CONTEXT-AWARE PROGRAMMING LANGUAGES

TOMAS PETRICEK

Computer Laboratory
University of Cambridge

2014

ABSTRACT

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

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

We identify a number of program properties that were not connected before, but model some notion of context. Our examples include tracking different execution platforms (and their versions) in cross-platform development, resources available in different execution environments (e. g. GPS sensor on a phone and database on the server), but also more traditional notions such as variable usage (e. g. in liveness analaysis and linear logics) or past values in stream-based data-flow programming.

Our first contribution is the discovery of the connection between the above examples and their novel presentation in the form of calculi (*coeffect systems*). The presented type systems and formal semantics highlight the relationship between different notions of context. Our second and third contributions are two unified coeffect calculi that capture commonalities in the presented examples. In particular, our *flat coeffect calculus* models languages with contextual properties of the execution environment and our *structural coeffect calculus* models languages where the contextual properties are attached to the variable usage.

Although the focus of this thesis is on the syntactic properties of the presented systems, we also discuss their category-theoretical motivation. We introduce the notion of an *indexed* comonad (based on dualisation of the well-known monad structure) and show how they provide semantics of the two coeffect calculi.

CONTENTS

1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

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

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

PARALLEL PROGRAMMING. The rise of multi-core CPUs is a clear example of an external development influencing programming language research. As multi-core and multi-processor systems became ubiquitous, 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 [33], data-parallelism and also asynchronous computing [69].

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 became a standard very quickly and so the lack of good language abstractions was apparent.

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

Technologies like LINQ [44] make querying data significantly easier. But perhaps because accessing data became important more gradually, it was not easy to see that SQL queries, embedded as parameterized strings[2], are a poor solution *before* better approaches were developed.

This is even more the case for *type providers* – a recent feature in F# that integrates external data sources directly into the type system of the language and thus makes data explorable directly from the source code editor (through features such as auto-completion on object members). It is not easy to see the limitations of standard techniques (using HTTP requests to query REST services or parsing CSV files and using string-based lookup) until one sees just how much type providers change the data-scientist's workflow[3].

[1] In the UK, the open government data portal is available at: `http://data.gov.uk/`

[2] The dominant approach is demonstrated, for example, by a review of SQL injection prevention techniques by Clarke [14]

[3] 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 [56].

CONTEXT-AWARE PROGRAMMING. In this thesis, 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*.

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 expose such flaws. We look at a number of basic programs that rely on contextual information, we explain why the currently dominant solutions are inappropriate and then briefly outline how this thesis solves the problems.

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 in context-dependent programming are. In Chapter 4 and Chapter 5, we develop two simple calculi to understand and capture the structure of those problems and, finally, Chapter **??** unifies the two abstractions.

1.1 WHY CONTEXT-AWARE PROGRAMMING MATTERS

The phrase *context in which programs are executed* sounds rather abstract and generic. What notions of *context* can be identified in modern software systems? Different environments provide different resources (e. g. a database or GPS sensors), environments are increasingly diverse (e. g. different mobile platforms with multiple partially incompatible versions). 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 environment even more heterogeneous. At the same time, applications access rich data sources and need to be aware of provenance information and respect the security policies 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.** Libraries such as LINQ [44] let developers write code in a host language like C# and then cross-compile it to multiple targets (including SQL, OpenCL or JavaScript [42]). Part of the compilation (e. g. generating the SQL query) occurs at runtime and developers have no guarantee that it will succeed until the program is executed, because only subset of the host language is supported.

- **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 may have permission 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.

```
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

- **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). 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 have to be solved. However, this is because it is not apparent 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 4 examples in more detail. The examples are chosen to demonstrate two distinct forms of contexts that are studied in this thesis – first two are related to the program environment and the latter two are associated with individual variables of the program.

1.1.1 *Context awareness #1: Platform versioning*

The diversity across devices means that developers need to target an increasing number of platforms and possibly also multiple versions of each platform. For Android, there is a number called API level [29] 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 instance methods and properties 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[4]. 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 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[5] introducing the technique "Remember the mantra: if you haven't tried it, it doesn't work." Again, it would be reasonable to expect that statically-typed languages could provide a better solution.

1.1.2 *Context awareness #2: System capabilities*

Another example related to the previous one is when libraries use meta-programming techniques (such as LINQ [44] or F# quotations [67]) to translate code written in a subset of a host language to some other target language, such as SQL, OpenCL or JavaScript. For database access, this is a recently developed technique replacing embedded SQL discussed in the introduction, but it is a more generally important technique for programming in heterogeneous environments. It lets developers targets multiple runtimes that have limited execution capabilities.

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

```
var db = new NorthwindDataContext();

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

This appears as a perfectly valid code and 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 the 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 results (44100) for a LINQ error message *"Method X has no supported translation to SQL"* caused by a similar limitation.

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

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

1.1.3 *Context awareness #3: Confidentiality and provenance*

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

For example, consider the following code sample that accesses a database by building a SQL query using string concatenation. For the purpose of the demonstration, this example does not use LINQ, but an older approach with a parameterized SQL query written as a string:

```
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* [14] (an attacker could enter `"'; DROP TABLE Products --"` as their `name` and delete the database table "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 information about the value should be propagated to `query`. The `SqlCommand` object should then require arguments that can not directly contain user input (in an unchecked form).

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 the security context, such marking of values (but at run-time) is called *tainting* [32], but the technique is a special case of more general *provenance* tracking. This can be useful when working with data in other contexts. For example, data jounralsts might want to propagate meta-data about the quality and the information source – is the source trustworthy? Is the data up-to-date? Such meta-data could propagate to the result and tell us important information about the calculated results.

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

The final 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]`. We focus on writing stencil computations (such as image blurring, Conway's game of life or convolution) where all arrays are of the same size and the system provides a *cursor* pointing to a current location in the stencil. We assume that the keyword **cursor** returns the current location in the stencil.

The following example implements a simple one-dimensional cellular automaton, 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 $\langle -1, -1 \rangle$, $\langle 0, 0 \rangle$ and $\langle 1, 1 \rangle$ (based on the three accesses on the first line).

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

1.2 TOWARDS CONTEXT-AWARE LANGUAGES

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

The different applications 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 software framework (or library) 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.

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

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

let iPhoneMain() =
  createiPhoneListing(fetchLocalNews)

let windowsMain() =
  createWindowsListing(fetchLocalNews)
```

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

1.2.1 *Context-aware languages in action*

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

We assume that the language supports cross-compilation and splits the single program into three components: one for the server-side and two for the client-side, for iPhone and Windows platforms, respectively. The cross-compilation could be done in a way similar to Links [15], but we do not require explicit annotations specifying the target platform.

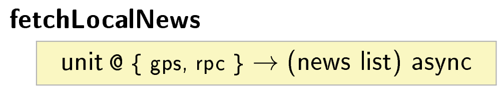
If we were writing the code using current mainstream technologies, we would have to create three completely separate components. The server-side would include the fetchNews function, which queries the database. The iPhone version would include fetchLocalNews, which gets the current GPS location and performs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

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

1.2.2 *Understanding context with types*

The approach advocated in this thesis is to track information about context requirements using the type system. To make this practical, the system needs to provide at least partial support for automatic type inference, as the information about context requirements makes the types more complex. An inspiring example might be the F# support for units of measure [38] – the user has to explicitly annotate constants, but the rest of the information is inferred automatically.

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



fetchLocalNews

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

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

The following summary shows the types of the functions from the code sample in Figure 2. These guide code generation by specifying which function should be compiled for which of the platforms, but they also provide documentation for the developers. In addition to function annotations, we also show the annotation attached to the password variable:

password	:	string @ sensitive
fetchNews	:	location @ { database } → news list
gpsLocation	:	unit @ { gps } → location
fetchLocalNews	:	location @ { gps, rpc } → news list
iPhoneMain	:	unit @ { ios, gps, rpc } → unit
windowsMain	:	unit @ { windows, gps, rpc } → unit

The example combines two separate notions of context. The variable password is annotated with a single (per-variable) annotation specifying tainting while functions are annotated with a set of resource requirements.

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

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

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

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

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

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

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

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

1.3 THEORY OF CONTEXT DEPENDENCE

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

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

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

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

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

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

The judgement states that the expression print x has a type unit. This is correct, but it ignores the important fact that the expression has a *side-effect* and prints a number to the console. In purely functional languages, this would not be possible. For example, in Haskell, the type would be IO unit meaning that the result is a *computation* that performs I/O effects and then returns unit value. Here, we look at another option for tracking effects, which is to extend the judgement with additional information about the effects. The judgement in a language with effect system would look as follows:

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

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

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

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

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

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

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

$$\begin{aligned}&(\mathsf{password}\!:\!\mathsf{string} @ \mathsf{sensitive}, \mathsf{cmd}\!:\!\mathsf{string}) @ \{\mathsf{database}\}\\ &\qquad \vdash \mathsf{query}(\mathsf{cmd}, \mathsf{password}) : \mathsf{news\ list}\end{aligned}$$

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

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

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

1.4 THESIS OUTLINE

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

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

- In Chapter 2 we give an overview of related work. Most importantly, we show that the idea of context-aware computations can be naturally approached from a number of directions developed recently in theories of programming languages (including type and effect systems, categorical semantics and sub-structural logics).

- Chapter 3 presents the first contribution of the thesis – the discovery of the connection between a number of existing programming language features that are related to context. The chapter presents type systems and semantics for a number of systems and analyses (including dataflow, liveness analysis, distributed programming and Haskell's type classes). Our novel presentation reveals their similarity.

- In Chapter 4 and Chapter 5, we present the key contributions of this thesis. We develop the *flat* and *structural* calculi, show how they capture important contextual properties and develop their categorical semantics using a notion based on comonads. Chapter **??** links the two systems using a single formalism that is capable of capturing both flat and structural properties and also discusses an alternative presentation that is more suitable for automatic type inference of coeffects.

- Related work is presented in Chapter 2 and throughout the thesis. One important direction deserves further exploration, and so Chapter **??** starts with a brief discussion of a different approach to tracking contextual information that arises from modal logics. Finally, the rest of Chapter **??** discusses approaches for implementing the presented theory in main-stream programming languages and concludes.

If there is a one thing that the reader should remember from this thesis, it is the fact that there is a unified notion of *context*, capturing many common scenarios in programming, and that programming language designers need to provide ways for working with this context (using *coeffects* or not). This greatly reduces the number of distinct concepts that software developers need to keep in mind of when building applications for the rich and diverse execution environments of the future.

2 PATHWAYS TO COEFFECTS

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

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

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

2.1 THROUGH TYPE AND EFFECT SYSTEMS

Introduced by Gifford and Lucassen [27, 43], 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 [70], communication in message-passing systems [36] and atomicity in concurrent applications [24].

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 [22] refers to this approach as *descriptive*[1].

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

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

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

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

Figure 3: Simple effect system

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

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

Figure 4: Simple effect system

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

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

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

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

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

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

2.2 THROUGH LANGUAGE SEMANTICS

Another pathway to coeffects leads through the semantics of effectful and context-dependent computations. In a pioneering work, Moggi [45] showed that effects (including partiality, exceptions, non-determinism and I/O) can be modelled uisng the category theoretic notion of *monad*.

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

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

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

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

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

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

2.2.1 *Effectful languages and meta-languages*

Moggi uses monads to define two formal systems. In the first formal system, a monad is used to model the *language* itself. This means that the semantics of a language is given in terms of a one specific monad and the semantics can be used to reason about programs in that language. To quote *"When reasoning about programs one has only one monad, because the programming language is fixed, and the main aim is to prove properties of programs"* [45, 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"* [45, p. 5].

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

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

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

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

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

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

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

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

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

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

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

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

2.2.2 *Marriage of effects and monads*

The work on effect systems and monads both tackle the same problem – representing and tracking of computational effects. The two lines of research have been joined by Wadler and Thiemann [86]. 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 [71]) and further developed by Atkey [5] in his work on *parameterized monads*.

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

2.2.3 *Context-dependent languages and meta-languages*

The theoretical parts of this thesis extend the work of Uustalu and Vene who use comonads to give the semantics of data-flow computations [77] and more generally, notions of *context-dependent computations* [76]. 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. [3] 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 [11] use *computational* comonads for intensional semantics[3]. In functional programming, Kieburtz [39] 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 [10], linking programming languages derived from modal logics to comonads. One such language has been reconstructed by Pfenning and Davies [58]. Nanevski et al. extend this work to Contextual Modal Type Theory (CMTT) [48], 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:

TODO: Maybe remove this from here, we repeate it in section 4

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* $C\alpha \to \alpha$

[3] The structure of computational comonad has been also used by the author of this thesis to abstract evaluation order of monadic computations [55].

- cobind *is a mapping* $(\mathsf{C}\alpha \to \beta) \to (\mathsf{C}\alpha \to \mathsf{C}\beta)$

such that, for all $f : \mathsf{C}\alpha \to \beta$ *and* $g : \mathsf{C}\beta \to \gamma$:

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

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

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

TODO: The following text is duplicated in chapter 4

The definition is similar to monad with "reversed arrows". Intuitively, the counit operation extracts a value α from a value that carries additional context $\mathsf{C}\alpha$. The cobind operation turns a context-dependent function $\mathsf{C}\alpha \to \beta$ 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 [51].

LANGUAGE SEMANTICS To demonstrate the approach of Uustalu and Vene, we consider the non-empty list comonad $\mathsf{C}\alpha = \mu\gamma.\alpha + (\alpha \times \gamma)$. 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:

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

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

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

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

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:

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

The semantics follows that of effectful computations using monads. A variable access is interpreted using $\mathsf{counit}_{\mathsf{C}}$ (obtain the value and ignore addi-

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

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

tional 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[4].

In languages inspired by modal logics, types can have the form $\Box\alpha$. In the work of Pfenning and Davies, this means a term that is provable with no assumptions. In distributed programming language ML5, Murphy et al. [46, 47] 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 [18] consider staged computations and interpret $\Box\alpha$ as a type of (unevaluated) expressions of type α.

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

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

- (*eval*) corresponds to $\mathsf{counit} : C^{\emptyset}\alpha \to \alpha$. It means that we can evaluate a closed (unevaluated) term and obtain a value. Note that the rule

[4] In fact, Pfenning and Davies [58, 48] never mention comonads explicitly. This is done in later work by Gabbay et al. [26], 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.

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 monads). The relation between CMTT and comonads has been suggested by Gabbay et al. [26], but the meta-language employed by CMTT does not directly correspond to comonadic operations. For example, our $\mathsf{let\ box}$ typing rule from Figure 5 is not a primitive of CMTT and would correspond to $\mathsf{box}(\Psi, \mathsf{letbox}(e_1, x, e_2))$. Nevertheless, the indexing in CMTT provides a useful hint for adding indexing to the work of Uustalu and Vene.

2.3 THROUGH SUB-STRUCTURAL AND BUNCHED LOGICS

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

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

In this section, we look at type systems that reconsider Γ in a number of ways. First of all, sub-structural type systems [87] restrict the use of variables in the language. Most famously linear type systems introduced by Wadler

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

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

Figure 6: Exchange, weakening and contraction typing rules

[84] 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 [49], 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 [50] (starting a fruitful line of research in software verification), but it is also interesting on its own.

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

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

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

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

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

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

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

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

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

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

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 $\alpha \rightarrow \beta$ 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 [49].

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

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

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

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

2.4 CONTEXT ORIENTED PROGRAMMING

The importance of context-aware computations is perhaps most obvious when considering mobile application, client/server web applications or even the internet of things. A pioneering work in the area using functional languages has been done by Serrano [62, 42] (which also inspired the example presented in Chapter 1). His HOP language supports cross-compilation and programs execute in different contexts. However, HOP is not statically type checked.

In the software engineering community, a number of authors have addressed the problem of context-aware computations. Hirschfeld et al. propose *Context-Oriented Programming* (COP) as a methodology [35]. The COP paradigm has been later implemented by programming language features.

Costanza [16] develops a domain-specific LISP-like language ContextL and Bardram [6] proposes a Java framework for COP.

Finally, the subject of context-awareness has also been addressed in work focusing on the development of mobile applications [8, 20]. Here, the *context* focuses more on concrete physical context (obtained from the device sensors) than context as an abstract language feature.

We approach the problem from a different perspective, building on the tradition of statically-typed functional programming languages and their theories.

2.5 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.

3 CONTEXT-AWARE SYSTEMS

Software developers as well as programming language researchers choose abstractions based not just on how appropriate they are. Other factors may include social aspects – how well is the abstraction known, how well is it documented and whether it is a standard tool of the *research programme*[1]. This may partly be why no unified context tracking mechanism has been developed so far.

In Chapter 1, we argued that context-awareness had, so far, only limited influence on the design of programming languages because it is a challenge that is not easy to see. However, many of the properties that this thesis treats uniformly as *coeffects* have been previously tracked by other means. This includes special-purpose type systems, systematic approaches arising from modal logic S4, as well as techniques based on abstractions designed for other purpose, most frequently monads.

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

3.1 STRUCTURE OF COEFFECT SYSTEMS

When introducing coeffect systems in Section 1.3, we related coeffect systems with effect systems. Effect systems track how program affects the environment, or, in other words capture some *output impurity*. In contrast, coeffect systems track what program requires from the environment, or *input impurity*.

Effect systems generally use judgements of the form $\Gamma \vdash e : \tau \,\&\, \sigma$, associating effects σ with the output type. In contrast, we choose to write coeffect systems using judgements of the form $\Gamma @ \sigma \vdash e : \tau$, associating the context requirements with Γ. Thus, we extend the traditional notion of free-variable context Γ with richer notions of context. Besides the notation, there are more important differences between effects and coeffects.

3.1.1 *Lambda abstraction*

The difference between effects and coeffects becomes apparent when we consider lambda abstraction. The typical lambda abstraction rule for effect systems looks as (*abs-eff*) in Figure 8. Wadler and Thiemann [86] explain how the effect analysis works as follows:

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

[1] Research programme, as introduced by Lakatos [40], is a network of scientists sharing the same basic assumptions and techniques.

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

Figure 8: Lambda abstraction for pure and effectful computations

This is the key property of *output impurity*. The effects are only produced when the function is evaluated and so the effects of the body are attached to the function. A recent work by Tate [71] uses the term *producer* effect systems for such standard systems and characterises them as follows:

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

The thunking is typically performed by a lambda abstraction – given an effectful expression e, the function $\lambda x.e$ is an effect free value (thunk) that delays all effects. As shown in the next section, contextual properties do not follow this pattern.

3.1.2 *Notions of context*

We look at three notions of context. The first is the standard free-variable context in λ-calculus. This is well understood and we use it to demonstrate how contextual properties behave. Then we consider two notions of context introduced in this thesis – *flat coeffects* refer to overall properties of the environment and *structural coeffects* refer to properties attached to individual variables.

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

The standard lambda abstraction (*abs-pure*), shown in Figure 8, splits the free-variable context of an expression into two parts. The value of the parameter has to be provided by the *call site* (dynamic scope) and the remaining values are provided by the *declaration site* (lexical scope). Here, the splitting is determined syntactically – the notation $\lambda x.e$ names the variable whose value comes from the call site.

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

FLAT COEFFECTS. In Section 1.2.1, we used *resources* in a distributed system as an example of flat coeffects. These could be, for example, a database, GPS sensor or access to the current time. We also outlined that such context requirements can be tracked as part of the typing assumption, for example, say we have an expression e that requires GPS coordinates and the current time. The context of such expression will be $\Gamma @ \{\, \mathsf{gps}, \mathsf{time} \,\}$.

The interesting case is when we construct a lambda function $\lambda x.e$, marshall it and send it to another node. In that case, the context requirements can be satisfied in a number of ways. When the same resource is available at the target machine (e. g.. current time), we can send the function with a context requirement and *rebind* the resource. However, if the resource is not available (e. g.. GPS on the server), we need to capture a *remote reference*.

In the example discussed here, $\lambda x.e$ would require GPS sensor from the declaration site (lexical scope) where the function is declared, which is attached to the current context as $\Gamma @ \{ \mathsf{gps} \}$. The current time is required from the caller of the function. So, the context requirement on the call site (dynamic scope) will be $r = \{ \mathsf{time} \}$. In coeffect systems, we attach this information to the function writing $\tau_1 \xrightarrow{r} \tau_2$.

We look at resources in distributed programming in more details in Section 3.2.2. The important point here is that in flat coeffect systems, contextual requirements are *split* between the call-site and declaration-site. Furthermore, in case of distributed programming, the resources can be freely distributed between the two sites.

STRUCTURAL COEFFECTS. On the one hand, variable context provides a *fine-grained tracking* mechanism of how context (variables) are used. On the other hand, flat coeffects let us track *additional information* about the context. The purpose of *structural coeffects* is to reconcile the two and to provide a way for fine-grained tracking of additional information related to variables in programs.

In Section 1.1.4, we used an example of tracking array access patterns. For every variable, the additional coeffect annotation keeps a range of indices that may be accessed relatively to the current cursor. For example, consider an expression $x[\mathbf{cursor}] = y[\mathbf{cursor} - 1] + y[\mathbf{cursor} + 1]$.

Here, the variable context Γ contains two variables, both of type Arr. This means $\Gamma = x{:}\mathsf{Arr},\ y!!\mathsf{Arr}$. For simplicity, we treat $\mathbf{cursor}$ as a language primitive. The coeffect annotations will be $(0,0)$ for x and $(-1,1)$ for y, denoting that we access only the current value in x, but we need access to both left and right neighbours in the y array. In order to unify the flat and structural notions, we attach this information as a *vector* of annotations associated with a *vector* of variable and write: $x{:}\mathsf{Arr},\ y{:}\mathsf{Arr} @ \langle (0,0), (-1,1) \rangle$. The unification is discussed in Chapter **??**.

Unlike in flat coeffects, in the structural systems, splitting of context determined by the syntax. For example, consider a function that takes y and contains the above body: $\lambda y.x[\mathbf{cursor}] = y[\mathbf{cursor} - 1] + y[\mathbf{cursor} + 1]$. Here, the declaration site contains x and needs to provide access at least within a range $(0,0)$. The call site provides a value for y, which needs to be accessible at least within $(-1,1)$. In this way, structural coeffects remove the non-determinism of flat coeffect systems.

Before looking at concrete flat and structural systems in mode details, we briefly overview some notation used in this thesis. As structural coeffects keep annotations as *vectors*, we use a number of operations related to scalars and vectors.

3.1.3 *Scalars and vectors*

The λ-calculus is asymmetric – it maps a context with *multiple* variables to a *single* result. An expression with free variables of types τ_i can be modelled by a function $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ with a product on the left, but a single value on the right. Effect systems attach effect annotations to the result τ. In coeffect systems, we attach a coeffect annotation to the context $\tau_1 \times \ldots \times \tau_n$.

Structural coeffects have one coeffect annotation per each variable. Thus, the annotation consists of multiple values – one belonging to each variable.

To distinguish between the overall annotation and individual (per-variable) annotations, we call the overall coeffect a *vector* consisting of *scalar* coeffects. This asymmetry also explains why coeffect systems are not trivially dual to effect systems.

It is useful to clarify how vectors are used in this thesis. Suppose we have a set $\mathcal{C}$ of *scalars* such that $r_1, \ldots, r_n \in \mathcal{C}$. A vector R over $\mathcal{C}$ is a tuple $\langle r_1, \ldots, r_n \rangle$ of scalars. We use bold-face letters like $\mathbf{r}, \mathbf{s}, \mathbf{t}$ for vectors and lower-case letters r, s, t for scalars[2]. We also say that a *shape* of a vector $\mathit{len}(\mathbf{r})$ (or more generally any container) is the set of *positions* in a vector. So, a vector of length n has shape $\{1, 2, \ldots, n\}$. We discuss containers and shapes further in Chapter **??** and also discuss how our use relates to containers of Abbott, Altenkirch and Ghani [2].

Just as in the usual multiplication of a vector by scalar, we lift any binary operation on scalars into a scalar-vector one. For any binary operation on scalars $\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, we define $s \circ \mathbf{r} = \langle s \circ r_1, \ldots, s \circ r_n \rangle$. Relations on scalars can be also lifted to vectors. Given two vectors $\mathbf{r}, \mathbf{s}$ of the same shape with positions $\{1, \ldots, n\}$ and a relation $\propto \subseteq \mathcal{C} \times \mathcal{C}$ we define $\mathbf{r} \propto \mathbf{s} \Leftrightarrow (r_1 \propto s_1) \wedge \ldots \wedge (r_n \propto s_n)$ Finally, we often concatenate vectors – for example, when joining two variable contexts. Given vectors $\mathbf{r}, \mathbf{s}$ with (possibly different) shapes $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, the associative operation for concatenation $\times$ is defined as $\mathbf{r} \times \mathbf{s} = \langle r_1, \ldots, r_n, s_1, \ldots, s_m \rangle$.

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

3.2 FLAT COEFFECT SYSTEMS

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

In this section, we look at a number of examples ranging from Haskell's type constraints and implicit parameters to distributed computing. For three of our examples – implicit parameters, liveness analysis and data-flow – we show an ad-hoc type system that captures their properties. This serves as a basis for Chapter 4, which develops a unified flat coeffect calculus.

3.2.1 *Implicit parameters and type classes*

Haskell provides two examples of flat coeffects – type class and implicit parameter constraints [85, 41]. Both of the features introduce additional *constraints* on the context requiring that the environment provides certain operations for a type (type classes) or that it provides values for named implicit parameters. In the Haskell type system, constraints C are attached to the types of top-level declarations, such as let-bound functions. The Haskell notation $\Gamma \vdash e : C \Rightarrow \tau$ corresponds to our notation $\Gamma @ C \vdash e : \tau$.

In this section, we present a type system for implicit parameters in terms of the coeffect typing judgement. We briefly consider type classes, but do no give a full type system.

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

IMPLICIT PARAMETERS. Implicit parameters are a special kind of variables that support dynamic scoping. They can be used to parameterise a computation (involving a long chain of function calls) without passing parameters explicitly as additional arguments of all involved functions.

The dynamic scoping means that if a function uses a parameter ?param then the caller of the function must set a value of ?param before calling the function. However, implicit parameters also support lexical scoping. If the parameter ?param is available in the lexical scope where a function (which uses it) is defined, then the function will not require a value from the caller.

A simple language with implicit parameters has an expression ?param to read a parameter and an expression[3] **letdyn** ?param $= e_1$ **in** e_2 that sets a parameter ?param to the value of e_1 and evaluates e_2 in a context containing ?param.

The fact that implicit parameters support both lexical and dynamic scoping becomes interesting when we consider nested functions. The following function does some pre-processing and then returns a function that builds a formatted string based on two implicit parameters ?width and ?size:

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

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

To keep examples in this chapter uniform, we do not use the Haskell notation and instead write $\tau_1 \xrightarrow{r} \tau_2$ for a function that requires implicit parameters specified r. In a purely dynamically scoped system, they would have to be defined when the user invokes the nested function. However, implicit parameters behave as a combination of lexical and dynamic scoping. This means that the nested function can capture the value of ?width and require just ?size. The following shows the two options:

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

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

This is not a complete list of possible typings, but it demonstrates the options. The *dynamic* case requires the parameter ?width twice (this may be confusing when the caller provides two different values). In the *mixed* case, the nested function captures the ?width parameter available from the declaration site. As a result, the latter function can be called as follows:

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

For different typings of format, different ways of calling it are valid. This illustrates the point made in Section 3.1.1 – flat coeffect systems may introduce certain non-determinism in the typing. The following section shows how this looks in the type system for implicit parameters.

TYPE SYSTEM. Figure 9 shows a type system that tracks the set of expression's implicit parameters. The type system uses judgements of the form

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

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

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

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

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

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

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

Figure 9: Coeffect rules for tracking implicit parameters

$\Gamma @ r \vdash e : \tau$ meaning that an expression e has a type τ in a free-variable context Γ with a set of implicit parameters specified by r. The annotations r, s, t are sets of pairs consisting of implicit parameter names and types, i. e. $r, s, t \subseteq \mathsf{Names} \times \mathsf{Types}$. The expressions include $?\mathsf{param}$ to read implicit parameter and **letdyn** to bind an implicit parameter. The types are standard, but functions are annotated with the set of implicit parameters that must be available on the call-site, i. e. $\tau_1 \xrightarrow{s} \tau_2$.

Accessing an ordinary variable (*var*) does not require any implicit parameters. The rule that introduces primitive context requirements is (*param*) – accessing a parameter $?\mathsf{param}$ of type τ requires it to be available in the context. The context may provide more (unused) implicit parameters thanks to the (*sub*) rule.

When we read the rules from the top to the bottom, application (*app*) and let binding (*let*) simply union the context requirements of the sub-expressions. However, lambda abstraction (*abs*) is where the example differs from effect systems. The implicit parameters required by the body $r \cup s$ can be freely split between the declaration-site ($\Gamma @ r$) and the call-site ($\tau_1 \xrightarrow{s} \tau_2$).

The union operation $\cup$ is not a disjoint union, which means that the values for implicit parameters can also be provided by both sites. For example, consider a function with a body $?\mathsf{a} + ?\mathsf{b}$. Assuming that the function takes and returns int, the following list shows 4 out of 9 possible valid typing. Full typing derivations can be found in Appendix ?:

$$\begin{array}{rclcll}
\Gamma @ \{?\mathsf{a} : \mathsf{int}\} & \vdash & \lambda x.?\mathsf{a} + ?\mathsf{b} & : & \mathsf{int} \xrightarrow{\{?\mathsf{b}:\mathsf{int}\}} \mathsf{int} & (1)\\
\Gamma @ \{?\mathsf{b} : \mathsf{int}\} & \vdash & \lambda x.?\mathsf{a} + ?\mathsf{b} & : & \mathsf{int} \xrightarrow{\{?\mathsf{a}:\mathsf{int}\}} \mathsf{int} & (2)\\
\Gamma @ \{?\mathsf{a} : \mathsf{int}\} & \vdash & \lambda x.?\mathsf{a} + ?\mathsf{b} & : & \mathsf{int} \xrightarrow{\{?\mathsf{a}:\mathsf{int}, ?\mathsf{b}:\mathsf{int}\}} \mathsf{int} & (3)\\
\Gamma @ \emptyset & \vdash & \lambda x.?\mathsf{a} + ?\mathsf{b} & : & \mathsf{int} \xrightarrow{\{?\mathsf{a}:\mathsf{int}, ?\mathsf{b}:\mathsf{int}\}} \mathsf{int} & (4)
\end{array}$$

The first two examples demonstrate why the system does not have the principal typing property. Both (1) and (2) are valid typings and they may both be desirable in certain contexts where the function is used.

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

The semantics is defined inductively over the typing derivation:

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

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

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

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

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

Monadic semantics using the reader monads differs as follows:

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

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

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

Figure 10: Semantics of a language with implicit parameters

SEMANTICS. Implicit parameters can be implemented by passing around a hidden dictionary that provides values to the implicit parameters. Accessing a parameter then becomes a lookup in the dictionary and the **letdyn** construct extends the dictionary. To elucidate how such hidden dictionaries are propagated through the program when using lambda abstractions and applications, we present a simple semantics for implicit parameters. The goal here is not to prove properties of the language, but simply to provide a better explanation. A detailed semantics in terms of indexed comonads is shown in Chapter 4.

For simplicity, we assume that all implicit parameters have the same type σ. In that setting, coeffect annotations r are just sets of names, i. e. $r, s, t \subseteq \mathsf{Names}$. Given an expression e of type τ that requires free variables Γ and implicit parameters r, our interpretation is a function that takes a product of variables from Γ together with a hidden dictionary of implicit parameters and returns τ:

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

The hidden dictionary is represented as a function from r to σ. This means that it provides a σ value for all implicit parameters that are required according to the typing. Note that the domain of the function is not the set of all possible implicit parameter names, but only a set of those that are specified by the type system.

A hidden dictionary also needs to be attached to all functions. A function $\tau_1 \xrightarrow{s} \tau_2$ is interpreted by a function that takes τ_1 together with a dictionary that defines values for implicit parameters in s:

$$[\![\tau_1 \xrightarrow{s} \tau_2]\!] = \tau_1 \times (s \to \sigma) \to \tau_2$$

The definition of the semantics is shown in Figure 10. Let binding can be viewed as a syntactic sugar for $(\lambda x.e_2)\ e_1$ and so it is omitted for brevity.

The (*var*) and (*param*) rules are simple – they project the appropriate variable and implicit parameter, respectively.

When an expression requires implicit parameters r, the semantics always provides a dictionary defined *exactly* on r. To achieve this, the (*sub*) rule restricts the function to r' (which is valid because $r' \subseteq r$).

The most interesting rules are (*abs*) and (*app*). In abstraction, we get two dictionaries f and g (from the declaration-site and call-site, respectively), which are combined and passed to the body of the function. The semantics prefers values from the call-site, which is captured by the $\uplus$ operation. In application, we first evaluate the expression e_1, then e_2 and finally call the returned function. The three calls use (possibly overlapping) restrictions of the dictionary as required by the static types.

Without providing a proof here, we note that the semantics is sounds with respect to the type system – when evaluating an expression, it provides it with a dictionary that is guaranteed to contain values for all implicit parameters that may be accessed. This can be easily checked by examining the semantic rules (and noting that the restriction and union always provide the expected set of parameters).

MONADIC SEMANTICS. Implicit parameters are related to the *reader monad*. The type $\tau_1 \times (r \to \sigma) \to \tau_2$ is equivalent to $\tau_1 \to ((r \to \sigma) \to \tau_2)$ through currying. Thus, we can express the function as $\tau_1 \to M\tau_2$ for $M\tau = (r \to \sigma) \to \tau$. Indeed, the reader monad can be used to model dynamic scoping. However, there is an important distinction from implicit parameters. The usual monadic semantics models fully dynamic scoping, while implicit parameters combine lexical and dynamic scoping.

The (*rdabs*) rule in Figure 10 shows a semantics that matches the usual monadic semantics using the reader monad. Note that the declaration-site dictionary is ignored and the body is called with only the dictionary provided by the call-site. This is a consequence of the fact that monadic functions are always pure values created using *unit*.

As we discuss later in Section 4.6.2, the reader monad can be extended to model rebinding. However, later examples in this chapter, such as liveness in Section 3.2.3 show that other context-aware computations cannot be captured by *any* monad.

TYPE CLASSES. Another type of constraints in Haskell that is closely related to implicit parameters are *type class* constraints [85]. They provide a principled form of ad-hoc polymorphism (overloading). When a code uses an overloaded operation (e.g. comparison or numeric operators) a constraint is placed on the context in which the operation is used. For example:

$$\begin{array}{l}\mathsf{twoTimes} :: \mathsf{Num}\ \alpha \Rightarrow \alpha \to \alpha \\ \mathsf{twoTimes}\ x = x + x\end{array}$$

The constraint $\mathsf{Num}\ \alpha$ on the function type arises from the use of the $+$ operator. Similarly to implicit parameters, type classes can be implemented using a hidden dictionary. In the above case, the function $\mathsf{twoTimes}$ takes a hidden dictionary that provides an operation $+$ of type $\alpha \times \alpha \to \alpha$.

Type classes could be modelled as a coeffect system. The type system would annotate the context with a set of required type classes. The typing of the body of $\mathsf{twoTimes}$ would look as follows:

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

Similarly, the semantics of a language with type class constraints can be defined in a way similar to implicit parameters. The interpretation of the body is a function that takes α together with a hidden dictionary of operations: $\alpha \times \mathsf{Num}_{\alpha} \rightarrow \alpha$.

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

summary. Implicit parameters are the simplest example of a system where function abstraction does not delay all impurities of the body. As discussed in Section 3.1.1, this is the defining feature of *coeffect* systems.

In this section, we have seen how this affects both the type system and the semantics of the language. In the type system, the (*abs*) rule places context-requirements on both the declaration-site and the call-site. For implicit parameters, this rule introduces non-determinism, because the parameters can be split arbitrarily. However, as we show in the next section, this is not always the case. Semantically, lambda abstraction *merges* two parts of context (implicit parameter dictionaries) that are provided by the call-site and declaration-site.

3.2.2 *Distributed computing*

Distributed programming was used as one of the motivating examples for coeffects in Chapter 1. This section explores the use case. We look at rebindable resources and cross-compilation. The structure of both is very similar to implicit parameters and type class constraints, but they demonstrate that there is a broader use for coeffect systems.

rebindable resources. The need for parameters that support dynamic scoping also arises in distributed computing. To quote an example discussed by Bierman et al. [9]: *"Dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources."*

Rebindable parameters are identifiers that refer to some specific resource. When a function value is marshalled and sent to another machine, rebindable resources can be handled in two ways. First, if the resource is available on the target machine, the parameter is *rebound* to the resource on the new machine. This is captured by dynamic scoping rules. Second, if the resource is not available on the target machine, the resource is either marshalled or a *remote reference* is created. This is captured by lexical scoping rules.

A practical language that supports rebindable resources is for example Acute [63]. In the following example, we use the construct **access** Res to represent access to a rebindable resource named Res. The following simple function accessed a database and a current date and filters values based on the date:

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

```
// Checks that input is valid; can run on both server and client
let validateInput = λname →
  name ≠ "" && forall isLetter name

// Searches database for a product; can run on the server-side
let retrieveProduct = λname →
  if validateInput name then Some(queryProductDb name)
  else None

// Client-side function to show price or error (for invalid inputs)
let showPrice = λname →
  if validateInput name then
    match (remote retrieveProduct()) with
    | Some p → showPrice (getPrice p)
    | None  → showError "Invalid input on the server"
  else showError "Invalid input on the client"
```

Figure 11: Sample client-server application with input validation

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

The type system and semantics for rebindable resources are essentially the same as those for implicit parameters. Primitive requirements are introduced by the **access** keyword. Lambda abstraction splits the resources non-deterministically between declaration-site (capturing remote reference) and call-site (representing rebinding). For this reason, we do not discuss the system in details and instead look at other uses.

CROSS-COMPILATION. A related issue with distributed programming is the need to target increasing number of diverse platforms. Modern applications often need to run on multiple platforms (iOS, Android, Windows or as JavaScript) or multiple versions of the same platform. Many programming languages are capable of targeting multiple different platforms. For example, functional languages that can be compiled to native code and JavaScript include, among others, F#, Haskell and OCaml [80].

Links [15], F# WebTools and WebSharper [68, 54], ML5 and QWeSST [46, 61] and Hop [42] go further and allow including code for multiple distinct platforms in a single source file. A single program is then automatically split and compiled to multiple target runtimes. This posses additional challenges – it is necessary to check where each part of the program can run and statically guarantee that it will be possible to compile code to the required target platform (safe *multi-targetting*).

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

Consider the client-server example in Figure 11. The retrieveProduct function represents the server-side, while showPrice is called on the client-side a.) Set based type system for cross-compilation, inspired by Links [15]

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

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

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

b.) Version number based type system, inspired by Android API level [19]

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

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

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

Figure 12: Two variants of coeffect typing rules for cross-compilation

and performs a remote call to the server-side function (how this is implemented is not our concern here). To ensure that the input is valid *both* functions call validateInput – however, this is fine, because validateInput uses only basic functions and language features that can be cross-compiled to both client-side and server-side.

In Links [15], functions can be annotated as client-side, server-side and database-side. F# WebTools [54] supports cross-compiled (mixed-side) functions similar to validateInput. However, these are single-purpose language features and they are not extensible. A practical implementation needs to be able to capture multiple different patterns – sets of environments (client, server, mobile) for distributed computing, but also Android API level [19] to cross-compile for multiple versions of the same platform.

TYPE SYSTEMS. Cross-compilation may seem similar to the tracking of resources (and thus to the tracking of implicit parameters), but it actually demonstrates a couple of new ideas that are important for flat coeffect systems. Unlike with implicit parameters, we will not present a specific existing system in this section – instead we briefly look at two examples that let us explore the range of possibilities.

In the first system, shown in Figure 12 (a), the coeffect annotations are sets of execution environments, i. e. $r, s, t \subseteq \{\mathsf{client}, \mathsf{server}, \mathsf{database}\}$. Subcoeffecting (*sub*) lets us ignore some of the supported execution environments; application (*app*) can be only executed in the *intersection* of the environments required by the two expressions and the function value.

Sub-coeffecting and application are trivially dual to the rules for implicit parameters. We just track supported environments using intersection as opposed to tracking of required parameters using union. However, this symmetry does not hold for lambda abstraction (*abs*), which still uses *union*. This models the case where the function can be executed in two different ways:

- The function is represented as executable code for an environment available at the call-site and is executed there, possibly after it is marshalled and transferred to another machine.
- The function body is compiled for the environment available at the declaration-site; the value that is returned is a remote reference to the code and function calls are performed as remote invocations.

This example ignores important considerations – for example, it is likely desirable to make this difference explicit and the implementation (and semantics) needs to be clarified. However, the example shows that the algebraic structure of coeffect annotations may be more complex. Here, using $\cap$ for application and $\cup$ for abstraction.

The second system, shown in Figure 12 (b) is inspired by the API level requirements in Android. Coeffect annotations are simply numbers representing the level ($r, s, t \in \mathbb{N}$). Levels are ordered increasingly, so we can always require higher level (*sub*). The requirement on function application (*app*) is the highest level of the levels required by the sub-expressions and the function. The system uses yet another variant of lambda abstraction (*abs*) – the requirements of the body are duplicated and placed on *both* the declaration-site and the call-site.

In addition to the work discussed already, ML5 [46] is another important work that looks at tracking of execution environments. It uses modalities of modal S4 to represent the environment – this approach is similar to coeffects, both from the practical perspective, but also through deeper theoretical links. We return to this topic in Chapter **??**.

3.2.3 *Liveness analysis*

Live variable analysis (LVA) [4] is a standard technique in compiler theory. It detects whether a free variable of an expression may be used by a program during its evaluation (it is *live*) or whether it is definitely not needed (it is *dead*). As an optimization, compiler can remove bindings to dead variables as they are never accessed. Wadler [83] describes the property of a variable that is dead as the *absence* of a variable.

FLAT LIVENESS ANALYSIS. In this section, we discuss a restricted form of liveness analysis. We do not track liveness of *individual* variables, but of the *entire* variable context. This is not practically useful, but it provides interesting insight into how flat coeffects work. A per-variable liveness analysis can be captured using structural coeffects and is discussed in Section 3.3.1. Consider the following two examples:

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

The body of the first function is just a constant 42 and so the context of the body is marked as *dead*. The parameter (call-site) of the function is not used and can also be marked as dead. Similarly, no variables from the declaration-site are used and so they are also marked as dead.

In contrast, the body of the second function accesses a variable value and so the body of the function is marked as *live*. In the flat system, we do not track *which* variable was used and so we have to mark both the call-site and declaration-site as live (this will be refined in structural liveness system).

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

$$\begin{array}{lll@{\qquad}lll@{\qquad}lll@{\qquad}lll}
\mathsf{L} \sqcup \mathsf{L} & = & \mathsf{L} & \mathsf{L} \sqcup \mathsf{D} & = & \mathsf{D} & \mathsf{L} \sqcap \mathsf{L} & = & \mathsf{L} & \mathsf{L} \sqcap \mathsf{D} & = & \mathsf{L} \\
\mathsf{D} \sqcup \mathsf{L} & = & \mathsf{D} & \mathsf{D} \sqcup \mathsf{D} & = & \mathsf{D} & \mathsf{D} \sqcap \mathsf{L} & = & \mathsf{L} & \mathsf{D} \sqcap \mathsf{D} & = & \mathsf{D}
\end{array}$$

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

$$\begin{array}{lll}
f : \tau_1 \xrightarrow{r} \tau_2 & g : \tau_2 \xrightarrow{s} \tau_3 & g \circ f : \tau_1 \xrightarrow{r \sqcup s} \tau_3
\end{array}$$

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

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

$$\begin{array}{lll}
f : \tau_1 \xrightarrow{r} \tau_2 & h : \tau_1 \xrightarrow{s} \tau_3 & \langle f, h \rangle : \tau_1 \xrightarrow{r \sqcap s} \tau_2 \times \tau_3
\end{array}$$

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

Figure 13: Liveness annotations with sequential and pointwise composition

FORWARD VS. BACKWARD & MAY VS. MUST. Static analyses can be classified as either *forward* or *backward* (depending on how they propagate information) and as either *must* or *may* (depending on what properties they guarantee). Liveness is a *backward* analysis – the requirements are propagated from variables to their declarations. The distinction between *must* and *may* is apparent when we look at an example with conditionals:

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

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

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

TYPE SYSTEM. A type system that captures whole-context liveness annotates the context with value of a two-point lattice $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$. The annotation L marks the context as *live* and D stands for a *dead* context. Figure 13 (a) defines the ordering $\sqsubseteq$, meet $\sqcup$ and join operations $\sqcap$ of the lattice.

The typing rules for tracking whole-context liveness are shown in Figure 14. The language now includes constants $c : \tau \in \Delta$. Accessing a constant (*const*) annotates the context as dead using D. This contrasts with variable access (*var*), which marks the context as live using L. A dead context (definitely not needed) can be treated as live context (which may be used) using the (*sub*) rule. This captures the *may* nature of the analysis.

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

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

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

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

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

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

Figure 14: Coeffect rules for tracking whole-context liveness

The (*app*) rule is best understood by discussing its semantics. The semantics uses *sequential composition* to compose the semantics of e_2 with the function obtained as the result of e_1. However, we need more than just sequential composition. The same input context is passed to the expression e_1 (in order to get the function value) and to the sequentially composed function (to evaluate e_2 followed by the function call). This is captured by *pointwise composition*.

Consider first *sequential composition* of (semantic) functions f, g annotated with r, s. The composed function $g \circ f$ is annotated with $r \sqcup s$ as shown in Figure 13 (b). The argument of the function $g \circ f$ is live only when the arguments of both f and g are live (1). When the argument of f is dead, but g requires τ_2 (2), we can evaluate f without any input and obtain τ_2, which is then passed to g. When g does not require its argument (3, 4), we can just evaluate g, without evaluating f. Here, the semantics *implements* the dead code elimination optimization.

Secondly, a *pointwise composition* passes the same argument to f and h. The parameter is live if either the parameter of f or h is live. The pointwise composition is written as $\langle f, h\rangle$ and it combines annotations using $\sqcap$ as shown in Figure 13 (c). Here, the argument is not needed only when both f and h do not need it (1). In all other cases, the parameter is needed and is then used either once (2, 3) or twice (4). The rule for function application (*app*) combines the two operations. The context Γ is live if it is needed by e_1 (which always needs to be evaluated) *or* when it is needed by the function value *and* by e_2.

The (*abs*) rule duplicates the annotation of the body, similarly to the cross-compilation example in Figure 12. When the body accesses any variables, it requires both the argument and the variables from declaration-site. When it does not use any variables, it marks both as dead. Finally, the (*let*) rule annotates the composed expression with the liveness of the expression e_2 – if the context of e_2 is live, then it also requires variables from Γ; if it is dead, then it does not require Γ or x. As further discussed later in Section ?, the (*let*) rule is again just a syntactic sugar for $(\lambda x.e_2)\ e_1$. Briefly, this follows from the simple observation that $r \sqcup (s \sqcap r) = r$.

$$\llbracket \Gamma @ \mathsf{L} \vdash x_i : \tau_i \rrbracket = \lambda(x_1, \ldots, x_n) \to x_i \tag{var}$$

$$\llbracket \Gamma @ \mathsf{D} \vdash c_i : \tau_i \rrbracket = \lambda() \to \delta(c_i) \tag{const}$$

$$\llbracket \Gamma @ \mathsf{L} \vdash e : \tau \rrbracket = \lambda x \to \llbracket \Gamma @ \mathsf{D} \vdash e : \tau \rrbracket\ () \tag{sub-1}$$

$$\llbracket \Gamma @ r \vdash e : \tau \rrbracket = \lambda x \to \llbracket \Gamma @ r \vdash e : \tau \rrbracket\ x \tag{sub-2}$$

$$\begin{aligned}&\llbracket \Gamma @ \mathsf{L} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \rrbracket = \lambda(x_1, \ldots, x_n) \to \\ &\qquad \lambda y \to \llbracket \Gamma, y : \tau_1 @ \mathsf{L} \vdash e : \tau_2 \rrbracket\ (x_1, \ldots, x_n, y)\end{aligned} \tag{abs-1}$$

$$\begin{aligned}&\llbracket \Gamma @ \mathsf{D} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 \rrbracket = \lambda() \to \\ &\qquad \lambda() \to \llbracket \Gamma, y : \tau_1 @ \mathsf{D} \vdash e : \tau_2 \rrbracket\ ()\end{aligned} \tag{abs-2}$$

$$\begin{aligned}&\llbracket \Gamma @ r \vdash e_1\ e_2 : \tau_2 \rrbracket = \lambda x \to \\ &\qquad \mathbf{let}\ g = \llbracket \Gamma @ r \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 \rrbracket\ x\ \mathbf{in}\ g\ ()\end{aligned} \tag{app-1}$$

$$\begin{aligned}&\llbracket \Gamma @ \mathsf{L} \vdash e_1\ e_2 : \tau_2 \rrbracket = \lambda x \to \\ &\qquad \mathbf{let}\ g = \llbracket \Gamma @ \mathsf{L} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \rrbracket\ x\ \mathbf{in}\ g\ (\llbracket \Gamma @ \mathsf{D} \vdash e_2 : \tau_1 \rrbracket\ ())\end{aligned} \tag{app-2}$$

$$\begin{aligned}&\llbracket \Gamma @ r \sqcup (s \sqcap t) \vdash e_1\ e_2 : \tau_2 \rrbracket = \lambda x \to \\ &\qquad \mathbf{let}\ g = \llbracket \Gamma @ r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \rrbracket\ x\ \mathbf{in}\ g\ (\llbracket \Gamma @ s \vdash e_2 : \tau_1 \rrbracket\ x)\end{aligned} \tag{app-3}$$

Figure 15: Semantics of a language with liveness analysis

EXAMPLES. Before looking at the semantics, we consider a number of simple examples to demonstrate the key aspects of the system. Full typing derivations are shown in Appendix ?:

$$(\lambda x \to 42)\ y \tag{1}$$

$$\mathsf{twoTimes}\ 42 \tag{2}$$

$$(\lambda x \to x)\ 42 \tag{3}$$

In the first case, the context is dead. In (1), the function's parameter is dead and so the overall context is dead, even though the argument uses a variable y – the semantics evaluates the function without passing it an actual argument. In the second case (2), the function is a variable that needs to be obtained and so the context is live. In the last case (3), the function accesses a variable and so its declaration-site is marked as requiring the context (*abs*). This is where structural coeffect analysis would be more precise – the system shown here cannot capture the fact that x is a bound variable.

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

We can represent such empty values using the option type (known as Maybe in Haskell). We use the notation $\tau + 1$ to denote option types. Given a context with variables x_i of type τ_i, the semantics is a function taking $(\tau_1 \times \ldots \times \tau_n) + 1$. When the context is live, it will be called with the left value (product of variable assignments); when the context is dead, it will be called with the right value (containing no information).

However, ordinary option type is not sufficient. We need to capture the fact that the representation depends on the annotation – in other words, the type is *indexed* by the coeffect annotation. The indexing is discussed in details in Section X. For now, it suffices to define the semantics using two separate rules:

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

The semantics of functions is defined similarly. When the argument of a function is live, the function takes the input value; when the argument is dead, the semantic function takes a unit as its argument:

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

Unlike with implicit parameters, the coeffect system for liveness tracking cannot be modelled using monads. Any monadic semantics would express functions as $\tau_1 \to M\,\tau_2$. Unless laziness is already built-in in the semantics, there is no way to call such function without first obtaining a value τ_1. The above semantics makes this possible by taking a unit 1 when the argument is not live.

In Figure 15, we define the semantics directly. We write $()$ for the only value of type 1. This appears, for example, in (*const*) which takes $()$ as the input and returns constant using a global dictionary δ. In (*var*), the context is live and so the semantics performs a projection. Sub-coeffecting is captured by two rules. A dead context can be treated as live using (*abs-1*); in other cases, the annotation is not changed (*abs-2*).

Lambda abstraction can be annotated in just two ways. When the body requires context (*abs-1*), the value of a bound variable y is added to the context Γ before passing it to the body. When the body does not require context (*abs-2*), it is called with $()$ as the input.

For application, there are 8 possible combinations of annotations. The semantics of some of them is the same, so we only need to show 3 cases. The rules should be read as ML-style pattern matching, where the last rule handles all cases not covered by the first two. In (*app-1*), we handle the case when the function g does not require its argument – e_2 is not used and instead, the function is called with $()$ as the argument. The case (*app-2*) covers the case when the expression e_1 does not require a context, but e_1 does. Finally, in (*app-3*), the same input (which may be either tuple of variables or unit) is propagated uniformly to both e_1 and e_2.

SUMMARY. Unlike with implicit parameters, the lambda abstraction for liveness analysis does not introduce non-determinism. It simply duplicates the context requirements. However, this still matches the property of coeffects that impurities cannot be thunked.

The semantics of liveness reveals a number of interesting properties too. Firstly, the semantics cannot be captured by any monad. Secondly, the system would not work without the coeffect annotations. The shape of the semantic function depends on the annotation (the input is either 1 or τ) and is *indexed* by the annotation. Finally, we discussed at length how the semantics of application arises from *sequential* and *pointwise* composition. This is another important aspect of coeffect systems – categorical semantics typically builds on *sequential* composition, but to model full λ calculus it needs more. For coeffect systems, we need *pointwise* composition where the same context is shared by multiple sub-expressions.

3.2.4 *Data-flow languages*

The Section 1.1.4 briefly demonstrated that we can treat array access as an operation that accesses a context. In case of arrays, the context is neighbourhood of a current location in the array specified by a cursor. In this section, we make the example more concrete, using a simpler and better studied programming model, data-flow languages.

Lucid [82] 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 $\mathsf{square}(x)$ represents a stream of squares calculated from the stream of values x.

The data-flow approach has been successfully used in domains such as development of real-time embedded application where many *synchronous languages* [7] build on the data-flow paradigm. The following example is inspired by the Lustre [30] language and implements program to count the number of edges on a Boolean stream:

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

let edgeCount =
  0 fby ( if edge then 1 + (prev edgeCount)
          else prev edgeCount )
```

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

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

When reading data-flow programs, we do not need to think about variables in terms of streams – we can see them as simple values. Most of the operations perform calculation just on the *current* value of the stream. However, the operation **fby** and **prev** are different. They require additional *context* which provides past values of variables (for **prev**) and information about the current location in the stream (for **fby**).

The semantics of Lucid-like languages can be captured using a number of mathematical structures. Wadge [81] originally proposed to use monads, while Uustalu and Vene later used comonads [75]. In Chapter 4, we extend the latter approach. However, the present chapter presents a sketch of a data-flow semantics defined directly on streams.

In the introductory example with array access patterns, we used coeffects to track the range of values accessed. In this section, we look at a simpler example – we only consider the **prev** operation and track the maximal number of *past values* needed. This is an important information for efficient implementation of data-flow languages. When we can guarantee that at most x past values are accessed, the values can be stored in a pre-allocated buffer rather than using e. g. on-demand computed lazy streams.

TYPE SYSTEM. A type system that tracks the maximal number of accessed past values annotates the context with a single integer. The current value is

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

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

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

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

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

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

Figure 16: Coeffect rules for tracking context-usage in data-flow language

always present, so 0 means that no past values are needed, but the current value is still available. The typing rules of the system are shown in Figure 16.

Variable access (*var*) annotates the context with 0; sub-coeffecting (*sub*) allows us to require more values than is actually needed. Primitive context-requirements are introduced in (*prev*), which increments the number of past values by one. Thus, for example, **prev** $(\textbf{prev}\ x)$ requires 2 past values.

The (*app*) rule follows the same intuition as for liveness. It combines *sequential* and *pointwise* composition of semantic functions. In case of data-flow, the operations combine annotations using $+$ and *max* operations:

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

Sequential composition adds the annotations. The function f needs m past values to produce a single τ_2 value. To produce two τ_2 values, we thus need $m+1$ past values of τ_1; to produce three τ_2 values, we need $m+2$ past values of τ_1, and so on. To produce n past values that are required as the input of g, we need $m+n$ past values of type τ_1. The pointwise composition is simpler. It uses the same stream to evaluate functions requiring m and n past values, and so it needs maximum of the two at most.

In summary, function application (*app*) requires maximum of the values needed to evaluate e_1 and the number of values needed to evaluate the argument e_2, sequentially composed with the function.

In function abstraction (*abs*), the requirements of the body are duplicated on the declaration-site and the call-site as in liveness analysis. If the body requires n past values, it may access n values of any variables – including those available in Γ, as well as the parameter x. Finally, the (*let*) rule simply adds the two requirements. This corresponds to the sequential composition operation, but it is also a rule that we obtain by treating let-binding as a syntactic sugar for $(\lambda x.e_2)\ e_1$.

EXAMPLE. As with the liveness example, the application rule might require more explanation. The following example is somewhat arbitrary, but it demonstrates the rule well. We assume that counter is a stream of positive

$$\llbracket \Gamma @ 0 \vdash x_i : \tau_i \rrbracket = \lambda\langle (x_0,\ldots,x_n)\rangle \rightarrow x_i \tag{\textit{var}}$$

$$\begin{aligned}&\llbracket \Gamma @ n+1 \vdash \textbf{prev}\ e : \tau \rrbracket = \lambda\langle \mathbf{v}_0,\ldots,\mathbf{v}_{n+1}\rangle \rightarrow \\ &\qquad \llbracket \Gamma @ n \vdash e : \tau \rrbracket\ \langle \mathbf{v}_1,\ldots,\mathbf{v}_{n+1}\rangle\end{aligned} \tag{\textit{prev}}$$

$$\begin{aligned}&\llbracket \Gamma @ n \vdash e : \tau \rrbracket = \lambda\langle \mathbf{v}_0,\ldots,\mathbf{v}_{n}\rangle \rightarrow \\ &\qquad \llbracket \Gamma @ n' \vdash e : \tau \rrbracket\ \langle \mathbf{v}_0,\ldots,\mathbf{v}_{n'}\rangle\end{aligned} \tag{\textit{sub}}$$

$$\begin{aligned}&\llbracket \Gamma @ n \vdash \lambda y.e : \tau_1 \xrightarrow{n} \tau_2 \rrbracket = \lambda\langle \mathbf{v}_0,\ldots \mathbf{v}_{n}\rangle \rightarrow \\ &\qquad \lambda(y,g) \rightarrow \llbracket \Gamma, y : \tau_1 @ n \vdash e : \tau_2 \rrbracket\ \langle (\mathbf{v}_0, y_0),\ldots,(\mathbf{v}_n, y_n)\rangle\end{aligned} \tag{\textit{abs}}$$

$$\begin{aligned}&\llbracket \Gamma @ \max(m, n+p) \vdash e_1\ e_2 : \tau_2 \rrbracket = \lambda(\mathbf{v}_0,\ldots,\mathbf{v}_{\max(m,n+p)}) \rightarrow \\ &\qquad \textbf{let}\ g = \llbracket \Gamma @ m \vdash e_1 : \tau_1 \xrightarrow{p} \tau_2 \rrbracket\ (\mathbf{v}_0,\ldots,\mathbf{v}_m) \\ &\qquad \textbf{in}\ g\ (\ \llbracket \Gamma @ n \vdash e_2 : \tau_1 \rrbracket\ (\mathbf{v}_0,\ldots,\mathbf{v}_n),\ldots, \\ &\qquad\qquad\quad \llbracket \Gamma @ n \vdash e_2 : \tau_1 \rrbracket\ (\mathbf{v}_p,\ldots,\mathbf{v}_{n+p})\ )\end{aligned} \tag{\textit{app}}$$

Figure 17: Semantics of a simple data-flow language

integers (starting from zero) and tick flips between 0 and 1. The full typing derivation is shown in Appendix ?:

```
( if  (prev tick) = 0
  then (λx → prev x)
  else (λx → x) )      (prev counter)
```

The left-hand side of the application returns a function depending on the *previous* value of tick. The resulting stream of functions flips between a function returning a current value and a function returning the previous value. If the current tick is 0, and the function is applied to a stream $\langle \ldots,4,3,2,1\rangle$ (where 1 is the current value), it yields the stream $\langle \ldots,4,4,2,2\rangle$.

To obtain the function, we need one past value from the context (for **prev** tick). The returned function needs either none or one past value (thus a subtyping rule is required to type it as requiring one past value). So, the annotations for (*app*) are $m = 1, p = 1$. The function is called with **prev** counter as an argument, meaning that the result is either the first or second past element. Given counter $= \langle \ldots,5,4,3,2,1\rangle$, the argument is $\langle \ldots,5,4,3,2\rangle$ and so the overall result is a stream $\langle \ldots,5,5,3,3\rangle$. From the argument, we get the requirement $n = 1$.

Using the (*app*) rule, we get that the overall number of past elements needed is $max(1, 1+1) = 2$. This should match the intuition about the code – when the first function is applied to the argument, the computation will first access **prev** tick (using one past value) and then **prev** (**prev** counter)) (using two past values).

SEMANTICS. The sample language discussed in this section is a *causal* data-flow language. This means that a computation can access *past* values of the stream (but not future values). In the semantics, we again need richer structure over the input.

Uustalu and Vene [76] model causal data-flow computations using a non-empty list $\mathsf{NeList}\ \tau = \tau \times (\mathsf{NeList}\ \tau + 1)$ over the input. A function $\tau_1 \rightarrow \tau_2$ is thus modelled as $\mathsf{NeList}\ \tau_1 \rightarrow \tau_2$. This model is difficult to implement efficiently, as it creates unbounded lists of past elements.

The coeffect system tracks maximal number of past values and so we can define the semantics using a list of fixed length. As with liveness, this is a data structure *indexed* by the coeffect annotation. We write τ^n for a list containing n elements (which can be also viewed as an n-element product $\tau \times \ldots \times \tau$).

As with the previous examples, our semantics interprets a judgement using a (semantic) function; functions in the language are modelled as functions taking a list of inputs:

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

Note that the semantics requires one more value than is the number of past values. This is because the first value is the current value and has to be always available, even when the annotation is zero as in (*var*).

The rules defining the semantics are shown in Figure 17. The semantics of the context is a *list of pairs*. To make the rules easier to follow, we write $\langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle$ for an n-element list containing pairs. Pairs that model the entire context such as $\mathbf{v}_1$ are written in bold. When we access individual variables, we write $\mathbf{v} = (x_1, \ldots, x_m)$ where x_i denote individual variables of the context.

In (*var*), the context is a singleton-list containing a product of variables, from which we project the right one. In (*prev*) and (*sub*), we drop some of the elements from the history (from the front and end, respectively) and then evaluate the original expression.

Lambda abstractions (*abs*) receives two lists of the same size – one containing values of the variables from the declaration-site $\langle \mathbf{v}_0, \ldots, \mathbf{v}_n \rangle$ and one containing the argument provided by the call-site $\langle y_0, \ldots, v_n \rangle$. The semantics applies the well-known *zip* operation on the lists and passes the result to the body.

Finally, application (*abs*) uses the input context in two ways, which gives rise to the two requirements combined using *max*. First, it evaluates the expression e_1 which is called with the past m values. The resulting function g is then sequentially composed with the semantics of e_2. To call the function, we need to evaluate e_2 repeatedly – namely, $p + 1$ times, which results in the overall requirement for $n + p$ past values.

SUMMARY. The most interesting point about the data-flow example is that it is remarkably similar to our earlier liveness example. In the type system, abstraction (*abs*) duplicates the context requirements and application (*abs*) arises from sequential and pointwise composition. We capture this striking similarity in Chapter 4. Before doing that, we look at one more example and then explore the *structural* class of systems.

3.2.5 *Permissions and safe locking*

In the implicit parameters and data-flow examples, the context provides additional resources or values that may be accessed at runtime. However, it may also track *permissions* or *capabilities* to perform some operation. Liveness can be seen as a trivial example – when the context is live, it contains a permission to access variables. In this section, we briefly consider a system for safe locking of Flanagan and Abadi [23] as one, more advanced example. Calculus of capabilities of Cray et al. [17] is discussed later in Section **??**.

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

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

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

The type system for safe locking associates a list of acquired locks with the context. Interestingly, the original presentation of the system uses a coeffect-style judgements of a form $\Gamma; p \vdash e : \tau$ where p is a list of accessible regions (protected by an acquired lock). Using our notation, the rule for **sync** looks as follows:

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

The rule requires that e_1 yields a value of a singleton type m. The type is added as an indicator of the locked region to the context $p \cup \{m\}$ which is then used to evaluate the expression e_2.

SUMMARY. Despite attaching annotations to the variable context, the system for safe locking uses effect-style lambda abstraction. Lambda abstraction associates all requirements with the call-site – a lambda function created under a lock cannot access protected memory available at the time of creation. It will be executed later and can only access the memory available then. This suggests that safe locking is perhaps better seen as an effect system.

Another interesting aspect is the extension to avoid deadlocks. In that case, the type system needs to reject programs that acquire locks in an invalid order. One way to model this is to replace $p \cup \{m\}$ with a *partial* operation $p \uplus \{m\}$ which is only defined when the lock m can be added to the set p. Supporting partial operations on coeffect annotations is an interesting extension which we discuss in Section ?. The extension also lets us capture systems with effect-style lambda abstraction such as safe locking.

3.3 STRUCTURAL COEFFECT SYSTEMS

In structural coeffect systems, the additional information are associated with individual variables. This is very often information about how the variables are used, or, in which contexts they are used.

In Chapter 1, we introduced the idea using an example that tracks array access patterns. Each variable is annotated with a range specifying which elements of the corresponding array may be accessed. In this section, we look at a number of examples. We first consider an example inspired by linear logic. Then we revisit liveness and data-flow, for which the structural system provides a more precise analysis. Finally, we look at a number of other practical uses including security, tainting and provenance tracking.

3.3.1 *Liveness analysis revisited*

The flat system for liveness analysis presented in Section 3.2.3 is interesting from a theoretical perspective, but it is not practically useful. In this section, we revisit the problem and define a structural system that tracks liveness per-variable.

STRUCTURAL LIVENESS. Recall two examples discussed earlier where the flat liveness analysis marked the whole context as (syntactically) live, despite the fact part of it was (semantically) dead:

```
let constant = λy → λx → y
let answer = (λx → x) 42
```

In the first case, the variable x is dead, but was marked as live. In the second example, the declaration-site of the answer value is dead, but was marked as live. This is because in both of the expressions, *some* variable is accessed. However, the (*abs*) rule of flat liveness has no way of determining *which* variables are used by the body – and, in particular, whether the accessed variable is the *bound* variable or some of the *free* variables.

As discussed earlier, we can resolve this by attaching a *vector* of liveness annotations to a *vector* of variables. In the first example, the available variables are y and x, so the variable context Γ is a vector $\langle y:\tau, x:\tau\rangle$. Only the variable y is used and so the annotated context is: $y:\tau, x:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$. When writing the contexts, we omit angle brackets around variables, but it should still be viewed as a vector. There are two important points:

- The fact that variables are now a vector means that we cannot freely reorder them. This guarantees that $x:\tau, y:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$ can not be confused with $y:\tau, x:\tau @ \langle \mathsf{L}, \mathsf{D}\rangle$. We need to define the type system in a way that is similar to sub-structural systems (discussed in Section 2.3) which provide explicit rules for manipulating the context.

- We choose to attach a vector of annotations to a vector of variables, rather than attaching individual annotations to individual variables. This lets us unify and combine flat and structural systems as discussed in Chapter **??**, but the alternative is briefly explored in Chapter **??**.

TYPE SYSTEM. The structural system for liveness uses the same two-point lattice of annotations $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ that was used by the flat system. We also use the $\sqcup, \sqcap$ and $\sqsubseteq$ operators that are defined in Figure 13.

The rules of the system are split into two groups. Figure 18 (a) shows the standard syntax-driven rules plus sub-coeffecting. In (*var*), the context contains just the single accessed variable, which is annotated as live. Other variables can be introduced using weakening. A constant (*const*) is accessed in an empty context, which also carries no annotations. The sub-coeffecting rule (*sub*) uses a point-wise extension of the $\sqsubseteq$ relation over two vectors as defined in Section 3.1.3.

In the (*abs*) rule, the variable context of the body $\Gamma, x:\tau_1$ is annotated with a vector $\mathbf{r} \times \langle s\rangle$, where the vector $\mathbf{r}$ corresponds to Γ and the singleton annotation s corresponds to the variable x. Thus, the function is annotated with s. Note that the free-variable context is annotated with vectors, but functions take only a single input and so are annotated with primitive annotations.

The (*app*) rule is similar to function applications in flat systems, but there is an important difference. In structural systems, the two sub-expressions a.) Ordinary, syntax-driven rules with sub-coeffecting

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

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 18: Structural coeffect liveness analysis

have separate variable contexts Γ_1 and Γ_2. Therefore, the composed expression just concatenates the variables and their corresponding annotations. (We can still use the same variable in both sub-expressions thanks to the structural contraction rule.)

The context Γ_1 is used to evaluate e_1 and is thus annotated with $\mathbf{r}$. The annotation for Γ_2 is more interesting. It is a result of sequential composition of two semantic functions – the first one takes the (multi-variable) context Γ_2 and evaluates e_2; the second takes the result of type τ_1 and passes it to the function $\tau_1 \xrightarrow{t} \tau_2$. The composition is defined as follows:

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

This definition is only for illustration and is revised in Chapter 5. The function g takes a product of multiple variables (and is annotated with a vector). The function f takes just a single value and is annotated with the scalar. As in the flat system, sequential composition is modelled using $\sqcup$ – but here, we use a scalar-vector extension of the operation. Finally, the (*let*) rule follows similar reasoning (and also corresponds to the typing of $(\lambda x.e_2)\ e_1$.

STRUCTURAL TYPING RULES. The structural typing rules are shown in Figure 18 (b). They mirror the rules know from sub-structural type systems (Section 2.3). Weakening (*weak*) extends the context with a single unused variable x and adds the D annotation to the vector of coeffects.

The variable is always added to the end as in the (*abs*) rule. However, the exchange rule (*exch*) lets us arbitrarily reorder variables. It flips the variables x and x' and their corresponding coeffect annotations in the vector. This is done by requiring that the lengths of the remaining, unchanged, parts of the vectors match.

Finally, contraction (*contr*) makes it possible to use a single variable multiple times. Given a judgement that contains variables y and z, we can derive a judgement for an expression where both z and y are replaced by a single variable x. Their annotations s, t are combined into $s \sqcap t$, which means that x is live if either z or y were live in the original expression.

EXAMPLE. To demonstrate how the system works, we consider the expression $(\lambda x \to v)\ y$. This is similar to an example where flat liveness mistakenly marks the entire context as live. Despite the fact that the variable y is accessed (syntactically), it is not live – because the function that takes it as an argument always returns v.

The typing derivation for the body uses (*var*) and (*abs*). However, we also need (*weak*) to add the unused variable x to the context:

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

The interesting part is the use of the (*app*) rule in the next step. Although the variable y is live in the expression y, it is marked as dead in the overall expression, because the function is annotated with D:

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

The application is written in two steps – the first one directly applies the (*app*) rule and the second one simplifies the coeffect annotation. The key part is the use of the scalar-vector operator $\mathsf{D} \sqcup \langle \mathsf{L} \rangle$. Using the definition of the scalar-vector extension, this equals $\langle \mathsf{D} \sqcup \mathsf{L} \rangle$ which is $\langle \mathsf{D} \rangle$.

SEMANTICS. When defining the semantics of flat liveness calculus, we used an indexed form of the option type $1 + \tau$ (which is 1 for dead contexts and τ for live contexts). In the semantics of expressions, the type wrapped the entire context, i.e. $1 + (\tau_1 \times \ldots \times \tau_n)$. In the structural version, the semantics wraps individual elements of the free-variable context pair: $(1 + \tau_1) \times \ldots \times (1 + \tau_n)$. For each variable, the type is indexed by the corresponding annotation. More formally:

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

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

Note that the product of the free variables is not an ordinary tuple of our language, but a special construction. This follows from the asymmetry of λ-calculus, as discussed in Section 3.1.3. Functions take just a single input and so they are interpreted in the same way as in flat calculus:

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

a.) Semantics of ordinary expressions

$$\llbracket x{:}\tau \,@\, \langle \mathsf{L} \rangle \vdash x : \tau \rrbracket = \lambda(x) \to x \qquad (\textit{var})$$

$$\llbracket () \,@\, \langle\rangle \vdash c : \tau \rrbracket = \lambda() \to \delta(c) \qquad (\textit{const})$$

$$\begin{aligned} &\llbracket \Gamma \,@\, \mathbf{r} \vdash \lambda y.e : \tau_1 \xrightarrow{\mathsf{s}} \tau_2 \rrbracket = \lambda \mathbf{v} \to \\ &\quad \lambda y \to \llbracket \Gamma, y{:}\tau_1 \,@\, \mathbf{r} \times \langle \mathsf{s} \rangle \vdash e : \tau_2 \rrbracket \; (\mathbf{v}, y) \end{aligned} \qquad (\textit{abs})$$

$$\begin{aligned} &\llbracket \Gamma_1, \Gamma_2 \,@\, \mathbf{r} \times (\mathsf{L} \sqcup \mathbf{s}) \vdash e_1\, e_2 : \tau_2 \rrbracket = \lambda(\mathbf{v_1}, \mathbf{v_2}) \to \\ &\quad \textbf{let } g = \llbracket \Gamma_1 \,@\, \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 \rrbracket \; \mathbf{v_1} \\ &\quad \textbf{in } g \; (\llbracket \Gamma_2 \,@\, \mathbf{s} \vdash e_2 : \tau_1 \rrbracket \; \mathbf{v_2}) \end{aligned} \qquad (\textit{app-1})$$

$$\begin{aligned} &\llbracket \Gamma_1, \Gamma_2 \,@\, \mathbf{r} \times (\mathsf{D} \sqcup \mathbf{s}) \vdash e_1\, e_2 : \tau_2 \rrbracket = \lambda(\mathbf{v_1}, \mathbf{v_2}) \to \\ &\quad \textbf{let } g = \llbracket \Gamma_1 \,@\, \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{\mathsf{D}} \tau_2 \rrbracket \; \mathbf{v_1} \textbf{ in } g \; () \end{aligned} \qquad (\textit{app-2})$$

b.) Semantics of structural context manipulation

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

$$\begin{aligned} &\llbracket \Gamma_1, y{:}\tau, x{:}\tau', \Gamma_2 \,@\, \mathbf{r} \times \langle \mathsf{t}, \mathsf{s} \rangle \times \mathbf{q} \vdash e : \sigma \rrbracket = \lambda(\mathbf{v_1}, y, x, \mathbf{v_2}) \to \\ &\quad \llbracket \Gamma_1, x{:}\tau', y{:}\tau, \Gamma_2 \,@\, \mathbf{r} \times \langle \mathsf{s}, \mathsf{t} \rangle \times \mathbf{q} \vdash e : \sigma \rrbracket \; (\mathbf{v_1}, x, y, \mathbf{v_2}) \end{aligned} \qquad (\textit{exch})$$

$$\begin{aligned} &\llbracket \Gamma_1, x{:}\tau, \Gamma_2 \,@\, \mathbf{r} \times \langle \mathsf{D} \rangle \times \mathbf{q} \vdash e[z \leftarrow x][y \leftarrow x] : \sigma \rrbracket = \lambda(\mathbf{v_1}, (), \mathbf{v_2}) \to \\ &\quad \llbracket \Gamma_1, y{:}\tau, z{:}\tau, \Gamma_2 \,@\, \mathbf{r} \times \langle \mathsf{D}, \mathsf{D} \rangle \times \mathbf{q} \vdash e : \sigma \rrbracket \; (\mathbf{v_1}, (), (), \mathbf{v_2}) \end{aligned} \qquad (\textit{contr-1})$$

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

Figure 19: Semantics of structural liveness

The rules that define the semantics are shown in Figure 19. To make the definition simpler, we are somewhat vague when working with products. We write variables of product type such as $\mathbf{v}$ in bold-face and individual values like x in normal face. We freely re-associate products and so $(\mathbf{v}, x)$ should not be seen as a nested product, but simply a product with a number of variables represented as another product $\mathbf{v}$ with one more variable x at the end. We shall be more precise in Chapter 5.

In (*var*), the context contains just a single variable and so we do not even need to apply projection; (*cosnt*) receives no variables and uses global constant lookup function δ. In (*abs*), we obtain two parts of the context and combine them into $(\mathbf{v}, x)$. This works the same way regardless of whether the variables are live or dead. For simplicity, we omit sub-coeffecting, which just turns some of the available values v_i to unit values $()$.

In application, we again need to "implement" dead code elimination. When the input parameter of the function g is live (*app-1*), we first evaluate e_2 and then pass the result to g. When the parameter is dead (*app-2*), we do not need to evaluate e_2 and so all values in $\mathbf{v_2}$ can be dead, i. e. $()$.

In the structural rules, (*weak*) receives context containing a dead variable as the last one. It drops the $()$ value and evaluates the expression in a context $\mathbf{v}$. Exchange (*exch*) simply swaps two variables. In contraction, we either duplicate a dead value (*contr-1*), or a live value (*contr-2*). In the latter, one of the duplicates may be dead and so we need to consider three separate cases.

SUMMARY. The structural liveness calculus is a typical example of a system that tracks per-variable annotations. In a number of ways, the system is simpler than the flat coeffect calculi. In lambda abstraction, we simply annotate function with the annotation of a matching variable (this rule is the same for all upcoming systems). In application, the *pointwise* composition is no longer needed, because the sub-expressions use separate contexts. On the other hand, we had to add weakening, contraction and exchange rules to let us manipulate the contexts.

The semantics of weakening demonstrates an important point about coeffects that may be quite confusing. When we read the *typing rule* from top to bottom, weakening adds a variable to the context. When we read the *semantic rule*, weakening drops a variable value from the context! This duality is caused by the fact that coeffects talk about context – they describe how to build the context required by the sub-expressions and so the semantics implements transformation from the context in the (typing) conclusion to the (typing) assumption.

The structural systems discussed in the upcoming sections are remarkably similar to the one shown here. We discuss two more examples in details to explore the design space, but we shall omit details that are the shared with the system in this section.

3.3.2 *Bounded variable use*

Liveness analysis checks whether a variable is used or unused. With structural coeffects, we can go further and track how many times is the variable accessed. Girard et al. [28] coined this idea as *bounded linear logic* and use it to restrict well-typed programs to polynomial-time algorithms. We first introduce the system in our, coeffect, style and then relate it with the original formulation.

BOUNDED VARIABLE USE. The system discussed in this section tracks the number of times a variable is accessed in the call-by-name evaluation. Although we look at an example that tracks *variable usage*, the same system could be used to track access to resources that are always passed as a reference (and behave effectively as call-by-name) and so the system is relevant for call-by-value languages too. To demonstrate the idea, consider the following term:

$$(\lambda v.x + v + v)\ (x + y)$$

When evaluated, the body of the function directly accesses x once and then twice indirectly, via the function argument. Similarly, y is accessed twice indirectly. Thus, the overall expression uses x three times and y twice.

As discussed in Chapter 5, the system preserves type and coeffect annotations under the β-reduction. Reducing the expression in this case gives $x + (x + y) + (x + y)$. This has the same bounds as the original expression – x is used three times and y twice.

TYPE SYSTEM. The type system in Figure 20 annotates contexts with vectors of integers. The rules have the same structure as those of the system for liveness analysis. The only difference is how annotations are combined – here, we use integer multiplication ($*$) and addition ($+$).

Variable access (*var*) annotates a variable with 1, meaning that it has been used once. An unused variable (*weak*) is annotated with 0. Multiple occura.) Ordinary, syntax-driven rules with sub-coeffecting

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

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

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

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

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

b.) Structural rules for context manipulation

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

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

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

Figure 20: Structural coeffect bounded reuse analysis

rences of the same variable are introduced by contraction (*contr*), which adds the numbers of the two contracted variables.

As previously, application (*app*) and let binding (*let*) combine two separate contexts. The second part applies a function that uses its parameter t-times to an argument that uses variables in Γ_2 at most $\mathbf{s}$-times (here, $\mathbf{s}$ is a vector of integers with an annotations for each variable in Γ_2). The sequential composition (modelling call-by-name) multiplies the uses, meaning that the total number of uses is $(t * \mathbf{s})$ (where $*$ is a multiplication of a vector by a scalar). This models the fact that for each use of the function parameter, we replicate the variable uses in e_2.

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

EXAMPLE. To type check the expression $(\lambda v.x + v + v)\ (x + y)$ discussed earlier, we need to use abstraction, application, but also the contraction rule. Assuming the type judgement for the body, abstractions yields:

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

To type-check the application, the contexts of e_1 and e_2 need to contain disjoint variables. For this reason, we α-rename x to x' in the argument $(x + y)$ and later join x and x' using the contraction rule. Assuming $(x' + y)$

is checked in a context that marks x' and y as used once, the application rule yields a judgement that is simplified as follows:

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

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

SEMANTICS. In the previous examples, we defined the semantics – somewhat informally – using a simple λ-calculus language to encode the model. More formally, this could be a Cartesian closed category. In that model, we can reuse variables arbitrarily and so it is not a good fit for modelling bounded reuse. Girard et al. [28] model their bounded linear logic in an (ordinary) linear logic where variables can be used at most once.

Following the same approach, we could model a variable τ, annotated with r as a product containing r copies of τ, that is τ^r:

$$\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle \vdash e : \tau \rrbracket \;:\; (\tau_1^{r_1} \times \ldots \times \tau_n^{r_n}) \to \tau$$

$$\text{where } \tau_i^{r_i} = \underbrace{\tau_i \times \ldots \times \tau_i}_{r_i - \text{times}}$$

The functions are interpreted similarly. A function $\tau_1 \xrightarrow{t} \tau_2$ is modelled as a function taking t-element product of τ_1 values: $\tau_1^t \to \tau_2$.

The rules that define the semantics of bounded calculus are mostly the same as (or easy to adapt from) the semantic rules of liveness in Figure 19. The ones that differ are those that use sequential composition (application and let binding) and the contraction rule, which represents pointwise composition.

In the following, we use variable names $\mathbf{v}_i$ for context containing multiple variables (where each variable may be available multiple times), i. e. have a type $\tau_1^{r_1} \times \ldots \times \tau_m^{r_m}$; We do not explicitly write the sizes of these vectors (number of variables in a context; number of instances of a variable) as these are clear from the coeffect annotations. We assume that Γ_2 contains n variables and that $s = \langle s_1, \ldots, s_n \rangle$:

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

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

In the (*contr*) rule, the semantic function is called with $s + t$ copies of a value for the x variable. The values are split between s and t separate copies of variables y and z, respectively.

The (*app*) rule is similar in that it needs to split the input variable context. However, it needs to split values of multiple variables – in $x_{i,j}$, the index i stands for an index of the variable while j is an index of one of multiple copies of the value. In the semantic function, the second part of the context consists of n variables where the multiplicity of each value is specified by the annotation s_i multiplied by t. The rule needs to evaluate the argument e_2 t-times and each call requires s_i copies of the i^{th} variable. To do this, we create contexts $\mathbf{y}_1$ to $\mathbf{y}_t$, each containing s_i copies of the variable (and so we require $s_i * t$ copies of each variable). Note that the contexts are created such that each value is used exactly once.

It is worth noting that the (*var*) rule requires exactly one copy of a variable and so the system tracks precisely the number of uses. However, the (*sub*) rule lets us ignore additional copies of a value. Thus, permitting (*sub*) rule is only possible if the underlying model is *affine* rather than *linear*.

BOUNDED LINEAR LOGIC. The system presented in this section differs from bounded linear logic (BLL) [28]. Using the terminology from Section 2.2.3, our system is written in the *language semantics* style, while BLL is written in the *meta-language* style.

This means that the terms and types of our system are the terms and types of ordinary λ-calculus, with the only difference that functions carry coeffect annotations. In BLL, the language of types is extended with a type constructor $!_k A$ (where A is a proposition, corresponding to a type τ in our system). The type denotes a value A that can be used at most k times.

As a result, BLL does not need to attach additional annotation to the variable context as a whole. The requirements are attached to individual variables and so our context $\tau_1, \ldots, \tau_n @ \langle k_1, \ldots, k_n \rangle$ corresponds to a BLL assumption $!_{k_1} A_1, \ldots, !_{k_n} A_n$.

Using the formulation of bounded logic (and omitting the terms), the weakening and contraction rules are written as follows:

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

The system captures the same idea as the structural coeffect system presented above. Variable access in bounded linear logic is simply an operation that produces a value $!_n A$ and so the system further introduces *dereliction* rule which lets us treat $!_1 A$ as a value A. We further explore difference between *language semantics* and *meta-language* and also revisit the BLL example in Chapter **??**.

SUMMARY. Comparing the structural coeffect calculus for tracking liveness and for bounded variable reuse reveals which parts of the systems differ and which parts are shared. In particular, both systems use the same vector operations ($\times$, $\langle - \rangle$) and also share the lambda abstraction rule (*abs*). They differ in the primitive values used to annotate used and unused variables (L, D and 1, 0, respectively) and in the operators used for sequential composition and contraction ($\sqcup$, $\sqcap$ and $*$, $+$, respectively). The algebraic structure capturing these operators is developed in Chapter 5.

The brief overview of bounded linear logic shows an alternative approach to tracking properties related to individual variables – we could attach annotations to the variables themselves rather than attaching a *vector* of annotations to the entire context. Our approach has two benefits – it lets you unify flat and structural systems (Chapter **??**) and it also makes it possible to build composed systems that mix both flat and structural properties.

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

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

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

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

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

Figure 21: Structural coeffect bounded reuse analysis

3.3.3 *Data-flow languages revisited*

When discussing data-flow languages in the previous section, we said that the context provides past values of variables. In Section 3.2.4, we tracked this as a *flat* property, which gives us a system that keeps the same number of past values for all variables. However, data-flow can also be adapted to a structural system which keeps the number of required past values individually for each variable. Consider the following example:

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

The value offsetZip adds values of left with previous values of right. To evaluate a current value of the stream, we need the current value of left and one past value of right. Flat system is not able to capture this level-of-detail and simply requires 1 past values of both streams in the variable context.

Turning a flat data-flow system to a structural data-flow system is a change similar to the one between flat ans structural liveness. In case of liveness analysis, we included the flat system only as an illustration (it is not practically useful).

For data-flow, the flat system is less precise, but still practically useful (simplicity may outweigh precision). As discussed in Section X, the structural system is necessary when we allow arbitrary recursion and cannot (easily) determine the number of required values statically.

TYPE SYSTEM. The type system in Figure 21 annotates the variable context with a vector of integers. This is similar as in the bounded reuse system, but the integers *mean* a different thing. Consequently, they are also calculated differently. We omit rules that are the same for all structural coeffect systems (exchange, lambda abstraction).

In data-flow, we annotate both used variables (*var*) and unused variables (*weak*) with 0, meaning that no past values are required. This is the same as in flat data-flow, but different from bounded reuse and liveness (where difference between using and not using a variable matters). Primitive requirements are introduced by the (*prev*) rule, which increments the annotation of all variables in the context.

In flat data-flow, we identified sequential composition and point-wise composition as two primitive operations that were used in the (flat) application. In the structural system, these are used in (*app*) and (*contr*), respectively.

Thus application combines coeffect annotations using $+$ and contraction using *max*. This contrasts with bounded reuse, which uses $*$ and $+$, respectively.

EXAMPLE. As an example, consider a function $\lambda x.\textbf{prev}\ (y+x)$ applied to an argument $\textbf{prev}\ (\textbf{prev}\ y)$. The body of the function accesses the past value of two variables, one free and one bound. The (*abs*) rule splits the annotations between the declaration-site and call-site of the function:

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

The expression always requires the previous value of y and adds it to a previous value of the parameter x. Evaluating the value of the argument $\textbf{prev}\ (\textbf{prev}\ y)$ requires two past values of y and so the overall requirement for the (free) variable y is 3 past values. In order to use the contraction rule, we rename y to y' in the argument:

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

The derivation uses (*app*) to get requirements $\langle 1,3 \rangle$ and then (*contr*) to take the maximum, showing three past values are sufficient.

Note that we get the same requirements when we perform β reduction of the expression. Substituting the argument for x yields the expression $\textbf{prev}\ (y + (\textbf{prev}\ (\textbf{prev}\ y)))$. Semantically, this performs stream lookups $y[1]$ and $y[3]$ where the indices are the number of enclosing $\textbf{prev}$ constructs.

SEMANTICS. To define the semantics of our structural data-flow language, we can use the same approach as when adapting flat liveness to structural liveness. Rather than wrapping the whole context in some wrapper (list or option type), we now wrap individual components of the product representing the variables in the context.

The result is similar as the structure used for bounded reuse. The only difference is that, given a variable annotated with r, we need $1+r$ values. That is, we need the current value, followed by r past values:

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

Despite the similarity with the semantics for bounded reuse, the values here *represent* different things. Rather than providing multiple copies of a value (out of which each can be used just once), the pair provides past values (that can be reused and freely accessed). To illustrate the behaviour we consider the semantics of the $\textbf{prev}$ construct and of the structural contraction rule:

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

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

In (*prev*), the semantic function is called with an argument that stores values of n variables, such that a variable x_i has values ranging from $x_{i,0}$ to x_{i,s_i+1}. Thus, there is one current value, followed by $s_i + 1$ past values. The expression e nested under **prev** requires only s_i past values and so the semantics simply drops the last value.

In the (*contr*) rule, the semantic function receives $max(s, t)$ values of a specific variable x. It needs to produce values for two separate variables, y and z that require s and t past values. Both of these numbers are certainly smaller than (or equal to) the number of values available. Thus we simply take the first values. Unlike in the contraction for BLL, the values are duplicated and the same values are used for both variables.

SUMMARY. Two of the structural examples shown so far (liveness and data-flow) extend an earlier flat version of a similar system. We discuss this relation in general later. However, a flat system can generally be turned into a structural one – although this only gives a useful system when the flat version captures properties related to variables.

The data-flow example demonstrates that the a flat system can also be turned into structural system. In general, this only works for systems where lambda abstraction duplicates context requirements (as in Figure 14).

3.3.4 *Security, tainting and provenance*

Tainting is a mechanism where variables coming from potentially untrusted sources are marked (*tainted*) and the use of such variables is disallowed in contexts where untrusted input can cause security issues or other problems. Tainting can be done dynamically as a runtime mark (e.g. in the Perl language) or statically using a type system. Tainting can be viewed as a special case of *provenance tracking*, known from database systems [13], 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 [78] and a well known attack known as SQL injection [32, 31]. In the latter chase, we want to check that SQL commands cannot be directly constructed from, potentially dangerous, inputs provided by the user. Consider the type checking of the following expression in a context containing variables id and msg:

```
let name = query("SELECT Name WHERE Id = %1",id)
msg + name
```

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

In runtime checking, all (string) values need to be wrapped in an object that stores Boolean flag (for tainting) or more complex data (for provenance). In static checking, the information need to be associated with the variables in the variable context.

CORE DEPENDENCY CALCULUS. The checking of tainting is a special case of checking of the *non-interference* property in *secure information flow*. There, the aim is to guarantee that sensitive information (such as credit card number) cannot be leaked to contexts with low secrecy (e.g. sent via an unsecured network channel). Volpano et al. [79] provide the first (provably) sound type system that guarantees non-inference and Sabelfeld et al. [60] survey more recent work. The checking of information flows has been also integrated (as a single-purpose extension) in the FlowCaml [64] language. Finally, Russo et al. and Swamy et al. [59, 66] show that the properties can be checked using a monadic library.

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

IMPLICIT FLOWS. An important aspect of secure information flow is called *implicit flows*. Consider the following example which returns either y or zero, depending on the value of x:

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

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

Abadi et al. realized that there is a number of analyses similar to secure information flow and proposed to unify them using a single model called Dependency Core Calculus (DCC) [1]. It captures other cases where some information about expression relies on properties of variables in the context where it executes. The DCC captures, for example, *binding time analysis* [72], which detects which parts of programs can be partially evaluated (do not depend on user input) and *program slicing* [73] that identifies parts of programs that contribute to the output of an expression.

COEFFECT SYSTEMS. The work outlined in this section is another area where coeffect systems could be applied. We do not develop coeffect systems for tracking of tainting, security and provenance in details, but briefly mention some examples in the upcoming chapters.

The systems work in the same way as the examples discussed already. For example, consider the tainting example with the query function calling an SQL database. To capture such tainting, we annotate variables with T for *tainted* and with U for *untainted*. Simply accessing variable does not introduce taint, but using a variable in certain contexts – such as in arguments of query – does introduce a taint. This is captured using the standard application rule (*app*):

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

The derivation assumes that query is a standard function that requires the parameters to be tainted (it does not have to be a built-in language construct). The argument is a variable and so it is not tainted in the assumptions.

In the conclusion, we need to derive an annotation for the variable id. To do this, we combine T (from the function) and U (from the argument). In case of tainting, the variable is tainted whenever it is already tainted *or* the function marks it as tainted. For different kinds of annotations, the composition would work differently – for example, for provenance, we could union the *set* of possible data sources, or even combine *probability distributions* modelling the influence of different sources on the value. However, expanding such ideas is beyond the scope of this thesis.

3.4 BEYOND PASSIVE CONTEXTS

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

There is a number of systems that also capture contextual properties, but that make it possible to *change* the context – not just be evaluating certain code block in a different scope (e.g. by wrapping it in prev in data-flow), but also by calling a function that, for example, acquires new capabilities and returns those to the caller. While this thesis focuses on systems with passive contexts, we briefly consider the most important examples of the *active* variant.

CALCULUS OF CAPABILITIES. Crary et al. [17] 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 [74] who developed an effect system (as discussed in Section 2.2.2) that uses lexically scoped *memory regions* to provide an efficient and controlled memory management.

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

```
let calculate = λinput →
  letrgn ρ in
  let x = ref$_\rho$ input in
  x := !x + 1; !x
```

The memory region ρ is a part of the context, but only in the scope of the body of **letrgn**. It is only available to the last two lines which allocate a memory cell in the region, increment a value in the region and then read it. The region is de-allocated when the execution leaves its lexical scope – there is no way to allocate a region inside a function and pass it back to the caller.

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

```
let calculate = λinput →
  letrgn ρ in
  let x = ref$_\rho$ input in
  x := !x + 1; x
```

The difference is that the example does not return the *value* of a reference using !x, but returns the reference x itself. The reference is allocated in a newly created region ρ. Together with the value, the function returns a *capability* to access the region ρ.

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

ACTIVE CONTEXTS. In a systems with passive contexts, we only need an annotation that specifies the required context. In semantics, this is reflected by having some structure (data type) $\mathcal{C}$ over the *input* of the function. Without giving any details, the semantics generally has the following structure:

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

Systems with active contexts require two annotations – one that specifies the context required before the call is performed and one that specifies how the context changes after the call (this could be either a *new* context or *update* to the original context). Thus the structure of the semantics would look as follows:

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

In case of Calculus of Capabilities, both of the structures could be the same and they could carry a set of available memory regions. In this thesis, we focus only on passive contexts. However, we briefly consider the problem of active contexts in the Section X of the future work chapter.

SOFTWARE UPDATING. Another example of a system that uses contextual information actively is dynamic software updating (DSU) [25, 34]. The DSU systems have the ability to update programs at runtime without stopping them. For example, Proteus developed by Stoyle et al. [65] 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*.

3.5 SUMMARY

This chapter served two purposes. The first aim was to present existing work on programming languages and systems that include some notion of *context*. Because there was no well-known abstraction capturing contextual properties, the languages use a wide range of formalisms – including from principled approaches based on comonads, ad-hoc type system extensions and static analyses as well as approaches based on monads. We looked at a number of applications inclding Haskell's implicit parameters and type classes, data-flow languages such as Lucid, liveness analysis and also a number of security properties.

The second aim of this chapter was to re-formulate the existing work in a more uniform style and thus reveal that all *context-dependent* languages share a common structure. In the upcoming three chapters, we identify the common structure more precisely and develop three calculi to capture it. We will then be able to re-create many of the examples discussed in this chapter by instantiating our unified calculi.

This chapter was divided into two major sections. First, we looked at *flat* systems, which track whole-context properties. Next, we look a *structural* systems, which track per-variable properties. Both of the variants are useful and important – for example, implicit parameters can only be expressed as *flat* system, but liveness analysis is only useful as *structural*. For this reason, we explore both of these variants in this thesis (Chapter 4 and Chapter 5, respectively). We can, however, unify the two variants into a single system discussed in Chapter **??**.

4 FLAT COEFFECT CALCULUS

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

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

4.1 INTRODUCTION

In the previous chapter, we looked at three important examples of systems that track whole-context properties. The type systems for whole-context liveness (Section 3.2.3) and whole-context data-flow (Section 3.2.4) have a very similar structure – their lambda abstraction duplicates the requirements and their application arises from the combination of *sequential* and *point-wise* composition.

The system for tracking of implicit parameters (Section 3.2.1), and similar systems for rebindable resources, differ in two ways. In lambda abstraction, they split the context requirements between the declaration-site and the call-site and they use only a single operator on the indices, typically $\cup$.

4.1.1 *Contributions*

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

- We present a *flat coeffect calculus* with a type system that is parameterized by a *flat coeffect algebra* and can be instantiated to obtain all of the three examples discussed (Section 4.2).
- We give the equational theory of the calculus and discuss type-preservation for call-by-name and call-by-value reduction (Section 4.4). We also extend the calculus with subtyping and pairs (Section 4.5).
- We present the semantics of the calculus in terms of *indexed comonads*, which is a generalization of comonads, a category-theoretical dual of monads (Section 4.3). The semantics provides deeper insight into how (and why) the calculus works.

4.1.2 *Related work*

The development in this chapter can be seen as a counterpart to the well-known development of *effect systems* [27] and the use of *monads* [45] in programming languages. The syntax and type system of the flat coeffect calculus follows similar style as effect systems [43, 70], but differs in the structure, as explained in the previous chapter, most importantly in lambda abstraction.

Wadler and Thiemann famously show a correspondence between effect systems to monads [86], relating effectful functions $\tau_1 \xrightarrow{\sigma} \tau_2$ to monadic computations $\tau_1 \rightarrow M^{\sigma}\tau_2$. In this chapter, we show a similar correspondence between *coeffect systems* and *comonads*. However, due to the asymmetry of λ-calculus, this is not a simple mechanical dualization.

The main purpose of the comonadic semantics presented in this chapter is to provide a semantic motivation for the flat coeffect calculus. The semantics is inspired by the work of Uustalu and Vene [76] who present the semantics of contextual computations (mainly for data-flow) in terms of comonadic functions $C\tau_1 \rightarrow \tau_2$. Our *indexed comonads* annotate the structure with information about the required context, i. e. $C^{\sigma}\tau_1 \rightarrow \tau_2$. This is similar to the recent work on *parameterized monads* by Katsumata [37].

4.2 FLAT COEFFECT CALCULUS

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

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

We discuss subtyping and pairs in Section 4.5. The type $\tau_1 \xrightarrow{r} \tau_2$ represents a function from τ_1 to τ_2 that requires additional context r. It can be viewed as a pure function that takes τ_1 *with* or *wrapped in* a context r.

In the categorical semantics, the function $\tau_1 \xrightarrow{r} \tau_2$ is modelled by a morphism $C^{r}\tau_1 \rightarrow \tau_2$. However, the object C^{r} does not exist as a syntactical value. This is because we use comonads to define the *semantics* rather than *embedding* them into the language as in the meta-language approaches (the distinction between the two approaches has been discussed in detail in Section 2.2.1). The annotations r are formed by an algebraic structure discussed next.

4.2.1 *Reconciling lambda abstraction*

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

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

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

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

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

4.2.2 *Flat coeffect algebra*

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

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

Definition 3. *A* flat coeffect algebra $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, relation* $\leqslant$ *and binary operations* $\circledast, \oplus, \wedge$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *and* $(\mathcal{C}, \oplus, \mathsf{ign})$ *are monoids,* $(\mathcal{C}, \leqslant)$ *is a pre-order and* $(\mathcal{C}, \wedge)$ *is a band (idempotent semigroup). That is, for all* $r, s, t \in \mathcal{C}$*:*

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

In addition, the following distributivity axioms hold:

$$\begin{array}{l}
(r \oplus s) \circledast t = (r \circledast t) \oplus (s \circledast t) \\
t \circledast (r \oplus s) = (t \circledast r) \oplus (t \circledast s)
\end{array}$$

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

- The monoid $(\mathcal{C}, \circledast, \mathsf{use})$ represents *sequential* composition of (semantic) functions. The laws of a monoid are required in order to form a category structure in the categorical model (Section 4.3).
- The monoid $(\mathcal{C}, \oplus, \mathsf{ign})$ represents *point-wise* composition, i. e. the case when the same context is passed to multiple (independent) computations. The monoid laws guarantee that usual syntactic transformations on tuples and the unit value (Section 4.5) preserve the coeffect.
- For the $\wedge$ operation, we require associativity and idempotence. The idempotence requirement makes it possible to duplicate the coeffects and place the same requirement on both call-site and declaration-site,

i. e. it makes the (*abs-df*) rule admissible. In some cases, the operator forms a monoid with the unit being the greatest element of the set.

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

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

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

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

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

4.2.3 *Understanding flat coeffects*

Before looking at the type system in Figure 22, let us clarify how the rules should be understood. The coeffect calculus provides both analysis of context dependence (type system) and semantics for context (how it is propagated). These two aspects provide different ways of reading the judgements $\Gamma @ r \vdash e : \tau$ and the typing rules used to define it.

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

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

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

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

The reason for this asymmetry follows from the fact that the context appears in a *negative position* in the semantic model (Section 4.3). It means that we need to be careful about using the words *split* and *merge*, because they can be read as meaning opposite things. To disambiguate, we always use the term *context requirements* when using the syntactic view and *context capabilities* or just *available context* when using the semantic view.

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

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

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

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

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

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

Figure 22: Type system for the flat coeffect calculus

4.2.4 *Flat coeffect types*

The type system for flat coeffect calculus is shown in Figure 22. Variables (*var*) and constants (*const*) are annotated with special values provided by the coeffect algebra. Following the top-down syntactic reading, the (*sub*) rule allows us to treat an expression with fewer context requirements as an expression with more context requirements.

The (*abs*) rule is defined as discussed in Section 4.2.1. The body is annotated with context requirements $r \wedge s$, which are then split between the context-requirements on the declaration-site r and context-requirements on the call-site s. Examples of the $\wedge$ operator are discussed in the next section.

In function application (*app*), context requirements of both expressions and the function are combined as discussed in Chapter 3. The pointwise composition $\oplus$ is used to combine the context requirements of the expression representing a function r and the context requirements of the argument, sequentially composed with the context-requirements of the function $s \circledast t$.

The type system also includes a rule for let-binding. The rule is *not* equivalent to the derivation for $(\lambda x.e_2)\ e_1$, but it represents one admissible typing derivation. We return to let-binding after looking a number of examples. Additional constructs such as pairs are covered in Section 4.5.

4.2.5 *Examples of flat coeffects*

The flat coeffect calculus generalizes the flat systems discussed in Section 3.2 of the previous chapter. We can instantiate it to a specific use just by providing a flat coeffect algebra. The following summary defines the systems for implicit parameters, liveness and data-flow. For the latter two, we obtain more general (but compatible) rule for lambda abstraction.

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

For simplicity, we assume that all parameters have the same type ρ and so the annotations only track sets of names. The definition uses set union for all three operations. Both variables and constants are are annotated with $\emptyset$ and the ordering is defined by $\subseteq$. The definition satisfies the flat coeffect algebra laws because $(S, \cup, \emptyset)$ is an idempotent, commutative monoid. The system has a single additional typing rule for accessing the value of a parameter:

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

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

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

As in Section 3.2.3, sequential composition $\circledast$ is modelled by the meet operation $\sqcap$ and point-wise composition $\oplus$ is modelled by join $\sqcup$. Two-point lattice is a commutative, idempotent monoid. The distributivity $(r \sqcup s) \sqcap t = (r \sqcap t) \sqcup (s \sqcap t)$ does not hold for *every* lattice, but it trivially holds for a two-point lattice used here.

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

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

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

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

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

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

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

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

4.2.6 *Typing of let binding*

Recall the (*let*) rule in Figure 22. It annotates the expression **let** $x = e_1$ **in** e_2 with context requirements $s \oplus (s \circledast r)$. This is a special case of typing of an expression $(\lambda x.e_2)\ e_1$, using the idempotence of $\wedge$ as follows:

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

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

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

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

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

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

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

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

4.3 CATEGORICAL MOTIVATION

The type system of flat coeffect calculus arises as a generalization of the examples discussed in Chapter 3, but we can also obtain it by looking at the categorical semantics of context-dependent computations. This is a direction that we explore in this section. Although the development presented here is interesting in its own, our main focus is *using* categorical semantics to motivate and explain the design of flat coeffect calculus.

4.3.1 *Categorical semantics*

As discussed in Section 2.2, categorical semantics interprets terms as morphisms in some category. For typed calculi, the semantics defined by $[\![-]\!]$ usually interprets typing judgements $x_1 : \tau_1 \ldots x_n : \tau_n \vdash e : \tau$ as morphisms $[\![\tau_1 \times \ldots \times \tau_n]\!] \to [\![\tau]\!]$.

As a best known example, Moggi [45] showed that the semantics of various effectful computations can be captured uniformly using the (*strong*) *monad* structure. In that approach, computations are interpreted as $\tau_1 \times \ldots \times \tau_n \to M\tau$ for some monad M. For example, $M\alpha = \alpha \cup \{\bot\}$ models partiality (maybe monad), $M\alpha = \mathcal{P}(\alpha)$ models non-determinism (list monad) and $M\alpha = (\alpha \times S)^S$ models side-effects (state monad). Here, the structure of a strong monad provides necessary "plumbing" for composing monadic computations.

Following similar approach to Moggi, Uustalu and Vene [76] showed that (*monoidal*) *comonads* uniformly capture the semantics of various kinds of context-dependent computations [76]. For example, data-flow computations over non-empty lists $\mathsf{NEList}\,\alpha = \alpha + (\alpha \times \mathsf{NEList}\,\alpha)$ are modelled using the non-empty list comonad.

The monadic and comonadic model outlined here represents at most a binary analysis of effects or context-dependence. A function $\tau_1 \to \tau_2$ performs *no* effects (requires no context) whereas $\tau_1 \to M\tau_2$ performs *some* effects and $C\tau_1 \to \tau_2$ requires *some* context. In the next section, we introduce *indexed comonads*, which provide a more precise analysis and let us model computations with context requirements r as functions $C^r\tau_1 \to \tau_2$ using an *indexed comonad* C^r.

4.3.2 *Introducing comonads*

In category theory, *comonad* is a dual of *monad*. Informally, we get a comonad by taking a monad and "reversing the arrows". More formally, one of the equivalent definitions of comonad looks as follows:

Definition 4. *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 $f : C\alpha \to \beta$ *and* $g : C\beta \to \gamma$*:*

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

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

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

From the functional programming perspective, we can see C as a parametric data type such as NEList. The counit operations 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.

As mentioned earlier, Uustalu and Vene [76] use comonads to model dataflow computations. They describe infinite (coinductive) streams and non-empty lists as example comonads.

Example 5 (Non-empty list)**.** *A non-empty list is a recursive data-type defined as* $\mathsf{NEList}\ \alpha = \alpha + (\alpha \times \mathsf{NEList}\ \alpha)$*. We write* $\mathbf{inl}$ *and* $\mathbf{inr}$ *for constructors of the left and right cases, respectively. The type* NEList *forms a comonad together with the following* counit *and* cobind *mappings:*

$$\begin{array}{rll}
\mathsf{counit}\ l & = h & \text{when } l = \mathbf{inl}\ h \\
\mathsf{counit}\ l & = h & \text{when } l = \mathbf{inr}\ (h, t) \\
\\
\mathsf{cobind}\ f\ l & = \mathbf{inl}\ (f\ l) & \text{when } l = \mathbf{inl}\ h \\
\mathsf{cobind}\ f\ l & = \mathbf{inr}\ (f\ l,\ \mathsf{cobind}\ f\ t) & \text{when } l = \mathbf{inr}\ (h, t)
\end{array}$$

The counit operation returns the head of the non-empty list. Note that it is crucial that the list is *non-empty*, because we always need to be able to obtain a value. The cobind defined here returns a list of the same length as the original where, for each element, the function f is applied on a *suffix* list starting from the element. Using a simplified notation for list, the result of applying cobind to a function that sums elements of a list gives the following behaviour:

$$\mathsf{cobind}\ \mathsf{sum}\ (7, 6, 5, 4, 3, 2, 1, 0) = (28, 21, 15, 10, 6, 3, 1, 0)$$

The fact that the function f is applied to a *suffix* is important in order to satisfy the *left identity* law, which requires that $\mathsf{cobind}\ \mathsf{counit}\ l = l$.

It is also interesting to examine some data types that do *not* form a comonad. As already mentioned, list $\mathsf{List}\ \alpha = 1 + (\alpha \times \mathsf{List}\ \alpha)$ is not a comonad, because the counit operation is not defined for the value $\mathbf{inl}\ ()$. Similarly, the Maybe data type defined as $1 + \alpha$ is not a comonad for the same reason. However, if we consider flat coeffect calculus for liveness, it appears natural to model computations as function $\mathsf{Maybe}\ \tau_1 \to \tau_2$. To use such model, we first need to generalise comonads to *indexed comonads*.

4.3.3 *Generalising to indexed comonads*

The flat coeffect algebra includes a monoid $(\mathcal{C}, \circledast, \mathsf{use})$, which defines the behaviour of sequential composition, where the annotation use represents a variable access. An indexed comonad is formed by a data type (object mapping) $C^r\alpha$ where the annotation r determines what context is required.

Definition 5. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with binary operator* $\circledast$ *and unit* use*, an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{r,s})$ *where:*

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

such that, for all $f : C^r\alpha \to \beta$ *and* $g : C^s\beta \to \gamma$ *and the identity* $\mathsf{id}_s : C^s\alpha \to C^s\alpha$*:*

$$\begin{array}{rr}
\mathsf{cobind}_{\mathsf{use},s}\ \mathsf{counit}_{\mathsf{use}} = \mathsf{id} & \textit{(left identity)} \\
\mathsf{counit}_{\mathsf{use}} \circ \mathsf{cobind}_{r,\mathsf{use}}\ f = f & \textit{(right identity)} \\
\mathsf{cobind}_{r \circledast s, t}\ (g \circ \mathsf{cobind}_{r,s}\ f) = (\mathsf{cobind}_{s,t}\ g) \circ (\mathsf{cobind}_{r, s \circledast t}\ f) & \textit{(associativity)}
\end{array}$$

Rather than defining a single mapping C, we are now defining a family of mappings C^r indexed by a monoid structure. Similarly, the operation $\mathsf{cobind}_{r,s}$ operation is now also formed by a *family* of mappings for different pairs of indices r, s. To be fully precise, cobind is a family of natural transformations and we should include α, β as indices, writing $\mathsf{cobind}^{\alpha,\beta}_{r,s}$. For the purpose of this thesis, it is sufficient to treat cobind as a family of mappings or, when it does not introduce ambiguity, view it as a single mapping.

The counit operation is not defined for all $r \in \mathcal{C}$, but only for the unit use. We still include the unit as an index writing $\mathsf{counit}_{\mathsf{use}}$, but this is merely for symmetry. Crucially, this means that the operation is defined only for some special contexts.

If we look at the indices in the laws, we can see that the left and right identity require use to be the unit of $\circledast$. Similarly, the associativity law implies the associativity of the $\circledast$ operator.

The category that models sequential composition is formed by the unit arrow counit together with the (associative) composition operation that composes computations with contextual requirements as follows:

$$\begin{aligned} -\hat{\circ}- &: (C^{r}\tau_1 \to \tau_2) \to (C^{s}\tau_2 \to \tau_3) \to (C^{r \circledast s}\tau_1 \to \tau_3) \\ g \,\hat{\circ}\, f &= g \circ (\mathsf{cobind}_{r,s}\, f) \end{aligned}$$

The composition $\hat{\circ}$ best expresses the intention of indexed comonads. Given two functions with contextual requirements r and s, their composition is a function that requires $r \circledast, s$. The contextual requirements propagate *backwards* and are attached to the input of the composed function.

EXAMPLES. Any comonad can be turned into an indexed comonad using a trivial monoid. However, indexed comonads are more general and can be used with other data types, including indexed Maybe.

Example 6 (Comonads)**.** *Any comonad C is an indexed comonad with an index provided by a trivial monoid $(\{1\}, *, 1)$ where $1 * 1 = 1$ and C^1 is the underlying mapping C of the original comonad. The operations counit_1 and $\mathsf{cobind}_{1,1}$ are defined by the operations counit and cobind of the comonad.*

Example 7 (Indexed option)**.** *The indexed option comonad is defined over a monoid $(\{\mathsf{L}, \mathsf{D}\}, \sqcup, \mathsf{L})$ where $\sqcup$ is defined as earlier, i. e. $\mathsf{L} = r \sqcup s \iff r = s = \mathsf{L}$. Assuming 1 is the unit type inhabited by $()$, the mappings are defined as follows:*

$$\begin{array}{ll} C^{\mathsf{L}}\alpha = \alpha & \mathsf{cobind}_{r,s} \;:\; (C^{r}\alpha \to \beta) \to (C^{r \sqcup s}\alpha \to C^{s}\beta) \\ C^{\mathsf{D}}\alpha = 1 & \mathsf{cobind}_{\mathsf{L},\mathsf{L}}\; f\; x \;\;= f\; x \\ & \mathsf{cobind}_{\mathsf{L},\mathsf{D}}\; f\; () \;= () \\ \mathsf{counit}_{\mathsf{L}} : C^{\mathsf{L}}\alpha \to \alpha & \mathsf{cobind}_{\mathsf{D},\mathsf{L}}\; f\; () \;= f\; () \\ \mathsf{counit}_{\mathsf{L}}\; v = v & \mathsf{cobind}_{\mathsf{D},\mathsf{D}}\; f\; () \;= () \end{array}$$

The indexed option comonad models the semantics of the liveness coeffect system discussed in 3.2.3, where $C^{\mathsf{L}}\alpha = \alpha$ models a live context and $C^{\mathsf{D}}\alpha = 1$ models a dead context which does not contain a value. The counit operation extracts a value from a live context; cobind can be seen as an implementation of dead code elimination. The definition only evaluates f when the result is marked as live and is thus required, and it only accesses x if the function f requires its input.

The indexed family C^{r} in the above example is analogous to the Maybe (or option) data type $\mathsf{Maybe}\,\alpha = 1 + \alpha$. As mentioned earlier, this type does not permit (non-indexed) comonad structure, because $\mathsf{counit}\;()$ is not defined. This is not a problem with indexed comonads, because counit only needs to be defined on live context.

Example 8 (Indexed product)**.** *The semantics of implicit parameters is modelled by an indexed product comonad. We use a monoid $(\mathcal{P}(\mathsf{Id}), \cup, \emptyset)$ where Id is the set of (implicit parameter) names. As previously, all parameters have the type ρ. The data type $C^{r}\alpha = \alpha \times (r \to R)$ represents a value α together with a function that* *associates a parameter value* ρ *with every implicit parameter name in* $r \subseteq \mathsf{Id}$. *The cobind and counit operations are defined as:*

$$\begin{array}{ll} \mathsf{counit}_{\emptyset} : C^{\emptyset}\alpha \to \alpha & \mathsf{cobind}_{r,s} \;:\; (C^{r}\alpha \to \beta) \to (C^{r \cup s}\alpha \to C^{s}\beta) \\ \mathsf{counit}_{\emptyset}\ (a, g) = a & \mathsf{cobind}_{r,s}\ f\ (a, g) = (f(a, g|_{r}), g|_{s}) \end{array}$$

The definition of counit simply ignores the function and returns the value in the context. The cobind operation uses the restriction operation $f|_{r}$, which we already defined when discussing semantics of implicit parameters in Section 3.2.1 (indeed, cobind here captures an essential part of the semantics).

The function g in cobind is defined on the union of the implicit parameters, i. e. $r \cup s \to \rho$. When passing it to f, we restrict it to just r and when returning it as a result, we restrict it to s.

4.3.4 *Properties and related notions*

We discuss additional examples in Section 4.3.5, after we look at the remaining structure that is needed to define the semantics of flat coeffect calculus. Before doing so, we discuss additional properties and categorical structures that have been proposed mainly in the context of monads and effects and are related to indexed comonads.

SHAPE PRESERVATION. Ordinary comonads have the *shape preservation* property [53]. Intuitively, this means that the shape of the additional context does not change during the computation. For example, in the NEList comonad, the length of the list stays the same after applying cobind.

Indexed comonads are not restricted by this property of comonads. For example, given the indexed product monad, in the computation $\mathsf{cobind}_{r,s} f$ above, the shape of the context changes from containing implicit parameters $r \cup s$ to containing just implicit parameters s.

FAMILIES OF MONADS. When linking effect systems and monads, Wadler and Thiemann [45] propose a *family of monads* as the categorical structure. The dual structure, *family of comonads*, is defined as follows.

Definition 6. *A* family of comonads *is formed by triples* $(C^{r}, \mathsf{cobind}_{r}, \mathsf{counit}_{r})$ *for all* r *such that each triple forms a comonad. Given* r, r' *such that* $r \leqslant r'$, *there is also a mapping* $\iota_{r',r} : C^{r'} \to C^{r}$ *satisfying certain coherence conditions.*

Family of comonads is more restrictive than indexed comonad, because each of the data types needs to form a comonad separately. For example, our indexed option does not form a family of comonads (again, because counit is not defined on $C^{D}\alpha = 1$). However, given a family of comonads and indices such that $r \leqslant r \circledast s$, we can define an indexed comonad. Briefly, to define $\mathsf{cobind}_{r,s}$ of an indexed comonad, we use $\mathsf{cobind}_{r \circledast s}$ from the family, together with two lifting operations: $\iota_{r \circledast s, r}$ and $\iota_{r \circledast s, s}$.

PARAMETERIC EFFECT MONADS. Parametric effect monads introduced by Katsumata [37] (independently to our indexed comonads) are closely related to our definition. Although presented in a more general categorical framework (and using monads), the model defines unit operation only on the unit of a monoid and bind operation composes effect annotations using the provided monoidal structure.

4.3.5 Flat indexed comonads

Indexed comonads model the semantics of sequential composition, but additional structure is needed to model the semantics of the flat coeffect calculus. This is where the duality between monads and comonads can no longer help us, because context is propagated differently than effects in lambda abstraction and application.

Whereas Moggi [45] requires *strong* monad to model effectful λ-calculus, Uustalu and Vene [76] require *lax semi-monoidal* comonad to model λ-calculus with contextual properties. The structure requires a monoidal operation:

$$\mathsf{m} : C\alpha \times C\beta \to C(\alpha \times \beta)$$

The m operation is needed in the semantics of lambda abstraction. It represents merging of contexts and is used to merge the context of the declaration-site (containing free variables) and the call-site (containing bound variable). For example, for implicit parameters, this combines the additional parameters defined in the two contexts.

The semantics of flat coeffect calculus requires operations for *merging*, but also for *splitting* of contexts. These are provided by *lax* and *oplax* monoidal structures. In addition, we also need a lifting operation (similar to ι from Definition 6) to model sub-coeffecting.

Definition 7. *Given a flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$, *an* flat indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$ *and* $\mathsf{lift}_{r',r}$ *where:*

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

The $\mathsf{merge}_{r,s}$ operation is the most interesting one. Given two comonadic values with additional contexts specified by r and s, it combineds them into a single value with additional context $r \wedge s$. The $\wedge$ operation often represents *greatest lower bound*[1], elucidating the fact that merging may result in the loss of some parts of the contexts r and s. We look at examples of this operation in the next section.

The $\mathsf{split}_{r,s}$ operation splits a single comonadic value (containing a tuple) into two separate values. Note that this does not simply duplicate the value, because the additional context is also split. To obtain coeffects r and s, the input needs to provide *at least* r and s, so the tags are combined using the $\oplus$, which is often the *least upper-bound*[1].

Finally, $\mathsf{lift}_{r',r}$ is a family of operations that "forget" some part of a context. This models the sub-coeffecting operation and lets us, for example, forget some of the available implicit parameters, or turn a live context (containing a value) into a dead context (empty).

ALTERNATIVE DEFINITION. Although we do not require this as a general law, in all our systems, it is the case that $r \leqslant r \oplus s$ and $s \leqslant r \oplus s$. This allows a simpler definition of *indexed flat comonad* by expressing the split operation in terms of the lifting (sub-coeffecting) as follows:

$$\begin{aligned} \mathsf{map}_r\ f &= \mathsf{cobind}_{r,r}\ (f \circ \mathsf{counit}_{\mathsf{use}}) \\ \mathsf{split}_{r,s}\ c &= (\mathsf{map}_r\ \mathsf{fst}\ (\mathsf{lift}_{r \oplus s, r}\ c), \mathsf{map}_s\ \mathsf{snd}\ (\mathsf{lift}_{r \oplus s, s}\ c)) \end{aligned}$$

[1] The $\wedge$ and $\oplus$ operations are the greatest and least upper bounds for the liveness and data-flow examples, but not for implicit parameters. However, they are useful as an informal analogy.

The map_r operation is the mapping on functions that corresponds to the object mapping C^r. The definition is dual to the standard definition of map for monads in terms of bind and unit. The functions fst and snd are first and second projections from a two-element pair. To define the $\mathsf{split}_{r,s}$ operation, we duplicate the argument c, then use lifting to throw away additional parts of the context and then transform the values in the context.

This alternative is valid for our examples, but we do not use it for two reasons. Firstly, it requires duplication of the value c, which is not required elsewhere in our model. So, using explicit split, our model could be embedded in a linear or affine model. Secondly, it is similar to the definition that is needed for structural coeffects in Chapter 5 and so it makes the connection between the two system easier to see.

EXAMPLES. All examples of *indexed comonads* discussed in Section 4.3.3 can be extended into *flat indexed comonads*.

Example 9 (Monoidal comonads). *Just like indexed comonads generalise comonads, the additional structure of flat indexed comonads generalises symmetric semimonoidal comonads of Uustalu and Vene [76]. The flat coeffect algebra is defined as* $(\{1\}, *, *, *, 1, 1, =)$ *where* $1 * 1 = 1$ *and* $1 = 1$. *The additional operation* $\mathsf{merge}_{1,1}$ *is provided by the monoidal operation called* m *by Uustalu and Vene. The* $\mathsf{split}_{1,1}$ *operation is defined by duplication and* $\mathsf{lift}_{1,1}$ *is the identity function.*

Example 10 (Indexed option). *Flat coeffect algebra for liveness defines* $\oplus$ *and* $\wedge$ *as* $\sqcup$ *and* $\sqcap$, *respectively and specifies that* $\mathsf{D} \sqsubseteq \mathsf{L}$. *Recall also that the object mapping is defined as* $C^{\mathsf{L}}\alpha = \alpha$ *and* $C^{\mathsf{D}}\alpha = 1$. *The additional operations of a flat indexed comonad are defined as follows:*

$$\begin{aligned}
\mathsf{merge}_{\mathsf{L},\mathsf{L}}\ (a,b) &= (a,b) \\
\mathsf{merge}_{\mathsf{L},\mathsf{D}}\ (a,()) &= () \\
\mathsf{merge}_{\mathsf{D},\mathsf{L}}\ ((),b) &= () \\
\mathsf{merge}_{\mathsf{D},\mathsf{D}}\ ((),()) &= ()
\end{aligned}
\qquad
\begin{aligned}
\mathsf{split}_{\mathsf{L},\mathsf{L}}\ (a,b) &= (a,b) \\
\mathsf{split}_{\mathsf{L},\mathsf{D}}\ (a,b) &= (a,()) \\
\mathsf{split}_{\mathsf{D},\mathsf{L}}\ (a,b) &= ((),b) \\
\mathsf{split}_{\mathsf{D},\mathsf{D}}\ () &= ((),())
\end{aligned}
\qquad
\begin{aligned}
\mathsf{lift}_{\mathsf{L},\mathsf{D}}\ v &= () \\
\mathsf{lift}_{\mathsf{L},\mathsf{L}}\ v &= v \\
\mathsf{lift}_{\mathsf{D},\mathsf{D}}\ () &= ()
\end{aligned}$$

Without the indexing, the merge operations implements *zip* on option values, returning an option only when both values are present. The behaviour of the split operation is partly determined by the indices. When the input is *dead*, both values have to be dead (this is also the only solution of $\mathsf{D} = r \sqcap \mathsf{D}$), but when the input is *live*, the operation can perform implicit sub-coeffecting and drop one of the values.

Explicit sub-coeffecting using the (*sub*) rule is modelled by the lift operation. This can turn a *live* value v into a dead value $()$, or it can behave as identity. The behaviour is, again, determined by the index.

Example 11 (Indexed product). *For implicit parameters, both* $\wedge$ *and* $\oplus$ *are the* $\cup$ *operation and the relation* $\leqslant$ *is formed by the subset relation* $\subseteq$. *Recall that the data type* $C^r\alpha$ *is* $\alpha \times (r \to \mathsf{R})$ *where* R *is some representation of a parameter value. The additional operations are defined as:*

$$\begin{aligned}
\mathsf{split}_{r,s}\ ((a,b),g) &= ((a,g|_r),(b,g|_s)) \\
\mathsf{merge}_{r,s}\ ((a,f),(b,g)) &= ((a,b),f \uplus g) \\
\mathsf{lift}_{r',r}\ (a,g) &= (a,g|_r)
\end{aligned}
\qquad
\begin{aligned}
&\text{where } f \uplus g = \\
&\quad f|_{\mathit{dom}(f) \setminus \mathit{dom}(g)} \cup g
\end{aligned}$$

The split operation splits the tuple and restricts the function (representing available implicit parameters) to the required sub-sets. This corresponds to the definition in terms of lift, which performs just the restriction. The merge

operation is more interesting. It uses $\uplus$ operation that we defined when introducing implicit parameters in Section 3.2.1. It merges the values, preferring the definitions from the right-hand side (call-site) over left-hand side (declaration-site). Thus the operation is not symmetric.

Example 12 (Indexed list)**.** *Our last example provides the semantics of data-flow computations. The flat coeffect algebra is formed by* $(\mathbb{N}, +, max, min, 0, 0, \leqslant)$*. In a non-indexed version, the semantics is provided by a non-empty list. In the indexed semantics, the index represents the length of the storing past values. The data type is then a pair of the current value, followed by* n *past values. The mappings that form the flat indexed comonad are defined as follows:*

$$C^n\alpha = \underbrace{\alpha \times \ldots \times \alpha}_{(n+1)-\text{times}}$$

$$\begin{array}{l}
\mathsf{counit}_0\langle a_0\rangle = a_0 \\[1ex]
\mathsf{cobind}_{m,n}\ f\langle a_0, \ldots a_{m+n}\rangle = \\
\quad \langle f\langle a_0, \ldots, a_m\rangle, \ldots, f\langle a_n, \ldots, a_{m+n}\rangle\rangle \\[1ex]
\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) = \\
\quad \langle (a_0, b_0), \ldots, (a_{min(m,n)}, b_{min(m,n)})\rangle \\[1ex]
\mathsf{split}_{m,n}\langle (a_0, b_0), \ldots, (a_{max(m,n)}, b_{max(m,n)})\rangle = \\
\quad (\langle a_0, \ldots, a_m\rangle, \langle b_0, \ldots, b_n\rangle) \\[1ex]
\mathsf{lift}_{n',n}\langle a_0, \ldots, a_{n'}\rangle = \qquad (\text{when } n \leqslant n') \\
\quad \langle a_0, \ldots, a_n\rangle
\end{array}$$

The reader is invited to check that the number of required past elements in each of the mappings matches the number specified by the indices. The index specifies the number of *past* elements and so the list always contains at least one value. Thus counit returns the element of a singleton list.

The $\mathsf{cobind}_{m,n}$ operation requires $m+n$ elements in order to generate n past results of the f function, which itself requires m past values. When combining two lists, $\mathsf{merge}_{m,n}$ behaves as *zip* and produces a list that has the length of the shorter argument. When splitting a list, $\mathsf{split}_{m,n}$ needs the maximum of the required lengths. Finally, the lifting operation just drops some number of elements from a list.

4.3.6 *Semantics of flat calculus*

In Section 3.2, we defined the semantics of concrete (flat) context-dependent computations including implicit parameters, liveness and data-flow. Using the *flat indexed comonad* structure, we can now define a single uniform semantics that is capable capturing of all our examples, as well as other computations that can be modelled by the structure.

CONTEXTS AND FUNCTIONS. The modelling of contexts and functions generalizes the earlier concrete examples. We use the family of mappings C^r as an (indexed) data-type that wraps the product of free variables of the context and the arguments of functions:

$$\begin{array}{rcl}
[\![x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n @ r \vdash e : \tau]\!] & : & C^r(\tau_1 \times \ldots \times \tau_n) \rightarrow \tau \\
[\![\tau_1 \xrightarrow{r} \tau_2]\!] & = & C^r\tau_1 \rightarrow \tau_2
\end{array}$$

EXPRESSIONS. The definition of the semantics is shown in Figure 23. For readability, we write the definitions in a simple programming language nota-

$$\begin{aligned}
&[\![\Gamma @ \mathsf{use} \vdash x_i : \tau_i]\!]\ \mathit{ctx} = \pi_i\ (\mathsf{counit}_{\mathsf{use}}\ \mathit{ctx}) && (\mathit{var})\\
&[\![\Gamma @ \mathsf{ign} \vdash c_i : \tau]\!]\ \mathit{ctx} = \delta\ (c_i) && (\mathit{const})\\
&[\![\Gamma @ r \vdash e : \tau]\!]\ \mathit{ctx} = && (\mathit{sub})\\
&\qquad [\![\Gamma @ r' \vdash e : \tau]\!]\ (\mathsf{lift}_{r,r'}\ \mathit{ctx}) \qquad\qquad (\text{when } r' \leqslant r) &&\\
&[\![\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2]\!]\ \mathit{ctx} = \lambda v. && (\mathit{abs})\\
&\qquad [\![\Gamma, x : \tau_1 @ r \wedge s \vdash e : \tau_2]\!]\ (\mathsf{merge}_{r,s}\ (\mathit{ctx}, v)) &&\\
&[\![\Gamma @ r \oplus (s \circledast t) \vdash e_1\ e_2 : \tau_2]\!]\ \mathit{ctx} = && (\mathit{app})\\
&\qquad \textbf{let}\ (\mathit{ctx}_1, \mathit{ctx}_2) = \mathsf{split}_{r,s \circledast t}\ (\mathsf{map}_{r \oplus (s \circledast t)}\ (\lambda x.(x,x))\ \mathit{ctx}) &&\\
&\qquad \textbf{in}\ [\![\Gamma @ r \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2]\!]\ \mathit{ctx}_1\ (\mathsf{cobind}_{s,t}\ [\![\Gamma @ s \vdash e_2 : \tau_1]\!]\ \mathit{ctx}_2) &&
\end{aligned}$$

Figure 23: Categorical semantics of the flat coeffect calculus

tion as opposed to the point-free categorical style. However, it can be equally written using just the operations of flat indexed comonad together with i^{th} projection from a tuple represented by π_i, *curry* and *uncurry*, function composition, value duplication ($\Delta : A \to A \times A$) and function pairing (given $f : A \to B$ and $g : C \to D$ then $f \times g : A \times C \to B \times D$). These operations can be provided by e. g. a Cartesian Closed Category.

The semantics of variable access and abstraction are the same as in the semantics of Uustalu and Vene [76], modulo the indexing. The semantics of variable access (*var*) uses $\mathsf{counit}_{\mathsf{use}}$ to extract product of free-variables from the context and then projection π_i to obtain the variable value. Abstraction (*abs*) takes the context *ctx* and function argument v and merges their additional contexts using $\mathsf{merge}_{r,s}$. Assuming the context Γ contains variables of types $\sigma_1, \ldots, \sigma_n$, this gives us a value $C^{r \wedge s}((\sigma_1 \times \ldots \times \sigma_n) \times \tau_1)$. Assuming that n-element tuples are associated to the left, the wrapped context is equivalent to $\sigma_1 \times \ldots \times \sigma_n \times \tau_1$, which can then be passed to the body of the function.

The semantics of application is more complex. It first duplicates the free-variable product inside the context (using map_r and duplication). Then it splits this context using $\mathsf{split}_{r,s \oplus t}$. The two contexts contain the same variables (as required by sub-expressions e_1 and e_2), but different coeffect annotations. The first context (with index r) is used to evaluate e_1, resulting in a function $C^t\tau_1 \to \tau_2$. To obtain the result, we compose this with a function created by applying $\mathsf{cobind}_{s,t}$ on the semantics of sub-expression e_2, which is of type $C^{s \circledast t}\sigma_1 \times \ldots \times \sigma_n \to C^t\tau_1$.

Finally, constants (*const*) are modelled by a global dictionary δ and sub-coeffecting is interpreted by dropping additional context from the provided context *ctx* using $\mathsf{lift}_{r,r'}$ and providing it to the semantics of the assumption.

PROPERTIES. The categorical semantics can be used to embed context-dependent computations in functional programming languages, similarly to how monads provide a way of embedding effectful computations. More importantly, it also provides validation for the design of the type system developed in Section 4.2.4. As stated in the following theorem, the annotations in the type system match those of the semantic functions.

Remark 1 (Correspondence)**.** *In all of the typing rules of the flat coeffect system, the context annotations* r *of typing judgements* $\Gamma @ r \vdash e : \tau$ *and function types* $\tau_1 \xrightarrow{r} \tau_2$ *correspond to the indices of mappings* C^r *in the corresponding semantic function defined by* $[\![\Gamma @ r \vdash e : \tau]\!]$.

Proof. By analysis of the semantic rules in Figure 23. □

Thanks to the indexing, the statement of the remark is significantly stronger than for a non-indexed system, because it provides the justification for our choice of indices in the typing rules. In particular, we can see that the annotations follow from the annotations on primitive functions that define the semantics. Also, each function defining the semantics uses a distinct operation of the coeffect algebra and so the type system is the most general possible definition (within the comonadic framework we use).

4.4 EQUATIONAL THEORY

Each of the concrete coeffect calculi discussed in this chapter has a different notion of context, much like various effectful languages have different notions of effects (such as exceptions or mutable state). However, in all of the calculi, the context has a number of common properties that are captured by the *flat coeffect algebra*. This means that there are equational properties that hold for all of the systems we consider. Further properties hold for systems where the context satisfies additional properties.

In this section, we look at such shared syntactic properties. This accompanies the previous section, which provided a *semantic* justification for the axioms of coeffect algebra with a *syntactic* justification. Operationally, this section can also be viewed as providing a pathway to an operational semantics for two of our systems (implicit parameters and liveness), which can be based on syntactic substitution. As we discuss later, the notion of context for data-flow is more complex.

4.4.1 *Syntactic properties*

Before discussing the syntactic properties of general coeffect calculus formally, it should be clarified what is meant by providing "pathway to operational semantics" in this section. We do that by contrasting syntactic properties of coeffect systems with more familiar effect systems. Assuming $e_1[x \leftarrow e_2]$ is a standard capture-avoiding syntactic substitution, the following equations define four syntactic reductions on the terms:

$$\begin{array}{rcll}
(\lambda x.e_1)\ e_2 & \longrightarrow_{\mathsf{cbn}} & e_1[x \leftarrow e_2] & (\textit{call-by-name}) \\
(\lambda x.e_1)\ v & \longrightarrow_{\mathsf{cbv}} & e_1[x \leftarrow v] & (\textit{call-by-value}) \\
(\lambda x.e_1)\ e_2 & \longrightarrow_{\mathsf{seq}} & \textbf{glet}\ x = e_2\ \textbf{in}\ e_1 & (\textit{internalized sequencing}) \\
e & \longrightarrow_{\eta} & \lambda x.e\ x & (\eta\textit{-expansion})
\end{array}$$

The rules capture syntactic reductions that can be performed in a general calculus, without any knowledge of the specific notion of context. The **glet** notation models explicit sequencing of context-dependent computations an is inspired by Filinski [21]. In the rest of the section, we briefly outline the interpretation of the four rules and then we focus on call-by-value (Section 4.4.2) and call-by-name (Section 4.4.3) in more details.

The focus of this work is on the general coeffect system and so we do not discuss the operational semantics of the specific notions of context. However, some work in that area has been done by Brunel et al. [12].

CALL-BY-NAME. In call-by-name, the argument is syntactically substituted for all occurrences of a variable. It can be used as the basis for operational semantics of purely functional languages. However, using the rule in effectful languages breaks the *type preservation* property. For example, consider a language with effect system where functions are annotated with sets of effects such as $\{\mathsf{write}\}$. A function $\lambda x.y$ is a effect-free:

$$y\!:\!\tau_1 \vdash \lambda x.y : \tau_1 \xrightarrow{\emptyset} \tau_2 \;\&\; \emptyset$$

Substituting an expression e with effects $\{\mathsf{write}\}$ for y changes the type of the function by adding latent effects (without changing the immediate effects):

$$\vdash \lambda x.e : \tau_1 \xrightarrow{\{\mathsf{write}\}} \tau_2 \;\&\; \emptyset$$

Similarly to effect systems, substituting a context-dependent computation e for a variable y can add latent coeffects to the function type. However, this is not the case for *all* flat coeffect calculi. For example, call-by-name reduction preserves types and coeffects for the implicit parameters system. This makes the model suitable for languages such as Haskell.

CALL-BY-VALUE. The call-by-value evaluation strategy is often used by effectful languages. Here, an argument is first reduced to a *value* before performing the substitution. In effectful languages, value is defined syntactically. For example, in the *Effect* language [86], values are identifiers x or functions $(\lambda x.e)$.

The notion of *value* in coeffect systems differs from the usual syntactic understanding. A function $(\lambda x.e)$ does not delay all context requirements of the body e and may have immediate context requirements. Thus we say that e is a value if it is a value in the usual sense *and* has not immediate context requirements. We define this formally in Section 4.4.2.

The call-by-value evaluation strategy holds for a wide range of flat coeffect calculi, including all our three examples. However, it is rather weak – in order to use it, the concrete semantics needs to provide a way for reducing context-dependent term $\Gamma @ r \vdash e : \tau$ to a term $\Gamma @ \mathsf{use} e' : \tau$ with no context requirements.

INTERNALIZED SEQUENCING. The (*internalized sequencing*) rule captures an operational semantics where the language provides a construct representing *sequential composition* and expressions can be reduced to a normal form, consisting of a sequenced context-dependent operations. We choose to write the sequencing operation as **glet** to emphasize that this is a separate primitive and not an ordinary syntactic **let**. The normal form looks as follows:

$$\textbf{glet}\ x_1 = e_1\ \textbf{in}\ \ldots\ \textbf{glet}\ x_n = e_n\ \textbf{in}\ e$$

Here, the expressions $e_1, \ldots, e_n, e$ do not contain further nested **glet** constructs. This evaluation strategy provides context-dependent counterpart to operational view of monads developed by Filinsky [21]. We discuss how expressions reduce to the normal form in Section 4.4.4

The (*internalized sequencing*) rule is useful when defining a concrete semantics for a language that provides constructs for explicitly providing the context. For example, consider a language with implicit parameters where a parameter is defined by e_1 **with** $?p = e_2$. Semantics for such language would provide a reduction rule:

$$(\textbf{glet}\ x_1 = ?p\ \textbf{in}\ e)\ \textbf{with}\ ?p = e_1 \quad \leadsto \quad \textbf{let}\ x_1 = e_1\ \textbf{in}\ e$$

Here, the **glet** construct provides a way of sequentialising the context-requirements so that they can be discharged by matching constructs providing the required contexts. As discussed tfearlier, we focus on the general case and so we discuss when a flat coeffect calculus supports this form of evaluation (rather than looking at semantics for concrete systems).

LOCAL SOUNDNESS AND COMPLETENESS. Two desirable properties of calculi, coined by Pfenning and Davies [58], are *local soundness* and *local completeness*. They guarantee that the rules which introduce a function arrow (lambda abstraction) and eliminate it (application) are not too strong and sufficiently strong.

The local soundness property is witnessed by (call-by-name) β-reduction, which we discussed already. The local completeness is witnessed by the η-expansion rule. We discuss the flat coeffect algebra conditions under which the reduction holds in Section 4.4.3.

4.4.2 *Call-by-value evaluation*

As discussed in the previous section, call-by-value reduction can be used for most flat coeffect calculi, but it provides a very weak general model i. e. the hard work of reducing context-dependent term to a *value* has to be provided for each system. Syntactic category for values is defined as:

$$\begin{array}{lll} v \in \mathit{SynVal} & v & ::= \quad x \mid c \mid (\lambda x.e) \\ n \in \mathit{NonVal} & n & ::= \quad e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\ e \in \mathit{Expr} & e & ::= \quad v \mid n \end{array}$$

The category *SynVal* captures syntactic values, but a context-dependency-free value in coeffect calculus cannot be defined purely syntactically.

Definition 8. *An expression* e *is a* value, *written as* $val(e)$ *if it is a syntactic value, i. e.* $e \in \mathit{SynVal}$ *and it has no context-dependencies, i. e.* $\Gamma @ \mathsf{use} \vdash e : \tau$.

The call-by-value substitution substitutes a value, with context requirements use, for a variable, whose access is also annotated with use. Thus, it does not affect the type and context-requirements of the term:

Lemma 1 (Call-by-value substitution)**.** *In a flat coeffect calculus with a coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$*, given a value* $\Gamma @ \mathsf{use} \vdash v : \sigma$ *and an expression* $\Gamma, x : \sigma @ r \vdash e : \tau$*, then substituting* v *for* x *does not change the type and context requirements:* $\Gamma @ r \vdash e[x \leftarrow v] : \tau$.

Proof. By induction over the type derivation, using the fact that x and v are annotated with use and that Γ is treated as a set in the flat calculus. □

The substitution lemma 1 holds for all flat coeffect systems. However, proving that call-by-value reduction preserves typing requires an additional constraint on the flat coeffect algebra, which relates the $\wedge$ and $\oplus$ operations:

$$r \wedge t \ \leqslant\ r \oplus t \qquad \text{(approximation)}$$

Intuitively, this specifies that the $\wedge$ operation (splitting of context requirements) under-approximates the actual context capabilities while the $\oplus$ operation (combining of context requirements) over-approximates the actual context requirements.

The property holds for all three systems we consider – for implicit parameters, this is an equality; for liveness and data-flow (which both use a lattice), the greatest lower bound is smaller than the least upper bound.

Assuming $\longrightarrow_{\mathsf{cbv}}$ is call-by-value reduction that reduces the term $(\lambda x.e)\, v$ to a term exv, the type preservation theorem is stated as follows:

Theorem 1 (Call-by-value reduction). *In a flat coeffect system with the (*approximation*) property, if* $\Gamma @ r \vdash e : \tau$ *and* $e \longrightarrow_{\mathsf{cbv}} e'$ *then* $\Gamma @ r \vdash e' : \tau$*.*

Proof. Consider the typing derivation for the term $(\lambda x.e)\, v$ before reduction:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \tau_1 @ r \wedge t \vdash e : \tau_2}{\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{t} \tau_2} \qquad \Gamma @ \mathsf{use} \vdash v : \tau_1}{\Gamma @ r \oplus (\mathsf{use} \circledast t) \vdash (\lambda x.e)\, v : \tau_2}}{\Gamma @ r \oplus t \vdash (\lambda x.e)\, v : \tau_2}$$

The last step simplifies the coeffect annotation using the fact that use is a unit of $\circledast$. From Lemma 1 $e[x \leftarrow v]$ has the same coeffect annotation as e. As $r \wedge t \leqslant r \oplus t$, we can apply sub-coeffecting:

$$(\mathit{sub})\ \dfrac{\Gamma @ r \wedge t \vdash e[x \leftarrow v] : \tau_2}{\Gamma @ r \oplus t \vdash e[x \leftarrow v] : \tau_2}$$

Thus, the reduction preserves type and coeffect annotation (although this may not be the *only* typing of the original term). □

4.4.3 *Call-by-name evaluation*

In the call-by-name reduction of $(\lambda x.e_1)\ e_2$, the expression e_2 is substituted for all occurrences of the variable v in an expression e_1. As discussed in Section 4.4.1, the call-by-name strategy does not *in general* preserve the type of a program, but it does preserve the typing in some interesting cases. The typing is preserved for different reasons in two of our systems, so we briefly review the concrete examples.

DATA-FLOW. The type preservation property does not hold for data-flow. This case is similar to the example shown earlier with effectful computations. As a minimal example, consider the substitution of **prev** z for a variable y in a function $\lambda x.y$:

$$y{:}\tau_1, z{:}\tau_1 @ 0 \vdash \lambda x.y : \tau_1 \xrightarrow{0} \tau_2 \qquad \text{(before)}$$
$$z{:}\tau_1 @ 1 \vdash \lambda x.\mathbf{prev}\ z : \tau_1 \xrightarrow{1} \tau_2 \qquad \text{(after)}$$

After the substitution, the coeffect of the body is 1. The rule for lambda abstraction requires that $1 = \mathit{min}(r, s)$ and so the least solution is to set both r, s to 1. The substitution this affects the coeffects attached both to the function type and the overall context.

Semantically, the coeffect over-approximates the actual requirements – the code does not access previous value of the argument x. This cannot be caputred by a flat coeffect system, but can be captured using the structural system discussed in Chapter 5.

IMPLICIT PARAMETERS. In data-flow, there is no typing for the resulting expression that preserves the type of the function. However, this is not the case for all systems. Consider substituting an implicit parameter access $?p$ for a variable y:

$$\begin{aligned} y{:}\tau_1 \,@\, \emptyset &\vdash \lambda x.y : \tau_1 \xrightarrow{\emptyset} \tau_2 && \text{(before)} \\ \emptyset \,@\, \{?p\} &\vdash \lambda x.?p : \tau_1 \xrightarrow{\emptyset} \tau_2 && \text{(after)} \end{aligned}$$

The above shows one possible typing of the body – one that does not change the coeffects of the function type and attaches all additional coeffects (implicit parameters) to the context. In case of implicit parameters (and, more generally, systems with set-like annotations) this is always possible.

LIVENESS. In liveness, the type preservation also holds, but for a different reason. Consider substituting any expression e of type τ_1 with coeffects r for a variable y:

$$\begin{aligned} y{:}\tau_1 \,@\, \mathsf{L} &\vdash \lambda x.y : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 && \text{(before)} \\ \emptyset \,@\, \mathsf{L} &\vdash \lambda x.e : \tau_1 \xrightarrow{\mathsf{L}} \tau_2 && \text{(after)} \end{aligned}$$

In the original expression, both the overall context and the function type are annotated with L, because the body contains a variable access. An expression e can always be treated as being annotated with L (because L is the top element of the lattice) and so substitution does not change any coeffects.

REDUCTION THEOREM. The above examples (implicit parameters and liveness) demonstrate two particular kinds of coeffect algebra for which typing preservation holds. Proving the type preservation separately provides more insight into how the systems work and so we choose to consider separately.

Definition 9. *We call a flat coeffect algebra* top-pointed *if* use *is the greatest (top) coeffect scalar* ($\forall r \in \mathcal{C}\ .\ r \leqslant \mathsf{use}$) *and* bottom-pointed *if it is the smallest (bottom) element* ($\forall r \in \mathcal{C}\ .\ r \geqslant \mathsf{use}$)*.*

Liveness is an example of top-pointed coeffects as variables are annotated with L and $\mathsf{D} \leqslant \mathsf{L}$, while implicit parameters and data-flow are examples of bottom-pointed coeffects. For top-pointed flat coeffects, the substitution lemma holds without additional requirements:

Lemma 2 (Top-pointed substitution)**.** *In a top-pointed flat coeffect calculus with an aglebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$*, substituting an expression* e_s *with arbitrary coeffects* s *for a variable* x *in* e_r *does not change the coeffects of* e_r*:*

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

Proof. Using sub-coeffecting ($s \leqslant \mathsf{use}$) and a variation of Lemma 1. □

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

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

Lemma 3 (Bottom-pointed substitution). *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \circledast = \oplus$ *and the operation is also idempotent and commutative:*

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

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

The flat system discussed here is *flexible enough* to let us always re-associate new context requirements (arising from the substitution) with the free-variable context. In contrast, the structural system discussed in Chapter 5 is *precise enough* to keep the coeffects associated with individual variables – thus preserving typing in a complementary way.

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

Theorem 2 (Call-by-name reduction). *In a coeffect system that satisfies the conditions for Lemma 2 or Lemma 3, if* $\Gamma @ r \vdash e : \tau$ *and* $e \twoheadrightarrow_{\mathsf{cbn}} e'$ *then* $\Gamma @ r \vdash e' : \tau$.

Proof. For top-pointed coeffect algebra (using Lemma 2), the proof is similar to the one in Theorem 1, using the facts that $s \leqslant \mathsf{use}$ and $r \wedge t = r \oplus t$. For bottom-pointed coeffect algebra, consider the typing derivation for the term $(\lambda x.e)\, v$ before reduction:

$$\dfrac{\dfrac{\Gamma, x : \tau_s @ r \vdash e_r : \tau_r}{\Gamma @ r \vdash \lambda x.e_r : \tau_s \xrightarrow{r} \tau_r} \qquad \Gamma @ s \vdash e_s : \tau_s}{\Gamma @ r \oplus (s \circledast r) \vdash (\lambda x.e_r)\, e_s : \tau_r}$$

The derivation uses the idempotence of $\wedge$ in the first step, followed by the (*app*) rule. The type of the term after substitution, using Lemma 3 is:

$$\dfrac{\Gamma, x : \tau_s @ r \vdash e_r : \tau_r \qquad \Gamma @ s \vdash e_s : \tau_s}{\Gamma, x : \tau_r @ r \circledast s \vdash e_r[x \leftarrow e_s] : \tau_s}$$

From the assumptions of Lemma 3, we know that $\circledast = \oplus$ and the operation is idempotent, so trivially: $r \circledast s = r \oplus (s \circledast r)$ □

EXPANSION THEOREM. The η-expansion (local completeness) is similar to β-reduction (local soundness) in that it holds for some flat coeffect systems, but not for all. Out of the examples we discuss, it holds for implicit parameters, but does not hold for liveness and data-flow.

Recall that η-expansion turns e into $\lambda x.e\ x$. In the case of liveness, the expression e may require no variables (both immediate and latent coeffects are marked as D). However, the resulting expression $\lambda x.e\ x$ accesses a variable, marking the context and function argument as live. In case of data-flow, the coeffects are made larger by the lambda abstraction. We remedy this limitation in the next chapter.

However, the η-expansion preserves the type for implicit parameters and, more generally, for any flat coeffect algebra where $\oplus = \wedge$. Assuming $\twoheadrightarrow_{\eta}$ is the standard η-reduction:

Theorem 3 (η-expansion)**.** *In a bottom-pointed flat coeffect calculus with an algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *where* $\wedge = \oplus$, *if* $\Gamma @ r \vdash e : \tau_1 \xrightarrow{s} \tau_2$ *and* $e \rightarrow_\eta e'$ *then* $\Gamma @ r \vdash e' : \tau_1 \xrightarrow{s} \tau_2$.

Proof. The following derivation shows that $\lambda x.e\ x$ has the same type as e:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma @ r \vdash e : \tau_1 \xrightarrow{s} \tau_2 \qquad x : \tau_1 @ \mathsf{use} \vdash x : \tau_1}{\Gamma, x : \tau_1 @ r \oplus (\mathsf{use} \circledast s) \vdash e\ x : \tau_2}}{\Gamma, x : \tau_1 @ r \oplus s \vdash e\ x : \tau_2}}{\Gamma, x : \tau_1 @ r \wedge s \vdash e\ x : \tau_2}}{\Gamma @ r \vdash \lambda x.e\ x : \tau_1 \xrightarrow{s} \tau_2}$$

□

The derivation starts with the expression e and derives the type for $\lambda x.e\ x$. The application yields context requirements $r \oplus s$. In order to recover the original typing, this must be equal to $r \wedge s$. Note that the derivation is showing just one possible typing – the expression $\lambda x.e\ x$ has other types – but this is sufficient for showing type preservation.

In summary, flat coeffect calculi do not *in general* permit call-by-name evaluation, but there are several cases where call-by-name evaluation can be used. Among the examples we discuss, these include liveness and implicit parameters. Moreover, for implicit parameters (and more generally, any set-like flat coeffect algebra), the η-expansion holds as well, giving us both local soundness and local completeness as coined by Pfenning and Davies [58].

4.4.4 *Internalized sequencing*

The call-by-value and call-by-name evaluation strategies discussed in the previous section are the key techniques for defining equational theory of flat coeffects. In this section, we briefly discuss another approach that follows the style of generic operational semantics designed by Filinski [21] for effectful computations.

The idea is to embed sequencing of context-dependent computations as an explicit construct into the language and define a *normal form* that consists of sequenced expressions. The reduction to the normal form is generic for all coeffect systems (satisfying certain conditions), while the reduction of the normal form is provided by each concrete coeffect system. The extended language with the **glet** construct and its typing is defined as follows:

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

$$(\textit{glet})\ \dfrac{\Gamma @ s \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 @ r \vdash e_2 : \tau_2}{\Gamma @ r \oplus (s \circledast r) \vdash \textbf{glet}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

The newly introduced syntactic form n represents a normal form. The construct **glet** $x = e$ **in** n models an explicit sequencing of an expression e, followed by an expression n. Note that **glet** can only contain further **glet** constructs in the body, but not in the argument. The typing of the **glet** is the same as the typing of ordinary **let** construct – as discussed in Section 4.2.4, for the examples we consider, this gives a *more precise* typing than the typing of the term $(\lambda x.n)\ e$.

The reduction rules that produce a normal form are shown in Figure 24. The (*eval*) rule introduces **glet** by reducing a redex $(\lambda x.e_2)\ e_1$ to an expres-

$$(\lambda x.e_2)\ e_1 \quad \rightsquigarrow \quad \textbf{glet}\ x = e_1\ \textbf{in}\ e_2 \qquad\qquad (\textit{eval})$$

$$\begin{array}{l} (\textbf{glet}\ x = e_1\ \textbf{in}\ e_2)\ e_3 \quad \rightsquigarrow \\ \qquad \textbf{glet}\ x = e_1\ \textbf{in}\ (e_2\ e_3) \end{array} \qquad x \notin \mathit{fv}(e_3) \qquad\qquad (\textit{glet-app})$$

$$\begin{array}{l} \textbf{glet}\ x_2 = (\textbf{glet}\ x_1 = e_1\ \textbf{in}\ e_2)\ \textbf{in}\ e_3 \quad \rightsquigarrow \\ \qquad \textbf{glet}\ x_1 = e_1\ \textbf{in}\ (\textbf{glet}\ x_2 = e_2\ \textbf{in}\ e_3) \end{array} \qquad x_1 \notin \mathit{fv}(e_3) \qquad (\textit{glet-glet})$$

Figure 24: Reduction to normal form.

sion representing explicit sequencing of e_1 and e_2. The remaining two rules specify how **glet** distributes with other constructs (**glet** and application). In (*glet-app*), the **glet** construct appearing inside an application is lifted to the top-level; similarly (*glet-glet*) lifts a **glet** construct nested in the argument of another **glet**.

As with *call-by-value* and *call-by-name* strategies, the *internalized sequencing* strategy can only be used with coeffect systems satisfying certain conditions. The general form of the conditions is summarized in Appendix ?. Here, we briefly consider our three examples.

conditions. The (*eval*) reduction can be safely applied for any flat coeffect calculus which satisfies the condition that the typing of the **glet** expression (shown above) is equivalent or more precise than typing of the expression $(\lambda x.e_2)\ e_1$. More formally:

Definition 10. *A flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is* oriented *if for all* $s, s_1, s_2, r \in \mathcal{C}$ *such that* $s = s_1 \wedge s_2$ *it is the case that* $s \circledast (s \oplus r) \leqslant s_1 \circledast (s_2 \oplus r)$.

As discussed in Section 4.2.6, this condition is satisfied for all our three examples (strictly for liveness and data-flow; and by equality for implicit parameters). For the (*glet-app*) and (*glet-glet*) rules, additional conditions arise from the typing of the original and reduced expression (showed in Appendix ?). The results are summarized in the following two tables.

Assuming $\Gamma @ r_1 \vdash e_1 : \tau_1$ and $\Gamma @ r_2 \vdash e_2 : \tau_3 \xrightarrow{s} \tau_2$ and $\Gamma @ r_3 \vdash e_3 : \tau_3$, the typings of the original and reduced expressions in (*glet-app*) rule are:

	Before	After	Satisfied
Parameters	$\mathit{max}(r_1 + r_2, r_3 + s)$	$r_1 + \mathit{max}(r_2, r_3 + s)$	×
Liveness	$r_2 \sqcap (r_3 \sqcup s)$	$r_2 \sqcap (r_3 \sqcup s)$	✓
Data-flow	$(r_2 \cup r_1) \cup (r_3 \cup s)$	$(r_2 \cup (r_3 \cup s)) \cup r_1$	✓

Assuming $\Gamma @ r_1 \vdash e_1 : \tau_1$ and $\Gamma @ r_2 \vdash e_2 : \tau_2$ and $\Gamma @ r_3 \vdash e_3 : \tau_3$, the typings of the original and reduced expressions in (*glet-glet*) rule are:

	Before	After	Satisfied
Parameters	$r_3 + (r_2 + r_1)$	$(r_3 + r_2) + r_1$	✓
Liveness	r_3	r_3	✓
Data-flow	$r_3 \cup (r_2 \cup r_1)$	$(r_3 \cup r_2) \cup r_1$	✓

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

$$(\textit{typ})\ \dfrac{\Gamma @ r \vdash e : \tau \qquad \tau <: \tau'}{\Gamma @ r \vdash e : \tau'}$$

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

$$\boxed{\tau <: \tau'}$$

$$(\textit{sub-trans})\ \dfrac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$(\textit{sub-fun})\ \dfrac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2' \qquad r' \geqslant r}{\tau_1 \xrightarrow{r} \tau_2 <: \tau_1' \xrightarrow{r'} \tau_2'}$$

$$(\textit{sub-refl})\ \dfrac{}{\tau <: \tau}$$

Figure 25: Subtyping rules for flat coeffect calculus

This means that *internalized sequencing* provides a basis for operational semantics of liveness and implicit parameters, but not for data-flow languages. For other coeffect systems, the conditions in Appendix ? have to be re-examined.

4.5 SYNTACTIC PROPERTIES AND EXTENSIONS

The flat coeffect algebra introduced in Section 4.2 requires a number of laws. The laws are required for three distinct reasons – to be able to define the categorical structure in Section 4.3, to prove equational properties in Section 4.4 and finally, to guarantee intuitive syntactic properties for constructs such as λ-abstraction and pairs in context-aware calculi.

In this section, we look at the last point. We discuss what syntactic equivalences are permitted by the properties of $\wedge$ and we extend the calculus with pairs and units and discuss their syntactic properties. In the following section, we further develop subtyping relation for the calculus.

4.5.1 *Subtyping for coeffects*

The typing rules discussed in Section 4.2.4 include sub-coeffecting rule which makes it possible to treat an expression with smaller context requirements as an expression with greater context requirements. In the corresponding categorical semantics, this means that we can *drop* some of the provided context.

Figure 25 adds sub-typing on function types, making it possible to treat a function with smaller context requirements as a function with greater context requirements. The definition uses the standard reflexive and transitive $<:$ operator. As the (*sub-fun*) shows, the function type is contra-variant in the input and co-variant in the output. The (*typ*) rule allows using sub-typing on an expression type in the coeffect calculus.

$$[\![\Gamma @ r \vdash e : \tau']\!] = [\![\tau <: \tau']\!] \circ [\![\Gamma @ r \vdash e : \tau]\!] \qquad (\textit{typ})$$

$$[\![\tau <: \tau]\!] = \mathsf{id} \qquad (\textit{sub-refl})$$

$$[\![\tau_1 <: \tau_3]\!] = [\![\tau_2 <: \tau_3]\!] \circ [\![\tau_1 <: \tau_2]\!] \qquad (\textit{sub-trans})$$

$$[\![\tau_1 \xrightarrow{r} \tau_2 <: \tau_1' \xrightarrow{r'} \tau_2']\!] = \lambda f. \qquad (\textit{sub-fun})$$
$$[\![\tau_2 <: \tau_2']\!] \circ f \circ \mathsf{map}_r\ [\![\tau_1' <: \tau_1]\!] \circ \mathsf{lift}_{r',r}$$

Figure 26: Semantics of subtyping for flat coeffects

SEMANTICS. We follow the same approach as with the rest of the calculus and use a categorical semantics to explain (and confirm) the design of the sub-typing rules. The semantics of a judgement $[\![\tau <: \tau']\!]$ is a function $[\![\tau]\!] \to [\![\tau']\!]$. As shown in Figure 26, the seamntics of the sub-typing rule (*typ*) then just composes the semantics of the original expression with the conversion produced by the semantics of the sub-typing judgement.

The rest of the Figure 26 shows the rules that define the semantics of <:. The reflexivity and transitivity are just the identity function and function composition, respectively. The (*sub-fun*) case is interesting – recall that the semantics of a function $\tau_1' \xrightarrow{r'} \tau_2'$ is $C^{r'}\tau_1' \to \tau_2'$. To build the required function, we first drop unnecessary context using $\mathsf{lift}_{r',r} : C^{r'}\tau_1' \to C^{r}\tau_1'$ and use the map_r function to transform the nested τ_1' to τ_1. Then we evaluate the original function f and turn the resulting τ_2 into the required result of type τ_2'.

4.5.2 *Alternative lambda abstraction*

In Section 4.2.1, we discussed how to reconcile two typings for lambda abstraction (for implicit parameters, the lambda function splits context requirements using $r \cup s$; for data-flow it suffices to duplicate the requirement r). We introduced the $\wedge$ operation as a way of providing the additional abstraction. Here, we identify coeffect calculi for which the simpler (duplicating) rule is sufficient.

IDEMPOTENCE. Recall that $(\mathcal{C}, \wedge)$ is a band, meaning that $\wedge$ is idempotent and associative. The idempotence means that the context requirements of the body can be required from both the declaration site and the call site. Thus, the following (*idabs*) typing is valid (for reference, it is shown side-by-side with the ordinary lambda abstraction rule):

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

To derive (*idabs*), we use idempotence on the body annotation $r = r \wedge r$ and then use the standard (*abs*) rule. So, (*idabs*) follows from (*abs*), but the other direction is not necessarily the case. The condition below identifies coeffect calculi where (*abs*) follows from (*idabs*).

Definition 11. *A flat coeffect algebra* $(\mathcal{C}, \circledast, \oplus, \wedge, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is* strictly oriented *if for all* $s, r \in \mathcal{C}$ *it is the case that* $r \wedge s \leqslant r$.

$$(\textit{pair})\ \frac{\Gamma @ r \vdash e_1 : \tau_1 \qquad \Gamma @ s \vdash e_2 : \tau_2}{r \oplus s @ \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$(\textit{proj})\ \frac{\Gamma @ r \vdash e : \tau_1 \times \tau_2}{\Gamma @ r \vdash \pi_i\ e : \tau_i}$$

$$(\textit{unit})\ \frac{}{\Gamma @ \mathsf{ign} \vdash () : \mathsf{unit}}$$

Figure 27: Typing rules for pairs and units

Remark 2. *For a flat coeffect calculus with a strictly oriented algebra, the standard (abs) rule can be derived from the (idfun) rule.*

Proof. The following derives the conclusion of (*abs*) using (*abs*), sub-coeffecting, sub-typing and the fact that the algebra is *stricly oriented*:

$$(\textit{typ})\ \dfrac{(\textit{sub})\ \dfrac{(\textit{idabs})\ \dfrac{\Gamma, x : \tau_1 @ r \wedge s \vdash e : \tau_2}{\Gamma @ r \wedge s \vdash \lambda x.e : \tau_1 \xrightarrow{r \wedge s} \tau_2}}{\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{r \wedge s} \tau_2}\ (r \leqslant r \wedge s)}{\Gamma @ r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2}\ (r \leqslant r \wedge s)$$

□

The practical consequence of the Remark 2 is that, for strictly oriented coeffect calculi (such as our liveness and data-flow computations), we can use the (*idabs*) rule and get an equivalent type system. This alternative formulation removes the non-determinism that arises from the splitting of context requirements in the original (*abs*) rule.

SYMMETRY. The $\wedge$ operation is idempotent and associative. In all of the three examples considered in this chapter, the operation is also *symmetric*. To make our definitions more general, we do not require this to be the case for *all* flat coeffect systems. However, systems with symmetric $\wedge$ have the following property.

Remark 3. *For a flat coeffect calculus such that* $r \wedge s = s \wedge r$, *assuming that* r', s', t' *is a permutation of* r, s, t*:*

$$\frac{\Gamma, x : \tau_1, y : \tau_2 @ r \wedge s \wedge t \vdash e : \tau_3}{\Gamma @ r' \vdash \lambda x.\lambda y.e : \tau_1 \xrightarrow{s'} (\tau_2 \xrightarrow{t'} \tau_3)}$$

Intuitively, this means that the context requirements of a function with multiple arguments can be split arbitrarily between the declaration site and (multiple) call sites. In other words, it does not matter how the context requirements are satisfied.

4.5.3 *Language with pairs and unit*

The calculus introduced in Section 4.2 consisted only of variables, abstraction, application and let binding to show the key aspects of flat coeffect systems. Here, we extend it with pairs and the unit value to sketch how it can be turned into a full programming language. The syntax of the language is extended as follows:

$$\begin{aligned} e &::= \dots \mid () \mid e_1, e_2 \\ \tau &::= \dots \mid \mathsf{unit} \mid \tau \times \tau \end{aligned}$$

The typing rules for pairs and the unit value are shown in Figure 27. The unit value (*unit*) is annotated with the ign coeffect (the same as other constants). Pairs, created using the (e_1, e_2) expression, are annotated with a coeffect that combines the coeffects of the two sub-expressions using the *point-wise* operator $\oplus$. The operator models the case when the (same) available context is split and passed to two independent sub-expressions. This matches the semantics of pairs discussed shortly. Finally, the (*proj*) rule is uninteresting, because π_i can be viewed as a pure function.

PROPERTIES. Pairs and the unit value in a lambda calculus should form a monoid – associativity means that the expression $(e_1, (e_2, e_3))$ should be isomorphic to $((e_1, e_2), e_3)$ and that $((), e) \simeq e \simeq (e, ())$, where the isomorphism appropriately transforms the values, without affecting other properties (here coeffects) of the expression.

In the following, we assume that assoc is a pure function transforming a pair $(x_1, (x_2, x_3))$ to a pair $((x_1, x_2), x_3)$. We write $e \equiv e'$ when for all Γ, τ and r, it is the case that $\Gamma @ r \vdash e : \tau$ if and only if $\Gamma @ r \vdash e' : \tau$.

Theorem 4. *For a flat coeffect calculus with pairs and units, the following holds:*

$$\begin{aligned} \mathsf{assoc}\ (e_1, (e_2, e_3)) &\equiv ((e_1, e_2), e_3) && \text{(associativity)} \\ \pi_1\ (e, ()) &\equiv e && \text{(right unit)} \\ \pi_2\ ((), e) &\equiv e && \text{(left unit)} \end{aligned}$$

Proof. Follows from the fact that $(\mathcal{C}, \oplus, \mathsf{ign})$ is a monoid and assoc, π_1 and π_2 are pure functions (treated as contstants in the langauge). □

The above properties follow from the laws of the flat coeffect algebra. In addition, if the $\oplus$ operation is symmetric (which is the case for all our examples in this chapter), it also holds that swap $(e_1, e_2) \equiv (e_2, e_1)$.

4.6 RELATED WORK

Most of the related work leading to coeffects has already been discussed in Chapter 2 and we covered work related to individual concepts throughout the chapter. In this section, we do not repeat the discussion present elsewhere – we discuss one specific question that often arises when discussing coeffects and that is *when is coeffect (not) an effect?*

We start with a quick overview of the ways in which effects and coeffects differ and then we briefly look at one (but illustrative) example where the two concepts overlap. We focus mainly on the equivalence between the *categorical semantics*, which reveals the nature of the computations – rather than considering just the syntactic aspects of the type system.

4.6.1 *When is coeffect not a monad*

Coeffect systems differ from effect systems in three important ways:

- Semantically, coeffects capture very different notions of computation. As demonstrated in Chapter 2, coeffects track additional contextual properties required by a computation, many of which cannot be captured by a monad (e. g. liveness or data-flow).

- Syntactically, coeffect calculi use a richer algebraic structure with pointwise composition, sequential composition and context merging $(\oplus, \circledast, \wedge)$ while most effect systems only use a single operation for sequential composition (monadic bind).
- Syntactically, the second difference is in the lambda abstraction (*abs*). In coeffect systems, the context requirements of the body can be split between declaration-site and call-site, while monadic effect systems delay all effects.

Despite the differences, our implicit parameters example can be also represented by a monad. Semantically, the *reader* monad is equivalent to the *product* comonad. Syntactically, we use the $\cup$ operation for all three operations of the coeffect algebra. However, the last point requires us to extend monadic lambda abstraction.

4.6.2 *When is coeffect a monad*

As discussed in Section 3.2.1, one of our examples, implicit parameters, can be also captured by a monad. However, *just* a monad is not enough because lambda abstraction in effect systems does not provide a way of splitting the context requirements between declaration-site and call-site (or combining the implicit parameters available in the scope where the function is deined and those specified by the caller).

CATEGORICAL RELATIONSHIP. Before looking at the necessary extensions, consider the two ways of modelling implicit parameters. We assume that the function $r \rightarrow \sigma$ is a lookup function for reading implicit parameter values that is defined on a set r. The two definitions are:

$$C^r\tau = \tau \times (r \rightarrow \sigma) \qquad (\textit{product comonad})$$
$$M^r\tau = (r \rightarrow \sigma) \rightarrow \tau \qquad (\textit{reader monad})$$

The *product comonad* simply pairs the value τ with the lookup function, while the *reader monad* is a function that, given a lookup function, produces a τ value. As noted by Orchard [52], when used to model computation semantics, the two representations are equivalent:

Remark 4. *Computations modelled as $C^r\tau_1 \rightarrow \tau_2$ using the product comonad are isomorphic to computations modelled as $\tau_1 \rightarrow M^r\tau_2$ using the currying/uncurrying isomoprhism.*

Proof. The isomorphism is demonstrated by the following equation:

$$\begin{aligned}
&C^r\tau_1 \rightarrow \tau_2 = \\
&(\tau_1 \times (r \rightarrow \sigma)) \rightarrow \tau_2 \\
&\tau_1 \rightarrow ((r \rightarrow \sigma) \rightarrow \tau_2) \\
&\tau_1 \rightarrow M^r\tau_2
\end{aligned}$$ □

The equivalence holds for monads and comonads (as well as *indexed* monads and comonads), but it does not extend to *flat* indexed comonads which also provide the $\mathtt{merge}_{r,s}$ operation to model context merging.

DELAYING EFFECTS IN MONADS. In the syntax of the language, the above difference is manifested in the (*abs*) rules for monadic effect systems and comonadic coeffect systems. The following listing shows the two rules side-by-side, using the coeffect notation for both of them:

$$(\mathit{cabs})\ \frac{\Gamma, x{:}\tau_1 \,@\, r \cup s \vdash e : \tau_2}{\Gamma \,@\, r \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2} \qquad (\mathit{mabs})\ \frac{\Gamma, x{:}\tau_1 \,@\, r \cup s \vdash e : \tau_2}{\Gamma \,@\, \emptyset \vdash \lambda x.e : \tau_1 \xrightarrow{r \cup s} \tau_2}$$

In the comonadic (*cabs*) rule, the implicit parameters of the body are split. However, the monadic rule (*mabs*) places all requirements on the call-site. This follows from the fact that monadic semantics uses the unit operation in the interpretation of lambda abstraction:

$$[\![\lambda x.e]\!] \;=\; \mathsf{unit}\ (\lambda x.[\![e]\!])$$

The type of unit is $\alpha \to M^{\alpha}\emptyset$, but in this specific case, the α is instantiated to be $\tau_1 \to M^{r \cup s}\tau_2$ and so this use of unit has a type:

$$\mathsf{unit}\ :\ (\tau_1 \to M^{r \cup s}\tau_2) \to M^{\emptyset}(\tau_1 \to M^{r \cup s}\tau_2)$$

In order to split the implicit parameters of the body ($r \cup s$ on the left-hand side) between the declaration-site ($\emptyset$ on the outer M on the right-hand side) and the call-site ($r \cup s$ on the inner M on the right-hand side), we need an operation (which we call delay) with the following signature:

$$\mathsf{delay}_{r,s}\ :\ (\tau_1 \to M^{r \cup s}\tau_2) \to M^{r}(\tau_1 \to M^{s}\tau_2)$$

The operation reveals the difference between effects and coeffects – intuitively, given a function with effects $r \cup s$, it should execute the effects r when wrapping the function, *before* the function actually perforsm the effectful operation with the effects. The remaining effects s are delayed as usual, while effects r are removed from the effect annotation of the body.

Another important aspect of the signature is that the function needs to be indexed by the coeffect annotations r, s. The indices determine how the input context requirements $r \cup s$ are split – and thus guarantee determinism of the function.

The operation cannot be implemented in a useful way for most standard monads, but the reader monad is, indeed, an exception. It is not difficult to see how it can be implemented when we expand the definitions of $M^{r}\tau$:

$$\mathsf{delay}_{r,s}\ :\ (\tau_1 \to (r \cup s \to \sigma) \to \tau_2) \to ((r \to \sigma) \to \tau_1 \to (s \to \sigma) \to \tau_2)$$

RESTRICTING COEFFECTS IN COMONADS. As just demonstrated, we can extend monads so that the reader monad is capable of capturing the semantics of implicit parameters, including the splitting of implicit parameter requirements in lambda abstraction. Can we also go the other way round and *restrict* the comonadic semantics so that all requirements are delayed as in the (*mabs*) rule, thus modelling fully dynamically scoped parameters?

This is, indeed, also possible. Recall that the semantics of lambda abstraction in the flat coeffect calculus is modelled using $\mathsf{merge}_{r,s}$. The operation takes two contexts (wrapped in a comonad $C^{r}\alpha$), combines their carried values and additional contextual information (implicit parameters). To obtain the (*mabs*) rule, we can restrict the first parameter, which corresponds to the declaration-site context:

$$\begin{aligned} &\mathsf{merge}_{r,s}\ : C^{r}\alpha \times C^{s}\beta \to C^{r \cup s}(\alpha \times \beta) && (\mathit{normal})\\ &\mathsf{merge}_{r,s}\ : C^{\emptyset}\alpha \times C^{s}\beta \to C^{s}(\alpha \times \beta) && (\mathit{restricted}) \end{aligned}$$

In the (*restricted*) version of the operation, the declaration-site context requires no implicit parameters and so all implicit parameters have to be satisfied by the call-site. The semantics using the restricted version corresponds to the (*mabs*) rule shown above.

The idea of restricting the operations of the coeffect calculus semantics could be used more generally. We could allow any of the coeffect algebra operations $\circledast, \wedge, \oplus$ to be *partial* and thus the restricted (fully dynamically-scoped) version of implicit parameters could be obtained just by changing the definition of $\wedge$. Similarly, we could obtain e. g. a fully lexically-scoped version of the system. Similar idea has been used for the semantics of effectful computations by Tate [71].

4.7 CONCLUSIONS

This chapter presented the *flat coeffect calculus* – a unified system for tracking contextual properties that are *whole-context*, meaning that they are related to the execution environment or the enitre context in which computations are executed. This is the first of the two *coeffect calculi* developed in this thesis.

The flat coeffect calculus is parameterized by a *flat coeffect algebra* that captures the structure of the information tracked by the type system. We demonstrated how to instantiate the system to capture three specific systems discussed earlier in Chapter 3, namely liveness, data-flow and implicit parameters.

Next, we introduced the notion of *flat indexed comonad*, which is a generalization of comonad, equipped with additional operations needed to provide categorical semantics of the flat coeffect calculus. The indices of the flat indexed comonad operations correspond to the coeffect annotations in the type system and provide a foundation for the design of the calculus.

Finally, we discussed the equational theory for flat coeffect calculus. Although each concrete instance of flat coeffect calculus models different notion of context, there are syntactic properties that hold for all flat coeffect systems satisfying certain additional conditions. In particular, two *subject reduction* theorems prove that the operational semantics for two classes of flat coeffect calculi (including liveness and implicit parameters) can be based on call-by-name reduction.

In the upcoming chapter, we move from *flat* coeffect calculi, tracking whole-context properties to *structural* coeffect calculi, tracking per-variable information, thus covering examples from the second half of Chapter 3.

5 STRUCTURAL COEFFECT LANGUAGE

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

In Chapter 3, we identified two notions of context. We generalized the class of flat calculi that capture whole-context properties in Chapter 4. In this chapter, we turn our attention to *structural* coeffect calculi that capture per-variable properties.

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

5.1 INTRODUCTION

Two examples of flat systems from the previous chapter were liveness and data-flow. As discussed in 3.3, these are interesting for theoretical reasons. However, tracking liveness of the whole context is not practically useful. Structural versions of liveness and data-flow let us track more fine-grained properties. Moreover, the equational theory of flat coeffect calculus did not reveal many useful properties for flat liveness and data-flow. As we show in this chapter, this is not the case with structural versions.

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

5.1.1 *Contributions*

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

- We present a *structural coeffect calculus* with a type system that is parameterized by a *structural coeffect algebra* and can be instantiated to obtain all of the three examples discussed (Section 5.2).
- We give the equational theory of the calculus. We prove the type-preservation property for all structural calculi for both call-by-name and call-by-value (Section 5.4).
- We show how to extend indexed comonads introduced in the previous section to *structural indexed comonads* and use them to give the seman-

tics of structural coeffect calculus (Section 5.3). As with the flat version, the categorical semantics provides a motivation for the design of the calculus.

5.1.2 *Related work*

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

The work in this chapter is more closely related to sub-structural type systems [87]. While sub-structural systems remove some or all of *weakening*, *contraction* and *exchange* rules, our systems keep them, but use them to manipulate both the context and its annotations.

Our work follows the language semantics style in that we provide a structural semantics to the terms of ordinary λ-calculus. The most closely related work has been done in the meta-langauge style, which extends the terms and types with constructs working with the context explicitly. This includes Contextual Modal Type Theory (CMTT) [48], where variables may be of a type $A[\Psi]$ denoting a value of type A that requires context Ψ. In CMTT, $A[\Psi]$ is a first-class type, while structural coeffect systems do not expose coeffect annotations as stand-alone types.

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

5.2 STRUCTURAL COEFFECT CALCULUS

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

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

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

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

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

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

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

5.2.1 *Structural coeffect algebra*

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

Definition 12. *A* structural coeffect scalar $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *is a set* $\mathcal{C}$ *together with elements* $\mathsf{use}, \mathsf{ign} \in \mathcal{C}$*, relation* $\leqslant$ *and binary operations* $\circledast, \oplus$ *such that* $(\mathcal{C}, \circledast, \mathsf{use})$ *and* $(\mathcal{C}, \oplus, \mathsf{ign})$ *are monoids and* $(\mathcal{C}, \leqslant)$ *is a pre-order. That is, for all* $r, s, t \in \mathcal{C}$*:*

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

In addition, the following distributivity axioms hold:

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

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

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

Definition 13. *A* structural coeffect algebra *is formed by a structural coeffect scalar* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *equipped with the following additional structures:*

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

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

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

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

5.2.2 *Structural coeffect types*

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

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

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

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

STRUCTURAL RULES. The remaining rules, shown in Figure 28 (b), are not syntax-directed. They allow different transformation of the free variable context. We include sub-coeffecting (*sub*) as one of the rules, allowing sub-coeffecting on coeffect scalars belonging to individual variables. The remaining rules capture *weakening*, *exchange* and *contraction* known from sub-structural systems.

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

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

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

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

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

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

b.) Structural rules for context manipulation:

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

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

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

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

Figure 28: Type system for the structural coeffect calculus

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

5.2.3 *Understanding structural coeffects*

The type system for structural coeffects appears more complicated when compared to the flat version, but it is in many ways simpler – it removes the ambiguity arising from the use of $\wedge$ in lambda abstraction and, as discussed in Section 5.4, has a cleaner equational theory.

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

This suggests that $\wedge$ and $\oplus$ serve two roles in flat coeffects. First, they are used as over- and under-approximations of $\times$. This is demonstrated by the (*approximation*) requirement introduced in Section 4.4.2, which requires that $r \wedge t \leqslant r \oplus t$. Semantically, flat abstraction combines available context, potentially discarding parts of it (under-approximation), while flat application splits available context, potentially duplicating parts of it (over-approximation)[1].

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

LET BINDING. The other aspect that makes structural systems simpler is that they remove the need for separate let binding. As discussed in Section 4.2.6, flat calculi include let binding that gives a *more precise* typing than combination of abstraction and application. This is not the case for structural coeffects.

Remark 5 (Let binding). *In a structural coeffect calculus, the typing of* $(\lambda x.e_2)\ e_1$ *is equivalent to the typing of* $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$.

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

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

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

5.2.4 *Examples of structural coeffects*

The structural coeffect calculus can be instantiated to obtain the structural coeffect calculi presented in Section 3.3. Two of them – structural data-flow and structural liveness provide a more precise tracking of properties that can be tracked using flat systems. Formally, a flat coeffect algebra can be turned into a structural coeffect algebra (by dropping the $\wedge$ operator), but this does not always give us a meaningful system – for example, it is not clear why one would associate implicit parameters with individual variables.

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

Example 13 (Structural liveness). *The structural coeffect algebra for liveness is formed by* $(\mathcal{L}, \sqcap, \sqcup, \mathsf{L}, \mathsf{D}, \sqsubseteq)$, *where* $\mathcal{L} = \{\mathsf{L}, \mathsf{D}\}$ *is the same two-point lattice as in the flat version, that is* $\mathsf{D} \sqsubseteq \mathsf{L}$ *with a join* $\sqcup$ *and meet* $\sqcap$.

Example 14 (Structural data-flow). *In data-flow, context is annotated with natural numbers and the flat coeffect algebra is formed by* $(\mathbb{N}, +, \mathit{max}, 0, 0, \leqslant)$.

For the two examples that have both flat and structural version, obtaining the strucutral coeffect algebra is easy. As shown by the examples above, we simply omit the $\wedge$ operation. The laws required by a structural coeffect algebra are the same as those required by the flat version and so the above definitions are both valid. Similar construction can be used for the *optimized data-flow* example from Section 4.2.5.

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

[1] Because of this duality, earlier version of coeffects in [57] used $\wedge$ and $\vee$.

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

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

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

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

Example 15 (Bounded variable reuse)**.** *The structural coeffect algebra for tracking bounded variable use is given by* $(\mathbb{N}, *, +, 1, 0, \leqslant)$

Similarly to the structural calculus for data-flow, the calculus for bounded variable reuse annotates each variable with an integer. However, the integer denotes how many times is the variable *accessed* rather than how many *past values* are needed. The resulting type system is the one shown in Figure 18 in Chapter 3.

5.3 CATEGORICAL MOTIVATION

When introducing structural coeffect systems in Section 3.3, we included a concrete semantics of structural liveness and bounded variable reuse. In this section, we generalize the examples using the notion of *structural indexed comonad*, which is an extension of *indexed comonad* structure. As in the previous chapter, the main aim of this section is to motivate and explain the design of the structural coeffect calculus shown in Section 5.2. The semantics highlights the similarities and differences between the two systems.

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

5.3.1 *Semantics of vectors*

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

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

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

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

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

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

5.3.2 *Indexed comonads, revisited*

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

Recall the definition from Section 4.3.3, which defines an indexed comonad over a monoid $(\mathcal{C}, \circledast, \mathsf{use})$ as a triple $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{r,s})$. The triple consists of a family of object mappings C^r, and two mappings that involve context-dependent morphisms of the form $C^r\tau \to \tau'$.

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

Definition 14. *Given a monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *with a point-wise extension of the* $\circledast$ *operator to a vector (written as* $t \circledast \boldsymbol{s}$*) and an operation lifting scalars to vectors* $\langle - \rangle$*, an* indexed comonad *over a category* $\mathcal{C}$ *is a triple* $(C^r, \mathsf{counit}_{\mathsf{use}}, \mathsf{cobind}_{s,\boldsymbol{r}})$*:*

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

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

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

5.3.3 *Structural indexed comonads*

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

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

Definition 15. *Given a structural coeffect algebra formed by* $(\mathcal{C}, \circledast, \oplus, \mathsf{use}, \mathsf{ign}, \leqslant)$ *with operations* $\langle - \rangle$ *and* $\circledast$*, a* structural indexed comonad *is an indexed comonad over the monoid* $(\mathcal{C}, \circledast, \mathsf{use})$ *equipped with families of operations* $\mathsf{merge}_{r,s}$, $\mathsf{split}_{r,s}$, $\mathsf{dup}_{r,s}$ *and* $\mathsf{lift}_{r',r}$ *where:*

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

Such that the following equalities hold:

$$\begin{aligned} \mathsf{merge}_{r,s} \circ \mathsf{split}_{r,s} &\equiv \mathsf{id} \\ \mathsf{split}_{r,s} \circ \mathsf{merge}_{r,s} &\equiv \mathsf{id} \end{aligned}$$

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

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

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

5.3.4 *Semantics of structural caluculus*

The concrete semantics for liveness and bounded variable use shown in Sections 3.3.1 and 3.3.2 suggests that semantics of structural coeffect calculi tend to be more complex than semantics of flat coeffect calculi. The complexity comes from the fact that we need a more expressive representation of the variable context – e. g. a vector of optional values – and that the structural system needs to pass separate variable contexts to the sub-expressions.

The latter aspect is fully captured by the semantics shown in this section. The earlier point is left to the concrete notion of structural coeffect. Our model still gives us the flexibility of defining the concrete representation of variable vectors. We explore a number of examples in Section 5.3.5 and start by looking at the unified categorical semantics defined in terms of *structural indexed comonads*.

$$\llbracket x\!:\!\tau \mathbin{@} \langle \mathsf{use} \rangle \vdash x : \tau \rrbracket\ \mathit{ctx} = \mathsf{counit}_{\mathsf{use}}\ \mathit{ctx} \qquad (\mathit{var})$$

$$\llbracket \Gamma \mathbin{@} \mathsf{ign} \vdash c_i : \tau \rrbracket\ \mathit{ctx} = \delta\ (c_i) \qquad (\mathit{const})$$

$$\begin{array}{l} \llbracket \Gamma_1, \Gamma_2 \mathbin{@} \mathbf{r} \times (t \circledast \mathbf{s}) \vdash e_1\ e_2 : \tau_2 \rrbracket\ \mathit{ctx} = \\ \qquad \mathbf{let}\ (\mathit{ctx}_1, \mathit{ctx}_2) = \mathsf{split}_{\mathbf{r}, t \circledast \mathbf{s}}\ \mathit{ctx} \\ \qquad \mathbf{in}\ \llbracket \Gamma_1 \mathbin{@} \mathbf{r} \vdash e_1 : \tau_1 \xrightarrow{t} \tau_2 \rrbracket\ \mathit{ctx}_1\ (\mathsf{cobind}_{t,\mathbf{s}}\ \llbracket \Gamma_2 \mathbin{@} \mathbf{s} \vdash e_2 : \tau_1 \rrbracket\ \mathit{ctx}_2) \end{array} \qquad (\mathit{app})$$

$$\begin{array}{l} \llbracket \Gamma \mathbin{@} \mathbf{r} \vdash \lambda x.e : \tau_1 \xrightarrow{s} \tau_2 \rrbracket\ \mathit{ctx} = \lambda v. \\ \quad \llbracket \Gamma, x\!:\!\tau_1 \mathbin{@} \mathbf{r} \times \langle s \rangle \vdash e : \tau_2 \rrbracket\ (\mathsf{merge}_{\mathbf{r}, \langle s \rangle}\ (\mathit{ctx}, v)) \end{array} \qquad (\mathit{abs})$$

$$\begin{array}{l} \llbracket \Gamma, x\!:\!\tau_1 \mathbin{@} \mathbf{r} \times \langle \mathsf{ign} \rangle \vdash e : \tau \rrbracket\ \mathit{ctx} = \\ \quad \mathbf{let}\ (\mathit{ctx}_1, \_) = \mathsf{split}_{\mathbf{r}, \langle \mathsf{ign} \rangle}\ \mathit{ctx}\ \ \mathbf{in}\ \ \llbracket \Gamma \mathbin{@} \mathbf{r} \vdash e : \tau \rrbracket\ \mathit{ctx}_1 \end{array} \qquad (\mathit{weak})$$

$$\begin{array}{l} \llbracket \Gamma_1, x\!:\!\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s \rangle \times \mathbf{q} \vdash e : \tau \rrbracket\ \mathit{ctx} = \\ \quad \llbracket \Gamma_1, x\!:\!\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s' \rangle \times \mathbf{q} \vdash e : \tau \rrbracket\ (\mathsf{nest}_{\mathbf{r}, \langle s \rangle, \langle s' \rangle, \mathbf{q}}\ \mathsf{lift}_{s,s'}\ \mathit{ctx}) \end{array} \qquad (\mathit{sub})$$

$$\begin{array}{l} \llbracket \Gamma_1, y\!:\!\tau_2, x\!:\!\tau_1, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle t, s \rangle \times \mathbf{q} \vdash e : \tau \rrbracket\ \mathit{ctx} = \\ \quad \llbracket \Gamma_1, x\!:\!\tau_1, y\!:\!\tau_2, \Gamma_2 \mathbin{@} \mathbf{r} \times \langle s, t \rangle \times \mathbf{q} \vdash e : \tau \rrbracket \\ \qquad (\mathsf{nest}_{\mathbf{r}, \langle t,s \rangle, \langle s,t \rangle, \mathbf{q}}\ \mathsf{swap}_{t,s}\ \mathit{ctx}) \end{array} \qquad (\mathit{exch})$$

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

Assuming the following auxiliary definitions:

$$\begin{array}{l} \mathsf{swap}_{t,s}\ :\ C^{\langle t,s \rangle}(\alpha \hat{\times} \beta) \to C^{\langle s,t \rangle}(\beta \hat{\times} \alpha) \\ \mathsf{swap}_{t,s}\ \mathit{ctx} = \\ \quad \mathbf{let}\ (\mathit{ctx}_1, \mathit{ctx}_2) = \mathsf{split}_{\langle t \rangle, \langle s \rangle}\ \mathit{ctx} \\ \quad \mathbf{in}\ \mathsf{merge}_{\langle s \rangle, \langle t \rangle}\ (\mathit{ctx}_2, \mathit{ctx}_1) \end{array}$$

$$\begin{array}{l} \mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s'},\mathbf{t}}\ :\ (C^{\mathbf{s}}\beta \to C^{\mathbf{s'}}\beta') \to C^{\mathbf{r}\times\mathbf{s}\times\mathbf{t}}(\alpha \hat{\times} \beta \hat{\times} \gamma) \to C^{\mathbf{r}\times\mathbf{s'}\times\mathbf{t}}(\alpha \hat{\times} \beta' \hat{\times} \gamma) \\ \mathsf{nest}_{\mathbf{r},\mathbf{s},\mathbf{s'},\mathbf{t}}\ f\ \mathit{ctx} = \\ \quad \mathbf{let}\ (\mathit{ctx}_1, \mathit{ctx}') = \mathsf{split}_{\mathbf{r}, \mathbf{s}\times\mathbf{t}}\ \mathit{ctx} \\ \quad \mathbf{let}\ (\mathit{ctx}_2, \mathit{ctx}_3) = \mathsf{split}_{\mathbf{s},\mathbf{t}}\ \mathit{ctx}' \\ \quad \mathbf{in}\ \mathsf{merge}_{\mathbf{r}, \mathbf{s'}\times\mathbf{t}}\ (\mathit{ctx}_1, \mathsf{merge}_{\mathbf{s'},\mathbf{t}}\ (f\ \mathit{ctx}_2, \mathit{ctx}_3)) \end{array}$$

Figure 29: Categorical semantics of the structural coeffect calculus

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

$$\begin{array}{rcl} \llbracket x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n \mathbin{@} \langle r_1, \ldots, r_n \rangle \vdash e : \tau \rrbracket & : & C^{\langle r_1, \ldots, r_n \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n) \to \tau \\ \llbracket \tau_1 \xrightarrow{r} \tau_2 \rrbracket & = & C^{\langle r \rangle}\tau_1 \to \tau_2 \end{array}$$

Note that the instances of flat indexed comonad ignored the fact that the variable context wrapped in the data structure is a product. This is not generally the case for the structural indexed comonads – the definitions shown in Section 5.3.5 are given specifically for $C^{\langle r_1, \ldots, r_n \rangle}(\tau_1 \hat{\times} \ldots \hat{\times} \tau_n)$ rather than more generally for $C^{r}\alpha$. The need for examining the structure of the variable context is another reason for using $-\hat{\times}-$ when interpreting expressions in contexts.

EXPRESSIONS. The semantics of strucutural coeffect calculi is shown in Figure 29. As in the previous chapter, the semantics is written in a programming language style using constructs such as let-binding rather than using a categorical (point-free) notation. As before, the semantics can be written using standard primitives (currying, uncurrying, function pairing etc.).

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

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

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

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

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

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

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

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

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

Remark 6 (Correspondence)**.** *In all of the typing rules of the structural coeffect system, the context annotations r and s of typing judgements $\Gamma @ r \vdash e : \tau$ and function types $\tau_1 \xrightarrow{s} \tau_2$ correspond to the indices of mappings C^r and $C^{\langle s \rangle}$ in the corresponding semantic function defined by $[\![\Gamma @ r \vdash e : \tau]\!]$.*

Proof. By analysis of the semantic rules in Figure 29. □

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

5.3.5 *Examples of structural indexed comonads*

The categorical semantics for structural coeffect calculus can be easily instantiated to give a semantics of a concrete calculus. In this section, we look at the three examples discussed throughout this chapter – structural liveness and data-flow and bounded variable reuse. Some aspects of the earlier two examples will be similar to flat versions discussed in Section 4.3 – they are based on the same data structures (option and a list, respectively), but the data structures are composed differently. Generally speaking – rather than having a data structure over a product of variables, we now have a vector of variables over a specific data structure.

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

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

Example 16 (Indexed list for data-flow)**.** *The indexed list model of data-flow computations is defined over a structural coeffect algebra $(\mathbb{N}, +, max, 0, 0, \leqslant)$. The data type $C^{\langle n_1, \ldots, n_k \rangle}$ is indexed by required number of past variables for each individual variable. It is defined over a vector of variables $\alpha_1 \hat{\times} \ldots \hat{\times} \alpha_k$ and it keeps a product containing a current value followed by n_i past values:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 17 (Indexed list for bounded reuse)**.** *The indexed list model of bounded variable reuse is defined over a structural coeffect algebra* $(\mathbb{N}, *, +, 1, 0, \leqslant)$*. The data type* $C^{\langle n_1,\ldots,n_k\rangle}$ *is a vector containing* n_i *values of* i*-th variable:*

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

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

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

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

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

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

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

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

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

Example 18 (Structural indexed option)**.** *Given a structural coeffect algebra formed by* $(\{\mathsf{L},\mathsf{D}\},\sqcap,\sqcup,\mathsf{L},\mathsf{D},\sqsubseteq)$ *and the indexed option data type* D^l*, such that* $D^{\mathsf{D}}\alpha = 1$ *and* $D^{\mathsf{L}}\alpha = \alpha$*, the data type for structural indexed option comonad is:*

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

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

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

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

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

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

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

5.4 EQUATIONAL THEORY

Similarly to the flat version, each concrete instance of the structural coeffect calculus has a different notion of context and thus a different operational interpretation. As before, the properties of the flat coeffect algebra guarantee that certain equational properties hold for all instances of the calculus.

We start the discussion by briefly considering the key aspects that make the equational theory of flat and structural coeffects different.

5.4.1 *From flat coeffects to structural coeffects*

When discussing equational theory for the flat calculus in Section 4.4, we noted that no single technique works universally for all flat coeffect calculi. We considered multiple different reductions that can be used as the basis for operational semantics for calculi satisfying different additional properties.

The structural coeffect calculus has more desirable equational properties. In particular, we can prove both β-reduction and η-expansion using just the properties of strucutral coeffect algebra. For this reason, we focus on these two reductions in this section. Using the terminology of Pfenning and Davies [58], the structural coeffect calculus satisfies both the *local soundness* and the *local completeness* properties.

SUBSTITUTION FOR FLAT COEFFECTS. The more interesting variant of the substitution lemma for flat coeffects (Lemma 3) required all operations of the flat coeffect algebra to coincide. This enables the substitution to preserve the type of expressions, because all additional requirements arising as the result of the substitution can be associated with the declaration context. For example, consider the following example where implicit parameter ?offset is substituted for the variable y:

$$\begin{array}{rll}
y : \mathsf{int} \,@\, \emptyset \vdash \lambda x.y + \mathsf{?total} & : \mathsf{int} \xrightarrow{\{\mathsf{?total}\}} \mathsf{int} & \text{(before)} \\
()\,@\,\{\mathsf{?offset}\} \vdash \lambda x.\mathsf{?offset} + \mathsf{?total} & : \mathsf{int} \xrightarrow{\{\mathsf{?total}\}} \mathsf{int} & \text{(after)}
\end{array}$$

The typing judgement obtained in (*after*) preserves the type of the expression (function value) from the original typing (*before*). This is possible thanks to the non-determinism involved in lambda abstraction – as all operators of the flat coeffect algebra used here are ∪, we can place the additional requirement on the outer context. Note that this is not the *only* possible typing, but it is *permissible* typing.

Here, the flat coeffect calculus gives us typing with limited *precision*, but enough *flexibility* to prove the substitution lemma.

SUBSTITUTION FOR STRUCTURAL COEFFECTS. In contrast, the substitution lemma for structural coeffects can be proven ecause structural coeffect systems provide enough *precision* to identify exactly with which variable should a context requirement be associated.

The following example shows a situation similar to the previous one – here, we use structural data-flow calculus (with the **prev** construct to obtain previous value of an expression) and we substitute $w+z$ for y:

$$\begin{array}{rll} y\!:\!\mathsf{int} @ \langle 2 \rangle \vdash \lambda x.\mathbf{prev}\ (x + \mathbf{prev}\ y) & : \mathsf{int} \xrightarrow{1} \mathsf{int} & \text{(before)} \\ w\!:\!\mathsf{int}, z\!:\!\mathsf{int} @ 2{*}\langle 1,1 \rangle \vdash \lambda x.\mathbf{prev}\ (x + \mathbf{prev}\ (w+z))) & : \mathsf{int} \xrightarrow{1} \mathsf{int} & \text{(after)} \\ w\!:\!\mathsf{int}, z\!:\!\mathsf{int} @ \langle 2,2 \rangle \vdash \lambda x.\mathbf{prev}\ (x + \mathbf{prev}\ (w+z))) & : \mathsf{int} \xrightarrow{1} \mathsf{int} & \text{(final)} \end{array}$$

The type of the function does not change, because the structural type system associates the annotation 1 with the bound variable x and the substitution does not affect how the variable x is used.

The other aspect demonstrated in the example is how the coeffect of the substituted variable affects the free-variable context of the substituted expression. Here, the original variable y is annotated with 2 and we substitute it for an expression $w+z$ with free variables w, z annotated with $\langle 1,1 \rangle$. The substitution applies the operation $\circledast$ (modelling sequential composition) to the annotation of the new context – in the above example $2{*}\langle 1,1 \rangle = \langle 2,2 \rangle$.

5.4.2 *Holes and substitution lemma*

As demonstrated in the previous section, reduction (and substitution) in the structural coeffect calculus may need to replace a *single* variable with a *vector* of variables. Furthermore, we may also need to substitute for multiple variables in the variable context.

For example, the expression $\lambda x.x+x$ is type-checked by type-checking x_1+x_2, contracting x_1 and x_2 and then applying lambda abstraction. During the reduction $(\lambda x.x+x)\ (y+z)$ we thus need to substitute y_1+z_1 for x_1 and y_2+z_2 for x_2. This is similar to substitution lemma in other structural variants of λ-calculus, for example in the bunched typing system [49]. To express the substitution lemma, we define the notion of a *context with holes*

Definition 16 (Context with holes)**.** *A* context with holes *is a context such as* $x_1:\tau_1,\ldots,x_k:\tau_k @ \langle r_1,\ldots,r_k \rangle$, *where some of the variable typings* $x_i:\tau_i$ *and corresponding coeffects* r_i *are replaced by* holes *written as* $-$.

$$\Delta[-@-]_n = \Delta[\underbrace{-@-\,|\,\ldots\,|\,-@-}_{n-\mathit{times}}]$$

$$\begin{array}{lll} \Delta[-@-]_n & := -, \Gamma @ \langle - \rangle \times s & \text{where } \Gamma @ s \in \Delta[-@-]_{n-1} \\ \Delta[-@-]_n & := x\!:\!\tau, \Gamma @ \langle r \rangle \times s & \text{where } \Gamma @ s \in \Delta[-@-]_{n} \\ \Delta[-@-]_0 & := () @ \langle \rangle & \end{array}$$

A context with n holes may either start with a hole, followed by a context with $n-1$ holes, or it may start with a variable followed by a context with n holes. Note that the definition ensures that the location of variable holes correspond to the locations of coeffect annotation holes. Given a context with holes, we can fill the holes with other contexts using the *hole filling* operation and obtain an ordinary coeffect-annotated context.

Definition 17 (Hole filling). *Given a context with* n *holes* $\Delta @ \boldsymbol{s} \in \Delta[-@-]_n$*, the hole filling operation written as* $\Delta @ \boldsymbol{s}[\Gamma_1 @ \mathbf{r_1} \mid \ldots \mid \Gamma_n @ \mathbf{r_n}]$*, which replaces the holes by the specified variables and corresponding coeffect annotations, is defined as:*

$$\begin{aligned}
-, \Delta @ \langle - \rangle \times \boldsymbol{s}\, [\Gamma_1 @ \boldsymbol{r}_1 \mid \Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \ &= \ \Gamma_1, \Gamma_2 @ \boldsymbol{r}_1 \times \boldsymbol{r}_2 \\
&\quad \text{where } \Gamma_2 @ \boldsymbol{r}_2 = \Delta @ \boldsymbol{s}[\Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \\
x_1 : \tau, \Delta @ \langle r_1 \rangle \times \boldsymbol{s}\, [\Gamma_1 @ \boldsymbol{r}_1 \mid \Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \ &= \ x_1 : \tau, \Gamma_2 @ \langle r_1 \rangle \times \boldsymbol{r}_2 \\
&\quad \text{where } \Gamma_2 @ \boldsymbol{r}_2 = \Delta @ \boldsymbol{s}[\Gamma_1 @ \boldsymbol{r}_1 \mid \Gamma_2 @ \boldsymbol{r}_2 \mid \ldots] \\
() @ \langle\rangle\, [\,] \ &= \ () @ \langle\rangle
\end{aligned}$$

When we substitute an expression with coeffects $\mathbf{t}$ (associated with variables Γ) for a variable that has coeffects s, the resulting coeffects of Γ need to combine $\mathbf{t}$ and s. Unlike in the flat coeffect systems, the structural substitution does not require all coeffect algebra operations to coincide and so the combination is more interesting than in the bottom-pointed substitution for flat coeffects, where it used the only available operator (Lemma 3).

Substitution can be seen as sequential composition. Informally – we first need to obtain the value of the expression (requiring $\mathbf{t}$) and then use it in context with requirements s. Thus the free variables of the expression *after* substitution are annotated with $s \circledast \mathbf{t}$, using the (scalar-vector extension) of the sequential composition operator $\circledast$.

Lemma 4 (Multi-nary substitution). *Given an expression with multiple holes filled by variables* $x_i : \tau_i$ *with coeffects* s_k*:*

$$\Gamma @ \boldsymbol{r}\, [x_1 : \tau_1 @ \langle s_1 \rangle \mid \ldots \mid x_k : \tau_k @ \langle s_k \rangle] \vdash e_r : \tau_r$$

and a expressions e_i *with free-variable contexts* Γ_i *annotated with* $\mathbf{t_i}$*:*

$$\Gamma_1 @ \boldsymbol{t}_1 \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma_k @ \boldsymbol{t}_k \vdash e_k : \tau_k$$

then substituting the expressions e_i *for variables* x_i *results in an expression with a context where the original holes are filled by contexts* Γ_i *with coeffects* $s_i \circledast \mathbf{t_i}$*:*

$$\Gamma @ \boldsymbol{r}\, [\Gamma_s @ s_1 \circledast \boldsymbol{t}_1 \mid \ldots \mid \Gamma_s @ s_k \circledast \boldsymbol{t}_k] \vdash e_r[x_1 \leftarrow e_1] \ldots [x_k \leftarrow e_k] : \tau_r$$

Proof. By induction over $\vdash$, using the multi-nary aspect of the substitution in the proof of the contraction case. See Appendix ?. □

5.4.3 *Reduction and expansion*

TODO: Exterminate!

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

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

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

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

Proof. Following from Lemma **??** and β-equality. □

Structural coeffect systems also exhibit η-equality, therefore satisfying both *local soundness* and *local completeness* conditions set by Pfenning and Davies [58]. This means that abstraction does not introduce too much, and application does not eliminate too much.

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

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

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

5.5 CONCLUSIONS

TODO: (...)

6 APPENDIX A

6.1 INTERNALIZED SUBSTITUTION

6.1.1 *First transformation*

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

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

to

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

meaning

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

6.1.2 *Second transformation*

Second transformation

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

or

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

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

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

require

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

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] 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.

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

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

[25] 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.

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

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

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

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

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

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

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

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

[41] 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.

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

[43] 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.

[44] 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.

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

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

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

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

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

[50] 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.

[51] D. Orchard. Programming contextual computations.

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

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

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

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

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

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

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

[59] 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.

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

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

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

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

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

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

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

[71] 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.

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

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

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

[75] 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.

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

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

[78] 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.

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

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

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

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

[83] 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.

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

[85] 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.

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

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