. *A* structural indexed comonad *comprises a functor* $\mathsf{D} : \Sigma_{n:\mathcal{S}}.(\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$ *where* $(\mathbb{I}, \bullet, \mathrm{I})$ *is a strict monoidal category,* $\hat{1} \in \mathcal{S}$ *and* $\mathsf{D}^{\hat{1}} : \mathbb{I}^{\hat{1}} \to [\mathbb{C}^{\hat{1}}, \mathbb{C}]$ *is an indexed comonad (with* $(\delta^{\hat{1}}_{X,Y})_A : \mathsf{D}^{\hat{1}}_{X \bullet Y} A \to \mathsf{D}^{\hat{1}}_X \mathsf{D}^{\hat{1}}_Y A$ *and* $(\varepsilon_{\mathrm{I}})_A : \mathsf{D}^{\hat{1}}_{\mathrm{I}} A \to A$*) and a* structural comultiplication *natural transformation:*

$$(\delta^n_{r,S})_{A^n} : \mathsf{D}^n_{r \bar{\bullet} S} A^n \to \mathsf{D}^{\hat{1}}_r \mathsf{D}^n_S A^n$$

where $A^n \in \mathbb{C}^n$, $r \in \mathbb{I}$, $S \in \mathbb{I}^n$ *and* $\bar{\bullet} : \mathbb{I} \times \mathbb{I}^n \to \mathbb{I}^n$ *is the monoid left action for* $\bullet$ *lifting scalar coeffects to shaped coeffects (e.g., the scalar-vector version of* $\bullet$*). Analogous laws to monoid left actions for unitality and associativity hold for structural comultiplication:*

$$\varepsilon_{\mathrm{I}} \circ \delta^n_{\mathrm{I},r} = \mathrm{id} \qquad \mathsf{D}^{\hat{1}}_r \delta^n_{S,T} \circ \delta^n_{r, S \bar{\bullet} T} = \delta^{\hat{1}}_{r,S} \mathsf{D}^n_T \circ \delta^n_{r \bar{\bullet} S, T}$$

Note the indexed comonad comultiplication $\delta^{\hat{1}}$ *for associativity.*

Structural indexed comonads provide composition for morphisms $f : \mathsf{D}^n_S A^n \to B$ and singleton-shaped $g : \mathsf{D}^{\hat{1}}_r B \to C$:

$$g \hat{\circ} f = g \circ \mathsf{D}^{\hat{1}}_r f \circ \delta^n_{r,S} : \mathsf{D}^n_{r \bar{\bullet} S} A^n \to C$$

Note that this composition is asymmetric: the left morphism and right morphisms have different shapes. To compose morphisms which both have non-trivial context shapes requires additional structure for manipulating contexts (shown in the next section).

Example 5 (Bounded reuse). $\mathsf{B} : \Sigma_{n:\mathbb{N}}, (\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$ *has a structural indexed comonad structure with the indexed comonad* $\mathsf{B}^{\hat{1}}$ *(Example 3) and the following structural comultiplication:*

$$\begin{aligned}
&\delta^n_{r,S} = \lambda(\langle a^1_1, \ldots, a^1_{r*S_1}\rangle, \ldots, \langle a^n_1, \ldots a^n_{r*S_n}\rangle).\\
&\;(\,(\,\langle a^1_1, \ldots, a^1_{S_1}\rangle, \qquad\qquad\qquad \ldots, \langle a^n_1, \ldots, a^n_{S_n}\rangle),\\
&\quad(\,\langle a^1_{(S_1+1)}, \ldots, a^1_{(S_1+1)+S_1}\rangle, \ldots, \langle a^n_{(S_n+1)}, \ldots, a^n_{(S_n+1)+S_n}\rangle),\\
&\quad\;\ldots\\
&\quad(\,\langle a^1_{(r-1)*s_1+1}, \ldots, a^1_{r*S_1}\rangle, \quad \ldots, \langle a^n_{(r-1)*S_n+1}, \ldots, a^n_{r*S_n}\rangle))
\end{aligned}$$

The input is an n*-variable context containing* r *times* S_i *copies of* a^i *for each variable. The output has* r *copies of a single* n*-variable context containing* S_i *copies of* a^i *for each variable. Thus,* $\delta^n_{r,S}$ *partitions the incoming context into* r*-sized contexts.*

Note that in the case of the flat system, a structural indexed comonad collapses to a regular indexed comonad on $\mathsf{D}^{\hat{1}}$.

6.5.3 *Splitting and merging contexts*

Indexed comonads and structural indexed comonads give a semantics for sequential composition of contextual computations. However, this does not provide enough structure for a full semantics of the coeffect calculus. Core to the semantics of abstraction and application is the merging and splitting of contexts. Recall the free-variable contexts and coeffects in the (abs) and (app) rules:

$$\text{(app)}\,\frac{\Gamma_1 @ R \vdash e_1 \ldots \quad \Gamma_2 @ S \vdash e_2 \ldots}{\Gamma_1, \Gamma_2 @ R \underline{\times} (T \circledast S) \vdash e_1\ e_2 \ldots} \qquad \text{(abs)}\,\frac{\Gamma, v : \sigma @ R \overline{\times} \langle S \rangle \vdash e \ldots}{\Gamma @ R \vdash \lambda v.e \ldots : \sigma \xrightarrow{S} \ldots}$$

Reading (app) bottom-up, the context of the application is split into two contexts for each subterm e_1 and e_2. Reading (abs) bottom-up, the context of the abstraction is merged with the singleton context of the parameter. Capturing these notions in the denotational semantics requires some additional structure.

A (non-indexed) comonadic semantics for the λ-calculus requires a *monoidal comonad* with operation $m_{A,B} : FA \times FB \to F(A \times B)$ [**?**]. Previously, we defined a similar operation for the semantics of a flat coeffect system, with an indexed monoidal operation $m^{R,S}_{A,B}$ for merging contexts. Dually, contexts were split with $n^{R,S}_{A,B}$ [**?**]. We used two operations for combining and splitting the coeffect annotations, respectively. Here we generalize these as $\overline{\times}$ and $\underline{\times}$ using a shape-indexed version.

Definition 15. $D : \Sigma_{n:\mathcal{S}}, (\mathbb{I}^n \to [\mathbb{C}^n, \mathbb{C}])$ *is a* indexed lax (semi)monoidal functor *and/or* colax (semi)monoidal functor *if it has the following natural transformations respectively:*

$$m^{n,m}_{R,S} : D^n_R A \times D^m_S B \to D^{n \diamond m}_{R \overline{\times} S}(A \times B)$$
$$n^{n,m}_{R,S} : D^{n \diamond m}_{R \underline{\times} S}(A \times B) \to D^n_R A \times D^m_S B$$

In both, shape descriptions are combined by $\diamond$. The first operation models context merging and combines coeffects using $\overline{\times}$. The second models context splitting, with $\underline{\times}$ for the pre-split coeffect.

Example 6 (Bounded reuse)**.** *For bounded reuse,* B *is an indexed lax and colax semimonoidal functor with the following operations:*

$$m^{n,m}_{R,S} = \lambda(\langle a_1, ..., a_n \rangle \times \langle b_1, ..., b_m \rangle).(\langle a_1, ..., a_n \rangle, \langle b_1, ..., b_m \rangle)$$
$$n^{n,m}_{R,S} = \lambda(\langle a_1, ..., a_n \rangle, \langle b_1, ..., b_m \rangle).(\langle a_1, ..., a_n \rangle \times \langle b_1, ..., b_m \rangle)$$

i. e., $m^{m,m}_{R,S}$ *takes a pair of contexts and merges them simply by replacing the product in* $\mathbb{C}$ *which pairs the two arguments (written using* $\times$*) with products inside of* B *(written using tuple notation* (x, y)*). The operation* $n^{n,m}_{R,S}$ *is the inverse.*

Example 7 (Implicit parameters)**.** *For implicit parameters,* $I^\star$ *is an indexed lax and colax semimonoidal functor with operations:*

$$m^{\star,\star}_{R,S} = \lambda((a, \gamma_R), (b, \gamma_S)).((a, b), \gamma_R \cup \gamma_S)$$
$$n^{\star,\star}_{R,S} = \lambda((a, b), \gamma).((a, \gamma|_R), (b, \gamma|_S))$$

As in Example 4, $\gamma|_R$ *and* $\gamma|_S$ *restrict the set of implicit parameters* γ *to the variable-values pairs for variables in* R *and* S.

$$(\quad\text{var})\ \frac{}{[\![x : \tau@\langle\mathsf{use}\rangle \vdash x : \tau]\!] = \varepsilon_I : D^{\hat{1}}_{I}[\![\tau]\!] \to [\![\tau]\!]}$$

$$(\quad\text{abs})\ \frac{[\![\Gamma, x{:}\sigma@R\overline{\times}\langle s\rangle \vdash e : \tau]\!] = g : D^{n\diamond\hat{1}}_{R\overline{\times}\langle s\rangle}}{[\![\Gamma@R \vdash \lambda x.e : \sigma \xrightarrow{s} \tau]\!] = \Lambda(g \circ m^{n,\hat{1}}_{R,\langle s\rangle}) : D^{n}_{R}[\![}$$

$$(\quad\text{app})\ \frac{[\![\Gamma_1@R \vdash e_1 : \sigma \xrightarrow{t} \tau]\!] = g_1 : D^{n}_{R}[\![\Gamma_1]\!] \to (D^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!] \Rightarrow [\![\tau]\!]) \qquad [\![\Gamma_2@S \vdash e_2 : \tau]\!] = g_2 : D^{m}_{S}}{[\![\Gamma_1, \Gamma_2@R \underline{\times} (t \circledast S) \vdash e_1\, e_2 : \tau]\!] = \Lambda^{-1} g_1 \circ (\mathsf{id} \times (D^{\hat{1}}_{\langle t\rangle}\, g_2 \circ \delta^{m}_{\langle t\rangle,S})) \circ n^{n,m}_{R,t\circledast S} : D^{n\diamond m}_{R\underline{\times}(t\circledast S)}[\![}$$

$$(\quad\text{ctx})\ \frac{[\![\Gamma@R \vdash e : \tau]\!] = f : D^{n}_{R}[\![\Gamma]\!] \to [\![\tau]\!] \qquad [\![\Gamma'@R' \rightsquigarrow \Gamma@R, \emptyset]\!] = c : D^{m}_{R'}[\![\Gamma']\!] \to D^{n}_{R}[\![\Gamma}{[\![\Gamma'@R' \vdash e : \tau]\!] = f \circ c : D^{m}_{R'}[\![\Gamma']\!] \to [\![\tau]\!]}$$

(weak) $[\![\Gamma, x{:}\tau@R \underline{\times} \langle\mathsf{ign}\rangle \rightsquigarrow \Gamma@R, \emptyset]\!] = \pi_1 \circ n^{n,\hat{1}}_{R,\langle\mathsf{ign}\rangle} : D^{n\diamond\hat{1}}_{R\underline{\times}\langle\mathsf{ign}\rangle}([\![\Gamma]\!] \times [\![\sigma]\!]) \to D^{n}_{R}[\![\Gamma]\!]$

(contr) $[\![\Gamma_1, x{:}\tau, \Gamma_2@R \underline{\times} \langle s \oplus t\rangle \underline{\times} Q \rightsquigarrow \Gamma_1, y{:}\tau, z{:}\tau, \Gamma_2@R \overline{\times} \langle s\rangle \overline{\times} \langle t\rangle \overline{\times} Q, [y, z \mapsto x]]\!] = m^{n,\hat{1},m}_{R,\langle s\oplus t\rangle,Q} \circ (\mathsf{id} \times \Delta_{s,t} \times \mathsf{id}) \circ n^{n}_{R}$

(exch) $[\![\Gamma_1, y{:}\sigma, x{:}\tau, \Gamma_2@R \underline{\times} \langle t\rangle \underline{\times} \langle s\rangle \underline{\times} Q \rightsquigarrow \Gamma_1, x{:}\tau, y{:}\sigma, \Gamma_2@R \overline{\times} \langle s\rangle \overline{\times} \langle t\rangle \overline{\times} Q, \emptyset]\!] = m^{n,\hat{1},\hat{1},m}_{R,\langle s\rangle,\langle t\rangle,Q} \circ (\mathsf{id} \times \mathsf{swap} \times \mathsf{id}) \circ n^{n,}_{R,}$

where $\mathsf{swap} : A \times B \to B \times A$ **and** Λ, Λ^{-1} denote currying and uncurrying respectively

Figure 20: Denotational semantics for the coeffect calculus

6.5.4 *Putting it together*

The semantics of the structural coeffect calculus $[\![-]\!]$ is defined in Figure 20, using the structures described in the previous sections.

core rules. The denotation in (*var*) maps a context of the singleton shape $\hat{1}$ containing just a single variable τ (with coeffect I) to a τ value using the counit operation.

The premise of (*abs*) takes a context of shape $n \diamond \hat{1}$ with coeffects $R\overline{\times}\langle s\rangle$ and a free-variables context consisting of Γ and an additional variable x. The denotation $g : D^{n\diamond\hat{1}}_{R\overline{\times}\langle s\rangle}[\![\Gamma, \nu : \sigma]\!] \to [\![\tau]\!]$ is pre-composed with m, such that its context is obtained by merging the declaration-site context (Γ) and call-site context (σ):

$$g \circ m^{n,\hat{1}}_{R,\langle s\rangle} : (D^{n}_{R}[\![\Gamma]\!] \times D^{\hat{1}}_{\langle s\rangle}[\![\sigma]\!]) \to [\![\tau]\!]$$

This is uncurried to give a denotation from a context to an exponential object representing the abstraction, where the singleton-shaped context becomes the source of the exponential.

The application rule (*app*) has two sub-expressions for the function and argument, with denotations requiring two distinct contexts:

$$g_1 : D^{n}_{R}[\![\Gamma_1]\!] \to (D^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!] \Rightarrow [\![\tau]\!]) \qquad g_2 : D^{m}_{S}[\![\Gamma_2]\!] \to [\![\sigma]\!]$$

The target of g_1 is an exponential object with singleton shape for the parameter of type σ. To evaluate g_1 and g_2, the semantics of (*app*) splits the incoming context over Γ_1, Γ_2 using n:

$$D^{n\diamond m}_{R\underline{\times}(\langle t\rangle \bullet S)}([\![\Gamma_1]\!] \times [\![\Gamma_2]\!]) \xrightarrow{n^{n,m}_{R,\langle t\rangle \bullet S}} D^{n}_{R}[\![\Gamma_1]\!] \times D^{m}_{\langle t\rangle \bullet S}[\![\Gamma_2]\!]$$

Since e_2 computes the parameter for function e_1, the denotation g_2 must be sequentially composed with the parameter part of g_1. Thus, the structural indexed comonad is used with g_2 to compute the correct context for the parameter of g_1:

$$D^{m}_{\langle t\rangle \bar{\bullet} S}[\![\Gamma_2]\!] \xrightarrow{\delta^{m}_{t,S}} D^{\hat{1}}_{\langle t\rangle} D^{m}_{S}[\![\Gamma_2]\!] \xrightarrow{D^{\hat{1}}_{\langle t\rangle} g_2} D^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!]$$

This is composed with the previous equation by lifting to the right-component of the product:

$$D^{n}_{R}[\![\Gamma_1]\!] \times D^{m}_{S}[\![\Gamma_2]\!] \xrightarrow{\mathrm{id}\times(D^{\hat{1}}_{\langle t\rangle} g_2 \circ \delta^{m}_{t,S})} D^{n}_{R}[\![\Gamma_1]\!] \times D^{\hat{1}}_{\langle t\rangle}[\![\sigma]\!]$$

This equation computes the calling context and parameter context for the function e_1, which is then composed with the uncurried g_1 denotation as shown in the (*app*) rule in Figure 20.

STRUCTURAL RULES. In Figure 20, (*ctx*) composes the denotation of an expression with a transformation c providing the semantic structural rules. The semantics of structural rules are defined by using $n^{n,m}_{R,S}$ to split contexts, transforming the components, and merging the transformed contexts using $m^{n,m}_{R,S}$. The (*contr*) rule uses an additional operation which duplicates a variable inside a context:

$$\Delta_{r,s} : D^{\hat{1}}_{r\oplus s} A \to D^{\hat{1}\diamond\hat{1}}_{\langle r\rangle \bar{\times} \langle s\rangle}(A \times A)$$

Example 8. *We demonstrate the semantics with a concrete example for the bounded reuse calculus. Consider the following:*

$$f : \mathbb{Z} \xrightarrow{2} \mathbb{Z}, x : \mathbb{Z}@\langle 2,4\rangle \vdash (\lambda z.z + z)\ (f\ x)$$

We elide the function body's denotation prior to contraction, $g = [\![x : \mathbb{Z}, y : \mathbb{Z} \vdash (+x)y : \mathbb{Z}]\!]$[3]. *The term therefore has denotation:*

$$[\![@\langle\rangle \vdash \lambda z.(+z)z : \mathbb{Z} \xrightarrow{2} \mathbb{Z}]\!] = \Lambda(g \circ \Delta_{\langle 1\rangle,\langle 1\rangle} \circ m^{0,1}_{\langle\rangle,\langle 1\rangle}) \tag{4}$$

$$\begin{aligned}
&[\![f : \mathbb{Z} \xrightarrow{2} \mathbb{Z}, x : \mathbb{Z}@\langle 1,2\rangle \vdash fx : \mathbb{Z}]\!] \\
&\quad = \Lambda^{-1}\varepsilon_1 \circ (\mathrm{id} \times (D\varepsilon_1 \circ \delta^{1}_{2,\langle 1\rangle})) \circ n^{1,1}_{\langle 1\rangle,\langle 2\rangle} \\
&\quad = \Lambda^{-1}\varepsilon_1 \circ n^{1,1}_{\langle 1\rangle,\langle 2\rangle}
\end{aligned} \tag{5}$$

$$\begin{aligned}
&[\![f : \mathbb{Z} \xrightarrow{2} \mathbb{Z}, x : \mathbb{Z}@\langle 2,4\rangle \vdash (\lambda z.(+z)z)\ (fx) : \mathbb{Z}]\!] \\
&\quad = \Lambda^{-1}(4) \circ (\mathrm{id} \times D(5) \circ \delta^{2}_{2,\langle 2,1\rangle}) \circ n^{0,2}_{\langle\rangle,\langle 4,2\rangle} \\
&\quad = g \circ \Delta_{r,s} \circ m^{0,1}_{\langle\rangle,\langle 1\rangle} \circ (\mathrm{id} \times D(5) \circ \delta^{2}_{2,\langle 2,1\rangle}) \circ n^{0,2}_{\langle\rangle,\langle 4,2\rangle}
\end{aligned} \tag{6}$$

where (5) *and* (6) *are simplified. We "run" this semantics on some input, evaluating each step of the denotation as a function. We write context objects, e.g.,* $D^{2}_{\langle R,S\rangle}(A, B)$ *as* $\langle (a_1, ..., a_R), (b_1, ..., b_R)\rangle$ *and products of contexts in* $\mathbb{C}$, *e.g.,* $D^{n}_{R}A \times D^{m}_{S}B$, *as* $(a \times b)$.

3 the full semantics has $[\![+]\!] : D^{0}_{\langle\rangle} 1 \to (D^{1}_{1}\mathbb{Z} \Rightarrow (D^{1}_{1}\mathbb{Z} \Rightarrow \mathbb{Z}))$

$$\begin{array}{rll}
& \langle (f_1, f_2), (x_1, x_2, x_3, x_4) \rangle & : \mathsf{D}^2_{\langle 2,4 \rangle}((\mathsf{D}^1_{\langle 2 \rangle}\mathbb{Z} \Rightarrow \mathbb{Z}) \times \mathbb{Z}) \\
\xrightarrow{n^{0,2}_{\langle\rangle,\langle 2,4\rangle}} & \langle\rangle \times \langle (f_1, f_2), (x_1, x_2, x_3, x_4) \rangle & : \mathsf{D}^0_{\langle\rangle} 1 \times \mathsf{D}^2_{\langle 2,4 \rangle} (\textit{as above}) \\
\xrightarrow{id \times \delta^2_{2,\langle 1,2\rangle}} & \langle\rangle \times \langle\langle f_1, (x_1, x_2)\rangle, \langle f_2, (x_3, x_4)\rangle\rangle & \\
& & : \mathsf{D}^0_{\langle\rangle} 1 \times \mathsf{D}^1_{\langle 2 \rangle}\mathsf{D}^2_{\langle 2,1 \rangle}((\mathsf{D}^1_{\langle 2 \rangle}\mathbb{Z} \Rightarrow \mathbb{Z}) \times \mathbb{Z}) \\
\xrightarrow{id \times \mathsf{D}n^{1,1}_{\langle 1\rangle,\langle 2\rangle}} & \langle\rangle \times \langle \langle f_1 \rangle \times \langle x_1, x_2 \rangle, \langle f_2 \rangle \times \langle x_3, x_4 \rangle \rangle & \\
& & : \mathsf{D}^0_{\langle\rangle} 1 \times \mathsf{D}^1_{\langle 2 \rangle}(\mathsf{D}^1_{\langle 1 \rangle}(\mathsf{D}^1_{\langle 2 \rangle}\mathbb{Z} \Rightarrow \mathbb{Z}) \times \mathsf{D}^1_{\langle 2 \rangle}\mathbb{Z}) \\
\xrightarrow{id \times \mathsf{D}(\Lambda^{-1}\varepsilon_1)} & \langle\rangle \times \langle f_1 \langle x_1, x_2 \rangle, f_2 \langle x_3, x_4 \rangle \rangle & : \mathsf{D}^0_{\langle\rangle} 1 \times \mathsf{D}^1_{\langle 2 \rangle}\mathbb{Z} \\
\xrightarrow{m^{0,1}_{\langle\rangle,\langle 1\rangle}} & \langle f_1 \langle x_1, x_2 \rangle, f_2 \langle x_3, x_4 \rangle \rangle & : \mathsf{D}^1_{\langle 2 \rangle}\mathbb{Z} \\
\xrightarrow{\Delta^2_{\langle 1\rangle,\langle 1\rangle}} & \langle (f_1 \langle x_1, x_2 \rangle, f_2 \langle x_3, x_4 \rangle) \rangle & : \mathsf{D}^1_{\langle 1,1 \rangle}(\mathbb{Z} \times \mathbb{Z}) \\
\xrightarrow{g} & [\![+]\!] \; \langle f_1 \langle x_1, x_2 \rangle\rangle \; \langle f_2 \langle x_3, x_4 \rangle\rangle & : \mathbb{Z}
\end{array}$$

6.5.5 *Soundness, with respect to equational theory*

Our denotational semantics for the coeffect calculus is sound with respect the equational theory of Section 6.4. That is:

Theorem 8 (Soundness)**.**

$$\Gamma @ R \vdash e \equiv e' : \tau \Rightarrow [\![\Gamma @ R \vdash e : \tau]\!] \equiv [\![\Gamma @ R \vdash e' : \tau]\!]$$

Proof of this follows from an interesting result which we first unpack: determining whether $[\![\Gamma @ R \vdash e_1 : \tau]\!] \equiv [\![\Gamma @ S \vdash e_2 : \tau]\!]$ follows from a proof on coeffect annotations that $R = S$.

Lemma 9. *Every coeffect algebra axiom corresponds to an axiom of one of the categorical structures introduced here (indexed (structural) comonad or indexed (co)lax monoidal functor).*

For example, the monoid axiom $X \circledast \mathsf{use} = X$ for scalar coeffects corresponds to indexed comonad axiom $\mathsf{D}^{\hat{1}}_X \varepsilon_{\mathsf{use}} \circ \delta^{\hat{1}}_{X,\mathsf{use}} = id_{\mathsf{D}^{\hat{1}}_X}$ (which requires the monoid axiom to hold). This lemma follows from our derivation of the indexed categorical structures here. They are not derived *ad hoc* but systematically as (lax) *homomorphisms* (structure preserving maps) between the structure of coeffect annotations in $\mathbb{I}$ and the structure of denotations in $\mathbb{C}$.

Proposition 1. *An indexed comonad on* D *witnesses that* D *is a* colax monoid homomorphism *between the (strict) monoidal categories* $(\mathbb{I}, \circledast, \mathsf{use})$ *and* $([\mathbb{C}, \mathbb{C}], \circ, 1_{\mathbb{C}})$ *(endofunctor composition).*

Unpacking this, a *monoid homomorphism* maps between the underlying sets of two monoids, preserving the monoid structure of one into the other, i. e., given monoids $(X, \bullet, I)$ and $(Y, \otimes, E)$ then a monoid homomorphism is a mapping $F : X \to Y$ such that:

$$FX \otimes FY \equiv F(X \bullet Y) \qquad E \equiv FI$$

The axioms of each monoid are preserved trivially by these equalities, e. g., $FX \equiv F(X \bullet I) \equiv FX \otimes FI \equiv FX \otimes E \equiv FX$. In a categorical setting, monoids are taken on categories with binary operations as bifunctors. A homomorphism is *lax* if the above equalities are instead morphisms (which we say *witness* the homomorphism) and *colax* if these morphisms go in opposite direction. Thus, a colax monoid homomorphism is witnessed by:

$$\delta : FX \otimes FY \leftarrow F(X \bullet Y) \qquad \varepsilon : E \leftarrow FI$$

Note our choice of morphism names. F no longer preserves the monoid axioms *up to equality* but has axioms on δ and ε, e.g., $\mathsf{F}(X \bullet I) \xrightarrow{\delta} \mathsf{F}X \otimes \mathsf{F}I \xrightarrow{\mathrm{id} \otimes \varepsilon} \mathsf{F}X \otimes \mathsf{E}$ equals $\mathsf{F}X \xrightarrow{\mathrm{id}} \mathsf{F}X$.

Our indexed comonad definition is equivalent to D being a colax homomorphism between a monoidal category $(X, \bullet, I)$ and the monoidal category of $\mathbb{C}$ endofunctors $(Y, \otimes, \mathsf{E}) = ([\mathbb{C}, \mathbb{C}], \circ, 1_{\mathbb{C}})$, with endofunctor composition $\circ$ and the trivial endofunctor $1_{\mathbb{C}}$. The indexed comonads axioms are the axioms of the colax homomorphism. Equivalently, D is a *colax monoidal functor*.

A similar approach is taken to deriving the remaining structures.

Proposition 2. *A structural indexed comonad provides* $\delta^{n}_{R,S} : \mathsf{D}^{\hat{1}}_{r}\mathsf{D}^{n}_{S}A^{n} \leftarrow \mathsf{D}^{n}_{r \circledast S}A^{n}$ *which witnesses that* D *is a colax homomorphism between the following* monoid left-actions *for* $(\mathbb{I}, \circledast, \mathsf{ign})$ *and* $([\mathbb{C}, \mathbb{C}], \circ, 1_{\mathbb{C}})$*:*

$$\begin{array}{rcccl} (r : \mathbb{I}) & \circledast & (\langle s_1, ..., s_n \rangle : \mathbb{I}^n) & = & \langle r \circledast s_1, ..., r \circledast s_n \rangle : \mathbb{I}^n \\ (\mathsf{D}^{\hat{1}}_{r} : [\mathbb{C}, \mathbb{C}]) & \circ & (\mathsf{D}^{n}_{S} : [\mathbb{C}^n, \mathbb{C}]) & = & \mathsf{D}^{\hat{1}}_{r}\mathsf{D}^{n}_{S} : [\mathbb{C}^n, \mathbb{C}] \end{array}$$

The axioms are the lax versions of the monoid left-action laws.

The lax and colax indexed monoidal operations $\mathsf{m}^{n,m}_{R,S}$ and $\mathsf{n}^{n,m}_{R,S}$ follow a similar derivation but as lax and colax monoid homomorphism between composite monoids on coeffect annotations and shapes and $\times$ in $\mathbb{C}$. The details are elided here.

Returning to soundness, our semantics is therefore defined in terms of structures whose axioms correspond to axioms of the syntactic equational theory. Consequently, semantic proofs correspond to syntactic proofs, modulo naturality laws and product/exponent laws in $\mathbb{C}$. This result holds in the general coeffect calculus and semantics since every semantic structure has a unique corresponding structure on coeffect annotations (i. e., $(\mathcal{C}, \circledast, \mathsf{use})$ for sequential composition of unary denotations, $(\mathcal{C}, \veebar)$ for splitting contexts, $(\mathcal{C}, \barwedge)$ for joining contexts).

Example 9. *Section 6.4.1 showed* η*-equality for structural systems, which uses the properties (1)* $\veebar = \barwedge = \times$ *for structural systems and (2)* $s \circledast \langle \mathsf{use} \rangle = \langle s \circledast \mathsf{use} \rangle = \langle s \rangle$*. The semantics here is sound with respect to* η*-equality, the proof of which uses the corresponding axioms (1)* $\mathsf{n}^{n,m}_{R,S} \circ \mathsf{m}^{n,m}_{R,S} = \mathrm{id}$ *and (2)* $\mathsf{D}^{\hat{1}}_{s}\varepsilon_{\mathsf{use}} \circ \delta^{m}_{s,\mathsf{use}}$ *(structural indexed comonad unit law, Definition 14).*

The accompanying technical report shows the full semantic proofs for $\beta\eta$-equality whose structure corresponds exactly to the syntactic proofs with the corresponding coherence conditions [?].

6.6 RELATED WORK

We expand briefly on the overview of related work in Section 6.2.5.

The (*storage*) rule for bounded linear logic explains the contextual requirements induced by proposition reuse [?]:

$$(\text{storage})\frac{!_{\overline{Y}}\Gamma \vdash A}{!_{X\overline{Y}}\Gamma \vdash !_{X}A}$$

where $X\overline{Y} = \langle XY_1, .., XY_n \rangle$ is the scalar multiple of a vector. This rule is akin to the δ^n operation of structural indexed comonads, indeed, we can model it exactly using $\delta^{n}_{X,\overline{Y}}$ and the lifting D^{n}_{X}.

In BLL, the modality $!_X$ is a constructor and may appear both on the left- and right-hand sides of $\vdash$. In this paper, reuse bounds annotate typing rules, thus there is no constructor corresponding to bounded reuse in the language; reuse bounds are meta-level. Our choice to work at the meta-level means that the coeffect calculus provides a unified analysis and semantics to different notions of context and its term language is that of standard λ-calculus.

Previously we briefly introduced indexed comonads [**?**] without derivation. Here we derived indexed comonads as colax homomorphisms. This is dual to the *parametric effect monad* structure defined as a lax homomorphism [**?**]. Our semantics requires additional structure not needed for effects due to the asymmetry inherent in the λ-calculus.

The necessity modality $\Box$ in S4 logic corresponds to a comonad with lax monoidal functor structure $\mathsf{m} : \Box A \times \Box B \to \Box(A \times B)$. Bierman and de Paiva [**?**] defined a term language corresponding to a natural deduction S4, where contexts contain sequences of $\Box$-wrapped assumptions $x_1 : \Box A_1, \ldots x_n : \Box A_n$. Modelling these judgments does not require a context splitting operation n unlike in our approach. Our approach can be thought of as having a single $\Box$ modality over the context which allows both flat whole-context dependence and structural per-variable dependence.

6.7 CONCLUSIONS

In this paper, we looked at two forms of context-dependence analysis – *flat* coeffect systems that track whole-context requirements (such as implicit parameters, resources, or platform version) and *structural* coeffects that track per-variable requirements (such as usage or data access patterns). The newly introduced structural system makes applications such as liveness, bounded reuse, and dataflow analysis (from our earlier work) practically useful. With the move towards cross-platform systems running in diverse environments, analysing context dependence is vital for reasoning and compilation. The coeffect calculus provides a foundation for further study, similar to the type-and-effect discipline.

We presented the system together with its syntactic equational theory and categorical semantics. The equational theory is presented in order to explain how the systems work, but it also provides a basis for an operational semantics for concrete systems. Exploring these, and their connection to the denotational semantics, is further work.

7 COEFFECT META-LANGUAGE

Both flat coeffect calculus and structural coeffect calculus (presented in the past two chapters) use indexed comonads to define the semantics of the langauge. In this section, we follow the meta-language style and embed indexed comonads into the language – the type constructor $C^r \alpha$ becomes a first-class value and we add language constructs corresponding to primitive operations of the indexed comonad.

7.1 INTRODUCTION

7.2 TYPE SYSTEM

7.3 OPERATIONAL PROPERTIES

7.4 CATEGORICAL SEMANTICS

7.5 APPLICATIONS

7.5.1 *Meta-programming*

7.5.2 *Mobile computations*

7.6 RELATED WORK

This chapter is closely related to Contextual Modal Type Theory (CMTT) of Nanevski et al. However they develop their language using model logic as a basis, while we use categorical foundations as the basis - leading to a different system.

7.7 SUMMARY

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

$$(\mathit{app})\ \frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$(\mathit{abs})\ \frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}$$

$$(\mathit{letbox})\ \frac{\Gamma \vdash e_1 : C^{r \oplus s}\alpha \qquad \Gamma, x : C^{r}\alpha \vdash e_2 : \beta}{\Gamma \vdash \textbf{let box}\ x = e_1\ \textbf{in}\ e_2 : C^{s}\beta}$$

$$(\mathit{eval})\ \frac{\Gamma \vdash e : C^{e}\alpha}{\Gamma \vdash !e : \alpha}$$

$$(\mathit{sub})\ \frac{\Gamma \vdash e : C^{s}\alpha}{\Gamma \vdash e : C^{r}\alpha}\ (s \leqslant r)$$

Figure 21: Type system for the coeffect meta-language λ_{cm}

8 CONCLUSIONS

How to make this practical?
codo notation / computation expressions
embedding algebraic things

BIBLIOGRAPHY

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

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

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