CONTEXT-AWARE PROGRAMMING LANGUAGES

TOMAS PETRICEK

Computer Laboratory
University of Cambridge

2014

[March 14, 2014 at 3:26 – classicthesis version x]

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.

1.1 PROGRAMMING LANGUAGE AND INNOVATION

We briefly consider two practical concerns that led to the development of new programming languages. This helps to explain why context-aware programming is of importance, which we cover in the next section. The examples are by no means representative, but they illustrate the 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 [29], data-parallelism and also asynchronous computing [58].

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 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 [37] 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.

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[1].

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 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 [47].

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 **??** unifies the two abstractions.

1.2 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 [37], OpenCL or JavaScript [35]) a part of the compilation (e. g. SQL generation) 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.

1.2.1 *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 [25] 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[2]. 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

2 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[3]: *"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.

1.2.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use metaprogramming techniques (such as LINQ [37] or F# quotations [56]) 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.

1.2.3 *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.

3 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 [28].

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.

1.2.4 *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, wrap-

around behaviour we could pad the input with a known number of elements from the other side of the array.

1.3 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.

1.4 CONTEXT-AWARE LANGUAGES IN ACTION

So, how should a context-aware language look? Surely, there is a wide range of options, but I hope I convinced you that it needs to be *context-aware* in some way! I'll write my pseudo-example in a language that looks like F#, is fully statically typed and uses type inference.

I think that type inference is particularly important here - we want to check quite a few properties that should not that difficult to infer (If I call a function that needs GPS, I need to have GPS access!) Writing all such information by hand would be very cumbersome.

So, let's say that we want to write a client/server news reader where the news are stored in a database on a server. When a client (iPhone or Windows phone) runs, we get GPS location from the phone and query the server that needs to connect to the "News" database using a password defined somewhere earlier (perhaps loaded from a server-side config file):

```
let lookupNews(location) =
  let db = query("News", password)
  selectNews(db, location)
```

```
let readNews() =
  let loc = gpsLocation()
  remote {
    lookupNews(loc)
  }

let iPhoneMain() =
  createCocoaWidget(readNews)

let windowsMain() =
  createMetroWidget(readNews)
```

The idea is that `lookupNews` is a server-side function that queries the "News" database based on the specified `location`. This is called from the client-side by `readNews` which get the current GPS position and uses a `remote { .. }` block to invoke the `lookupNews` function remotely (how exactly would this be written is a separate question - but imagine a simple REST request here).

Then, we have two main functions, `iPhoneMain` and `windowsMain` that will serve as two entry points for iPhone and Windows build of the client-side application. They are both using a corresponding platform-specific function to build the user interface, which takes a function for reading news as an argument.

If you wanted to write and compile something like this today, you could use F# in Xamarin Studio to target iPhone and Window phone, but you'd either need two separate end-application projects or a large number of unmaintainable `#if` constructs. Why not just use a single project, if the application is fairly simple?

I imagine that a context-aware statically typed language would let you write the above code and if you inspected the types of the functions, you would see something like this:

```
password      :   string { sensitive }
lookupNews    :   Location -{ database }-> list<News>

gpsLocation   :   unit -{ gps }-> Location
readNews      :   unit -{ rpc, gps }-> Async<list<News>>

iPhoneMain    :   unit -{ cocoa, gps, rpc }-> unit
windowsMain   :   unit -{ metro, gps, rpc }-> unit
```

The syntax is something that I just made up for the purpose of this article - it could look different. Some information could even be mapped to other visual representations (e.g. blueish background for the function body in your editor). The key thing is that we can learn quite a lot about the context usage:

- `password` is available in the context, but is sensitive and so we cannot return it as a result from a function that is called via an RPC call. - `lookupNews` requires database access and so it can only run on the server-side or on a thick client with local copy of the database. - `gpsLocation` accesses GPS and since we call it in `readNews`, this function also requires GPS (the requirement is propagated automatically). - We can compile the program for two client-side platforms - the entry points require GPS, the ability to make RPC calls and Cocoa or Metro UI platform, respectively.

When writing the application, I want to be always able to see this information (perhaps similarly to how you can see type information in the various F# editors). I want to be able to reference multiple versions of base libraries - one for iPhone and another for Windows and see all the API functions at the same time, with appropriate annotations. When a function is available on both platforms, I want to be able to reuse the code that calls it. When some function is available on only one platform, I want to solve this by designing my own abstraction, rather than resorting to ugly `#if` pragmas.

Then, I want to take this single program (again, strutured using whatever abstractions I find appropriate) and compile it. As a result, I want to get a component (containing `lookupNews`) that I can deploy to the server-side and two packages for iPhone and Windows respectively, that reference only one or the other platform.

1.5 COEFFECTS: THEORY OF CONTEXT DEPENDENCE

If you're expecting a "Download!" button or (even better) a "Buy now!" button at the end of this article, then I'll disappoint you. I have no implementation that would let you do all of this. My work in this area has been (so far) on the theoretical side. This is a great way to understand what is *actually* going on and what does the *context* mean. And if you made it this far, then it probably worked, because I understood the problem well enough to be write a readable article about it!

1.5.1 *Brief introduction to type systems*

I won't try to reproduce the entire content of the thesis in this introduction – but I will try to give you some background in case you are interested (that should make it easier to look at the papers above). We'll start from the basics, so readers familiar with theory of programming languages can skip to the next section.

Type systems are usually described in the form of *typing judgement* that have the following form:

$$\Gamma \vdash e : \tau \tag{1}$$

The judgement means that, given some variables described by Γ, the expression or program e has a type τ. What does this mean? For example, what is a type of the expression `x + y`? Well, this depends - in F# it could be some numeric type or even a string, depending on the types of `x` and `y`. That's why we need the variable context Γ which specifies the types of variables. So, for example, we can have:

$$x : \mathtt{int}, y : \mathtt{int} \vdash x + y : \mathtt{int} \tag{2}$$

Here, we assume that the types of `x` and `y` (on the left hand side) are both `int` and as a result, we derive that the type of `x + y` is also an `int`. This is a valid typing, for the expression, but not the only one possible - if `x` and `y` were of type `string`, then the result would also be `string`.

1.5.2 *Checking what program does with effect systems*

Type systems can be extended in various interesting ways. Essentially, they give us an approximation of the possible values that we can get as a result. For example refinement types [**?**] can estimate numerical values more precisely (e.g. less than 100). However, it is also possible to track what a program does - how it *affects* the world. For example, let's look at the following expression that prints a number:

$$x : int \vdash print\ x : unit \tag{3}$$

This is a reasonable typing in F# (and ML languages), but it ingores the important fact that the expression has a *side-effect* and prints the number to the console. In Haskell, this would not be a valid typing, because `print` would return an `IO` computation rather than just plain `unit` (for more information see IO in Haskell [4].

However, monads are not the only way to be more precise about side-effects. Another option is to use effect system [**?**] which essentially annotates the result of the typing judgement with more information about the *effects* that occur as part of evaluation of the expression:

$$x : int \vdash print\ x : unit\&\{IO\} \tag{4}$$

The effect annotation is now part of the type - so, the expression has a type `unit & { io }` meaning that it does not return anything useful, but it performs some I/O operation. Note that we do not track what *exactly* it does - just some useful over-approximation. How do we infer the information? The compiler needs to know about certain language primitives (or basic library functions). Here, `print` is a function that performs I/O operation.

The main role of the type system is dealing with composition - so, if we have a function `read` that reads from the console (I/O operation) and a function `send` that sends data over network, the type system will tell us that the type and effects of `send (read ())` are `unit & io, network`.

Effect systems are a fairly established idea - and they are a nice way to add better purity checking to ML-like languages like F#. However, they are not that widely adopted (interestingly, checked exceptions in Java are probably the most major use of effect system). However, effect systems are also a good example of general approach that we can use for tracking contextual information...

1.5.3 *Checking what program requires with coeffect systems*

How could we use the same idea of *annotating* the types to capture information about the context? Let's look at a part of the program from the case study that I described earlier:

$$pass : string \vdash query(\text{"}News\text{"}, pass) : NewsDb \tag{5}$$

The expression queries a database and gets back a value of the `NewsDb` type (for now, let's say that `"News"` is a constant string and `query` behaves

4 http://www.haskell.org/haskellwiki/IO_inside

like the SQL type provider in F# [5] and generates the `NewsDb` type automatically).

What information do we want to capture? First of all, we want to add an annotation saying that the expression requires *database access*. Secondly, we want to mark the `pass` variable as *secure* to guarantee that it will not be sent over an unsecured network connection etc. The *coeffect typing judgement* representing this information looks like this:

$$pass : string^{\{secure\}}@\{database\} \vdash query("News", pass) : NewsDb \quad (6)$$

Rather than attaching the annotations to the *resulting type*, they are attached to the variable *context*. In other words, the equation is saying – given a variable `pass` that is marked as secure and additional environment providing database access, the expression `query("News", pass)` is well typed and returns a `NewsDb` value.

As a side-note, it is well known that *effects* correspond to *monads* (and Haskell uses monads as a way of implementing limited effect checking). Quite interestingly, *coeffects* correspond to the dual concept called *comonads* and, with a syntactic extension akin to the `do` notation or *computation expressions* [**?**], you could capture contextual propreties by adding comonads to a language.

1.6 SUMMARY

I started by explaining the motivation for my work - different problems that arise when we are writing programs that are aware of the context in which they run. The context includes things such as execution environment (databases, resources, available devices), platform and framework (different versions, different platforms) and meta-data about data we access (sensitivity, security, provenance).

This may not be preceived as a major problem - we are all used to write code that deals with such things. However, I believe that the area of *context-aware* programming is a source of many problems and pains - and programming languages can help!

In the second half of the article, I gave a brief introduction to *coeffects* – a programming language theory that can simplify dealing with context. The key idea is that we can use types to track and check additional information about the *context*. By propagating such information throughout the program (using type system that is aware of the annotations), we can make sure that none of the errors that I used as a motivation for this article happen.

5 http://www.pinksquirrellabs.com/post/2013/12/09/The-Erasing-SQL-type-provider.aspx

2 INTRODUCTION

In the rest of the chapter, we first look at the distinguishing factor of two of the presented coeffect systems (how they differ from other existing work). Then we look at a number of concrete problems that are discussed in greater details in later chapters. We use the examples to demonstrate the real-world problems that motivate our work. (Theoretical and other motivations are discussed later in Chapter 3).

2.1 CONTEXT AND LAMBDA ABSTRACTION

Our work on context-aware programming languages connects two directions in existing research on the theory of programming languages. On one side, effect systems [24] and monadic computations [38, 75] provide a detailed and established method for tracking what effects programs have – that is, how they affect the environment where they execute. On the other side, the work on comonadic notions of computations [65] shows how to use the mathematical dual of monads – comonads – to give categorical semantics of context-dependent computations.

Effect systems introduced track actions such as memory operations or communication. They are described by typing judgments of a form $\Gamma \vdash e : \alpha, \sigma$ where Γ is the context of a program (typically available variables), e is the expression (program) itself, α is the type of values returned by the program (e.g. integer or boolean) and σ is a set of possible effects. The judgment states that, given the context Γ, an expression has a type α and can only perform effects specified by the set σ. Wadler and Thiemann [75] explain how this shapes effect analysis of a lambda abstraction – that is, how effect systems analyze the effects associated with a definition of a function:

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

This means that, when a programmer defines a function, the system records that executing the function will perform the effects of the body of the function. However, simply defining a function is an effect-free computation. Tate [60] calls such effect systems *producer* effect systems and generalizes the idea of function to more general "thunking":

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

In contrast to the static analysis of (producer) effect systems, 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.

2.2 WHY COEFFECTS MATTER

This section gives three examples of context-dependent computations whose properties can be captured by the tree calculi presented in this thesis. We look at an example of the *flat coeffect calculus* (Chapter 4), *structural coeffect calculus* (Chapter 5) and *coeffect meta-language* (Chapter **??**).

2.2.1 *Flat coeffect calculus*

The flat coeffect calculus associates a single piece of information with the context. As an example, we look at a simple distributed programming language that includes the concept of *resources*. A resource may be accessed using a special construct **access** Res. The following example shows a function taht lists recent events – it accesses the resource News (representing a database) and a resource Clock (with the current time):

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

Consider a scenario where the function is *declared* on the server-side and then transferred and *executed* on the client-side. The resource News represents a database that is only available on the server-side and so the function needs to keep a remote reference to the server. However, the Clock resource may (or may not) be available on the client-side. If the resource is available on the client, then it may be re-bound and the function will use the current client's data – for example, to accommodate for time-zone changes.

This example demonstrates how lambda abstraction behaves for context-dependent computations. The context requirement of the function body is a set of resources $\{\mathsf{Clock}, \mathsf{News}\}$. The context requirements are split between the *declaration-site* and *call-site*. However, there are multiple possible splittings. The splitting $\{\mathsf{News}\} \cup \{\mathsf{Clock}\}$ models the case when the database is accessed from the server, but time is taken from the client, while the splitting $\{\mathsf{Clock}, \mathsf{News}\} \cup \{\}$ corresponds to the case when both resources are accessed from the server.

2.2.2 *Structural coeffect calculus*

The calculus used in the previous section annotates the entire context with a single value – such as the set of required resources. However, sometimes it is desirable to annotate not the entire context, but individual variables of the context.

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 $\mathsf{a}_{[\mathsf{x}]}$. To track information about individual variables, we use a product-like operation $\times$ on tags to mirrors the product structure of variables. For example:

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

The annotations 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, respectively. If we

substitute c for both a and b, we need to combine the context-requirements and take the maximum of the requirements of the individual variables:

$$(\mathsf{c} : \mathsf{stream})\ @\ \mathsf{max}(5, 10) \vdash \mathsf{c}_{[5]} + \mathsf{c}_{[10]} : \mathsf{nat}$$

This version of the calculus removes the non-determinism of the lambda abstraction from the previous version. As we associate information with individual variables, lambda abstraction creates a function that requires the context requirements associated with the variable that is being abstracted over.

For example, if we wrapped the earlier example above in a function taking a (and using b from the declaration-site) then the function would have context requirements 5 – that is the number associated with the variable a.

2.2.3 *Coeffect meta-language*

TODO: Give some example that uses the coeffects metalanguage style. This will probably be meta-programming with open expressions (similarly to what Pfenning and Nanevski do with contextual modal type theory).

2.3 WHY CONTEXT MATTERS

We claimed earlier that context-dependent comptuations are becoming increasingly common and our work is focused on annotating the (free-variable) context with additional information. The importance of context can be also demonstrated by looking at a technology that focuses solely on making the context richer – F# type providers.

TODO: Say more about type providers – they extend the context Γ so that it can be lazily loaded and it can be huge. Potentially, it could be also annotated with additional meta-data...

2.4 CONTRIBUTIONS

TODO: We present three calculi that model common notions of context-dependence and can be used as basis for developing context-aware programming langauges with static type systems.

3 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 3.2) and extend the duality to category theory, looking at the categorical dual of monads known as *comonads* (Section 3.3). Finally we look at logically inspired type systems that are closely related to our structural coeffects (Section 3.4).

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.

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 [31], and the subject has also been addressed in mobile computations [7, 17]. In programming languages, Costanza [13] develops a domain-specific LISP-like language ContextL and Bardram [5] 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 [34] 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)
```

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:

```
(?width :: Int) ⇒ String → ((?size :: Int) ⇒ String → 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* [74]. 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

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

that the function takes a hidden parameter – a dictionary that provides the operation $+ :: \alpha \to \alpha \to \alpha$. Thus, the type $\mathsf{Num}\ \alpha \Rightarrow \alpha \to \alpha$ can be viewed as $(\mathsf{Num}_\alpha \times \alpha) \to \alpha$. 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. [8, 52]. 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 [69].

Links [12], F# libraries [57, 45], ML5 and QWeSST [39, 51] and Hop [35] 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 [12], functions can be annotated as client-side, server-side and database-side. F# WebTools [45] 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].

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 [39] suggest that a contextual treatment is more appropriate. We return to ML5 when discussing semantics in Section 3.3.3.

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 [20].

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:

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

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

The type system for safe locking associates 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.

2 Android Developer guide [16] demonstrates how difficult it is to solve the problem without language support.

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 [71] 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* [6] build on the data-flow paradigm. The following example is inspired by the Lustre [26] 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 [70] originally proposed to use monads, while Uustalu and Vene later used comonads [64].

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) [3] 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 [72] 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:

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

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:

$$\textbf{let}\ \mathsf{defaultArg} = \lambda \mathsf{cond} \to \lambda \mathsf{input} \to$$
$$\quad \textbf{if}\ \mathsf{cond}\ \textbf{then}\ 42\ \textbf{else}\ \mathsf{input}$$

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:

$$\textbf{let}\ \mathsf{offsetZip} = 0\ \textbf{fby}\ (\mathsf{left} + \textbf{prev}\ \mathsf{right})$$

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

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 [11], 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 [67] and a well known attack known as SQL injection [28, 27]. 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. [68] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [50] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the FlowCaml [53] language. Finally, Russo et al. and Swamy et al. [49, 55] 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 $\{L, H\}$ represents low and high secrecy, respectively with $L \leqslant H$ 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:

```
if x > 0 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) [1]. It captures other cases where some information about expression relies on properties of variables in the context where it executes. The DCC captures, for example, *binding time analysis* [61], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [62] 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. [14] 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 [63] who developed an *effect system* (discussed in Section 3.3.2) 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:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{calculate} = \lambda \mathsf{input} \rightarrow \\
\quad \textbf{letrgn}\ \rho\ \textbf{in} \\
\quad \textbf{let}\ \mathsf{x} = \textbf{ref}_{\rho}\, \mathsf{input}\ \textbf{in}\ !\mathsf{x}
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{calculate} = \lambda \mathsf{input} \rightarrow \\
\quad \textbf{letrgn}\ \rho\ \textbf{in} \\
\quad \textbf{let}\ \mathsf{x} = \textbf{ref}_{\rho}\, \mathsf{input}\ \textbf{in}\ \mathsf{x}
\end{array}
$$

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) [22, 30] is the ability to update programs at runtime without stopping them. The Proteus system developed by Stoyle et al. [54] 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.

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

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

Figure 2: Simple effect system

3.2 THROUGH TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [24, 36], 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 [59], communication in message-passing systems [32] and atomicity in concurrent applications [21].

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 [19] refers to this approach as *descriptive*[3].

SIMPLE EFFECT SYSTEM The structure of a simple effect system is demonstrated in Figure 2. 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 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 3 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.

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

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

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

Figure 3: Simple effect system

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 2, 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).

3.3 THROUGH LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [38] 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 $f : \alpha \to M\beta, g : \beta \to M\gamma$*:*

$$\mathsf{bind}\ \mathsf{unit} = \mathsf{id} \qquad \textit{(left identity)}$$
$$\mathsf{bind}\ f \circ \mathsf{unit} = f \qquad \textit{(right identity)}$$
$$\mathsf{bind}\ (\mathsf{bind}\ g \circ f) = (\mathsf{bind}\ f) \circ (\mathsf{bind}\ g) \qquad \textit{(associativity)}$$

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.

3.3.1 *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"* [38, 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"* [38, 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[4].

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

4 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.

pret 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.

3.3.2 *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 [75]. 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 [60]) and further developed by Atkey [4] 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.

3.3.3 *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 [66] and more generally, notions of *context-dependent computations* [65]. 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. [2] 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 [10] use *computational* comonads for intensional semantics[5]. In functional programming, Kieburtz [33] 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 [9], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [48]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [41], 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* $(C\alpha \to \beta) \to (C\alpha \to C\beta)$

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

$$\mathsf{cobind}\ \mathsf{counit} = \mathsf{id} \qquad \textit{(left identity)}$$

$$\mathsf{counit} \circ \mathsf{cobind}\ f = f \qquad \textit{(right identity)}$$

$$\mathsf{cobind}\ (\mathsf{cobind}\ g \circ f) = (\mathsf{cobind}\ f) \circ (\mathsf{cobind}\ g) \qquad \textit{(associativity)}$$

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 [44].

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 calcu-

5 The structure of computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [46].

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

Figure 4: Typing for a comonadic language with contextual staged computations

lates 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:

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

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:

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

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).

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[6].

6 In fact, Pfenning and Davies [48, 41] never mention comonads explicitly. This is done in later work by Gabbay et al. [23], 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.

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. [39, 40] 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 $\Box\alpha \to \alpha$). Finally, Davies and Pfenning [15] 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 4 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 $\mathsf{counit} : C^{\emptyset}\alpha \to \alpha$. 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 $\mathsf{cobind} : (C^{\Psi}\alpha \to \beta) \to C^{\Psi,\Phi}\alpha \to C^{\Phi}\beta$. It means that given a term which requires variable context Ψ, Φ (expression 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

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

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

Figure 5: Exchange, weakening and contraction typing rules

monads). The relation between CMTT and comonads has been suggested by Gabbay et al. [23], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our let box typing rule from Figure 4 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.

3.4 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 [76] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler [73] 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 [42], 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 to the development of separation logic [43] (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 5. 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.

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

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

Figure 6: Exchange, weakening and contraction rules for bunched typing

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 6. 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 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 [42].

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.

3.5 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.

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* [59] provide a static analysis of effects and *monads* [38] provide a unified semantics to different notions of effect. Wadler and Thiemann unify the two approaches [75], *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 [65] (the dual of monads) and by languages derived from modal logic [48, 41], 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 [65]. 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 [24], track *effects* of computations, such as memory access or message-based communication [32]. 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 [75]:

> *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 $C^s\tau_1 \to \tau_2$ 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 [34] 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 [34], 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 7 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:

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

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-

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

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

Figure 7: 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. 8 detects whether all variables are dead ($C^{\mathsf{D}}\Gamma$) or whether at least one variable is live ($C^{\mathsf{L}}\Gamma$). Variable access (*var*) is annotated with L and constant access with D. That is, if $c \in \mathbb{N}$ then $C^{\mathsf{D}}\Gamma \vdash c : \mathsf{int}$. A dead context may be marked as live by letting $\mathsf{D} \sqsubseteq \mathsf{L}$ 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 $r = s = \mathsf{L}$). 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. 7 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 $r = \mathsf{L}$ and does not contain value when $r = \mathsf{D}$).

4.1.3 *Efficient dataflow.*

Dataflow languages (e. g. [71]) declaratively describe computations over streams. In *causal* data flow, program may access past values – in this setting, a func-

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

Figure 8: Selected coeffect rules for liveness analysis

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

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

Figure 9: Selected coeffect rules for causal data flow

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

Figure 9 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 [65] 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):

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

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 $C^r\tau_1 \to \tau_2$. 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.

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

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

Figure 10: 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 $S = \{\mathsf{D}, \mathsf{L}\}$ where $\mathsf{D} \sqsubseteq \mathsf{L}$. Variables are annotated with the top element $e = \mathsf{L}$ 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 10 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 $C^{r}\tau_1 \to \tau_2$.

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 [38]. Dually, Uustalu and Vene showed that (*monoidal*) *comonads* capture various kinds of context-dependent computation [65].

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 [65]. 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* $A \in \mathrm{obj}(\mathcal{C})$ *then* $C^r A \in \mathrm{obj}(\mathcal{C})$ *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:*

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

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

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

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

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

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 [65] additionally require a (*lax semi-*) *monoidal comonad* structure, which provides a monoidal operation $m : CA \times CB \to C(A \times B)$ for merging contexts (used in the semantics of abstraction).

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

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

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

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

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

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

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

EXAMPLE Uustalu and Vene model causal dataflow computations using the non-empty list comonad $\mathsf{NEList}\ A = A \times (1 + \mathsf{NEList}\ A)$ [65]. 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.

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

Figure 11: Categorical semantics for the coeffect calculus

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

4.3.2 *Categorical Semantics.*

Figure 11 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 $\mathsf{m}_{r,s}$ 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 $\mathsf{m}_{r,s}$ match the coeffects of the terms. The semantics of application is more complex. It first duplicates the free-variable values inside the context and then splits this context using $\mathsf{n}_{r,s\oplus t}$. The two contexts (with different coeffects) are passed to the two subexpressions, where the argument subexpression, passed a context $(s \oplus t)$, is coextended to produce a context s which is passed into the parameter of the function subexpression (*cf.* given $f : A \to (B \to C)$, $g : A \to B$, then $\mathit{uncurry}\, f \circ (\mathrm{id} \times g) \circ \Delta : A \to C$).

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

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 [48]) 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 [18, 38].

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:

$$\frac{C^s\Gamma \vdash e_1 : \tau_1 \qquad C^{r_1 \wedge r_2}(\Gamma, x : \tau_1) \vdash e_2 : \tau_2}{C^{r_1 \vee (r_2 \oplus s)}\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2} \tag{7}$$

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

$$\frac{C^s\Gamma \vdash e_1 : \tau_1 \qquad C^r(\Gamma, x : \tau_1) \vdash e_2 : \tau_2}{C^{r \vee (r \oplus s)}\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2} \tag{8}$$

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 (8). However, we also consider the cases when a syntactic substitution $e_2[x \leftarrow e_1]$ has the coeffects specified by the above rule (8) 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 ($r \vee e = e$); in the case of resources, e is the *least* element ($r \vee e = r$) 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 [34].

4.5 RELATED AND FURTHER WORK

This paper follows the approaches of effect systems [24, 59, 75] and categorical semantics based on monads and comonads [38, 65]. 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 [65] and indexed monads of Atkey [4], Wadler and Thiemann [75]. Interestingly, *indexed* comonads are *more general* than comonads, capturing more notions of context-dependence (§4.1).

COMONADS AND MODAL LOGICS. Bierman and de Paiva [9] 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 [48, 9, 41], building a *metalanguage* (akin to Moggi's monadic metalanguage [38]) that includes $\Box$ as a type constructor. For example, in [48], 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 [38, 65]. We follow the semantic approach.

Nanevski et al. extend an S4 term language to a *contextual* modal type theory (CMTT) [41]. 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:

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

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

The structural coeffect system is related to bunched typing [**?**] (but generalizes it by adding indices). We are currently investigating how to use structural coeffects to capture fine-grained context-dependence properties such as secure information flow [68] 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 $C^r\tau_1 \to \tau_2$, 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 $a_{[x]}$. To track information about individual variables, we use a product-like operation $\times$ on tags to mirrors the product structure of variables. For example:

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

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:

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

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:

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

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

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: $C^{15\times(10\vee(0\times 0))}(a, (a, b))$. Using a simple normalization mechanism (described later), this can be further reduced to $C^{(15\vee 10)\times 10}(a, b)$. 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:*

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

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 $C^r\Gamma \Rightarrow_c C^r\Gamma$.

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 $C^s\tau_1 \to \tau_2$.

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 $C^{r_1}\Gamma_1 \Rightarrow_c C^{r_2}\Gamma_2$ 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 $C^r\Gamma \vdash e : \alpha$

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

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

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

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

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

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

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

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

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

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

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

Lemma 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:

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

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 $C^{r\times s}\Gamma, v : \tau$). 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 13. Using contexts with holes, we can now formulate the general substitution lemma for λ_{Cs}.

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

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

Figure 13: 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 4 (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 14: 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* $\mathrm{obj}(\mathcal{C}) \times \mathrm{obj}(\mathcal{C})$ *to an object* $\mathrm{obj}(\mathcal{C})$ *and families of mappings:*

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

And with a family of mappings $\iota_{r,s} : T^r A \to T^s A$ *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:

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

The type of the $m_{r,s}$ 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 $n_{r,s}$, which serves as the dual of $m_{r,s}$. 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 $n_{r,s}$ 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 $\hat{T}^r$ in the same way as in Section **??**. However, we still cannot freely manipulate the variables in the context. Given a context modelled as $T^{r \times s}(A \hat{\times} B)$, we can lift a morphism f to $\hat{T}^{r \times s}(f)$, 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 $T^r A$ from $T^r(A \hat{\times} B)$.

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 14. 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 $m_{r,s}$).

The rule λ_{Cs} -Sapp now uses the operation $n_{s,(r \vee t)}$ 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:

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

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: (...)

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] 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.

[29] 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.

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

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

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

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

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

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

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

[37] 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.

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

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

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

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

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

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

[44] D. Orchard. Programming contextual computations.

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

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

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

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

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

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

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

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

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

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

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

[56] 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.

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

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

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

[60] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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